A Beginner to Pro. Learn how to build Advanced Flutter Apps

BEGINNING FLUTTER 3.0 WITH DART

SANJIB SINHA

Beginning Flutter 3.0 with Dart

A Beginner to Pro. Learn how to build Advanced Flutter 3.0 Apps

Sanjib Sinha

This book is for sale at http://leanpub.com/beginningflutterwithdart

This version was published on 2022-05-29

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. Getting Started

Before getting started, let me tell you one thing. Always use the latest Provider package - https://pub.dev/packages/provider[1] for state management. And always maintain the Null Safety - https://flutter.dev/docs/null-safety[2].

I also strongly recommend to read the latest and updated articles on Flutter - https://sanjibsinha.com/category/flutter [3].

Download latest Flutter 3.0

To start with, we need to download the Flutter framework.

That is our first task. We need to go to The installation page of Flutter - https://flutter.dev/docs/get-started/install[4] page from where we will download and install Flutter according to your operating system.

If you have been working with Flutter 2.*, then just issue the following command:

```
flutter upgrade
```

However, Flutter gets updated and upgraded quite often. For example, Text Buttons to take inputs from users are no longer the same now.

We will start with Windows, first.

Want to read more old and archived Flutter related Articles and resources?

For more old and archived Flutter related Articles and Resources[5]

Before that we want to make one thing clear.

What is new in Flutter 3.0

A few days ago Google announced the official release of Flutter 3.0. Let's see what is new in Flutter 3.

[1] https://pub.dev/packages/provider
[2] https://flutter.dev/docs/null-safety
[3] https://sanjibsinha.com/category/flutter/
[4] https://flutter.dev/docs/get-started/install
[5] https://sanjibsinha.com

Firstly, with reference to mobile application development, there has not been a great change. Structurally what we have been doing, will continue to do.

Certainly a change in here and there had taken place. In the next section we'll discuss that and will take a deep dive.

Secondly, flutter web sections have got the makeover. Certainly it has become better.

Finally, a lot of changes have taken place in the desktop part.

Three months ago Google Flutter and Dart team announced Flutter support for Windows.

Flutter 3.0 is stable for macOS and Linux.

With reference to macOS and Linux, a lot of changes have taken place.

Now it's ready for production on all desktop platforms. As a result, we can now create platform-rendered menu bars on macOS. To do that we can use the "PlatformMenubar" widget that inserts platform-only menus.

It also supports accessibility services such as screen-readers, accessible navigation, and inverted colours.

Full support for international text input on all desktop platforms is also there.

Web updates in flutter 3.0

Let's talk about the Web updates in flutter 3.0.The "ImageDecoder" API plays an important role in web applications.

Therefore flutter web now uses the "ImageDecoder" API in browsers that support it.

And there are more.

We can also use other widgets in flutter web.

For example we can think of the splash screen, loading indicator etc.

Most importantly, Flutter 3.0 has improved the performance.

It is faster than before.

As an outcome it builds frames 20 percent faster which is a significant progress.

Last but not least, we should mention two interesting things that have caught our attention.

Firstly, Flutter 3.0 supports foldable mobile devices. Secondly, Flutter 3.0 supports Material Design 3.

In the next section we'll talk about it in detail.

Who should read this book?

Are you an absolute beginner who without having any prior knowledge of programming language wants to build a mobile application? Well, then this book is for you. This book is not for intermediate or experienced learners or developers.

We will try to add two things to our knowledge so that we will start building your mobile application. First we will learn Flutter, a framework or tool that helps us to build the mobile application. Second, we will learn a programming language called Dart, with which Flutter works.

If we do not understand the basic syntax and semantics of Dart, we will not be able to understand the internal activities of Flutter.

We will learn both, Flutter and Dart side by side. For instance, if we find something like function and object or named parameters in a constructor, we will learn that concept in Dart.

If you have no knowledge of programming language, or you have not written a single line of code, you need not worry. We will go very smooth, we will have plenty of screenshots that will explain what we are going to do. We will also learn the basic concepts of programming language through Dart; it is important, because otherwise we will not be able to understand how Flutter framework works.

If you have any question, please do not hesitate to send me a mail at:

sanjib12sinha@gmail.com

Flutter for Windows

Clicking the download button will automatically start downloading ziped Flutter in your Download folder. It would be around 700 MB in size. While extracting the file it would take around 1.30 GB place of your hard drive. You may copy that extracted file to elsewhere, or you may keep it there (figure 1.1).

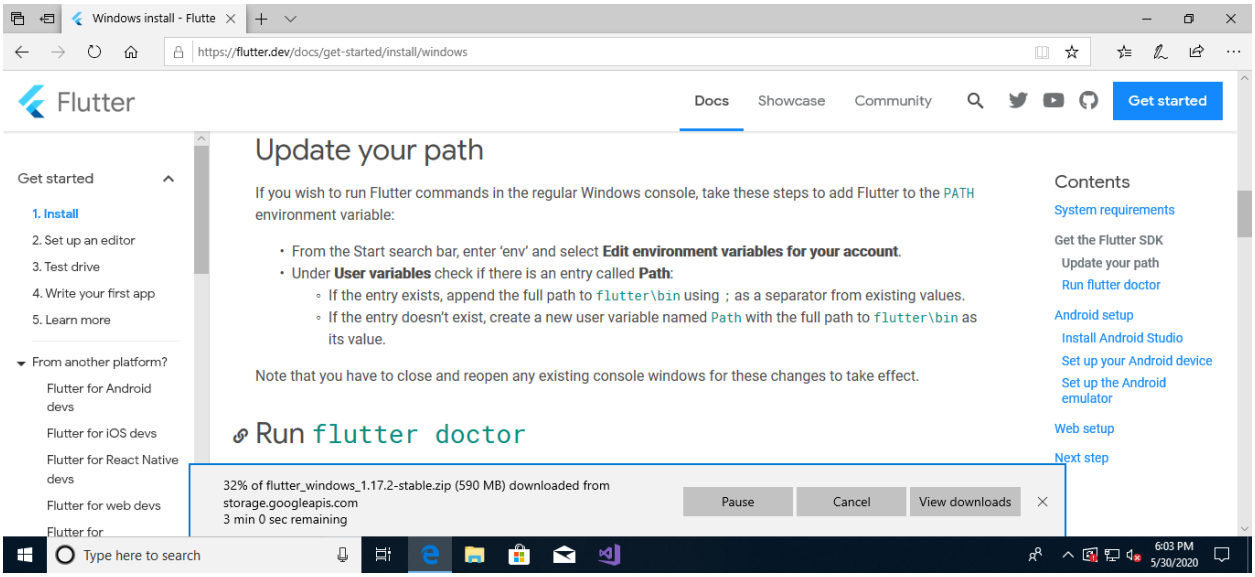

Figure 1.1 – Downloading Flutter for Windows

We have kept the extracted flutter folder there and created a new 'environment' path for the user. Because we want to work through the command prompt, in future, we have created this global environment path. Creating a new environment variable path in any Windows operating system is also easy. In the Windows 10 operating system, we type 'environment variable' in the search prompt, it will automatically open up the related window for us.

We can copy and paste the whole path there as the following:

```
1  "C:\Users\Downloads\flutter\bin".
```

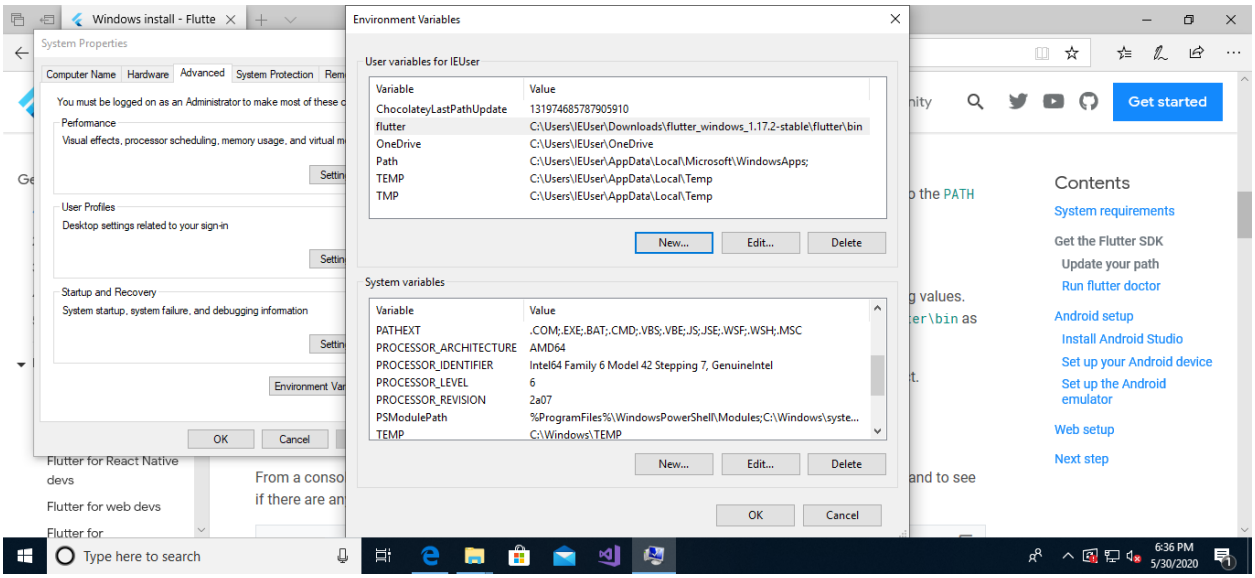

Figure 1.2 – Creating the new environment variable path in Windows 10

Now, we can open the command prompt and type ‘flutter doctor’ to see whether we have any Flutter related IDE installed already. It will also check whether we have any connected device or not.

We have not installed Android Studio or any other Flutter related IDE beforehand. The command ‘flutter doctor’ has detected that (Figure 1.3).

To work with Flutter, we need a good IDE. In fact, when we were downloading Flutter, it indicated that we should install Android Studio or any good IDE where we would have a connected device. The connected device is nothing but a virtual mobile device where we can see and test our mobile application.

Android Studio should be the best choice. It is widely used and Flutter home page also suggests to download and install that IDE.

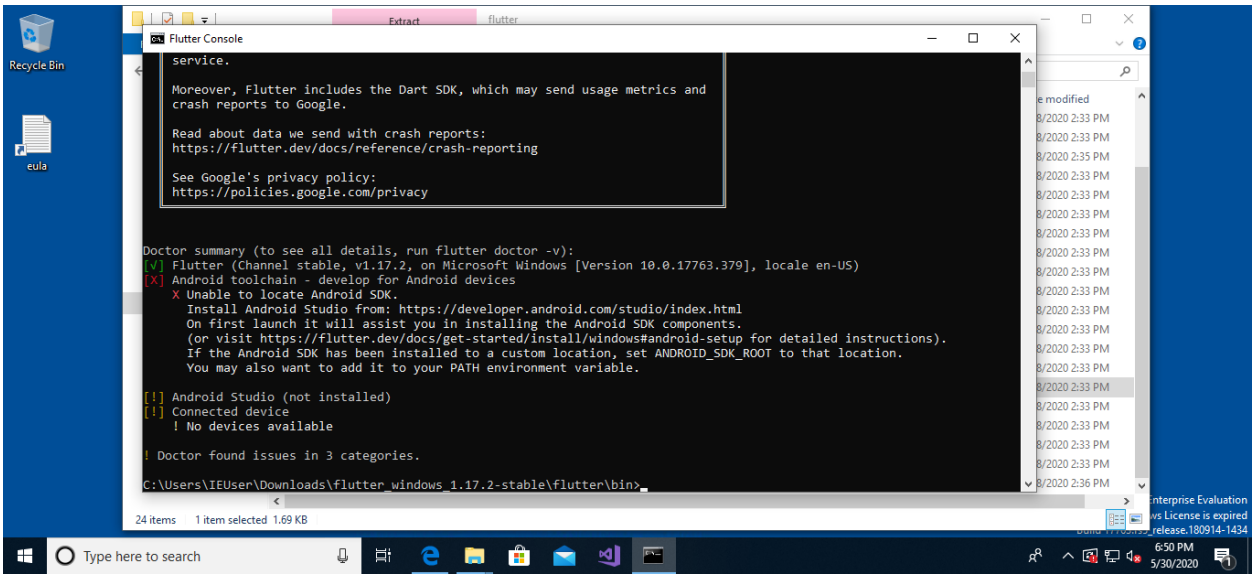

Figure 1.3 – Flutter Doctor Summary in Windows 10

In Flutter Doctor summary, we have found that Android Studio has not been installed and there is no device available.

We will do that in our macOS and Linux machines, because we will use any one of that operating system to learn Flutter and Dart together.

Flutter for macOS and Linux

Downloading Flutter for macOS and Linux is same. It will download the “flutter_linux_1.17.2-stable.tar.xz” file in your “Downloads” folder.

Next we will issue the following command to extract Flutter, on our terminal:

```
//code 1.1
tar xf flutter_linux_1.17.2-stable.tar.xz
```

Now we can copy this extracted 'flutter' directory to a suitable place, where we will build our first mobile application. In the 'Documents' directory, we have created another directory named 'development'. We will keep the extracted 'flutter' directory there.

Just like Windows 10, we will now set the global path for 'flutter', so that we can use 'flutter' command, anywhere in our machine, in the future.

We will do that using 'vim' or 'nano' text editor, that works on the terminal. By the way, the commands are same for any macOS or Linux operating system.

If you type the following command, the nano text editor will open up the 'bashrc' file.

```
//code 1.2
nano ~/.bashrc
```

At the end of the 'bashrc' file we will add this line:

```
//code 1.3

export PATH=$PATH:/home/ss/Documents/development/flutter/bin:$PATH
```

We have to mention the full path as given above. We have kept our extracted 'flutter/bin' folder in the '/home/ss/Documents/development' directory.

Our next step will be to download the Android Studio. Download the zipped folder and extract it anywhere in the machine. We have kept it in our '/home/' directory. Next, issue this command:

```
//code 1.4
ss@ss-desktop:~$ cd android-studio/bin/
ss@ss-desktop:~/android-studio/bin$ ./studio.sh
```

It will open up the Android Studio for us (figure 1.4). Once the Android Studio opens up, you can go to the 'open folder' option and choose the flutter project we have created already. How we have created it, we will come to that point in a minute.

Before that, we need to see the Android Studio and our newly created virtual device.

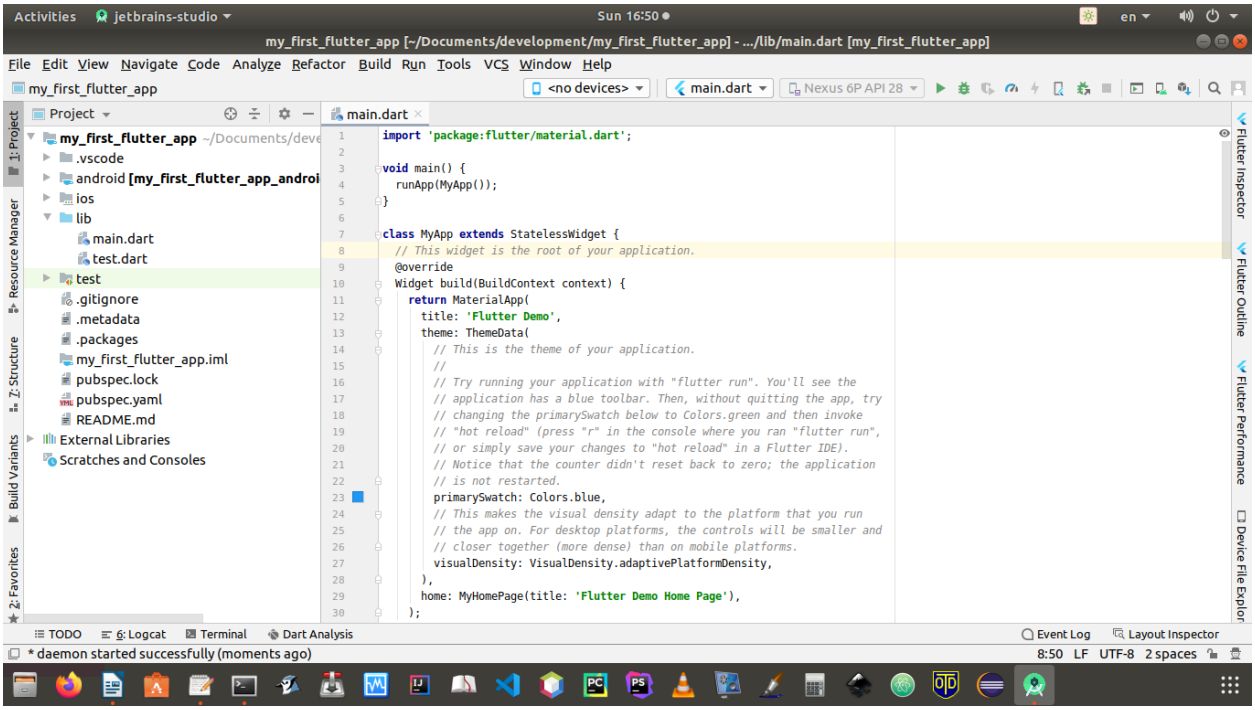

Figure 1.4 – Android Studio and our first flutter project

Before opening the Android Studio, we have opened up our terminal, and typed the following commands to reach to the newly installed 'flutter' directory.

```
//code 1.5
ss@ss-desktop:~$ cd Documents/development/flutter/
ss@ss-desktop:~/Documents/development/flutter$ flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, v1.17.2, on Linux, locale en_IN)

[✓] Android toolchain - develop for Android devices (Android SDK version 29.0.3)
[✓] Android Studio (version 3.5)
[✓] Android Studio (version 4.0)
[✓] IntelliJ IDEA Community Edition (version 2019.3)
[✓] VS Code (version 1.43.2)
[!] Connected device
    ! No devices available

! Doctor found issues in 1 category.
ss@ss-desktop:~/Documents/development/flutter$
```

As you have seen in the above output, 'flutter doctor' has found only one issue. It has not found any connected device. Otherwise, we have already installed Android Studio (version 4.0), which is the

latest at the time of writing this book. We have also installed IntelliJ IDEA Community Edition, and we have also Visual Studio Code IDE.

We can use the virtual device from Android Studio, but we can use the Visual Studio Code IDE or IntelliJ IDEA Community Edition IDE for writing our code. They will automatically synchronize with the connected device.

Once you have installed, please check the Flutter latest version and issue the following command:

```
flutter upgrade
```

However, before that we need to create our first flutter project with the help of flutter command as the following:

```
//code 1.6
flutter create my_first_flutter_app
```

Remember one thing. When we want to create a new flutter project, we should always create like this. The naming convention is important here. We can only use the underscore between the words. No hyphen or space is allowed.

Now the time has come to go back to the Android Studio. We will pick up the 'open folder' option and choose to open the newly created flutter project. We have named it as: 'my_first_flutter_app'.

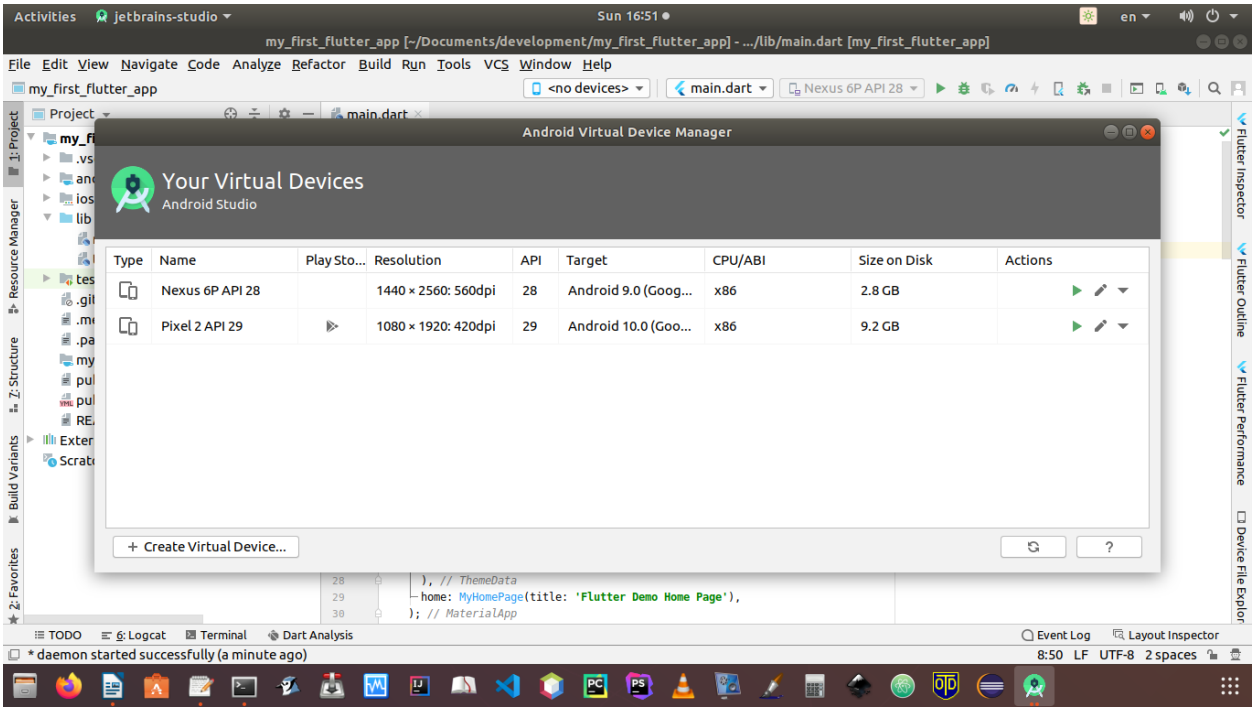

Figure 1.5 – Open the Android Virtual Device (AVD) manager from tools menu

To open up the connected device, we need to open the Android Virtual Device manager, or AVD manager in short.

You will get that from the 'tools' menu.

Select any one of them and click the 'green' play button on the far right hand side of any virtual device. It will automatically open up the 'connected device' (Figure 1.6).

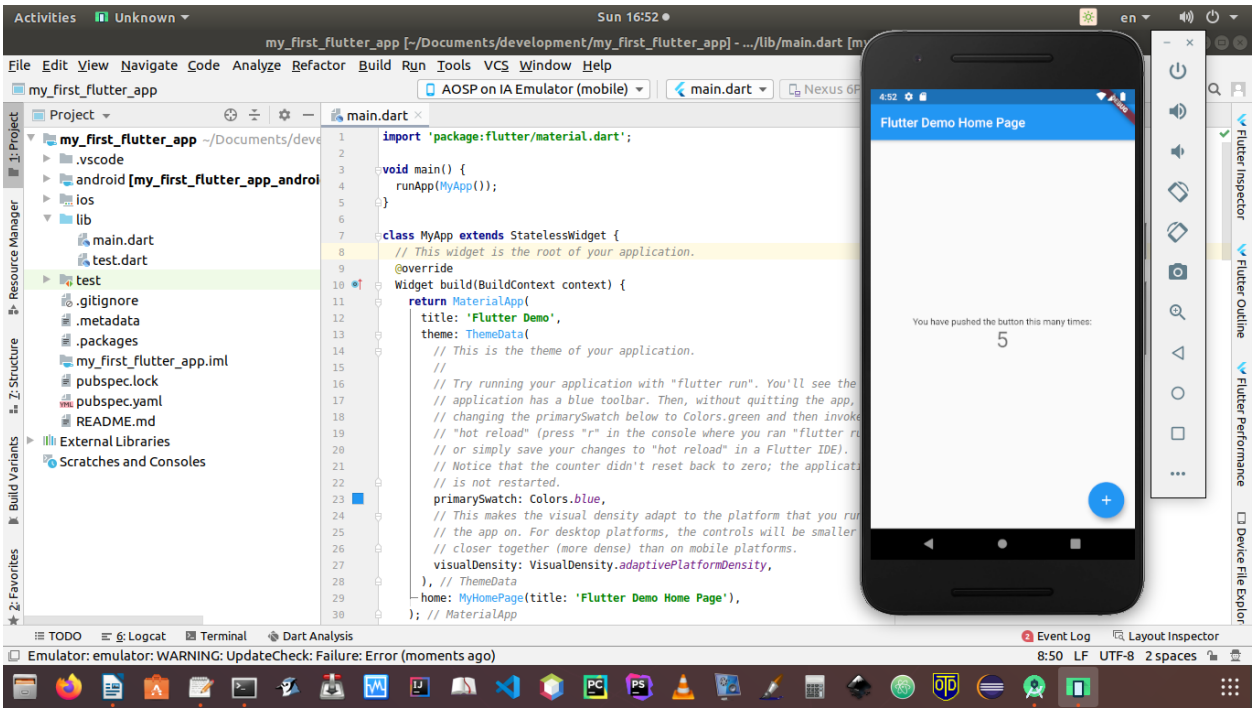

Figure 1.6 – We have the connected device on which we can test our first mobile application

Now everything is ready. We can start building our first mobile application from scratch using Flutter and Dart. Before closing down this section, we should know a few good tips. Usually, the beginners encounter a few errors when they try to run the command:

```
flutter doctor
```

If it gives any error, try this command:

```
flutter doctor --android-licenses
```

It will ask you to accept the license. Accept it, and it will not give any error anymore. Another problem often gives trouble to the new developers.

As a beginning Flutter developer, people often are stuck with this issue. They cannot launch the virtual mobile device while working with Android Studio.

We want that every code we write should reflect on the virtual device. It can be done by going to the 'AVD manager' from tools. But sometimes an ugly error pops up its head and tells that

‘/dev/kvm permission denied’. In Ubuntu 18 or Mac OS, you can give user the permission by issuing this command:

```
//code 1.7
sudo chmod 777 -R /dev/kvm
```

But it has a drawback. If someone else uses your machine, then the other user also gets the permission. The best remedy is – give permission to yourself only by the following commands:

```
//code 1.8
sudo apt install qemu-kvm
sudo adduser your-username kvm
sudo chown your-username /dev/kvm
```

It will solve the issue for ever. Now you can launch any virtual device you want. You can launch the device with your Android Studio, and work with any other IDE like IntelliJ or Visual Studio.

What are important concepts in Dart

In dart everything is Object. What does that mean actually? Let's learn the important concepts.

Are you planning to learn Flutter? Then learn the Dart Programming Language. At least, learn the important concepts in Dart. And the basics.

It will help you to grasp the concepts of Flutter-Framework easily.

We presume, you are a beginner. You have no coding background.

But do not worry. We will start from the beginning. Let us learn the important concepts first.

Firstly, Dart is an Object-Oriented-Programming Language.

What does that mean?

It means in Dart, everything is Object. To understand this, look at the world around you.

We cannot live without this planet. Therefore, if this planet is an object, it must be very powerful. Because this "object" planet contains several other "objects".

We, humans are also a kind of "object". Think about the trees. Trees are also objects.

Basically, the Object-Oriented-Programming Language emulates the real world. As a result, we can quite easily build relationships between different "types" of "objects".

Now, as you probably notice, another word "type" enters our Dart-Vocabulary.

Every object has a "type".

And we should define the "Type" before we create an "Object".

We can define the "type" of the Planet by the following way.

```
1  class Planet {}
```

The same way, we can create another object Tree. And we can define the type the same way.

```
1  class Tree {}
```

We can create a planet object, or a tree object in a very simple way.

```
1  var planet = Planet();
2  var tree = Tree();
```

Since these two objects are of different types, they are not equal.

Subsequently, we can check that in the top-level main() function.

```
void main() {
var planet = Planet();
var tree = Tree();
planet.runtimeType == tree.runtimeType ? print('Yes, Planet is Tree')
    : print('No, Planet is not Tree.');

// Output: No, Planet is not Tree.
}

class Planet {}
class Tree {}
```

As a result, we have learned two most important concepts.

Dart has many “types” of “objects”. We will learn them as we progress. We will also learn how to write a Class, and create an Object in detail.

As an absolute beginner you just remember a few rules.

```
Dart is an Object-Oriented-Programming.

Everything in Dart is Object.

Everything we can place in a variable is an Object.

Every Object has its own Type. Therefore two different type of Objects are not same.

Last but not least. We can place any “Type” of “Object” in a “Variable”. As a result\
, a “Variable” can change the value of that “Type” of “Object” only. But the “Variab\
le” cannot change the “Type”.
```

We can take a look at the following code. The syntax may be unknown. However, as a beginner you do not have to worry.

We will learn them in the coming sections.

```dart
main() {
/// everything in Dart is Object
/// Object is an instance of Class
///

Person? person = Person();
person.name = 'I am a Person Object in Dart';
print(person.name);
// I am a Person Object in Dart

Employee? emp = Employee();
emp.person = Person();
emp.person!.name = 'I am an Employee Object with name X';
print(emp.person!.name);
// I am an Employee Object with name X
/// Now we can place the objects in variables
///
var placeHolder = emp;
placeHolder.person!.name = 'I am a variable, I can change my value.'
    ' But I cannot change my Type.';
print(placeHolder.person!.name);
// I am a variable, I can change my value. But I cannot change my Type.
/// changing value
///
placeHolder.person!.name = 'I am a variable, so I am changing my value.'
    ' But I cannot change my Type.';
print(placeHolder.person!.name);
// I am a variable, so I am changing my value. But I cannot change my Type.

// placeHolder = person;
// we cannot change the Type
}

class Person {
late String? name;
}

class Employee {
late Person? person;
}

/**
I am a Person Object in Dart
```

```
I am an Employee Object with name X
I am a variable, I can change my value. But I cannot change my Type.
I am a variable, so I am changing my value. But I cannot change my Type.

*/
```

Please read the comments we have written in the above code.

That will explain many things.

A Few Words About DART IDE

We need a good Integrated Development Environment or IDE to build Application in Dart Programming Language.

We can use any good IDE.

However, our first choice is Visual Studio Code.

Visual Studio Code is a source-code editor made by Microsoft for Windows, Linux and macOS.

It has many features. Most importantly we can debug our code. And besides, it has support for syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git.

The Android Studio is also good. Even you can download and install the IntelliJ IDEA Community Edition.

To test our code, we also need to install Dart SDK in our local system.

If you install Flutter and Dart plugin in any IDE, that will also work. In that case, you don't need Dart SDK in your Operating System.

Installing any IDE in Windows is comparatively easy. Download the ".exe" file from the official web site and double-click to launch it.

This is recommended. You can also download the ZIP file and unpack it to the program files. You will find the android-studio bin folder where you can launch the respective ".exe" files.

However, downloading the ".exe" file from the official web site and launching it online is recommended.

Installing Android Studio on MAC is not a complicated process.

You need to launch Android Studio DMG file and then drag and drop the Android Studio into the Applications folder. After that, the launching process is easy.

The set-up Wizard will guide you through the rest part. For development, you need to download the Android SDK components.

I will recommend to use Linux as the main operating system.

Why?

Because Android as a framework will always execute better on top of the Linux Kernel.

Installing Dart SDK in Linux is also easy.

Why do we need the Dart SDK?

Because it has the libraries and command-line tools that we need to develop all kinds of Dart applications, web, command-line or server apps.

To install Dart, first open your terminal, and then you can issue the following commands:

```
    sudo apt-get update

    sudo apt-get install apt-transport-https

    sudo sh -c 'curl https://dl-ssl.google.com/linux/linux_signing_key.pub | apt-key\
 add -'

    sudo sh -c 'curl https://storage.googleapis.com/download.dartlang.org/linux/debi\
an/dart_stable.list > /etc/apt/sources.list.d/dart_stable.list'
After that, install the stable release of the Dart SDK.


sudo apt-get update

sudo apt-get install dart
```

After that you can check your Dart version:

```
$ dart --version
Dart SDK version: 2.15.1 (stable) (Tue Dec 14 13:32:21 2021 +0100) on "linux_x64"
```

Installing Android Studio on Linux, is quite simple and user friendly.

You don't have to issue any command-line instructions.

Download the ZIP file and unpack it to either "/usr/local/" or "/opt/" for shared users.

Now, navigate to the "/android-studio/bin/" directory and execute the "studio.sh" file with the help of this command:

```
./studio.sh
```

If it asks to install the required libraries for 64-bit Linux machines.

According to the instruction, install it.

If you are a first time user of Android Studio, you can import previous Android Studio settings.

Or, you may skip it by clicking the OK button.

The Android Studio Wizard will guide you to set it up; remember, this set-up includes downloading Android SDK components.

It is required for development. Moreover, in the configure section you can install Flutter and Dart plugins.

To get maximum of Android Studio in a 64-bit Linux machine, we need to install some 32-bit libraries.

The following command-line instructions will work for us.

```
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386 lib32z1 libbz2-1.0:\
i386
```

The command will ask for the root password. For the 64-bit Fedora, the command is different.

```
sudo yum install zlib.i686 ncurses-libs.i686 bzip2-libs.i686
```

Now, we are ready to work in Android Studio.

Comparatively, installing the IntelliJ Community Edition is much easier. You can install it from the Ubuntu Software Center.

Open the Software centre and type the IntelliJ Community Edition. It will show up.

You can also install the IntelliJ Community through command-line instruction on the terminal:

```
sudo snap install intellij-idea-community –classic
```

The applications in the Ubuntu Software centre are basically “snap” packages.

Therefore, if we already have “snap” packages installed in our machine, we can install it through the terminal.

After the primary installation is over, don’t forget to install the Dart plugins.

We keep our Dart files in the “bin” folder, and run the program to get the output on the Console below.

Why Dart Language?

We can answer this question in one sentence.

The Dart Programming Language is a great fit for both – Mobile Apps and Web Apps. It is free and open source.

Moreover, the Dart repository is available at https://github.com/dart-lang. And at the same time, you may get the feel of the language at the official website: https://www.dartlang.org/.

Although we can use the Dart for both mobile and web development, yet it is more popular for building Mobile Application.

For the same reason, the Dart became popular along with the Flutter framework for developing cross-platform mobile apps.

What is Dart and flutter?

Flutter is an Open-Source UI SDK developed by Google.

The framework Flutter allows the development of iOS and Android applications.

Since the Flutter uses Dart as the programming language, learning Dart helps us to learn the Flutter also.

Dart is an Open-Source, programming language. It is easy to learn.

Therefore we can learn Dart to build stable, and creates high-performance applications.

This book serves as a good introduction to the Dart Programming Language.

Why?

Because we have designed it to give you a complete idea about how Dart works.

For small operations, you can use the online code editor: https://dartpad.dartlang.org.

However, for package building and creating projects, we need the Code editors like Visual Studio Code, Android Studio, or IntelliJ Community.

The Visual Studio Code also has Dart language testing support.

It is easy to install required Plugins. Furthermore, if we want to build Mobile Applications, using Dart and Flutter, they are more useful.

Let us examine why we should learn the Dart?

What is Dart best for?

Firstly, it is extremely productive.

Secondly, if you already know any object-oriented programming language such as C++, C# or Java, it is easy to learn the Dart language.

Thirdly, if you are an absolute beginner, then it is good that you start with the Dart which has clear and concise syntax.

Finally, you will also have great support of rich and powerful core libraries and thousands of packages.

As an absolute beginner, you don't have to worry about the libraries now. We will learn together to use them later when the time comes.

The performance of the Dart is high across mobile devices and the web because it optimizes the compilation power.

Besides, its portability rate is extremely good.

It compiles to ARM and x86 code so that Dart mobile apps can run on iOS and Android and beyond.

Now here is a complimentary note for the absolute beginners.

There is a difference between ARM and X86 processors.

The ARM processors follow a RISC (Reduced Instruction Set Computer) architecture, while x86 processors are CISC (Complex Instruction set Architecture).

All together, because of these features, x86 processors are considered to be faster than ARM. And at the same time it is fast for web apps also.

Let us see our first Dart code.

```
main() {
print("Hello World!");
}

//output
Hello World!
```

Let us write some more console based code to get the feel of Dart.

At the same time, we will know what are the most basic syntax of the Dart and how they work together.

```
main() {
print("Hello World!");
//calling a function
doSomething();
}
//define a function
doSomething(){
print("Do something!")
//calling a function inside another function
    lifeIsShort();
}
//defining another function
lifeIsShort(){
print("Life is too short to do so many things.");
}
```

In the above code, we have defined two functions first.

Next, we have nested another function inside one function.

finally, we have called them together through a single function.

However, there is a mistake in our code. It is an intended mistake so that you understand how debugging takes place in Dart.

Let us watch the output:

```
bin/main.dart:12:24: Error: Expected ';' after this.
print("Do something!")
```

We have not place a semicolon after displaying an output.

Let us correct it and run the program again. In Android Studio you may use “Shift+F10” to run the code.

Now it is OK.

```
Hello World!
Do something!
Life is too short to do so many things.
```

We have learned many things in our first code.

The very first lesson teaches us the most basic thing of all knowledge. You learn from your mistake.

Is Dart easy?

Yes, the Dart programming language is very yes.

However, we should always be careful about the syntactical errors. Missing a semicolon or a dollar sign before a variable could be a big game changer.

Especially, when you are going to build a large scale mobile application on iOS or Android, be very careful about these small mistakes.

Syntax-wise Dart has similarities with C, C#, Python, Java and JavaScript.

You have seen how we have used the “comments” in our code.

Try to contribute as much comment as possible to clear your standpoint as a developer.

It is necessary because when another person reads your code, she will understand it. Moreover she will visualize it as you have visualized your code while writing.

We will discuss comments at the right time.

We have started our code with the top level function “main()”.

It is required and special in nature because here the application executes.

So inside the main() function we have called a function “doSomething()” which has a nested-function inside it “lifeIsShort()”.

Each function gives a display output with the function “print()”. It is a handy way to display any output. Consequently, we have covered many things in our first program.

Variable in Dart

In Dart everything is an Object. Therefore a variable in Dart stores references to an Object.

Let us create a variable and initialise.

```
1  var name = 'Mutudu';
```

As the variables store references, in the above example the same thing happens.

The variable called “name” contains a reference to a “String” object with a value of “Mutudu”.

Every Object in Dart has a “type”. In the above code, the “type” of the “name” variable is inferred to be a “String”.

Why?

Because we have not specified it, or mentioned it.

On the contrary, we could have specified the “type”.

```
String name = 'Mutudu';
```

There is another possibility. An Object may not be restricted to any particular “type”.

In that case, we can initialise it like the following way.

```
Object name = 'Mutudu';
```

With reference to the above discussion, we have learned a few important facts about a variable.

In Dart, a variable is always associated with a “type”. We can either specify the “type”, or we can allow the variable to infer it.

Like any other programming language, the Dart has many types. We will discuss built in types in the coming section.

So far we have seen only one “Type”. The String.

However, there are other types also. The integer, the boolean and a few others.

We have also learned that the Dart allows us to be strongly-typed. And at the same time we can be duck-typed.

The phrase “strongly-typed” means we specify the “type”. When we do not specify the “type”, it is known as the “duck-typed”.

If we don’t want to be specific about the “type”, we can use keywords like “var”, “Object”, or “dynamic”.

To understand the mechanism behind storing reference is simple.

Whenever we declare and initialise a variable, a spot on the memory is booked.

And in that spot the object is stored. And the variable always refers to that spot.

Since the name is “variable”, this reference can change.

However, a question remains. Suppose we do not initialise the variable.

```
var name;
```

In that case, does this variable store any reference? Does it contain any value?

In the next section we will discuss that.

What is null safety in Dart

Dart and Flutter, both, practice the “null-safety”. A variable must have a value.

The word “NULL” in computer means “no value”. It also means “undefined”, “uninitialized” or “empty” value. The word “NULL” neither means 0. Nor, it means a blank String. Null safety in Dart means, by default, the “type” in Dart code is not NULL. Each “type” must have a well-defined value.

What does that mean? When we declare a “type” for a variable, the variable must have a value.

```
void main() {
int x;
x = 0;
print(x);
x = 82;
int y;
y = x;
print(y);
}
// output
0
82
```

However, a variable might have a null value. Although we have declared a “type” for that variable. In that case, the Dart ensures that we have assigned a correct value for that variable. Otherwise, it will throw an error.

Suppose we have written the following code.

```
void main() {
int x;

print(x);

}
// output
line 4 • The non-nullable local variable 'x' must be assigned before it can be used.\
 (view docs)
Try giving it an initializer expression, or ensure that it's assigned on every execu\
tion path.
```

The Dart and Flutter, both, practice the “null-safety”. That is why it throws an error.

But, we can avoid such error. If we add a “?” sign after the declared “type”.

Consider a code snippet like the following.

```
void main(List<String> args) {
int? x;
// output is null
print(x);
}
// output
null
```

We have added a “?” sign to its type declaration. As a result, the variable might have a “null” value.

Otherwise, it would have thrown an error.

From Dart 2.12 and Flutter 2 a variable cannot have a “NULL” value. However, if we add the “?” to its type declaration, there will be no error.

If we do not add the “?” sign to its type declaration, the following variable cannot be null.

```
var i = 42;
```

For the same reason, the following code will throw an error.

```
int? a = null;
int b = a;
// a value of type int? cannot be assigned to a variable of 'type' int
print(b);
```

Before Dart 2.13, suppose we created the Dart Application the following way.

```
dart create -t console-full my_application
```

In such case, we need to migrate to the “NULL-SAFETY”.

```
cd my_application
dart migrate --apply-changes
```

For example, run the following code in your console and watch the output.

```dart
import 'package:dart_for_beginner/null_test.dart';

void main(List<String> args) {
int? x;
x = null;
// output is null
print(x);
List<String?> theListThatCanHoldNullValue = ['first', 'second', null];
print(theListThatCanHoldNullValue.length); // 3
print(theListThatCanHoldNullValue[2]);
String? _aTextThatCouldBeNullButNot() => 'Not Null';
String? notNull = _aTextThatCouldBeNullButNot();
print(notNull);
final aStudent = Student();
final aSchool = School();
aStudent.school = aSchool;
aSchool.student = aStudent;
print('A student gets a school ' + aStudent.school.name);
print('A school gets a student ' + aSchool.student.name);
}
```

The output is as follows.

```
null
3
null
Not Null
A student gets a school L'cole
A school gets a student A Student
```

For full code snippet please visit the GitHub repository.

What is Null Safety in Flutter

We always explicitly tell the Dart, or tell the Flutter that a variable is “NULL”.

In the previous section we have discussed the “Null-Safety” in the Dart programming language. We have learned the basic concept of the “Null-Safety”. The same rule on the “Sound-Null-Safety” works in Flutter. However, in a different way.

Firstly, let us check what is meant by the term “Sound-Null-Safety”?

From the Dart 2.12 and the Flutter 2, this principle of the Sound-Null-Safety becomes mandatory.

What does this principle say?

Every “type” in the Dart, and in the Flutter, is the “non-nullable”. In other words, we explicitly tell the Dart, or tell the Flutter that a variable is “NULL”. Otherwise we must use a “?” sign after the declared “type”.

```
int? x;
```

What does the above code convey?

It says, if we do not supply any integer value the value of the variable x would be the “null”.

We have already learned in the previous section that a “NULL” value means a “undefined”, “uninitialized” or “empty” value. In other word, it is a “no value”.

But can we build a Mobile Application with the “No-Value”? Or, with a “Null”?

We know that it is not possible.

Therefore the Dart and the Flutter follows the “Sound-Null-Safety” design principle.

It optimizes the compiler. The application runs faster. Moreover, if we place the “Null-Check” in right place, the application does not crash suddenly. In addition, it does not throw any error.

There are more.

Let us a consider a simple Flutter Application. In this application we consistently provide a “NULL” value.

However, we cannot supply a Null value where the “type” String is necessary.

In that case, we need to check it first. If it is NULL, then an alternative String value is provided.

Let us see the code now.

```
import 'package:flutter/material.dart';

void main(List<String> args) {
runApp(const MyApp(
    title: null,
));
}

class MyApp extends StatelessWidget {
const MyApp({Key? key, required this.title}) : super(key: key);

final String? title;

@override
```

```
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'It can\'t be NULL',
    home: MyHomePage(title: null),
    );
}
}

class MyHomePage extends StatelessWidget {
const MyHomePage({
    Key? key,
    required this.title,
}) : super(key: key);

final String? title;

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text(
        title != null ? 'It\'s not NULL value' : 'It\'s a NULL value',
        ),
    ),
    body: Center(
        child: Container(
        padding: const EdgeInsets.all(20),
        child: Text(
            title != null ? 'It\'s not NULL value' : 'It\'s a NULL value',
            style: const TextStyle(
            fontSize: 30,
            fontWeight: FontWeight.bold,
            ),
        ),
        ),
    ),
    );
}
}
```

Throughout the Flutter Application we have assigned a NULL value for the “title” .

```
runApp(const MyApp(
    title: null,
));
```

But, in the end, we have to check.

```
title != null ? 'It\'s not NULL value' : 'It\'s a NULL value',
```

As we pass the “NULL” value to the “title” parameter. We need to pass it through the Constructor.

```
const MyApp({Key? key, required this.title}) : super(key: key);

final String? title;
...
```

We have added the “?” sign after the “type”. As a result, the “value” of the variable could be “NULL”.

For full code please visit the GitHub repository.

What is “type” in Dart and Flutter

The “type” plays an Important role in Dart and Flutter. We must work using the correct “type”.

As a beginner we need to understand what is “type” in the Dart and in the Flutter. In this section, we will also learn what are the “built-in-type” that the Dart provides.

We have already learned what a “variable” is. Now, any variable can hold different “type” of Data. The data could be an integer. Or, it could be a String. Or, a Boolean.

To begin with we can declare the “type” before a “variable”. To do that we need to specify the “type”.

Is it an integer?

Then we will write like this:

```
int? x;
```

Why we have used a “?” sign? The reason has been explained before.

It means, this variable could have a “NULL” value.

At the same time, we could have written like this:

```
var x = 1;
```

In that case, we have not specified the “type”. But the Dart will infer that the “type” of the variable would be an integer.

This is called “type-annotation”. Although this is optional, yet it is a good practice that we specify the “type” before a variable.

When we do not specify the type, the Dart and the Flutter applies runtime checks.

Since it is optional, the Dart and the Flutter uses type inference.

If you have interest t learn more, see the language tour.

Declaring type is good, because if we do not supply proper type, the Dart static analyzer warns us. The bugs are caught in the compile time. As a result, we can correct the bugs earlier.

Consider the code below:

```
void printAListOfInteger(List<int> a) => print(a);

void main() {
List<int> list = [];
list.add(1);
list.add(2);
printAListOfInteger(list);
}
```

In the above code, we have declared a list of integers. Therefore, before a variable like “list”, we need to specify the particular “type”.

The output is:

```
 Running with sound null safety 
Connecting to VM Service at ws://127.0.0.1:33047/ToG8jrELdFw=/ws
[1, 2]
```

We could have written the code in the following way also.

```
// Both the code below will work fine
// first option
void main() {
var list = <int>[];
list.add(1);
list.add(2);
printAListOfInteger(list);
}
// second option
void main() {
dynamic list = <int>[];
list.add(1);
list.add(2);
printAListOfInteger(list);
}
```

In the above code we have not specified the “type” of the variable. However, when we assign a value, we mention the “type”.

Therefore, we can do the both. Either, we can specify the “type” on the left hand side. Or, we can use the “type” on the right hand side.

But, we need to maintain the right “type” always.

Built-in Types in Dart

Type does not mean only “non-nullable” value.

That is why, we do not write “Object”. We write “Object?”.

The “NULL” or “no-value” is also another “type” in Dart.

The Dart language has special support for the following types.

```
Numbers (int, double)
Strings (String)
Boolean (bool)
Lists (List, also known as arrays)
Sets (Set)
Maps (Map)
Runes (Runes; often replaced by the characters API)
Symbols (Symbol)
The value null (Null)
```

As we progress, we will check each type.

When we assign a correct “type” to a declared variable, it is called a “literal”.

```
String aText = 'A String';
```

In the above code, ‘A String’ is a String literal.

Some other types also have special roles in the Dart language:

```
Object: The superclass of all Dart classes except Null.
    Future and Stream: Used in asynchrony support.
    Iterable: Used in for-in loops and in synchronous generator functions.
    Never: Indicates that an expression can never successfully finish evaluating. Mo\
st often used for functions that always throw an exception.
    dynamic: Indicates that you want to disable static checking. Usually you should \
use Object or Object? instead.
    void: Indicates that a value is never used. Often used as a return type.
```

We initialise an object of any of these special types using a literal.

For example, “Hello John Smith” is a string literal. And, “false” is a boolean literal.

Consider this code:

```
main() {
    String? saySomething;
    bool? isNull;
    if(saySomething == null){
        print("It is $isNull");
    }
    else {
        print("It is not $isNull");
    }
    }

// output

It is null
```

Since the String variable is the “NULL”, the output is:

```
It is null
```

How to implement types in Flutter

In Flutter we use Dart built in types all the time. A few types are used more often than other.

In our previous section we have discussed the Dart built in types. Here we will see how we can implement types in Flutter.

This demonstration is for beginners. Moreover, I presume you do not have any coding background. In that case, just be patient. You need not create any Flutter Application.

Just watch how Dart built in types are used in Flutter.

By the way, we have seen that there are several built in types. Although we do not use all at once. In addition, we need a few types more often than other.

Between all built in types, we mostly use the common “types”. The Numbers, the String, the Boolean and the List and Map. Besides, we will also use some special “types”. The Object and Class, the “void”, The “Future and Stream”.

However, if you are a beginner, you do not worry. We will learn them later.

In fact, as we progress, we will see the usages of the common types, more and more.

Let us consider a very simple Flutter Application. In this Flutter Application, we have used built in types like the Numbers that include the integer and double.

We have used the double “type” to add the margin and padding. In one case we have used the Boolean. Subsequently, to build a Row, we have used the List.

Meanwhile, the “type” String is used in many cases.

Let us first see the code. Please read the comments where we have mentioned each “type”.

```
import 'package:flutter/material.dart';

/// how we can use Dart built in types in Flutter
/// here is a simple demonstration for beginners

void main() {
runApp(const MyApp());
}

class MyApp extends StatelessWidget {
const MyApp({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    /// using String type
    title: 'Dart Built in Types',

    /// using Boolean type
    debugShowCheckedModeBanner: false,
    home: MyAppHome(),
    );
}
}

class MyAppHome extends StatelessWidget {
const MyAppHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        /// using String type
        title: const Text('Dart Built in Types'),
    ),
    body: Center(
        /// using type List
        child: Row(
        children: [
```

```
            Container(
            /// using type double
            ///
            margin: const EdgeInsets.all(20.0),
            padding: const EdgeInsets.all(10.0),
            color: Colors.lime,

            /// using String type
            child: const Text('First Row'),
            ),
            Container(
            /// using type double
            padding: const EdgeInsets.all(10.0),
            color: Colors.amber,

            /// using String type
            child: const Text('Second Row'),
            ),
        ],
        ),
    ),
    );
}
}
```

If we run this Flutter Application, what do we see?

An AppBar Widget at the top of the Home Screen. The AppBar displays a text, whose value is basically the “type” String.

In the body section, we see two colored boxes that are placed in a Row. Similarly, the Row Widget uses a the “type” List.

To sum up, a Flutter Application always uses the Dart built in types. In many ways.

However, a beginner with no coding background will not understand these elementary concepts so easily.

Therefore, we need to explore more. Firstly, we need to learn Dart Programming Language. In the coming sections, we will learn them.

Secondly, after learning the basic concepts of the Dart, we will again come back to the Flutter.

Finally, we will implement the Dart programming concepts in Flutter.

Difference between final and constant

In the similar vein, the keywords “final” and “const” both obstruct to modify value.

In the Dart the keyword “const” represents a constant that modifies value. On the other hand, the keyword “final” means single-assignment. It appears that they are very like each other in some way.

Why? Because they both modify values.

The same rule applies to the Flutter also.

However, there are some differences between these two keywords.

To understand the difference let us consider this code.

```
final String s = 's';
// s = 'x'; // The final variable 's' can only be set once.
const int? num = null;
// num = 1; // Constant variables can't be assigned a value.
```

Firstly, we have set the value of the variable “s”. The “type” is String. As a result, when we try to set the value again, it throws an error.

Secondly, we have assigned a value to a constant variable “num”. As a consequence, when we try to reassign a new value, it throws an error.

In the similar vein, they both obstruct to modify value.

In this context, we have not discussed the Class, Object and the “Object-Oriented-Programming” yet.

Although, the difference between the “final” and the “const” is better understood in that perspective.

Therefore, let us imagine a Class Foo which has one “final” variable. And one “const” variable.

Inside a Class we always use the “static” keyword before the “const”. Subsequently, we can access the “const” variable directly. We do not have to create an instance.

On the contrary, for the “final” variable, we have initialised the value through the Class Constructor.

A “final” variable instance variable must be initialised through the Class Constructor.

```
class Foo {
final String f;
static const int i = 1;
Foo(this.f);
}
main() {
Foo f1 = Foo('foo1');
// f1.f = ''; // f cannot be used as a setter because it's final
Foo f2 = Foo('foo2');
print(f1.f);
print(f2.f);
// Foo.i = 2; // Constant variables can't be assigned a value.
}
```

In the above code, when we try to change the “final” variable by the instance, it throws an error.

Why?

Because we have already once set the value.

In the case of the “const” variable, we have also set the value once inside the Class. Therefore, we cannot assign a new value again. It also throws an error.

Here is the full code with the output. For a proper understanding, we have commented out the errors.

```
class Foo {
final String f;
static const int i = 1;
Foo(this.f);
}
main() {
final String s = 's';
// s = 'x'; // The final variable 's' can only be set once.
const int? num = null;
// num = 1; // Constant variables can't be assigned a value.
print(s);
print(num);

Foo f1 = Foo('foo1');
// f1.f = ''; // f cannot be used as a setter because it's final
Foo f2 = Foo('foo2');
print(f1.f);
print(f2.f);
// Foo.i = 2; // Constant variables can't be assigned a value.

```

```
/*
    * The output:
s
null
foo1
foo2
    * */
}
```

Further, when we will create a Flutter Application we will use both – the “final”, and the “const”.

However, before we start building a Flutter Application we need to understand the core concepts of the “Object-Oriented-Programming”.

So stray tuned. And, keep reading.

Introduction to core libraries in Dart

In Dart, we need to take help from the core libraries. Why? Because when we build an Application in the Flutter, we need them.

For an example, we need to check whether a number is the “Even” or “Odd”?

Dart has a rich set of core libraries that helps us in such cases. The Dart core libraries provide essentials for such programming tasks.

Consider a small Dart program.

```
void main() {
int? aNumber = 5;
if(aNumber.isEven) {
    print('The number $aNumber is Even');
} else {
    print('The number $aNumber is Odd');
}
}
// output
The number 5 is Odd
```

The “isEven” is a property that returns true if and only if this integer is even. In our case, the number is not even. As a result we get the output: “The number 5 is Odd”.

In this section, we will have a gentle introduction to the “Core Libraries” in the Dart. In the Advanced level, we will delve deep into it.

We can manage such small everyday programming tasks using the Dart core libraries. And they are automatically imported.

We have also learned about the “built-in-types“. The int and double provide support for Dart’s built-in numerical data types. An object of the type bool is either true or false.

Both ‘int’ and ‘double’ types are subtypes of ‘num’.

The ‘num’ type includes basic operators such as “ +, -, /, and * ”. They represent ‘plus, minus. division and multiplication’ signs.

We can also call them arithmetic operators. It includes another operator the “modulo”. It represents the “remainder” and the sign is: ‘%’.

In the next section we will discuss the String. After that we will see how List and Map work in the Dart. They provide data structures for managing collections of objects.

Let us see some interesting examples:

```
void main() {
int? aNumber = 5;
if(aNumber.isOdd) {
    print('The number $aNumber is Odd');
} else {
    print('The number $aNumber is Even');
}
}

    //output
    The number 5 is Odd
```

We can also turn a string to a double number.

Let us change the above code a little bit. Just like the integer, to “String” also has some properties.

```
    main() {
    var one = int.parse('1');
    var doubleToString = double.parse('23.564');
    print(one);
    print(doubleToString);
    if(one.isOdd && doubleToString.isFinite){
        print('The first number is an odd number and the second one is a double ${do\
ubleToString} and a finite number.');
    } else {
        print('It is an even number and the second one is not a double ${doubleToStr\
ing} and a non-finite number.');
    }
```

```
}
// The output is quite expected. Both statements are true so the // // relational op\
eration takes to this output:
// output
    1
23.564
```

The first number is an odd number and the second one is a double 23.564 and a finite number.

We can do the vice versa too.

Therefore we can turn an integer to string.

```
    main() {
    int myNUmber = 542;
    double myDouble = 3.42;
    String numberToString = myNUmber.toString();
    String doubleToString = myDouble.toString();
    if ((numberToString == '542' && myNUmber.isEven) && (doubleToString == '3.42' &&\
 myDouble.isFinite)){
        print('Both have been converted from an even number ${myNUmber} and a finite\
 double ${myDouble} to string. ');
    } else {
        print('Number and double have not been converted to string.');
    }
}

    //output
    Both have been converted from an even number 542 and a finite double 3.42 to str\
ing.
```

As we progress, we will find, Dart is extremely flexible language. As a consequence, the syntax is simple to remember.

Moreover, we get lot of help from the core libraries.

What is a Boolean in Dart and Flutter

The Boolean literals ‘true’ and ‘false’ have type ‘bool’. They are compiled time constants.

The Boolean is a logical data type that can have only the values “TRUE” or “FALSE”.

For example, in the Dart or in the Flutter, Boolean conditionals are often used for many purposes.

There are several built-in Boolean conditional in the Dart, or the Flutter.

We have already seen such examples in the previous section.

```
void main() {
int? aNumber = 5;
if(aNumber.isOdd) {
    print('The number $aNumber is Odd');
} else {
    print('The number $aNumber is Even');
}
}

    //output
    The number 5 is Odd
```

The Boolean conditional decides which section of code to execute.

According to the above example, if the number was “ODD” a certain output will come out. If not, the other section of code would have executed.

In case of for loops, the Boolean conditional is used also. We will see that later.

The Boolean value is named after English mathematician George Boole, who pioneered the field of mathematical logic.

We have already seen that Dart has a type called ‘bool’.

The Boolean literals ‘true’ and ‘false’ have type ‘bool’. They are compiled time constants.

Consider this code:

```
main() {
bool isTrue = true;
bool isFalse = false;
if(isFalse || isTrue){
    print("It is true.");
}
}
// output
It is true.
```

When the choice is between Boolean value “TRUE” or “FALSE”. The “TRUE” wins.

On the other hand, consider the code below.

```
main() {
bool isTrue = true;
bool isFalse = false;
if(isFalse && isTrue)
{
    print("It is True."); // dead code
} else {
    print("It is False.");
}
}
// output
It is False.
```

When the choice is between Boolean value “TRUE” and “FALSE”. The “FALSE” wins.

We can prove in a more realistic way.

```
main() {
bool isTrue = true;
bool isFalse = false;
if(isFalse && isTrue)
{
    print('${isTrue.toString()}');
} else {
    print('${isFalse.toString()}');
}
}
// output
false
```

Therefore, we have two boolean literals. The “true” and “false”.

By using ‘if control logic’ we can organise our code.

Is a Boolean 1 or 0?

The constant value “true” is 1. And the constant value “false” is 0.

However, when the condition is between “1 AND 0”, it is always 0. However, when the condition is between “1 OR 0”, it is always 1.

Therefore, arithmetically we can draw a conclusion.

The “1 AND 0” is like “0 * 1” or “0 / 1”. On the other hand, the “1 OR 0” stands for either “1 + 0 “, or “1 – 0”.

Hence, between the “true” and the “false”, the “OR” relational operator always chooses the “true”. On the other hand, Between the “true” and the “false”, the “AND” relational operator always chooses the “false”.

This is an extremely important concept in computer science.

Why?

Because, in our control structure, we always depend on such conditional.

What is Boolean logic examples?

We organise our code according to this conditional. Whether a statement is “TRUE” or “FALSE” really matters.

As we progress, we will find more examples.

Just to remember, we follow this principle. Between the Good and the Bad if we use the OR relational operator, it is considered to be a dead code.

Why?

Because, the rule is simple. The Good OR the Bad? The answer is the Good.

On the contrary, the Good AND the Bad? The answer is the Bad.

for an instance, let us write the code below to test the Boolean logic.

```
void main() {
bool? toBe = true;
bool? notToBe = false;

/// isExisting becomes false
bool? isExisting = toBe && notToBe;

if (isExisting) {
    /// since this part only works if this condition is true
    /// this part of code won't be executed
    print('I exist.');
} else {
    /// this will be executed
    print('I don\'t exist.');
}

/// doIExist becomes true
bool? doIExist = toBe || notToBe;
if (doIExist) {
    /// since this part only works if this condition is true
    /// this part of code will be executed
```

```
    print('I exist.');
} else {
    print('I don\'t exist.');
}
}
```

When we will discuss the “if-else” logic structure, we will see more examples. We will also learn about the “ternary” operator.

For full code, please visit the respective GitHub repository.

What are String values in Dart and Flutter

The Unicode support makes Dart powerful. Mobile and web applications in any language is possible.

We use the “String” to represent text in Dart and Flutter. And it plays an important role any Flutter Mobile or Web Application.

A string can be either single or multi line.

We write the single line String using either the single or double quotes.

However, the multi line String is written using triple quotes.

Watch the following code. They are all valid Dart “String”.

The String is immutable.

Although we cannot change a String, we can perform any Boolean operation on a String in Flutter.

As a result, it creates a new String. Or, it tests whether the String is NULL or not.

```
void main() {
String? dart = 'Dart for Flutter';
print(dart);
String? newdart = dart.substring(0, 5);
print(newdart);

}

/**
* /// output

Dart for Flutter
Dart
///
*/
```

How do you use a string in flutter?

A Dart string is a sequence of UTF-16 code units.

For absolute beginners, let us give a short note on UTF-8, UTF-16, and UTF-32.

They all store Unicode but uses different bytes.

Let us first try to understand the advantages of using UTF-16 code over the other two.

Let us know about UTF-8.

Where ASCII characters represent the majority of texts, UTF-8 has an advantage.

Like ASCII, UTF-8 also encodes all characters into 8 bits.

It is the opposite for UTF-16. In UTF-16 ASCII is not predominant.

As a consequence, the UTF-16 has an advantage. UTF-16 remains at just 2 bytes for most characters.

However, UTF-32 tries to cover all possible characters in 4 bytes.

That means, processors have extra load making it pretty bloated.

The Unicode support makes Dart very powerful.

So that we can make mobile and web applications in any language.

Let us see one example where we have tried some Bengali script.

```
main(List<String> arguments) {
    //print("Hello World ${IdeaProjects.calculate()}");
    String bengaliString = "□□□□□ □□□□";
    String englisgString = "This is some English text.";
    print("Here is some Bengali script - ${bengaliString} and some English script ${\
englisgString}");
    }

// And we have the output here:

//output

    Here is some Bengali script - □□□□□ □□□□ and some English script This is some En\
glish text.
```

How do you put string in a Dart?

While handling strings, we should remember a few things.

We can use both single quote(‘’) and double quote(“”).

```
    main(List<String> arguments) {
    String stringWithSingleQuote = 'I\'m a single quote';
    String stringWithDoubleQuote = "I'm a double quote.";
    print("Using delimiter in single quote - ${stringWithSingleQuote} and using deli\
miter in double quote - ${stringWithDoubleQuote}");
    }
//output

Using delimiter in the single quote - I'm a single quote and using delimiter in the \
double quote - I'm a double quote
```

We can use the delimiter in both cases.

But the double quote is more helpful in such cases.

We have put the value of expression inside a string.

How?

We have used our variable in this way: “${stringWithSingleQuote}”.

We can use the variable the following way also.

```
print(”$stringWithSingleQuote”);
or
print($stringWithSingleQuote);
```

How do you variable a string in flutter?

String interpolation, concatenation and multi-line is quite easy in Dart.

Consider this code where we have put a String variable inside another String.

```
String? anExampleOfConcatenation = 'Dart ' + 'for ' + 'Flutter';
print(anExampleOfConcatenation);
String? anotherExampleOfConcatenation = 'Dart ' 'for ' 'Flutter';
print(anotherExampleOfConcatenation);
String? anExampleOfInterploation = anotherExampleOfConcatenation;
print('$anExampleOfInterploation is of '
    '${anExampleOfInterploation.length} characters');

// output
Dart for Flutter
Dart for Flutter
Dart for Flutter is of 16 characters
```

If you want to store some constant value inside a constant string, the value cannot be variables. Consider this code:

```
    main(List<String> arguments) {
    const aConstantInteger = 12;
    const aConstantBoolean = true;
    const aConstantString = "I am a constant string.";
    const aValidConstantString = "this is a constant integer: ${aConstantInteger}, a\
 constant boolean: ${aConstantBoolean}, a constant string: ${aConstantString}";
    print("This is a valid constant string and the output is: $aValidConstantString"\
);
    }

    //output
    This is a valid constant string and the output is: this is a constant integer: 1\
2, a constant boolean: true, a constant string: I am a constant string.
```

We have created a valid constant string by storing constant value inside them. The output is perfectly OK.

It will not work if you want to hold variable data inside a constant string.

We have changed the code 2.10 to this. As a result, it warns us with a compile time error.

```
    main(List<String> arguments) {
    var aConstantInteger = 12;
    var aConstantBoolean = true;
    var aConstantString = "I am a constant string.";
    const aValidConstantString = "this is a constant integer: ${aConstantInteger}, a\
 constant boolean: ${aConstantBoolean}, a constant string: ${aConstantString}";
    print("This is a valid constant string and the output is: $aValidConstantString"\
);
    }

And here is the output full of errors.

    //output

//Const variables must be initialized with a constant value. (view docs)
//Try changing the initializer to be a constant expression.

    bin/main.dart:9:63: Error: Not a constant expression.
    const aValidConstantString = "this is a constant integer: ${aConstantInteger}, a\
 constant boolean: ${aConstantBoolean}, a constant string: ${aConstantString}";
....
```

It did not work.

As we progress, we will learn more about string.

Understanding string is very important in making the mobile or the web applications in Flutter.

Lists in Dart

The List is the most “Common Collection” not only in the Dart, but in every Programming Language.

What do we mean by the term “Common Collection”?

The term “Common Collection” refers to another term the “Array”. The Array means an “ordered-group-of-objects”.

By the way, we use both terms equally. The List or the Lists. Both they stand for a List Object.

JavaScript array literals look like Dart lists.

Here is a sample code we may consider to understand why this concept is important:

```
main(List<String> arguments) {
List fruitCollection = ['Mango', 'Apple', 'Jack fruit'];
print(fruitCollection[0]);
}
//output
Mango
```

Let us consider another piece of code where we have created two List Objects.

One is of the type String. And the other is the type Integer.

```
main(List<String> arguments) {
List fruitCollection = ['Mango', 'Apple', 'Jack fruit'];
var myIntegers = [1, 2, 3];
print(myIntegers[2]);
print(fruitCollection[0]);
}
// output
3
Mango
```

What is the difference between these two code snippets?

In the above code, firstly, we have explicitly mentioned the “Type” of the variable. And, that is a List.

```
List fruitCollection = ['Mango', 'Apple', 'Jack fruit'];
```

But, secondly, we have not mentioned the “Type” of the variable.

```
var myIntegers = [1, 2, 3];
```

However, Dart infers that the variable has the “Type” List.

How do you use lists in darts?

We can always make the List variable as the “Dynamic”.

```
void main() {
var myIntegers = [1, 2, 3, 'non-integer object'];
print(myIntegers[3]);
}
// output
non-integer object
```

It did not raise any error. But, we could have explicitly declare that the List is of “Dynamic” type.

```
void main() {
List<dynamic> myIntegers = [1, 2, 3, 'non-integer object'];
print(myIntegers[3]);
}
```

But, we cannot write this way. It will raise an error.

```
void main() {
List<int> myIntegers = [1, 2, 3, 'non-integer object'];
print(myIntegers[3]);
}
//output
The element type 'String' can't be assigned to the list type 'int'.
```

The same thing will happen if we try to add a String Type to the List of Integers.

```
void main() {
var myIntegers = [1, 2, 3];
myIntegers.add(' a String');
print(myIntegers[3]);
}
// output
The argument type 'String' can't be assigned to the parameter type 'int'.
```

We can create the “Constant List”. But we cannot change the “Constant State” of the Constant List Object.

```
void main() {
var constantList = const [1, 2, 3];
constantList[1] = 1; // This line will cause an error.
}
// output
Uncaught Error: Unsupported operation: indexed set
```

From the Dart 2.3 we can use the “Spread-Operator”. Other way, we can call it the “Three-Dots”.

How do I add items to my Dart list?

Now, we can easily add an item to any List.

```
void main() {
var list = [1, 2, 3];
var list2 = [0, ...list];
print(list2);
}
// output
[0, 1, 2, 3]
```

We can add multiple items to the List now.

Consider a List which is NULL.

What happens if we try to add any item to it?

```
void main() {
var list;
var list2 = [0, ...list];
print(list2);
}
// output
Uncaught Error: TypeError: null: type 'JSNull' is not a subtype of type 'Iterable<dy\
namic>'
```

But one “?” solves the problem.

We have learned the “NULL-SAFETY” before.

Now, we can write the above code the following way.

```
void main() {
var list;
var list2 = [0, ...?list];
print(list2);
}
// output
[0]
```

This is known as the “NULL-AWARE-SPREAD-OPERATOR”.

Now we can add more different “Type” of items to the List which is NULL.

```
void main() {
var list;
var list2 = [0, 'A String', 1, 'Another String', ...?list];
print(list2[0]);
print(list2[1]);
print(list2[2]);
print(list2[3]);
print(list2.length);
}
// output
0
A String
1
Another String
4
```

We can clearly see that it is a List of “dynamic ”.

The index of a List starts with 0.

However, when we measure the length, it counts from 1.

Why Map is important in Dart

In the Dart, the Map literals came first, the literal "{ }" is a default to the Map type.

The Map in the Dart Programming language represents a collection of key-value pairs. We can retrieve a value from its associated key.

In any Programming Language, the Data Structure is a very important concept. Therefore we need to understand the basics of this key concept.

However, each key has exactly one value. In this section we will have an introductory note about the Dart Map.

Later, we will find more implementation in the Flutter Map section.

Why does the Dart Map play an important role in Flutter?

It is because the Map can be iterated. Now, there are different type of Maps. As a result, the order of iteration may change.

However, before the Dart Map, let us know about the Set first.

The Set is a kind of List. Although mainly unordered List. The order of the Set depends on its implementation.

The List is well indexed. Therefore, we can access by position. But in the Set, we cannot do that. Moreover, in the List, we can add duplicate items. The Set, however, does not allow that.

Syntax wise they are also different.

```
1 var lists = ['something'];
2 var sets = {'something'};
```

In Dart, a Set is an unordered collection of unique items.

Let us see a simple example of the Set.

```
main() {
var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(fruitCollection.lookup('Apple'));
}

//output
Apple
```

We can search the Set using the lookup() method. If the value is not there, it returns 'null'.

```
main() {
var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(fruitCollection.lookup('Something Else'));
}

//output
null
```

Remember Syntax wise Set and Map look same.

How do I create a Dart map?

Watch the following code.

```
var myInteger = {};
```

It does not create an empty Set, but an empty Map.

The syntax for the Map literals is similar to that of for the Set literals.

Why does it happen? Because in the Dart, the Map literals comes first, the literal "{ }" is a default to the Map type.

We can prove this by a simple test.

```
main() {
var myInteger = {};
if(myInteger.isEmpty){
    print("It is a map that has no key, value pair.");
} else print("It is a set that has no key, value pair.");
}
Watch the output:

//output

It is a map that has no key, value pair.
```

It means the map is empty.

If it was a set, we would have got the other output.

At present we need to remember, the Map is an object that associates keys and values.

How do Dart maps work?

Both keys and values can be of any “Type” of object.

```
Map<dynamic, dynamic> myProducts = {};
myProducts['first'] = 'TV';
myProducts[2] = 'Mobile';
myProducts['third'] = 'Refrigerator';
if (myProducts.containsValue('Mobile')) {
    print('Our products have ${myProducts[2]}');
}
```

Each key occurs only once, but you can use the same value multiple times.

Dart support for the maps is provided by the Map literals and the Map type.

Maps support spread operators (... and ...?). We have seen the three “...” in the List section.

```
var myProductsOne = {'final': 3, ...myProducts};
if (myProductsOne.containsValue('Mobile')) {
    print('Our products have ${myProductsOne[2]}');
}
if (myProductsOne.containsValue(3)) {
    print('Our products have ${myProductsOne['final']}');
}
// Our products have Mobile
// Our products have 3
```

We can easily print any value of the Map using the key.

```
main() {
var myProducts = {
    'first' : 'TV',
    'second' : 'Refrigerator',
    'third' : 'Mobile',
    'fourth' : 'Tablet',
    'fifth' : 'Computer'
};
print(myProducts['third']);
}
```

```
// output
// Mobile
```

The Dart understands that the ‘myProducts’ has the “Type” Map.

It is not mandatory that the Key and the Value should be of the same “Type”.

```
main() {
Map<int, String> myProducts = {
    1 : 'TV',
    2 : 'Refrigerator',
    3 : 'Mobile',
    4 : 'Tablet',
    5 : 'Computer'
};
print(myProducts[3]);
}
```

The output is – mobile.

However, once we declare the “Type”, we cannot change that. It will throw error.

```
myProducts['1'] = 'First';
/// The argument type 'String' can't be assigned to the parameter type 'int'.
```

Can we add a Set type collection of value inside a Map?

Yes, we can. Consider this code:

```
main() {
Set mySet = {1, 2, 3};
var myProducts = {
    1 : 'TV',
    2 : 'Refrigerator',
    3 : mySet.lookup(2),
    4 : 'Tablet',
    5 : 'Computer'
};
print(myProducts[3]);
}
// output
// 2
```

In the above code, we have injected a collection of the “Type” Set into the Map.

In Flutter, the List and the Map will play a very important role.

Therefore, later we will learn how we can Map a List.

What is map data type in Dart?

In Dart, Map is the “Dictionary-like-Data-Type”. And it exists in the “key-value” form.

We can declare Map in two ways.

Using Map Literals like the following way.

```
var maps = {};
```

Or we can use the Map Constructors.

Here is a sample of a few code snippets that will give you an idea about the Set and the Map.

```
void main() {
var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(fruitCollection.lookup('Apple'));
var anotherFruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(anotherFruitCollection.lookup('Something Else'));
var myInteger = {};
if (myInteger.isEmpty) {
    print("It is a map that has no key, value pair.");
} else
    print("It is a set that has no key, value pair.");

var actors = {
    'first': 'De Nero',
    'second': 'Pacino',
    'third': 'Willis',
    'fourth': 'Morgan',
    'fifth': 'Hackman',
    1: 'Someone Else',
};
print(actors[1]);
Map<int, String> maps = {};
maps[1] = 'First';

/// maps['1'] = 'First';
/// The argument type 'String' can't be assigned to the parameter type 'int'.
Map<dynamic, dynamic> myProducts = {};
myProducts['first'] = 'TV';
myProducts[2] = 'Mobile';
myProducts['third'] = 'Refrigerator';
if (myProducts.containsValue('Mobile')) {
```

```
    print('Our products have ${myProducts[2]}');
}
var myProductsOne = {'final': 3, ...myProducts};
if (myProductsOne.containsValue('Mobile')) {
    print('Our products have ${myProductsOne[2]}');
}
if (myProductsOne.containsValue(3)) {
    print('Our products have ${myProductsOne['final']}');
}
// Our products have Mobile
// Our products have Mobile
// Our products have Mobile
// Our products have 3
}
```

Arithmetic Operators in Dart

Using Arithmetic Operators is easy in Dart. But we need to know the special operators also.

What do the usual Arithmetic Operators do? We know that. Nothing new. The Addition, the Subtraction, the Multiplication and the Division. The Arithmetic Operators in Dart do the same thing.

However, there are more. Suppose we want the remainder of an integer division. In Dart Programming Language we can do that.

Subsequently, we can divide a double by a double number, but the result could be an integer "Type". We can also do such operations.

Let us try some simple Arithmetic Operations in the Dart.

```
void main() {
int a = 10;
int b = 3;
int y = a + b;
print(y); // 13
int x = a - b;
print(x); // 7
int z = a * b;
print(z); // 30
}
```

Till now, everything goes fine. But if we want to divide the "int a" y "int b", it no longer remains an integer. It becomes another "Type" double.

Therefore we cannot write this code.

```
int u = a / b; // an error
```

We can solve that problem by the following line of code.

```
int u = a ~/ b;
print(u); // 3
```

Therefore we can change a “Type” double to the “Type” integer by this special Arithmetic Operator – “~/”.

The same way, we can get a remainder.

```
int v = a % b;
print(v); // 1
```

The operator “%” gives us the remainder which is an integer.

Dart also supports both prefix and postfix increment and decrement operators.

```
++a;
print(a); // 11
a++;
print(a); // 12
--a;
print(a); // 11
a--;
print(a); // 10
```

In Dart, when you use operators, you actually create expressions.

What does that mean?

```
++a = a + 1; // this is the expression value
```

A postfix increment operator means the same.

Let us see some examples.

```
main() {
int aNum = 12;
double aDouble = 2.25;
var theResult = aNum ~/ aDouble;
print(theResult);
}

//output
5
```

In the above code, the special operator has displayed an integer. Not a double.

However, if we had divided it in a plain way, it would give us different value.

```
main(List<String> arguments) {
int aNum = 12;
double aDouble = 2.25;
var theResult = aNum / aDouble;
print(theResult);
}
```

Here is the output:

```
//output
5.333333333333333
```

While using any Arithmetic Operator, we need to be careful to assign the value.

If we explicitly declare the “Type” or specify the “Type”, we need to be careful about using the correct Arithmetic Operator.

Any good IDE, like Visual Studio Code, gives us warning beforehand.

In the coming section we will learn the Equality and the Relational operators.

Equality and relational operators

The operator does not only check the “Equality” it also expresses the “Relation” between two Objects.

We need to understand the Equality and Relational operators from a beginner’s point of view. Therefore we consider a simple example firstly.

```
int? a = 10;
int? b = 10;
```

Secondly, we have explicitly declared the “Type” of the both Objects. Always remember, in Dart, everything is Object.

However, if you have no coding background, you should not know what Objects are.

Yet, in the above code, both variables represent integer “Type”. And, moreover, they are assigned the same constant value 10.

As a result, we can say, they are equal.

```
void main() {
int? a = 10;
int? b = 10;
if(a == b) {
    print('$a and $b are equal.');
} //**output** - 10 and 10 are equal.
}
```

Why this is called “Equality and Relational” operator?

The operator does not only check the “Equality” it also expresses the “Relation” between two Objects.

The other “Equality and Relational Operators” are as follows.

```
Operator        Meaning
==         Equal; see discussion below
!=         Not equal
>         Greater than
<         Less than
>=         Greater than or equal to
<=         Less than or equal to
```

Now, we can check some examples. It will give us clear ideas.

```
main() {
int firstNum = 40;
int secondNum = 41;
if (firstNum != secondNum){
    print("$firstNum is not equal to the $secondNum");
} else print("$firstNum is equal to the $secondNum");
}

//output
40 is not equal to the 41
```

The output is quite expected. The integer value 40 is not equal to 41.

```
main() {
int firstNum = 40;
int secondNum = 40;
int thirdNum = 74;
int fourthNum = 56;
if (firstNum == secondNum || thirdNum == fourthNum){
    print("If choice between 'true' or 'false', the 'true' gets the precedence.");
} else print("If choice between 'true' or 'false', the 'false' gets the precedence."\
);
}

//output
If choice between 'true' or 'false', the 'true' gets the precedence.
```

We have used the “Logical Operator” “||” which means the “OR”. In the coming section we will learn what the “Logical Operator” is.

In the following code, we have used the “Logical Operator” “&&” which means the “AND”.

```
main() {
int firstNum = 40;
int secondNum = 40;
int thirdNum = 74;
int fourthNum = 56;
if (firstNum == secondNum && thirdNum == fourthNum){
    print("If choice between 'true' and 'false', in this case the 'true' gets the pr\
ecedence.");
} else print("If choice between 'true' and 'false', in this case the 'false' gets th\
e precedence.");
}
```

```
12
13 //output
14 If choice between 'true' or 'false', in this case the 'false' gets the precedence.
```

Let us consider more example.

```
1  main(List<String> arguments) {
2  int aNUmber = 35;
3  if(!(aNUmber != 150) && aNUmber <= 150){
4      print("It's true");
5  } else print("It's false.");
6  }
```

The “>=” operator means greater than, or equal to.

It is the “>” greater than. And, it is the “<” less than.

Finally, in the rare case, we use the identical() function to test whether two objects are equal or not. We will discuss that later.

Let us see some more code samples.

```
1  class Foo {
2  Object? m = 'M';
3  Object? n = 1;
4  String? s;
5  int? i;
6  }
7
8  void main() {
9  int? a = 10;
10 int? b = 10;
11 if(a == b) {
12     print('$a and $b are equal.');
13 } // 10 and 10 are equal.
14 int firstNum = 40;
15 int secondNum = 40;
16 int thirdNum = 74;
17 int fourthNum = 56;
18 if (firstNum == secondNum || thirdNum == fourthNum){
19     print("If choice between 'true' or 'false', the 'true' gets the precedence.");
20 } else {
21     print("If choice between 'true' or 'false', the 'false' gets the precedence.");
22 }
```

```
print(identical(a, b)); // true
int? x = 10;
double? y = 10.5;
print(identical(x, y)); // false
Foo foo = Foo();
print(identical(foo.m.hashCode, foo.n.hashCode));
print(identical(foo.s, foo.i));
}
// output
10 and 10 are equal.
If choice between 'true' or 'false', the 'true' gets the precedence.
true
false
false
true
```

For details, see Operators.

What are Logical Operators

Logical operators either can combine two Boolean expressions or invert any Boolean expression

Logical Operators in the Dart Programming Language help us to do two things. One is to combine two Boolean expressions. And the other is to invert any Boolean expression.

We have seen Boolean expression before.

```
int? x;
int? y;
if(x == y){
    print('Both $x and $y are NULL');
}
```

However, the above Boolean expression will not be true if the variable “y” has a value.

```
int? x;
int? y = 10;
bool z = x == y;
if(!z) {
    print('$x and $y are unequal');
}
```

Since “x” and “y” are unequal, we can invert the expression, by this Boolean expression.

```
if(!z)
```

Therefore, "!" is a Boolean expression. It is known as "NOT".

The other two Boolean expressions are "| |" and "&&". The first one, "| |" means Logical "OR". And the second one "&&" means Logical "AND".

Consider the following code where we combine two Boolean expressions with the help of Logical Operator "&&".

```
int? x;
int? y = 10;
if(x == y && x!= y){
    print('Both $x and $y are NULL');
} else {
print('$x is NULL but $y is an integer type.');
}
```

Watch the above code and try to predict the result.

Firstly, the value of "x" is no longer equal to the value of "y". Therefore, the Logical AND, "&&" compares between one TRUE and FALSE. In such cases, the "if condition" is not TRUE. As a result, the "else condition" will work.

And the output is.

```
null is NULL but 10 is an integer type.
```

To clarify, there are three logical operators. The "and", "or" , and "not" . The meaning of these operators is similar to their meaning in English.

For example, here is the Operator and the Meaning.

```
Operator        Meaning
!expr               inverts the following expression (changes false to true, and vice vers\
a)
||                logical OR
&&                logical AND
```

Logical Operators in Dart

let us see some different types of Boolean expressions and usage of Logical operators.

```dart
void main() {
int? x;
int? y;
if(x == y){
    print('Both $x and $y are NULL');
}
int? m = 10;
int? n = 20;
if(m == n || m != n){
    // False OR True => **TRUE** will prevail
    print('Both $m and $n have unequal value');
} else {
    print('Both $m and $n have equal value');
}
// Output: Both 10 and 20 have unequal value
int? a = 5;
int? b = 5;
if(a == b || a != b){
    // True OR False => **TRUE** will prevail
    print('Both $a and $b have equal value');
} else {
    print('Both $a and $b have unequal value');
}
// output: Both 5 and 5 have equal value
if((a == b || a != b) && a != b){
    // True AND False => **FALSE** will prevail
    print('If part will work.');
} else {
    print('Else part will work.');
}
// output: Else part will work.
/// Conditional Expression
(a == b)? print('If part will work.'): print('Else part will work.');
// output: If part will work.
(a != b)? print('If part will work.'): print('Else part will work.');
// output: Else part will work.
}
```

For full code snippet, please visit the respective GitHub Repository.

Assignment Operators

The assignment operator is of two types. Simple and Compound.

In Dart, we assign a new value to a variable. We do this operation with the help of the “assignment operator”. The assignment operator looks like “=”.

For an example, let us consider this code.

```
int? a = 10;
int? b = 20;
a = b;
```

In the above code, we have firstly assigned a value to the variable “a”. The value is 10. Besides, we assigned 20 to the variable “b”.

However, in the next step, we have assigned the value of “b” to “a”.

As a result, the new value of “a” becomes 20 now.

Therefore, an assignment operator is used to assign a new value to a variable

What happens if the value of a variable is NULL? Since the Dart Programming Language practices “NULL SAFETY”, we must be careful.

Consider this code:

```
void main() {
int? firstNum = 10;
int? secondNum;
if(firstNum == 10) print("The value of $firstNum is set.");
if (secondNum == null) print("It is true.");
secondNum ??= firstNum;
print(secondNum);
}
```

In the above code, the value of the variable “secondNum” is NULL. That is why we used “??=” operator. It is also another kind of assignment operator.

As a result, the new value of the variable “secondNum” becomes 10.

```
//output
The value of 10 is set.
It is true.
10
```

However, these assignment operators are known as simple assignment operator.

The compound assignment operator is different. It combines an operation with an assignment.

Consider this code.

```
a += b;
// it assigns a value with an operation
a + b;
```

The above code first assigns the value of the variable “b” to the variable “a”.

Next, it also assigns an operation to the variable “a”. The operation is to add the value of “b” to “a”.

Let us summarise all assignment operators in one place.

```
main(){
  int? firstNum = 10;
  int? secondNum;
  if(firstNum == 10) print("The value of $firstNum is set.");
  if (secondNum == null) print("It is true.");
  secondNum ??= firstNum;
  print(secondNum);
  print("After using an assignment operator, the value changes.");
  secondNum += secondNum;
  print(secondNum);
  print("After using an assignment operator, the value changes again.");
  secondNum -= secondNum;
  print(secondNum);
  if (secondNum == null) {
    print("It is true.");
  } else {
    print("it is false, because the 'secondNUm' has the value of $secondNum now.");
  }
}
```

From the output, we can get an idea. We know how the simple and compound assignment operators work.

```
//output
The value of 10 is set.
It is true.
10
After using an assignment operator, the value changes.
20
After using an assignment operator, the value changes again.
0
it is false, because the 'secondNUm' has the value of 0 now.
```

The following table represents all type of simple and compound assignment operators.

```
=        *=        %=        >>>=        ^=
+=        /=        <<=        &=        |=
-=        ~/=        >>=
```

What is Conditional Expressions in Dart

Dart has three special operators that check the Conditional Expressions. It is a short-form of if-else statement.

We have already learned different types of operators in Dart. In this section we will learn Conditional Expressions. We can specify them as the “Conditional-Operators” also.

How do they look like?

The Operators look like “?” and “:”, and they replace if-else statements.

Firstly, let us explain it. Secondly, we will see how it works.

The “Question Mark” and the “Semicolon” serve a particular purpose in the Dart Programming Language.

After a hypothesis we put the “?” operator. And then we check between two conclusions with the “:” operator.

```
Hypothesis ? Conclusion One : Conclusion Two;
```

In the above code, if the “Hypothesis” is “True”, the “Conclusion One” must be “True”. Otherwise, the “Conclusion Two” becomes “True”.

Therefore, these two operators in Dart let us evaluate expressions.

We run the same operation with a long-version if-else statements.

Let us go through a code snippet which will explain the concept.

```
main() {
int? firstNum = 10;
int? secondNum;
var first = firstNum != 10 ? 'Not $firstNum' : '$firstNum';
print(first); // 10
/// the long version with if-else
if (firstNum != 10) {
    print('Not $firstNum');
} else {
    print('$firstNum'); // 10
}
var second = secondNum ?? 10;

/// if secondNum was not NULL the second would return its value
/// in this case, secondNum was NULL
/// therefore second expression, or 10 was returned
print('$second'); // 10
}
```

By the way, the Dart language also allows us to check whether the value is NULL or not. In such cases, we use another operator “??”.

```
Hypothesis ?? Conclusion;
```

If the hypothesis is NULL we get the conclusion at once. If the hypothesis is not NULL, it becomes the “conclusion”.

In the above code, the same thing happens.

```
int? secondNum;
var second = secondNum ?? 10;

/// if secondNum was not NULL the second would return its value
/// in this case, secondNum was NULL
/// therefore second expression, or 10 was returned
print('$second'); // 10
```

Basically it checks whether the “hypothesis” is NULL or not.

We have saved all code snippets in one place. Please check the respective branche of this GitHub Repository.

Relation between Flutter and Dart

We have found out that Flutter is a framework or tool that we need to create beautiful mobile applications. Flutter is written in Dart programming language. To understand how Flutter works, we need to understand Dart also.

Before digging deep to find out the relation between Flutter and Dart, let us try to understand one key concept of programming. There are two distinct parts of programming. One is abstraction and the other is concretion. We need to convert our abstract ideas into concretion, or a concrete shape or form.

Any mobile application is an abstract idea. We need a tool like Flutter to give it a concrete shape. Take a more real life example. Justice is an abstraction, but law is a tool. When we say that 'justice is done', the abstract idea of Justice gets a concrete shape. And it is 'done' by using the tool called 'law'.

We hope that now we get a more clear picture why we need a tool like Flutter. Because we want to convert our abstract idea of making a mobile application we need a tool like Flutter.

While we use Flutter, we will encounter many terms like function, class, constructor, positional parameter, named parameter, object, Widget, etc, etc.

As an absolute beginner if you search the Internet, you will find that before Flutter developers used either Ionic or React Native to build mobile applications. Android developers used Java also. You can use Java to build Android application. Therefore, Flutter is not doing anything new. People used to do that before using other tools. Reading until this point, we may ask, then why we should learn Flutter. We could have learned something else,some other tools. Some other languages.

This question is pertinent to our discussion.

Flutter has some benefits. You enjoy some more privileges not enjoyed by other developers. To use the Flutter tool, you need to learn one programming language – Dart, and Flutter has a single code-base that can be used to build Android and native iOS mobile application.

It is a specialty, special advantage not enjoyed by all who use other tools.

Therefore, as an absolute beginner, you need to remember that Dart is a programming language. And Flutter uses Dart language building mobile applications on top of Dart platform. Flutter has some more components, such as Software Development Kit or SDK, flutter engine, foundation libraries and Widgets that are design specific.

As a programming language, Dart has other functionalities.

We can build web or desktop applications with Dart. Feel free to differ, but Dart seems to be a mixture of C and Java. If you have already learned these two languages, Dart will appear to be less daunting. Flutter runs in the Dart virtual machine. Just like we have seen earlier in Java Virtual Machine or JVM.

We do not want to be more specific on Flutter internals, as this book is aimed for the absolute beginners.

To be more specific on how Flutter and Dart work together, we will create another Dart project in our IntelliJ IDEA Community Edition IDE. In that Dart project we will learn Dart simultaneously as we progress with Flutter in a different project. We hope that will make sense.

To create a separate Dart project, we will open our IntelliJ IDE and add the Dart and Flutter plugins first. After that, we will open up our IntelliJ IDE to create a console based Dart application (Figure 1.7).

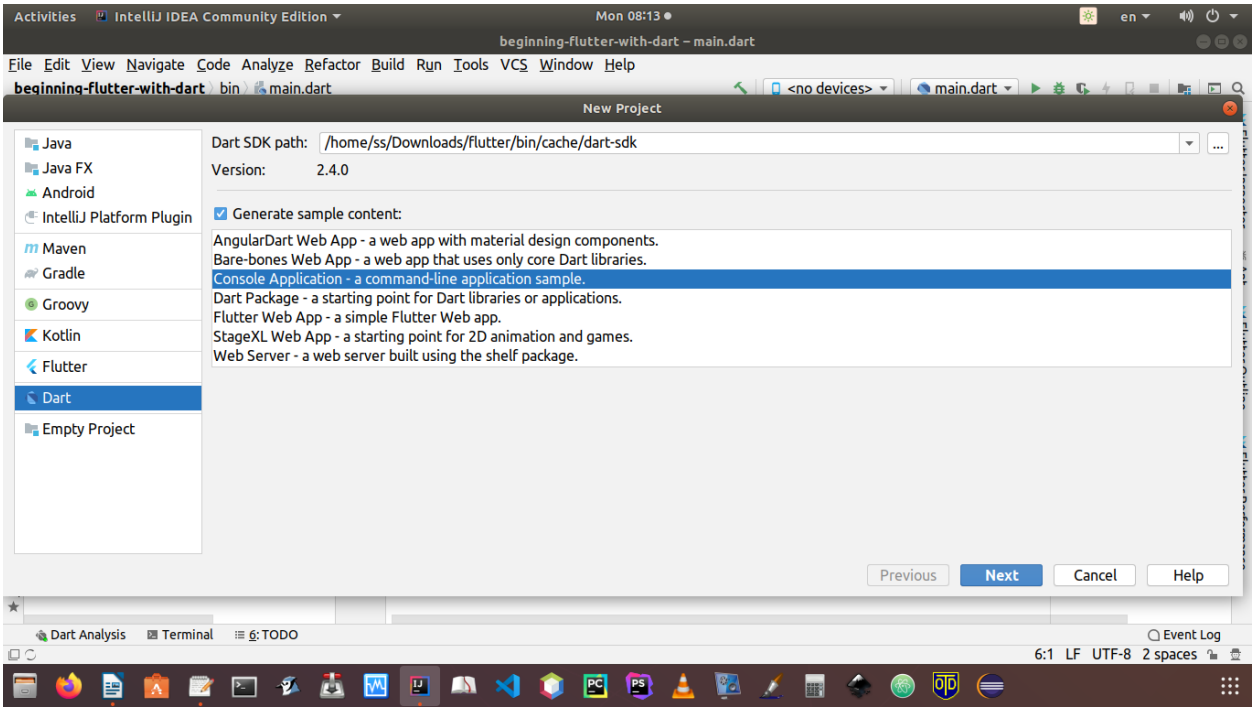

Figure 1.7 – A Dart Console project in IntelliJ IDE

As in the above image, we are going to create a Dart Console application. In this command line application sample, we will write different type of Dart code to learn the language basics.

The language basics is enough to give us a brief and primary idea about how Flutter works and builds a mobile application.

We are keeping these two projects separate. For Dart we have a console sample application named 'beginning_flutter_with_dart' and for the Flutter project we have 'my_first_flutter_app'.

We have saved these two separate projects in two separate folders.

When we have created the Dart project, it comes up with two '.dart' files. The 'main.dart' is in the 'bin' folder, and the 'beginning_flutter_with_dart.dart' file is in the 'lib' folder.

Like 'C', 'C++' or 'Java', Dart application runs through the 'main()' function. Therefore, the 'main.dart' file in the 'bin' folder is the main file through which our Dart console sample application will run.

What is the role of the 'lib' folder? We will place other dart files in the 'lib' folder, just like the

'beginning_flutter_with_dart.dart' file.

When we open up the 'beginning_flutter_with_dart.dart' file, we find a function inside.

```
// code 1.9
int calculate() {
return 6 * 7;
}
```

And the 'main.dart' file has this code inside:

```
// code 1.10
import 'package:beginning_flutter_with_dart/beginning_flutter_with_dart.dart'
as beginning_flutter_with_dart;

main(List<String> arguments) {
print('Hello world: ${beginning_flutter_with_dart.calculate()}!');
}
```

If we run this code we get this output:

```
//output of code 1.10
Hello world: 42!
```

If you are an absolute beginner, it really makes no sense. Let us tolerate this for a moment and try to understand what is actually happening.

As a beginner, try to understand programming language from the standpoint of natural language. In a natural language, we start with letters or alphabets. Then we form words by arranging those alphabets. After that we need to learn grammar, which is a set of rules that teaches us to make meaningful sentences. Only after learning to create sentences, we can think of writing a paragraph, an essay, a story, even a novel.

With the help of a programming language we also try to write an application, a software. A natural language works with words, and a programming language deals with data. This Dart console application gives us two types of data. One is 'String' data that gives the output: Hello World; and, it also gives us an 'integer' data, which gives us an output of 42.

Watch the code 1.9 carefully, it is a function of integer data type, and it has a name 'calculate()', and it returns an integer value of multiplication between two numbers 6 and 7. Quite predictably it has returned 42.

Therefore, we have learned one important concept, a function returns a value. If we mention what type of data type it will return, it will return that data type and in the main() function, we can call this function and get the desired output.

However, in the main() function, there are many more things that we should also discuss. If we take a look at the code 1.10, we see that in the 'main.dart' file we have imported the 'beginning_flutter_-with_dart.dart' file from the 'lib' folder as a package. When we have created the Dart application, it automatically gives it a name, the same name that we used while creating the console sample application.

Now, inside the print() function, that usually prints an output, Dart uses that name (beginning_-flutter_with_dart) and adds a '.' symbol to call the 'calculate()' function. Why this is happening? It is because Dart is an object-oriented programming language, and in Dart treats everything as an object. Through that object Dart calls any function as it has called here.

Now let us change the code 1.10 to this:

```
// code 1.11
import 'package:beginning_flutter_with_dart/beginning_flutter_with_dart.dart'
as an_object;

main(List<String> arguments) {
print('Hello world: ${an_object.calculate()}!');
}
```

Simultaneously, we have changed the code 1.9 to this, where we have changed the inside value of calculate() function.

```
// code 1.12
int calculate() {
return 6 * 12;
}
```

If we run this program, it works and gives us this output:

```
//output of code 1.11
Hello world: 72!
```

We have learned a few important concepts. Dart converts everything into an object. We can call that object by any name. At the time of creation, the object was named 'beginning_flutter_with_dart'; later we have changed the name to a more generic name, such as 'an_object'. After changing the name, we have called the same function that was written into the Dart file, inside the 'lib' folder.

Can we create another function in the 'lib' folder?

Let us try. Creating a new function is very simple process. We will click the second mouse on the 'lib' folder in our IntelliJ IDE, it will automatically ask for creating different types of file. We have chosen a Dart file and named it 'a_new_function'. A new Dart file is generated inside the 'lib' folder.

We are trying to add two numbers through that function and returns its value. The code snippet looks like this:

```
// code 1.13
int addingTwoNumbers(var x, var y){
return x + y;
}
```

Now we will call this function inside the main() function just like before.

```
// code 1.14
import 'package:beginning_flutter_with_dart/beginning_flutter_with_dart.dart'
as an_object;
import 'package:beginning_flutter_with_dart/a_new_function.dart'
as a_new_function;

main(List<String> arguments) {
print('Hello world: ${an_object.calculate()}!');
print('Adding 10 and 20: ${a_new_function.addingTwoNumbers(10, 20)}');
}
```

The output is quite predictable.

```
//output of code 1.14
Hello world: 72!
Adding 10 and 20: 30
```

We have passed two variables ‘x’ and ‘y’ as parameters through the function addingTwoNumbers(var x, var y); instead of using the word ‘var’ we could have written ‘int’. Dart is strongly typed programming language, so we can mention what data type we are passing. Otherwise, we can use only ‘var’, that stands for variable, and Dart will automatically infer it as integers; in this case we wanted to return an integer data type.

If you look at the meaning of the word in natural language,the word ‘function’ is used as noun as well as verb. When you use it as noun, one of the meanings tells us something like this: the actions and activities assigned to or required or expected of a person or group. And as a verb, its meaning is quite straight forward: perform as expected when applied.

In the programming paradigm, a function() does not always return something. It could be void. That means it does not return anything. Take a look at the main() function. Before the main() function, have seen any data type like ‘int’ or ‘String’? The main() function always calls other functions and gives us the output.

Can we create a void function and call it inside a main function?

Yes, we can do. Let us add a void function inside the ‘a_new_function.dart’ file and the code 1.13 looks like this:

```
// code 1.15
int addingTwoNumbers(var x, var y){
return x + y;
}

void doNothing(){
print('Do nothing');
}
```

Now we can call this void function just like any other regular function, inside the man() function.

```
// code 1.16
import 'package:beginning_flutter_with_dart/beginning_flutter_with_dart.dart'
as an_object;
import 'package:beginning_flutter_with_dart/a_new_function.dart'
as a_new_function;

main(List<String> arguments) {
print('Hello world: ${an_object.calculate()}!');
print('Adding 10 and 20: ${a_new_function.addingTwoNumbers(10, 20)}');
a_new_function.doNothing();
}

//output
Hello world: 72!
Adding 10 and 20: 30
Do nothing
```

Since inside the void function we have used a print() function and passed a String object - 'Do nothing'. We get the same output.

We may ask, what is the function of this functions inside the Flutter project? Do this functions will have to do anything with our first mobile application?

To get that answer, we need to close our Dart project for a time being and move to the Flutter project 'my_first_flutter_app'. Let us open the Android Studio.

When the Flutter project was created it came with a main() file, just like we have just seen in the Dart project.

The code snippet is quite long, but we want to see the whole code here, because we want to see if we can find something familiar as an absolute beginner. So far, we have learned to create functions, and we have heard that everything in Dart is object, but we still do not understand it very much.

Let us see the whole code snippets, without the comments, first.

```
// code 1.17
import 'package:flutter/material.dart';

void main() {
runApp(MyApp());
}

class MyApp extends StatelessWidget {
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(

        primarySwatch: Colors.blue,

        visualDensity: VisualDensity.adaptivePlatformDensity,
    ),
    home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
}
}

class MyHomePage extends StatefulWidget {
MyHomePage({Key key, this.title}) : super(key: key);

final String title;

@override
_MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
int _counter = 0;

void _incrementCounter() {
    setState(() {

    _counter++;
    });
}

```

```
@override
Widget build(BuildContext context) {

    return Scaffold(
    appBar: AppBar(

        title: Text(widget.title),
    ),
    body: Center(

        child: Column(

        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
            Text(
            'You have pushed the button this many times:',
            ),
            Text(
            '$_counter',
            style: Theme.of(context).textTheme.headline4,
            ),
        ],
        ),
    ),
    floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
    ),
    );
}
}
```

Watching the above code, we can say that, yes, we have found one familiar thing, a main() function. And the main()function here directly calls a function runApp(); inside that runApp() function, Flutter has passed a parameter MyApp(), which also looks like a function. But it is not a regular function. It is an object that has been instantiated from the class MyApp().

Because of this code our virtual device looks like the figure 1.6.

We will remove all the code and build our mobile application from the scratch so that we can follow the building process step by step.

But before that, we will try to understand the code. We have already found one familiar function main() inside the Flutter 'main.dart' file. The second most important thing we have noticed a

comment that tells us that '// This widget is the root of your application'.

Basically, in Flutter, Widget plays the key role. We can summarize that it is all about Widget. Our mobile application is a collection of many parent and child Widgets. The Widget tree contains different child Widgets, they draw the image on the mobile screen pixel by pixel.

At the top there is a header section, after that just below of the header starts the body part. Again, the body part contains many other Widgets. Things go on like this. In Flutter you cannot drag and drop your Widgets; we have to code them in different folders and after that we can import them as packages.

At the top of the Flutter 'main.dart' file we have seen this line:

```
1  import 'package:flutter/material.dart';
```

This 'material.dart' file has been supplied by Flutter. This file has many core functionalities that we can call in the 'main.dart' file.

We have also noticed a line like this:

```
1  class MyApp extends StatelessWidget {}
```

As an absolute beginner, we have learned function(), but we have not found out what class is. Moreover, we also do not know how 'class' and 'object' are connected; yet it is said that in Dart everything is object. Therefore, we will again, go back to our Dart console sample application; we will learn this core concepts and after that we will again come back to the Flutter project and build our mobile application from the scratch.

However, we feel that we need to understand the concepts of function in detail, especially in the context of object-oriented programming paradigm.

Functions and Objects

When we say: functions are objects in Dart, it seems confusing to the absolute beginners. The seasoned programmers may get the hint: Dart is an out and out object-oriented language. So even functions are objects and have a type called – Function.

It means many things. One of the key things is you can assign a function to a variable, and even you can pass a function as arguments to other functions. We have seen it in the Flutter 'main.dart' file as the following:

```
void main() {
runApp(MyApp());
}
```

To understand objects, you need to have an introduction to object-oriented programming. In this section, we will have an introduction to object-oriented programming. Otherwise, we cannot follow how Flutter works with its objects.

We have already seen how functions work. Although that was a basic introduction; we will cover this topic later in detail when we will discuss 'methods' in object-oriented programming.

Before writing a function, we need to remember a few major points:

```
1. It is a good practice to define a type of function. So type annotation is recomme\
nded.
2. Although Dart recommends type annotation, a function still works without any "typ\
e declaration". So you can omit the type and write it straight.
3. However, the most important thing to remember in Dart is: whatever value you want\
 to 'return', from a function, you need to change the 'type' of that function accord\
ingly. If you want an 'integer' value to 'return', you should change the 'type' of t\
he function to 'integer'.
4. For 'void', nothing is returned from a function. So whenever you use the keyword \
'void' before the function, you need to use the 'print(object)' option.
```

So far we have seen integer and String data types that return numbers and texts respectively. However, there is another major data type in every programming language, Dart is no exception. It is called 'boolean'. It returns either 'true' or 'false'. In our Flutter project, we will need this data type, especially while building the app logic.

Let us see some examples, it will give us a clear picture of how boolean data work. In our Dart project, in the 'lib' folder, we will create a new file 'boolean_function.dart'. We have written this following code inside that file:

```
// code 1.18
bool isTrue(){
return true;
}
bool isFalse(){
return false;
}
```

We should also change the 'main.dart' file accordingly.

```
// code 1.19
import 'package:beginning_flutter_with_dart/boolean_function.dart'
as boolean_object;

dynamic main(List<String> arguments) {

print('It is true: ${boolean_object.isTrue()}');
print('It is false: ${boolean_object.isFalse()}');

}
//output

It is true: true
It is true: false
```

If we did not mention the boolean type in the code 1.18, it would still work. But Dart strongly recommends to define the type. It makes your code more clear. By the way,we should know about a few other good practices.

According to the naming convention, it is recommended that a function name should always be like this: 'aFunction()'; it is called camel case. The initial word will start with lower case, and the next word should start with an upper case. If we want to mean a single verb action, we can use one word like 'build()', which Flutter has done in its 'main.dart' file.

In the Flutter code, it means, the Widget control is building the first application.

When we name a class, we will use Pascal case, such as 'MyApp' used in the Flutter 'main.dart' file; in this case,both words start with upper case.

The names of function and class should always be meaningful and should be synchronized with your application. If we take a close look at the Flutter 'main.dart' file, we will understand that every name of class, function is meaningful.

To understand this naming convention, we should take a look at the Flutter 'main.dart' code again. Watch this code snippets from code 1.17:

```
class MyApp extends StatelessWidget {
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(

        primarySwatch: Colors.blue,

```

```
        visualDensity: VisualDensity.adaptivePlatformDensity,
    ),
    home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
}
}
```

We can clearly see that the MaterialApp() calls another function ThemeData() inside it. Therefore, we can call another function inside a function.

We will go back to the Flutter project again, but before that, let us open our IntelliJ IDE and try to pass the 'ThemeData()' function inside the function 'MaterialApp() '. To do that we need to create 'passing_a_function_inside_function.dart'. Inside that file, we write a simple function that passes a function as its parameter.

```
// code 1.20
String MaterialApp(page()){
return page();
}
```

Now in the 'main.dart' file we call that 'MaterialApp(page())' function and try to pass another function as its parameter.

```
// code 1.21
import 'package:beginning_flutter_with_dart/passing_a_function_inside_function.dart'
as passing_function;

// ignore: always_declare_return_types
main(List<String> arguments) {

// ignore: always_declare_return_types
themeData(){
    return 'Home Page';
}

print('Passing function inside a function: ${passing_function.MaterialApp(themeData)\
}');

}

//output
Passing function inside a function: Home Page
```

We have just done what we have seen in the Flutter 'main.dart' file. Of course, we have done in a microscopic form. Flutter uses the same concepts in a much bigger way. But we have at least started understanding what is happening inside. Yet, we need to understand class and objects. When we use a function inside a class, it is usually called a method.

A class may extend functionalities of other classes. Flutter tool uses hundreds of such classes, and extend many more classes to use their functionalities and give the application a concrete shape.

Before going to understand classes and objects, let us go back to our Flutter project again. This time, we will remove the in-built code and after that, we will try to write a small piece of code, to change the appearance of the virtual device.

Building the mobile application from scratch

Let us open the Android Studio and start our virtual device. To start building our first mobile application, we need to remove all the in-built code from the 'main.dart' file.

Let us name our application "MyFirstApp". The very first thing we need to do is, write a main() function inside the 'main.dart' file. Any Flutter project will always launch through the main() function.

We are not going to do anything special. Displaying a text,such as 'My First Flutter app from scratch...' will be enough at the beginning. Let us write our code, the following way:

```
1  // code 1.22
2  import 'package:flutter/material.dart';
3
4  void main() {
5  runApp(MyFirstApp());
6  }
7
8  class MyFirstApp extends StatelessWidget {
9  Widget build(BuildContext context) {
10     return MaterialApp(home: Text('My First Flutter app from scratch...'),
11     );
12 }
13 }
```

Before running our code we will take a look at the virtual device that comes up with the creation of the Flutter project. Think about the code 1.17, which was created on the original 'main.dart' file. It created a virtual device,which displays a counter. Clicking the button raise the counter number by 1.

The virtual device initially looked like the following image (Figure 1.8).

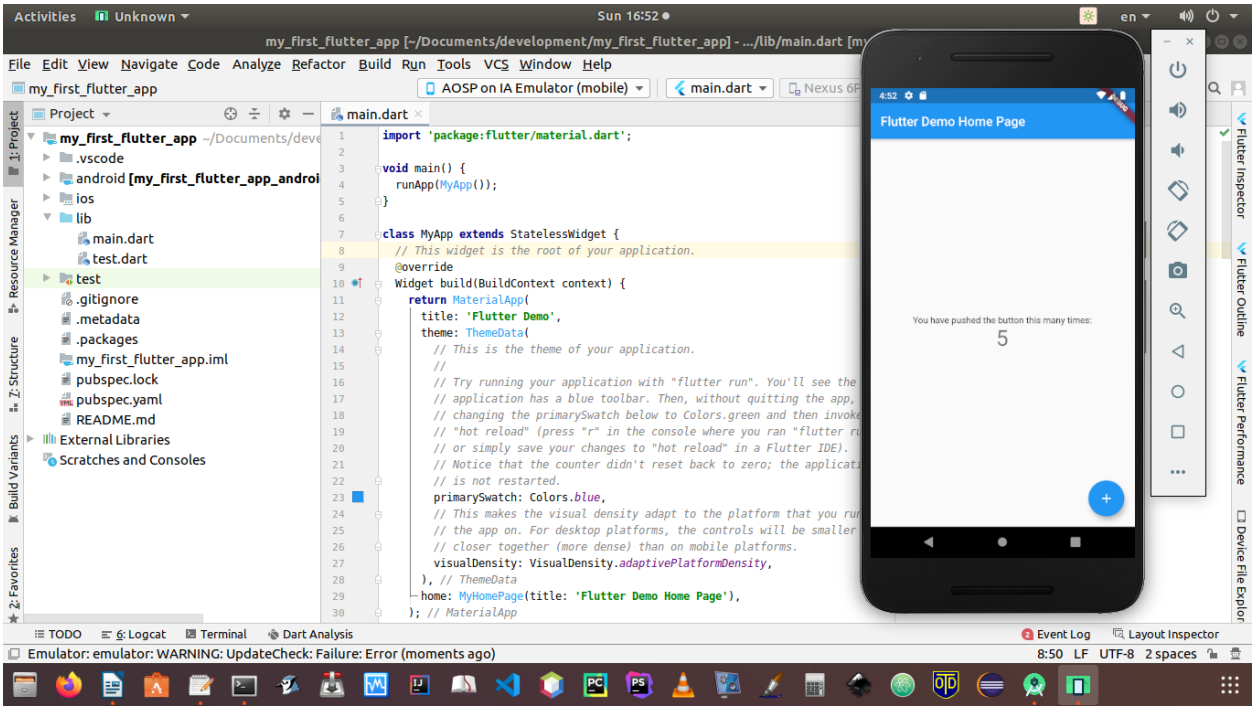

Figure 1.8 – The virtual device at the time of creation of Flutter App

Now we have changed the original 'main.dart' file and building our first flutter application from scratch. Let us run the new 'main.dart' file, and we get this output in our virtual device (Figure 1.9). It is not a good looking application, at present, but to add more functionalities we need to understand more core concepts, such as how class and object work in Dart.

We have kept the main() file as it was. Only we have changed our application's name from 'MyApp' to 'MyFirstApp'.

```
void main() {
runApp(MyFirstApp());
}
```

We have also changed the original class configuration. Now 'MyFirstApp(){}' class extends another class 'StatelessWidget(){}', which also passes the Widget method build(). The Widget build() method also passes another 'BuildContext' object called 'context' as its parameter. This build() method returns another class constructor 'MaterialApp()' that returns a 'named parameter' Text() class constructor. Inside the Text() class constructor we have passed a String parameter 'My First Flutter app from scratch...'.

As an absolute beginner, it appears to be very difficult if you do not know anything about class and objects. However, our first code works and change the appearance of the virtual device.

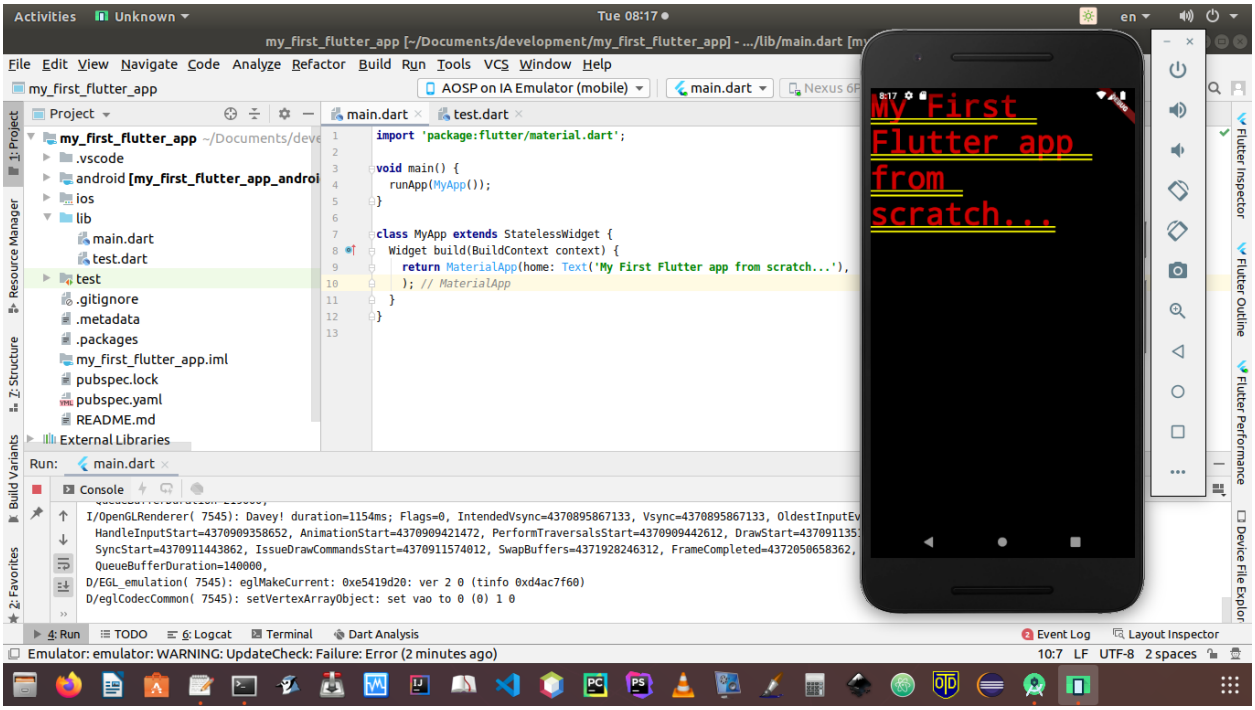

Figure 1.9 – Building our first mobile application from scratch

It does not look good. But we have just started building the application. It will look better as the application progresses. However, before progressing any further with our first Flutter application, we have to understand how class and object work. To understand that we need to go back to our Dart project again.

For more Flutter related Articles and Resources[6]

[6] https://sanjibsinha.com

2. Flutter and Dart Architecture: Understanding Class and Object

Flutter is a software development tool kit, or in short, a tool.

However, Dart is a programming language. And, Flutter uses Dart.

That is why we need to understand the basics of Dart programming language.

Why?

Because that will help us to understand the rules that Flutter Tool follows.

Does Flutter require Dart

The Flutter-Framework needs to implement the important concepts in Dart.

In our previous section we have seen what are important concepts in Dart. Flutter requires, not only Dart, but it also requires that you must have understood those concepts.

Why?

Because, firstly, the Flutter-Framework needs to implement the important concepts.

We will see that in a minute.

Secondly, we use Dart coding convention in Flutter. As a result a simple Flutter Application uses the same coding pattern. We have seen that in Dart. The Dart built-in “Type” is used. We can use the conditional expressions, and many more in Flutter.

Finally, an absolute beginner wants to learn the Flutter structure first.

Therefore, let us first create a simple Flutter Application.

flutter create first_flutter_app

As a result, we see the following code in our Visual studio IDE.

```
import 'package:flutter/material.dart';

void main() {
runApp(MyApp());
}

class MyApp extends StatelessWidget {
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
    ),
    home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
}
}

class MyHomePage extends StatefulWidget {
MyHomePage({Key key, this.title}) : super(key: key);

final String title;

@override
_MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
int _counter = 0;

void _incrementCounter() {
    setState(() {
    _counter++;
    });
}

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
```

```
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
            Text(
            'You have pushed the button this many times:',
            ),
            Text(
            '$_counter',
            style: Theme.of(context).textTheme.headline4,
            ),
        ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
      );
  }
  }
```

It is a long code. Moreover, an absolute beginner with no coding background, is not supposed to understand it.

Right?

However, we may try to understand the Structure of the Flutter Application.

Structure of a Flutter Application

Therefore, next, we are going to understand the structure as a beginner. It is presumed that we do not know coding. We have just started learning Dart.

In that scenario, the above code really does not make any sense. It might make sense when we understand the Structure. In addition we can analyse them after that.

Let us start with the first part.

```
import 'package:flutter/material.dart';
```

We have imported the Flutter Material library. So that we can build a Material App. Previously we have learned how to use the Dart core library.

The Dart and Flutter both start from the “main()” function. Inside the “main()” function we run another function “runApp()” and pass the root Widget.

We will learn about the “function” in Dart and Flutter in the next section.

What is Flutter Widget?

In Flutter everything is Widget. A Widget is nothing but a “Class”. And as a Widget it always builds other Widgets inside it.

Previously we have learned that a “Class” defines the “Type” of an “Object”.

Consequently, what we see is a Widget tree.

If you see a Flutter Application, it will clarify the concept of Widget Tree.

The root Flutter Application starts with a Widget tree. There are several Widgets under the root Widget.

The coming sections will show us what is a “function” and “class”. Till then, we just try to understand how the other part of the structure works.

```
void main() {
runApp(MyApp());
}
```

The root Widget “MyApp()” returns a “MaterialApp()” Widget inside its “build()” method.

As a consequence the “MaterialApp()” Widget, or the Class passes several arguments through its Constructor.

One of the arguments is “home”.

Let us see the code.

```
class MyApp extends StatelessWidget {
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
    ),
    home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
}
}
```

The ” MyHomePage()” is our home page. Moreover, it is the “subclass” of a Stateful Widget.

The same way, just before we have seen that the root Widget “MyAp()” is the “subclass” of a Stateless Widget.

We will not discuss the difference between the Stateful and the Stateless Widget at this point.

Why?

Because that is beyond the scope of an Absolute beginner. We will learn it later, when the proper time comes.

However, we have a basic idea how Flutter Application builds its Structure.

Next, we will learn what are “function” and “class” in Dart. Subsequently, we will come back again and try to understand the rest part of the Flutter Application Structure.

What is Dart and Flutter function

There are four functions in Dart and Flutter. And every function is an Object.

In Dart and Flutter everything is Object. Therefore a “function” is also an “Object”. We have already learned that every Object in Dart has a “Type”. As a result, the “function” in Dart and Flutter has also a “Type”.

The “function” type is Function. As usual it is a Class. The base class for all function types.

As an absolute beginner, it is difficult to understand how a Class becomes a “type” of an object. However, we have discussed this topic in our previous sections.

As of now, we do not have to understand how the Function class works in the background. We need to know a few basic things about a “function” in Dart and Flutter.

How do you use a Dart function?

Firstly, we will learn how we use a Dart function. Later we will learn how many functions are there in Dart and Flutter.

We can write a function in Dart in the following way.

```
1 bool isTrue() {
2 return true;
3 }
```

As we see, the function has a type, name and, moreover, it is returning the that type in its body.

We could have written this function this way also.

```
bool isTrue() => true;
```

Since the above function contain just one expression, we have used a shorthand syntax. We also call it the “arrow-syntax”.

Since any “function” in Dart and Flutter is an “Object”, we can assign it to a “variable”.

```
var testTrue = isTrue();
```

The same way, we can define another function of same type and assign it to a variable.

```
bool isFalse() {
return false;
}

var testFalse = isFalse();
```

How many functions are there in Dart and Flutter?

There are four categories of functions in Dart and Flutter. We will see them in a minute.

However, before that, let us try to understand why we need a function.

As we can see in the above code, a function is a set of statements that take inputs. After taking the inputs, it returns an output. It is reusable. Consequently, we can call a function anywhere in our Dart and Flutter Application.

In the above code snippets the same thing occurs.

A Boolean type function returns a Boolean value. This is the simplest form of a function.

Sometimes, a function might not return anything.

```
void returnVoid() => print('Returning void');
```

When a function does not return any specific “type”, we call it “void”.

To put it simply, this is our first category of functions. The function has no “argument”, no “return-type”.

Let us see the second category. A function that has arguments and return type.

```
int addNumbers(int x, int y) {
return x + y;
}
print(addNumbers(50, 60)); // 110

int addTwoNumbers(int x, int y) => x + y;
print(addTwoNumbers(10, 20)); // 30
```

We have already seen the third category. A function that has no argument, but has a return type.

```
bool isTrue() {
return true;
}
```

The fourth category of function has arguments, but no return type.

```
void getName(String name) => print(name);
getName('John'); // John
```

Until now, we have got a basic idea about how the “function” works in Dart.

Next, we will see a simple example of how we can use a function in Flutter.

How do you use a function in flutter?

Later as we progress, we will see a lot of examples regarding this topic.

However, let us get the simplest idea about how we use a function in Flutter. Consider any Flutter Application. We will always find the “build()” method.

Right?

That is a function. And the “type” of any “build()” function is the “Widget”. And it usually takes an argument “context”.

Why?

The reason is simple. The “build()” method, or function, whatever you call it, returns a Widget. That is why, the “type” of a “build()” method is the Widget.

Let us take a look at any Flutter Application.

```
class MyApp extends StatelessWidget {
const MyApp({Key? key}) : super(key: key);

static const String _title = 'Flutter Animated TextStyle';

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: _title,
    debugShowCheckedModeBanner: false,
    home: Scaffold(
        appBar: AppBar(title: const Text(_title)),
        body: Column(
            children: const [
            ImplicitTextStyleAnimation(),
            SizedBox(
                height: 20.0,
            ),
            ExplicitTextStyleAnimation(),
            ],
        )),
    );
}
}
```

Forget about the rest part. Just take a look at this part only.

```
Widget build(BuildContext context) {
    return MaterialApp(...)
....
}
```

If we compare this part of any Flutter Application, we can see the similarity. It looks like any other simple Dart function.

Is not it?

Finally, let us see the full code snippets that we have used so far.

```
main() {
/// everything in Dart is Object
/// even functions are objects and have a type, Function
///
bool isTrue() {
return true;
}

var testTrue = isTrue();

bool isFalse() {
return false;
}

var testFalse = isFalse();

int addNumbers(int x, int y) {
return x + y;
}
print(addNumbers(50, 60));

int addTwoNumbers(int x, int y) => x + y;
print(addTwoNumbers(10, 20)); // 30

void returnVoid() => print('Returning void');

isTrue() ? print('It is $testTrue') : print('It is $testFalse');
isFalse() ? print('It is $testFalse') : print('It is $testTrue');
var result = addTwoNumbers(10, 20);
print(result);
returnVoid();
void getName(String name) => print(name);
getName('John'); // John

}
/**
// output

110
30
It is true
It is true
30
```

```
Returning void
John

*/
```

What are parameters in Dart

The positional parameters in Dart can either be named, or optional. But cannot be both at the same time.

In the previous section we have seen what is “function” in Dart and Flutter. Although we have not discussed “arguments”, or “parameters”, still we have had a glimpse of it.

```
int addNumbers(int x, int y) {
return x + y;
}
print(addNumbers(50, 60));

int addTwoNumbers(int x, int y) => x + y;
print(addTwoNumbers(10, 20)); // 30
```

In the above code we have passed two “arguments” to the “functions” – “addNumbers(int x, int y)” and “addTwoNumbers(int x, int y)”. In other words, we have defined the function parameters.

Since this function returns an expression, we can also use the “arrow” syntax. If the function body was a statement, we could not use the arrow syntax.

From now on, we will try to understand how we can pass “arguments”, or define “parameters”.

Remember, we are trying to understand the Dart programming language as an absolute beginner. So that, later, we can apply the same concepts in Flutter.

In the above code, we have seen the example of “positional parameters”.

Let us see more positional parameters.

```
String greet(String name, String message) {
    return '$name tells you $message';
}

print(greet('Json', 'How is everything?'));
```

Run the above program. We will get the following output.

```
Json tells you How is everything?
```

We can pass any number of arguments. Or, we can define any number of function parameters.

In addition, we can make any positional parameter optional.

However, if we want to define the “optional positional parameter”, we wrap the parameters with “[ ]” marks.

```
int multiplyThreeWithOptionalPositionalParameters(
    int x,
    int y, [
    int? z,
]) {
    var result = x * y;
    if (z != null) {
    result = x * y * z;
    }
    return result;
}

print(multiplyThreeWithOptionalPositionalParameters(10, 20));
print(multiplyThreeWithOptionalPositionalParameters(10, 20, 5));
```

As a result, we can call the function either “without” the “optional” parameter, or “with” the “optional” parameter.

If we run the program, it checks whether the value of the optional parameter is NULL or not NULL.

When the value is not given to any optional parameter, it takes the NULL value.

In that case, the program ignores the NULL value. An it multiplies only the first two positional parameters whose values are given.

Consequently, the output is as the following.

```
200
1000
```

However, we can tweak the above code to make it short.

We can use “=” to define default values for optional parameters. If we pass a constant value 1 as its default value, the long code statement reduces to a single expression.

```
int multiply(int x, int y, [int? z = 1]) => x * y * z!;
print(multiply(10, 20));
print(multiply(10, 20, 5));
```

Consequently, the output is as same as we have got before.

```
200
1000
```

Therefore the rule is: If a parameter is optional but can't be null, provide a default value.

Finally, we will take a look at the "named parameters". The "named parameters" are also optional. Unless we make them "required", the "named parameters" always remain optional.

In Flutter, we always find them.

How do we create the "named parameters"?

We pass the arguments or define function parameters inside two curly braces – "{ }".

Let us see an example.

```
/// named parameters are optional unless they are required
///
String sayHello({
    required String message,
    String? name,
}) {
    return message + name!;
}

print(sayHello(message: 'Hello', name: ' John'));
```

In the above code, we have made one "named" parameter "required".

As a result, when we call the function, we have to pass the value.

It is mandatory.

In Flutter, we will find a lot of "named parameters". Both. Optional, and with the "required" keyword.

One such example is TextStyle Widget.

A very common Widget in Flutter, the TextStyle Widget constructor uses only named parameters.

```
const TextStyle(
            color: Colors.red,
            fontSize: 25.0,
            fontWeight: FontWeight.bold,
            )
```

Not only the TextStyle Widget, every Widget constructor in Flutter uses only named parameters. And some of them are mandatory. That means they have used the “required” keyword in their definition.

The discussion on Dart and Flutter function is not over yet.

For this part we have used the following code snippet.

```
import 'package:flutter/cupertino.dart';


main() {
/// everything in Dart is Object
/// even functions are objects and have a type, Function
///
String greet(String name, String message) {
    return '$name tells you $message';
}

print(greet('Json', 'How is everything?'));

/// optional positional parameters
///
int multiply(int x, int y, [int? z = 1]) => x * y * z!;
print(multiply(10, 20));
print(multiply(10, 2, 10));

int multiplyThreeWithOptionalPositionalParameters(
    int x,
    int y, [
    int? z,
]) {
    var result = x * y;
    if (z != null) {
    result = x * y * z;
    }
    return result;
}
```

```dart

print(multiplyThreeWithOptionalPositionalParameters(10, 20));
print(multiplyThreeWithOptionalPositionalParameters(10, 20, 5));

/// named parameters are optional unless they are required
///
String sayHello({
    required String message,
    String? name,
}) {
    return message + name!;
}

print(sayHello(message: 'Hello', name: ' John'));

/// A function can have any number of required positional parameters. These can be f\
ollowed either by named parameters or
/// by optional positional parameters (but not both).
}
/**
// output

Json tells you How is everything?
200
200
200
1000
Hello John


*/
```

What are top level functions in Dart

When a “function” sits on the top of Class, Object, or Interface, we call it the top-level function.

Firstly, we need to know the definition of the top level functions. Secondly, we will learn how these “top-level” functions work in Dart. And, finally, we will see some code snippets.

Why?

Because we have never vouchsafed that interesting titbit before.

Let us first know what is the “top-level” function. What do we mean by the term “top-level”?

First we know, then we will flirt with the “top-level” function.

Right?

When a “function” sits on the top of every Class, Object, or Interface, we call it the top-level function. It is not nested inside any other function, or any structure, whatsoever.

As we just said. The “top-level” function sits on the top of any hierarchy of Class or Function.

The best example is the “main()” function in Dart and Flutter. It is the entry point.

Certainly, we can write another top level function on the top of the main() function. But, to declare it, we need to use the entry-point which is the main() function.

Let us see one simple example.

```
void aTopLevelFunction() {} // A top-level function
main() {
Function anyFunctionInsideTopLevelFunction;

// Comparing top-level functions.
anyFunctionInsideTopLevelFunction = aTopLevelFunction;
if (aTopLevelFunction == anyFunctionInsideTopLevelFunction) {
    print('A top level function is same as any function '
        ' inside a top-level function.');
}
}

// #output: A top level function is same as any function  inside a top-level functio\
n.
```

The output is quite obvious. It tells us the story in detail.

We have defined a top level function at the top of the main() function which is also a “top level” function. However, when we declare another Function inside the main() function, it is same as the top level function we have already declared.

What is a function?

A function may beguile you with different definitions.

Therefore, just remember one golden rule.

A “function” always returns a “value”. If we do not specify any value, it returns “NULL”.

What do we mean by a “value”? The “Type” of the “Function”. The type might be any built-in-types in Dart. Or any custom-type.

We have discussed built-in-types. But time has not come to discuss the “custom type”. We will learn Class and Objects in detail first. After that, we will flirt with the “custom type”.

If we do not specify any type, a function in Dart always returns NULL.

Let us see a simple example.

```
/// All functions return a value. If no return value is
/// specified, the statement returns null
///
catchYou() {}

if (catchYou() == null) {
    print('It returns null.');
}
// #output: It returns null.
```

There are two other types of functions in Dart. We will see a lot of examples in Flutter too.

For that reason, let us try to understand them.

Although we have not discussed Class and Object yet, still there are two types of functions that we declare inside a Class.

When we discuss them, we will also have a gentle introduction to Class and Object.

What are methods in Dart?

Declare a function inside a Class, it becomes a method. The nature of function remains the same. However the name only changes.

There are two types of methods. Static method, and Instance method.

The term “Instance” actually represents “Object”. When we create an Object, we also call it an Instance. We also refer to this process as – instantiating a class.

We can access the “Static Method” with the Class-Name. As a result, we do not have to create an Object. Or, in other words, do not have to “instantiate”.

Let us see an example.

```
class AClass {
static void aStaticMethod() {} // A static method
void anInstanceMethod() {} // An instance method
}
```

In the above code snippet, we have defined a Class. Next, we have declared two methods.

The first one is a static method. And, the second one is an instance method.

As a result, we can access the static method with the Class name. Meanwhile, we can access the instance method only by creating an instance, or an object.

Let us proceed with our code.

```
main() {
/// we can declare any function inside any top-level function
Function anyFunctionInsideTopLevelFunction;

// Comparing static methods.
anyFunctionInsideTopLevelFunction = AClass.aStaticMethod;
if (AClass.aStaticMethod == anyFunctionInsideTopLevelFunction) {
    print('Any function inside a top level function '
        ' is as same as a static method.');
}
}
//#output: Any function inside a top level function  is as same as a static method.
```

When we declare any function inside a top-level function, we can also assign a static method for its use.

Moreover, they become same.

However, for instance methods it is not true.

Difference between Static Method and Instance Method

Firstly, let us create two instances.

```
var firstInstanceOfAClass = AClass(); // first Instance of AClass
var secondInstanceOfAClass = AClass(); // second Instance of AClass
```

Next, we will access the instance method through this instance.

And, after that we can compare them.

```
class AClass {
static void aStaticMethod() {} // A static method
void anInstanceMethod() {} // An instance method
}

main() {

var firstInstanceOfAClass = AClass(); // first Instance of AClass
var secondInstanceOfAClass = AClass(); // second Instance of AClass

var aVariable = secondInstanceOfAClass;
anyFunctionInsideTopLevelFunction = secondInstanceOfAClass.anInstanceMethod;

// These closures refer to the second instance,
// so they're equal.
if (aVariable.anInstanceMethod == anyFunctionInsideTopLevelFunction) {
    print('These closures refer to the second instance,'
        ' so they\'re equal.');
}

// These closures refer to different instances,
// so they're unequal.
if (firstInstanceOfAClass.anInstanceMethod !=
    secondInstanceOfAClass.anInstanceMethod) {
    print('These closures refer to different instances,'
        ' so they\'re unequal.');
}
}
```

As we look at the code snippet, we can get the idea. Although two instances access the same "instance method", they are not same.

Look at the output now.

```
These closures refer to the second instance, so they're equal.
These closures refer to different instances, so they're unequal.
```

If you want to go through the full code snippet, here is it.

Please read the comments carefully. It may help you clear confusions.

```dart
void aTopLevelFunction() {} // A top-level function

class AClass {
static void aStaticMethod() {} // A static method
void anInstanceMethod() {} // An instance method
}

main() {
/// Every app must have a top-level main() function,
/// which serves as the entrypoint to the app.
/// The main() function returns void
/// it has an optional List<String> parameter for arguments.

/// everything in Dart is Object
/// even functions are objects and have a type, Function
///

/// All functions return a value. If no return value is
/// specified, the statement return null
///
catchYou() {}

if (catchYou() == null) {
    print('It returns null.');
}

/// we can declare any function inside any top-level function

Function anyFunctionInsideTopLevelFunction;

// Comparing top-level functions.
anyFunctionInsideTopLevelFunction = aTopLevelFunction;
if (aTopLevelFunction == anyFunctionInsideTopLevelFunction) {
    print('A top level function is same as any function '
        ' inside a top-level function.');
}

// Comparing static methods.
anyFunctionInsideTopLevelFunction = AClass.aStaticMethod;
if (AClass.aStaticMethod == anyFunctionInsideTopLevelFunction) {
    print('Any function inside a top level function '
        ' is as same as a static method.');
}
```

```
44
45  // Comparing instance methods.
46  var firstInstanceOfAClass = AClass(); // first Instance of AClass
47  var secondInstanceOfAClass = AClass(); // second Instance of AClass
48  var aVariable = secondInstanceOfAClass;
49  anyFunctionInsideTopLevelFunction = secondInstanceOfAClass.anInstanceMethod;
50
51  // These closures refer to the second instance,
52  // so they're equal.
53  if (aVariable.anInstanceMethod == anyFunctionInsideTopLevelFunction) {
54      print('These closures refer to the second instance,'
55          ' so they\'re equal.');
56  }
57
58  // These closures refer to different instances,
59  // so they're unequal.
60  if (firstInstanceOfAClass.anInstanceMethod !=
61      secondInstanceOfAClass.anInstanceMethod) {
62      print('These closures refer to different instances,'
63          ' so they\'re unequal.');
64  }
65  }
66  /**
67  // output
68
69  It returns null.
70  A top level function is same as any function  inside a top-level function.
71  Any function inside a top level function  is as same as a static method.
72  These closures refer to the second instance, so they're equal.
73  These closures refer to different instances, so they're unequal.
74
75
76  */
```

What is Class in Dart

A Class defines the Type of an Object. Moreover, we can create as many Objects as we can.

Do not be afraid. The heading might suggest, we are going to start with a Car Class. ☒ Thankfully, we do not. Firstly, we will try to define the “Class” in Dart. Secondly, we will try to explain how we can use the “variable” and “function” with the “Class”.

Finally, in the next section, we will try to understand how Flutter uses these concepts.

We are going to write this article for absolute beginner. Therefore, we will try to be as simple. We hope that is not sinful.

The Dart programming language is a “strongly-typed” language. Although it allows “duck-typed” style. As a result, if we do not mention the “Type”, the Dart infers the necessary “Type”.

Meanwhile we have also learned that in Dart everything is Object. And behind every Object, there is a Class. In essence, a Class actually defines the “Type” of the Object.

We need to understand this statement first. A Class defines an Object. If we cannot define the “Type”, or the “Class” properly, the Object might be defiled by the Class.

What does that mean?

It means, when we create an Object, we need to know the purpose.

For what purpose we want to use that object? According to that purpose we will define its type, or class.

Consider a simple example. Why do we need an Integer?

We may need an integer for many purposes. However, behind every purpose there is a common pattern. Some kind of arithmetic operations are there. Right?

For that reason, we need arithmetic operators.

Why we are going to learn Dart programming language? Because we want to build a web, or mobile application with Flutter-Framework.

And in Flutter, we need to handle with hundreds of Classes, which are commonly known as Widgets.

These Flutter classes, or widgets have thousands of methods and instance variables.

As a beginner, we need to know what are methods, instance variables firstly. When we learn to write them, we will automatically understand the Flutter in-built Widgets. Moreover, their behaviours.

Let us write a simple class so that we can create an object.

In Dart and Flutter, the top-level function main() is the entry point.

Therefore, we need to create any object inside the main() function.

However, before that, we should write our first Class.

```dart
class AClass {
String? iAmInstanceVariable = 'I am initialized';
String? anotherInstanceVariable;
void callMeMethod() {
    print('Call me method.');
}
}
main() {
/// Every object is an instance of a class
///
var anObject = AClass();
/// Objects have members consisting of functions and data
/// we will call them methods and instance variables, respectively
///
anObject.iAmInstanceVariable =
    'Although I was initialized, I am changing value.';

/// we can assign any object to a variable for later use
///
var anInstanceVariable = anObject;

/// now this variable can access any Object members
///
anInstanceVariable.anotherInstanceVariable = 'I am no longer NULL';
anInstanceVariable.callMeMethod();
}
// ##output## : Call me method.
```

If we read the comments inside the above code, we will understand the basic concepts.

Firstly, we create an Object the following way.

```dart
var anObject = AClass();
```

Next, we need to write down how this object should behave. That will define its type.

To determine the object's type, we use variables and functions.

When the variable and functions are used in a class, they are known as instance variable, and methods respectively.

As a result, in the following class, we have two instance variables, and one method.

```
class AClass {
String? iAmInstanceVariable = 'I am initialized';
String? anotherInstanceVariable;
void callMeMethod() {
    print('Call me method.');
}
}
```

In the above code, we have declared one class. Between two instance variables,one is initialised. And the other is not.

As the Dart practices NULL-SAFETY, we have used the “?” mark for the uninitialised instance variable.

```
String? anotherInstanceVariable;
```

How do you declare a class in dart?

We can declare another class where the method passes the instance variables.

Previously, we have learned how to work with the function. What kind of parameters are there.

To make things simple, we have used positional parameters. We have not used the optional positional parameter, nor we have used named parameter.

```
class AClassWithMethodThatPassesInstanceVariable {
String iAmInstanceVariable = 'I am initialized';
String? anotherInstanceVariable;
void sendMessage(String to, String from) {
    iAmInstanceVariable = to;
    anotherInstanceVariable = from;
}
}
```

Consequently, we can pass the method parameters in main() function. Certainly, before that we have to create an object.

```
main() {
/// Every object is an instance of a class
///
var anotherObject = AClassWithMethodThatPassesInstanceVariable();
anotherObject.sendMessage('John', 'Json');
print('${anotherObject.iAmInstanceVariable} sends '
    'message to ${anotherObject.anotherInstanceVariable}');
}
//#output: John sends message to Json
```

In Flutter we will find a lot of Class Constructors. Some APIs, especially the Flutter Widget Constructors — use only named parameters. And some of them are mandatory. That means while declaring the Class Constructor the keyword “required” has been used.

Let us see one simple example of Class Constructor.

```
class AClassWithConstructor {
String iAmInstanceVariable = 'I am initialized';
AClassWithConstructor(this.iAmInstanceVariable);
}
main() {
/// Every object is an instance of a class
///

var aConstructorObject = AClassWithConstructor('Pass a value');
print(aConstructorObject.iAmInstanceVariable);
}
// #output: Pass a value
```

After that, we can create a Class with named parameter.

Later, in Flutter, we will see a lot of named parameters. Every Widget uses them. However, the basic concepts are simple and follow the same design principle.

At the same time, we have made it “mandatory” by using the “required” keyword.

```dart
class AClassWithNamedconstructor {
String iAmInstanceVariable = 'I am initialized';
String? anotherInstanceVariable;
AClassWithNamedconstructor(this.iAmInstanceVariable);
AClassWithNamedconstructor.aNamedConstructor({
    required this.iAmInstanceVariable,
    this.anotherInstanceVariable,
});
}
main() {
/// Every object is an instance of a class
///

var aNamedConstructorObject = AClassWithNamedconstructor.aNamedConstructor(
    iAmInstanceVariable: 'I am a required parameter');

print(aNamedConstructorObject.iAmInstanceVariable);
}
// #output: I am a required parameter
```

These are the basic concepts that we need to understand about an Object and its Class Structure.

But this is an introduction only. The next thing that we need to know is how two Objects interact. The relationship between more than one Objects.

So stay tuned.

How to use Dart and Flutter Class

The Class is the Type of an Object. As a result, when we check the Type of an Object, we get the name of the Class.

We have already discussed the basics of Class and Objects. Now we know why we need a Class. And how to use Dart and Flutter Class. We have also learned how a class defines the “Type” of an Object. What are instance variables and methods.

However, we need a more concrete example. So that we can understand how Classes, or Widgets work in Flutter.

Therefore, firstly, we will define a Class in Dart. After that we will try to do the same thing in Flutter.

As a result, we will learn one key component of Dart and Flutter.

A class can use another Object as its instance variable. Not only that, besides its own methods, it can use methods of another class.

As an example, let us create a Class named Text in Dart.

```
class Text {
String? text;
TextStyle? style;
Text(this.text, {this.style});
}
```

We see that the Text class has two instance variables. One is a String. Which is not initialised.

As the second instance variable we have used an Object. The object “style” has a “Type”, or a blue print which has been defined in the Class – TextStyle.

In the Text class constructor we have passed one named parameters. It is not mandatory.

Why?

Because we have not used the “required” keyword.

In first place, we need to define the TextStyle Class also.

```
class TextStyle {
double? fontSize;
TextStyle({this.fontSize});
}
```

The TextStyle class is quite simple. It has one instance variable which is of the built-in type double.

In its class constructor we have passed that instance variable as a named parameter. Although it is not mandatory. Because we have not used the “required” keyword.

In the top-level main() function, we will first create a “size” object. We can set the value by mentioning it in the constructor.

Meanwhile, we can also create a Text object with the TextStyle now. It becomes much easier.

```
main() {
TextStyle size = TextStyle(fontSize: 30.0);
Text iAmTextWithStyle = Text(
    'I am Text',
    style: size,
);

print('${iAmTextWithStyle.text} with size ${size.fontSize}');
}
/**
// output
I am Text with size 30
*/
```

Moreover, we can check the “Type” of the newly created Text Object.

```
print('The Type of the Text Object is: ${iAmTextWithStyle.runtimeType}');
// #output: The Type of the Text Object is: Text
```

To get an object's type at runtime, we can use the Object property "runtimeType", which returns a Type object.

Keeping this Dart code in our mind, let us create a fresh Flutter App. Since Flutter has a Text Widget, we will see if we can get any similarity with our Dart code.

```
flutter create first_app
```

In fact, we will find the similar coding style in our Flutter Widgets.

However, in the following code, Text is a Widget which has a String passed as its Constructor parameter. And there are several other named parameters. The "fontSize" is one of them.

Remember, they are in-built. The Flutter-Framework ships with them. As a result, we do not have to write them as we had done in our Dart code.

In our Dart code, we have followed the same structure. Although that was purely for a better understanding.

```
child: const Text(
        'Dart and Flutter Demonstration',
        style: TextStyle(
            fontSize: 30.0,
            color: Colors.black45,
        ),
        ),
```

Just run the Flutter App.

Let us see the full code snippet. The Flutter-Framework uses its own Text and TextStyle Classes, or Widgets. Therefore, we do not have to worry about them.

But, to get an idea, we followed the same principle in our Dart code.

```
import 'package:flutter/material.dart';
void main() {
runApp(const MyApp());
}
class MyApp extends StatelessWidget {
const MyApp({Key? key}) : super(key: key);
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: MyHomePage(),
    );
}
}
class MyHomePage extends StatelessWidget {
const MyHomePage({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Dart and Flutter Class'),
    ),
    body: Container(
        margin: const EdgeInsets.all(30.0),
        padding: const EdgeInsets.all(30.0),
        child: const Text(
        'Dart and Flutter Demonstration',
        style: TextStyle(
            fontSize: 30.0,
            color: Colors.black45,
        ),
        ),
    ),
    );
}
}
```

We hope you get an idea about how the Dart classes and objects work in Flutter.

Flutter Structure for Beginner

Concepts of Dart Class influence the Flutter Structure. How it works? Let us learn.

In our previous section we have learned how we can create a Dart class and object. We have also learned that Class defines the “Type” of the Object. As a result, in Flutter, every Widget, or Class has its own “Type”. As a result, the Flutter Structure builds up adding one “Type” of Widget with another “Type” of Widget.

The Flutter structure starts with one Widget. After that, like a tree it starts adding up different branch of Widgets below. For that reason, we call it the Widget tree.

To understand properly, we will first see a simple Flutter App.

Although on the screen it is difficult to imagine how the Widgets are placed one below the other.

The Above Widget tree starts with the Scaffold Widget.

Under the Scaffold Widget we get two Widgets. On the left side we have the AppBar Widget. And on the right side, we have Column Widget. We could have placed the Column Widget under the Container Widget as we see in the diagram below.

If we compare the above structure with the Flutter App shown on the screen, we will find the similarity.

Each Widget is a separate class. However they are connected with each other.

How?

A Widget, or Class Constructor uses the named parameters to return several other Widgets.

As a result, they build the Flutter Structure together. Connecting with one another. To build the User Interface, the Widgets talk to each other. Passing messages between relentlessly.

In bare eyes we cannot see the mechanism that the Flutter-Framework manages for us.

To understand this mechanism, we can take a look at the full code snippet. The Widget tree starts in the entry point. The top-level function main() runs the Flutter App.

After that, the Widget tree starts building up.

```
1  import 'package:flutter/material.dart';
2
3  void main() {
4  runApp(const MyApp());
5  }
6
7  class MyApp extends StatelessWidget {
8  const MyApp({Key? key}) : super(key: key);
9
10 // This widget is the root of your application.
11 @override
12 Widget build(BuildContext context) {
13     return MaterialApp(
```

```
14      title: 'Flutter Demo',
15      theme: ThemeData(
16          primarySwatch: Colors.blue,
17      ),
18      home: const MyHomePage(),
19      );
20  }
21  }
22
23  class MyHomePage extends StatelessWidget {
24  const MyHomePage({Key? key}) : super(key: key);
25
26  @override
27  Widget build(BuildContext context) {
28      return Scaffold(
29      appBar: AppBar(
30          title: const Text('First Flutter App'),
31      ),
32      body: Column(
33          mainAxisAlignment: MainAxisAlignment.center,
34          children: [
35          Row(
36              mainAxisAlignment: MainAxisAlignment.center,
37              children: [
38              Container(
39                  margin: const EdgeInsets.all(20),
40                  padding: const EdgeInsets.all(20),
41                  child: Text(
42                  'First Row',
43                  style: Theme.of(context).textTheme.headline4,
44                  ),
45              ),
46              Container(
47                  margin: const EdgeInsets.all(20),
48                  padding: const EdgeInsets.all(20),
49                  child: Text(
50                  'Second Row',
51                  style: Theme.of(context).textTheme.headline4,
52                  ),
53              ),
54              ],
55          ),
56          Text(
```

```
57              'It\'s a Text Widget. Below an Image Widget.',
58              style: Theme.of(context).textTheme.headline5,
59          ),
60          Container(
61              margin: const EdgeInsets.all(20),
62              padding: const EdgeInsets.all(20),
63              width: 250,
64              height: 250,
65              child: Image.network(
66              'https://cdn.pixabay.com/photo/2021/11/13/23/06/tree-6792528_960_720.jpg\
67 ',
68              ),
69          ),
70          ],
71      ),
72      );
73  }
74  }
```

In the above code, we can clearly see the Flutter structure.

The body section of our Flutter App starts with the Scaffold. Then the Widget tree stretches below. The Column Widget has placed three Widgets vertically.

One of them is Row Widget which has again two Widgets as its branches.

How to test the Type of the Widget?

As we have discussed earlier, every Widget or Class defines the “Type”. Not only that, as the named parameters in its Constructor we pass another Widget. Each Widget represents different Type.

We can easily check the Type of any Widget.

Let us initialize two variables inside the build() method first.

Every variable in Dart refers to an object which is an instance of a Class.

Therefore we can usually use constructors to initialize variables.

```
class MyHomePage extends StatelessWidget {
const MyHomePage({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    Text text = const Text('This is Text Widget');
    Scaffold scaffold = const Scaffold();
...
```

After that, we can wrap one Text Widget with GestureDetector Widget. Subsequently, the GestureDetector Widget helps us to click the Widget.

Meanwhile we can get an output on our console.

```
GestureDetector(
            child: Text(
            'It\'s a Text Widget. Below an Image Widget.',
            style: Theme.of(context).textTheme.headline5,
            ),
            onTap: () {
            print(text.runtimeType);
            print(scaffold.runtimeType);
            },
        ),
```

When we click the Text, in our console, we get the following output.

Eventually, that tells us the Type of the Widgets.

```
Text
Scaffold
```

This simple test clearly shows each Widget represents different Type.

As a beginner we might wonder how a Widget, or Class extends other Class, or Widget.

Just like the above code, where the “MyApp” Widget extends the Stateless Widget.

It happens because Dart allows a Class to extend its capacity by inheriting from other Classes. But, a Class can extend only one Class.

Certainly we can overcome this limitation.

In the next section, we will discuss how we can extend more than one Classes in Dart.

So far, we have understood one key concept. Flutter has many in-built classes and objects that interacting with each other and build application. This is a complex process and it runs on Dart platform.

We have also seen that in any Dart code, 'package' or libraries play the key role. In a Flutter project, everything runs through the 'main.dart' file, or through the main() function.

Now it is impossible to write hundreds of classes and creating thousands of objects inside one 'main.dart' file. The key concept of object-oriented programming is modularity. Breaking the application into several different parts will enhance the ability to organize our code.

Therefore, we need to understand the key concepts of class and objects.

More In-Depth Introduction to Class and Objects

As we know, Dart is an object-oriented language with classes and objects. Every object is an instance of a class and all classes descend from Object.

For absolute beginners, it is a conflicting statement. Class is behind every object. And it says, behind classes there are objects. What does it actually mean? It means a class could have many classes inside. It can inherit properties and methods of other classes, when that class has an instance or object, we can say that the object is behind many classes.

In Dart, there is a concept called 'Mixin-based' inheritance. It means every class has exactly one superclass, and a class body can be reused in multiple hierarchies. This concept is a little bit advanced for absolute beginners. So we will cover 'Mixin-based' inheritance at the end of this book.

To begin with, we start with a simple class and an object. So far we have seen variables and functions. We have seen how we can pass variables as parameters.

Let us think about something that will hold variables and functions inside. We call it a class. You should think this way: an object has states, like a person object has 'colorOfEyes'. It is a state. A state can always be changed. Therefore, an object can also change its state. So, we can conclude that an object has some states and it can also change its states. The way or method that are used in changing object-states are called 'methods'. You can also call it 'function'. Many languages use the keyword function, and some other use the term method. Extending our person object, which has state like 'colorOfEyes', we can say that the person object has one method called 'wearContactLens()'. Now, using that method, the person object can change its state 'colorOfEyes' .

A class always acts like a blueprint of an object. What will be the states, and how those states can be changed, all are predefined in a class.

In the next section we will see a long list of code, which will create mainly two objects; a person from a Person class, and a robot object from a Robot class. If you do not understand it at the first glance, do not worry. It takes time.

How two objects interact

Before starting to learn how two objects interact, we must remember that Dart is an object-oriented programming language. It means, in Dart, everything is an object.

Behind every object, there is a blueprint or class that decides how an object will change its states using different methods.

We will discuss about objects in more detail, as we progress. For the sake of simplicity, you only know that we are all objects. Actually, any object-oriented language simulates the real world.

Here, in the coming application mobile game, we assume that one person object owns one robot object. The person object and the robot object have names, and they can do some actions. An object ideally should have a state and it can also change its state by using methods.

Consider a simple example. Besides having names, the robot object has state like 'number of bullets' that it can fire at something. We can change the state of the number of bullets by using a method called 'canFire(number of bullets)', passing the number of bullets as its parameter.

When a person owns a robot, he or she can use it to fire at some other person objects. And the person object that has been fired at, can strike back to the person object who fires at him.

By now, you understand that we are talking about four different objects. They are interacting with one another. In any application, be it a software, or a mobile application, or a web application, these interactions between various objects keep going on.

Our, programmer's job is to design the application most efficiently.

Okay, enough talking, let us watch some code now. Here is the first code snippet, where we have written two classes, Person and Robot. We have also created two objects, belonging to each of them.

In the bottom section, we have commented out some actions that we are going to write in the next code snippet.

```
1  // code 2.1
2  class Person{
3  String name;
4  Person(this.name);
5  }
6
7  class Robot{
8  String name;
9  Robot(this.name);
10 }
11
12 // ignore: always_declare_return_types
13 main(List<String> arguments) {
14 var personOne = Person('John');
15 var robotOne = Robot('ROBO_COP');
16 print('The first person object has a name : ${personOne.name}');
17 print('The first robot object has a name : ${robotOne.name}');
18
```

```
// robot.canFire(any number of bullets)
// person.getsARobot(robotOne)
// personOne.robotOne.fireAt(personTwo)
// personTwo.getsARobot(robotTwo)
// personTwo.robotOne.fireAt(personOne)
}

//output
The first person object has a name : John
The first robot object has a name : ROBO_COP
```

So, we have successfully created two objects, belonging to Person and Robot. Now the person must have this robot, which is called ROBO_COP.

A Person class is written like this:

```
class Person{
String name;
Person(this.name);
}
```

Inside the Person class, we have two things – one is a state (name), which is represented by a data type String. Therefore, it will be text. Next we have a special method called constructor. The name of the constructor should always be name of the class. Here the name of constructor is Person(). There could be many constructors, we will see to that later.

In this case, we have one constructor that passes a parameter 'this.name'. It means, through that special method or function called constructor we can change the state of the 'person' object. Now we are able to create different person objects with different names. A person can now own a Robot object. We have followed the same procedure for the Robot class.

By the way, the person constructor can also be written like this also.

```
class Person{
String name;

Person(String name){
    this.name = name;
}
}
```

We have commented out the parts of our code in the below, which we are going to implement in the future course of our program.

After owning this robot he can do some actions using that. Therefore, in our next code snippet we have added a few lines.

```
// code 2.2
class Person{
String name;

Person(String name){
    this.name = name;
}
}

class Robot{
String name;
int numberOfBullets;

Robot(String name){
    this.name = name;
}

int canFire(int numberOfBullets){
    this.numberOfBullets = numberOfBullets;
    return numberOfBullets;
}



}

void main(){
var personOne = Person("John");
var robotOne = Robot("ROBO_COP");
print("The first person object has a name : ${personOne.name}");
print("The first robot object has a name : ${robotOne.name}");

// robot.canFire(number of bullets)
// person.getsARobot(robotOne)
// personOne.robotOne.fireAt(personTwo)
// personTwo.getsARobot(robotTwo)
// personTwo.robotOne.fireAt(personOne)

print("${robotOne.name} can fire ${robotOne.canFire(100)} bullets");
}

// output:

```

```
The first person object has a name : John
The first robot object has a name : ROBO_COP
ROBO_COP can fire 100 bullets
```

So, every robot can fire and fire up to 100 bullets. Now, this person object John can use this robot to fire at someone else. To make that happen, we need to create another Person object and Robot object, but before that John must own this ROBO_COP.

In the next code snippet we will try that, first.

```
// code 2.3
class Person{
String name;
Robot robot;

Person(String name){
    this.name = name;
}

String getsARobot(Robot robot){
    this.robot = robot;
    return robot.name;
}
}

class Robot{
String name;
int numberOfBullets;

Robot(String name){
    this.name = name;
}

int canFire(int numberOfBullets){
    this.numberOfBullets = numberOfBullets;
    return numberOfBullets;
}



}

void main(){
```

```
var personOne = Person("John");
var robotOne = Robot("ROBO_COP");
print("The first person object has a name : ${personOne.name}");
print("The first robot object has a name : ${robotOne.name}");

// robot.canFire(number of bullets)
// person.getsARobot(robotOne)
// personOne.robotOne.fireAt(personTwo)
// personTwo.getsARobot(robotTwo)
// personTwo.robotOne.fireAt(personOne)

print("${robotOne.name} can fire ${robotOne.canFire(100)} bullets");
print("${personOne.name} has a robot called ${personOne.getsARobot(robotOne)}");
}
```

Let us watch the output to check whether this person John has owned this robot object or not.

```
The first person object has a name : John
The first robot object has a name : ROBO_COP
ROBO_COP can fire 100 bullets
John has a robot called ROBO_COP
```

Therefore, John has owned ROBO_COP. Now, he can use this robot to fire at someone. Right? But, to do that we need some more person and robot objects.

The next code snippet shows us the same thing.

```
// code 2.4
class Person{
String name;
Robot robot;

Person(String name){
    this.name = name;
}

String getsARobot(Robot robot){
    this.robot = robot;
    return robot.name;
}
}

class Robot{
```

```
String name;
int numberOfBullets;
Person person;

Robot(String name){
    this.name = name;
}

int canFire(int numberOfBullets){
    this.numberOfBullets = numberOfBullets;
    return numberOfBullets;
}

String fireAt(Person person){
    this.person = person;
    return person.name;
}


}

void main(){
var personOne = Person("John");
var robotOne = Robot("ROBO_COP");
print("The first person object has a name : ${personOne.name}");
print("The first robot object has a name : ${robotOne.name}");
var personTwo = Person("Hicky");
print("The second person object has a name : ${personTwo.name}");

// personOne.robotOne.fireAt(personTwo)
// personTwo.getsARobot(robotTwo)
// personTwo.robotOne.fireAt(personOne)

print("${robotOne.name} can fire ${robotOne.canFire(100)} bullets");
print("${personOne.name} has a robot called ${personOne.getsARobot(robotOne)}");
print("${personOne.name} uses ${personOne.getsARobot(robotOne)} "
    "to fire at ${robotOne.fireAt(personTwo)}");

}
```

Once a new person object Hicky comes into picture, John uses his robot to fire at the new person, Hicky. Here is the output:

```
//output
The first person object has a name : John
The first robot object has a name : ROBO_COP
The second person object has a name : Hicky
ROBO_COP can fire 100 bullets
John has a robot called ROBO_COP
John uses ROBO_COP to fire at Hicky
```

We can also manipulate the number of bullets in the run-time. Let us change the last line of the above code snippet.

```
print("${personOne.name} uses ${personOne.getsARobot(robotOne)} "
    "to fire ${personOne.robot.canFire(50)} bullets at ${robotOne.fireAt(personTwo)}\
");
```

It changes the output, if we re-run the code.

```
//output
The first person object has a name : John
The first robot object has a name : ROBO_COP
The second person object has a name : Hicky
ROBO_COP can fire 100 bullets
John has a robot called ROBO_COP
John uses ROBO_COP to fire 50 bullets at Hicky
```

Now, Hicky should be able to retaliate. Otherwise, how we can make our futuristic mobile application look interesting? Therefore, in the next code snippet, we have managed to solve that problem.

```
// code 2.5
class Person{
String name;
Robot robot;
Person person;

Person(String name){
    this.name = name;
}

String getsARobot(Robot robot){
    this.robot = robot;
    return robot.name;
```

```
14  }
15
16  String strikeBack(Person person){
17      this.person = person;
18      return person.name;
19  }
20  }
21
22  class Robot{
23  String name;
24  int numberOfBullets;
25  Person person;
26
27  Robot(String name){
28      this.name = name;
29  }
30
31  int canFire(int numberOfBullets){
32      this.numberOfBullets = numberOfBullets;
33      return numberOfBullets;
34  }
35
36  String fireAt(Person person){
37      this.person = person;
38      return person.name;
39  }
40
41
42  }
43
44  void main(){
45  var personOne = Person("John");
46  var robotOne = Robot("ROBO_COP");
47  print("The first person object has a name : ${personOne.name}");
48  print("The first robot object has a name : ${robotOne.name}");
49  var personTwo = Person("Hicky");
50  print("The second person object has a name : ${personTwo.name}");
51  var robotTwo = Robot("ROBO_MACHINE");
52  print("The second robot object has a name : ${robotTwo.name}");
53
54  print("${robotOne.name} can fire ${robotOne.canFire(100)} bullets");
55  print("${personOne.name} has a robot called ${personOne.getsARobot(robotOne)}");
56  print("${personOne.name} uses ${personOne.getsARobot(robotOne)} "
```

```
    "to fire ${personOne.robot.canFire(50)} bullets at ${robotOne.fireAt(personTwo)}\
");
print("${personTwo.name} has a robot called ${personTwo.getsARobot(robotTwo)}");
print("${personTwo.name} uses ${personTwo.getsARobot(robotTwo)} "
    "to fire ${personTwo.robot.canFire(100)} bullets at ${robotTwo.fireAt(personOne)}\
}");
print("${personTwo.name} strikes back at ${personTwo.strikeBack(personOne)}");

}
```

Look, in the above code snippet, we don't have any commented out sections anymore, because we have implemented everything that we have wanted to do. Therefore, the output changes as follows:

```
//output
The first person object has a name : John
The first robot object has a name : ROBO_COP
The second person object has a name : Hicky
The second robot object has a name : ROBO_MACHINE
ROBO_COP can fire 100 bullets
John has a robot called ROBO_COP
John uses ROBO_COP to fire 50 bullets at Hicky
Hicky has a robot called ROBO_MACHINE
Hicky uses ROBO_MACHINE to fire 100 bullets at John
Hicky strikes back at John
```

So, we can make this battle more interesting; moreover, we can add more features. However, to do that, we need to design the software, data structures, and algorithm in the most efficient manner.

All we have done is, we have created a few objects, building relationship between them by passing objects through various classes. After that they start interacting with each other. Of course, we can do this job more efficiently. But, to do that, we need to learn the language basics, first. Only after learning that, we can design our first Flutter application more efficiently.

More about classes and objects

At the very beginning, we have seen Person and Robot class. We have also seen how we have created various objects that interacted with each other. Now, in every programming language, it is a customary that when we explain class and object, we give examples of Car class. Some also use Dog and Cat class. Thinking that animal right could be violated, we have restricted ourselves to Car class.

Suppose we have a car class. It has two properties: name, and model number. It has also a method (outside object-oriented paradigm we call it function) called 'isTurnedOn(bool)' we have passed a

'boolean type' argument through that function or method. Consider it as the "action part" of the class 'Car'. When we pass 'boolean value true', the car starts and when we pass 'boolean value false', the car stops.

Now imagine a manufacturer company wants to build many cars that have separate names, model numbers but each one has one method 'isTurnedOn(bool)'. In this scenario, each car is an object or instance of 'Car' class. Consider the code below.

```
//code 2.6
main(List<String> arguments) {
var newCar = new Car();
newCar.carName = "Red Angel";
newCar.carModel = 256;
if(newCar.isTurnedOn(true)){
    print("${newCar.carName} starts. It has model number ${newCar.carModel}");
} else print("${newCar.carName} stops. It has model number ${newCar.carModel}");
}
class Car {
int carModel = 123;
String carName = "Blue Angel";
bool isTurnedOn(bool){
    return false;
}
}
```

It gives us this output:

```
//output
Red Angel stops. It has model number 256
```

Watch the 'Car' class. It has two properties or attributes (or states): 'carName' and 'carModel'. Treat them as variables, but since they are inside a class, we will call them properties, members, or attributes, or states. These values can be changed when we will create an instance. In fact, we have done the same, inside the 'main()' function.

The default values were '123' and 'Blue Angel'. But we have an output where the name changes to 'Red Angel'. And the model has been changed to '256'. We have created an instance or object of the 'Car' class, by simply writing this line:

```
var newCar = new Car();
```

Next , we have defined the name and the model number as:

```
newCar.carName = "Red Angel";
newCar.carModel = 256;
```

The next step is vital, because we have declared the method 'isTurnedOn(bool)' as 'true'.

```
if(newCar.isTurnedOn(true)){
print("${newCar.carName} starts. It has model number ${newCar.carModel}");
} else print("${newCar.carName} stops. It has model number ${newCar.carModel}");
```

Now according to our logic, if the method 'isTurnedOn(bbol)' is 'true', it should start. But in the output, we have seen that it 'stops'.

Why it happens?

It happens because in our 'Car' class, we have already set that value 'false.'

Let us change it to 'true' and see the output again:

```
//code 2.7
main(List<String> arguments) {
var newCar = new Car();
newCar.carName = "Red Angel";
newCar.carModel = 256;
if(newCar.isTurnedOn(true)){
    print("${newCar.carName} starts. It has model number ${newCar.carModel}");
} else print("${newCar.carName} stops. It has model number ${newCar.carModel}");
}
class Car {
int carModel = 123;
String carName = "Blue Angel";
bool isTurnedOn(bool){
    return true;
}
}

//Watch the output again:

//output
Red Angel starts. It has model number 256
```

From this example, we can conclude one thing: a class is a blueprint of an object. An object or an instance of a class is extremely powerful, it is not like simple variables, holding one reference to a spot in the memory where we can only store a value. Through an 'app' object we can run a large complicated application, moreover, we can make a series of complex layers of logic behind an object.

How Flutter and Dart work together

Now we have an introduction to class and object-oriented programming paradigms. Now in the light of new insights we have just learned, we can define Flutter more precisely. Flutter is a tool that builds native cross-platform (here cross-platform means for iOS and Android platforms, both) mobile applications with one programming language Dart and one code-base.

Flutter has its own SDK or Software Development Kit that converts our code to native machine code. SDK also helps us to develop our application more eloquently.

Due to the presence of the SDK Flutter works as a framework or Widget library that produces reusable User Interface or UI that builds different types of blocks, utility functions and packages.

Although Dart is an object-oriented programming language, for Flutter its role is more focused on front-end specific. It means with the help of Dart we will build different types of User Interfaces.

As we progress we will understand this core concepts more and more while building our first Flutter application. Understanding Dart language basic is important as it helps Flutter to build UI as code. Since Flutter is mainly different types of Widget trees, we have to write different types of code in Dart.

The advantage of Flutter is that it provides iOS specific code, as well as Android specific code from a single code-base. When we run a Flutter application on our smartphone, we actually see a bunch of Widgets. From the top to the bottom of the mobile screen Flutter divides its Widgets accordingly.

We may think these Widgets as controls that we create by writing code in Dart. In fact, we do not have to do the low level plumbing each time. Flutter framework and Widget libraries provide us the required assistance.

All we need to do is to memorize common rules of building basic Widgets, and moreover, we need to understand the core Dart language basic like function, class and object-oriented style of programming, different types of parameters and their roles in Flutter Widgets, etc.

As we have said, any Flutter application is a bunch of Widgets, we also mean that what we see on the mobile application is Widget trees. However complex application it appears to be, it is actually a bunch of Widget trees. Since there is no Visual Editor code assistance, there is no drag-and-drop facility. We have to code the entire application. Although it sounds daunting at the beginning, in reality, it is not. Because Flutter SDK has come up with almost every kind of solutions, we just need to add those functionalities.

We will find different types of buttons, text boxes, text decorations available. The Flutter API comes up with two distinct facilities; one is Utility functions, and the other is Widget libraries. They are written in Dart, and Dart complies every code we write with the help of SDK, and finally we get the native code for iOS and Android. The iOS platform has different types of buttons, so the Android. Flutter tackles this problem in its own way, it has a custom implementation. Every pixel is drawn on the mobile screen. For that reason, the platform specific limitations are tackled without any hitch.

If we think of a minimal Flutter app, it calls the runApp()function inside the main() function. We could have called the runApp() function with a Widget. Instead we passed a parameter MyFirstApp(), which is a class that extends 'StatelessWidget' class (code 1.22).

Now we are going to change the code 1.22 to get an idea of how Flutter can run minimally based only on runApp() function.

```
// code 2.8
import 'package:flutter/material.dart';

void main() {
runApp(
    Center(
    child: Text(
        'My First Flutter App is running!',
        textDirection: TextDirection.ltr,
    ),
    ),
);
}
```

We have run the Flutter default Center() class constructor inside the runApp() function. It automatically changes the look of the virtual device (Figure 2.1).

Of course, this is not the way one should build a Flutter app. We will also do not take this way. However, getting an idea of how a Flutter app runs will not hurt the learning process.

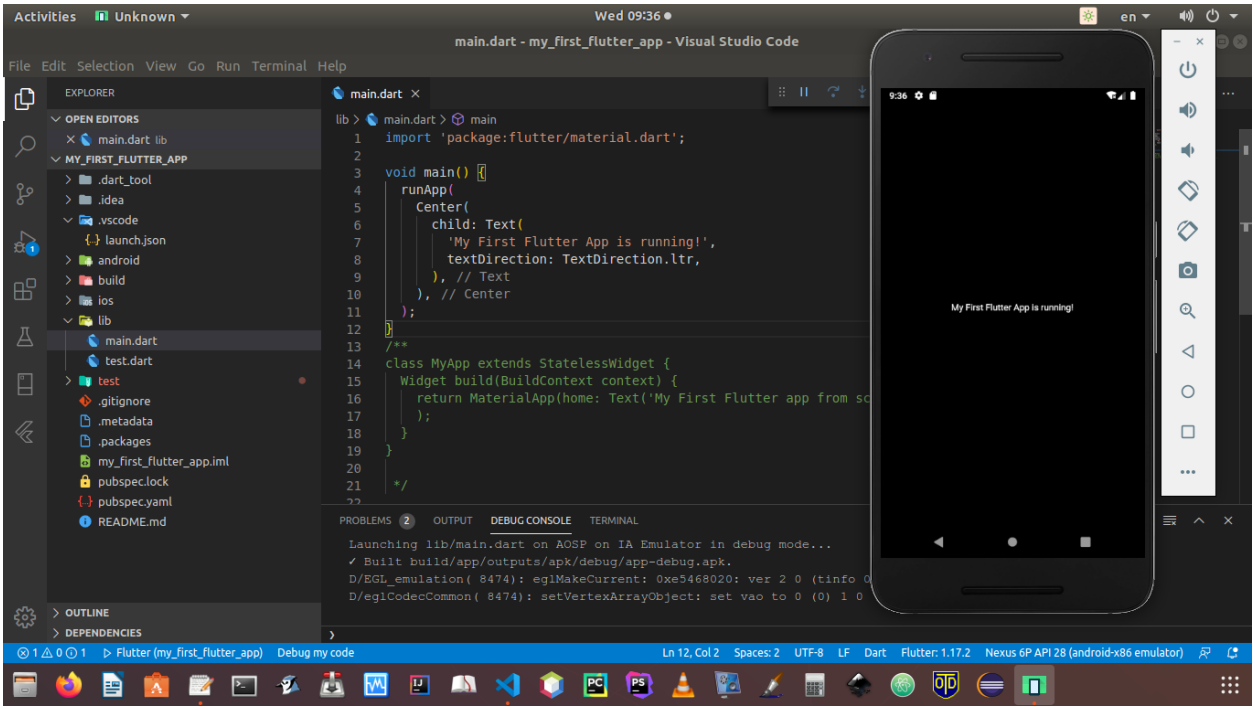

Figure 2.1 – A minimal version of Flutter App

The above image gives us another idea of writing our code, which is preferable especially for macOS and Linux users.

We can run the virtual device from Android Studio, and after that we can open the same Flutter project using Visual Studio IDE. The virtual device will automatically synchronize with Visual Studio (VS); the advantage of using VS IDE is it gives you chance to 'hot reload' property. Each time we change the code, we will just click the 'hot reload' button that hangs loosely over the VS IDE. We can immediately see the change on the connected Virtual Device.

Now, we will get back to our old code snippet that gave us an ugly looking text output (Figure 1.9).

We will start building our code from the following code that we had seen in code snippets 1.22: import 'package:flutter/material.dart';

```
void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
Widget build(BuildContext context) {
    return MaterialApp(home: Text('My First Flutter app from scratch...'),
    );
}
}
```

Let us try to understand how our first Flutter Application 'MyFirstApp' gives the text output. First of all, it is a generic class that extends another Widget class StatelessWidget(). The 'material.dart' file has defined all these in-built classes.

Inside our 'MyFirstApp' class we have called a Widget class method build() that passes an object 'context' that is an instance of class 'BuildContext'.

As we have said earlier, in Dart, everything is Widget. For that reason, inside the build() method of Widget class we have returned another Widget 'MaterialApp()', which draws everything from the 'material.dart' file.

We need to study this part of code more closely to know a few key concepts of Dart language. A function sometimes passes positional argument or parameter; and, sometimes a function passes named argument. Look at this line of code:

```
Widget build(BuildContext context)
```

Here, the 'context' object is a positional argument. But the 'MaterialApp()' Widget passes named argument, like this:

```
return MaterialApp(home: Text('My First Flutter app from scratch...'),
```

We need to understand this key concept, first. After that we will again come back to our Flutter project and keeps on building our first platform independent mobile application using Flutter.

Positional and Named argument

Whether in a class method or in a function, sometimes you need to pass values. We call them arguments or parameters, whichever you like.

Dart is so flexible, it gives ample opportunity to the developers to manipulate the parameters. We may use the default parameters, in such cases, you need to pass the parameters. That is compulsory. But there are two other options available in Dart. You can use 'positional parameter' or 'named parameter'.

Let us see that in a code where we have used default and positional parameters:

```
//code 2.9
//default parameters
String defaultParameters(String name, String address, {int age = 10}){
return "$name and $address and age $age";
}
//optional parameters
String optionalParameters(String name, String address, [int age] ){
return "$name and $address and $age";
}
void main(){
print(defaultParameters("John", "Jericho"));
print(optionalParameters("John", "Form Chikago"));
// overriding the default age
print(defaultParameters("JOhn", "Jericho", age : 20));
}
```

Inside the main() function, in our default parameter function, we have passed only two values: name and address. We did not pass the 'age'. We did not have to, because it had already been defined in our function: {int age = 10}. Remember to use the curly brace to define the default parameter.

Can we override the default parameter? Yes, we can. See this part inside the main() function:

```
// overriding the default age
print(defaultParameters("JOhn", "Jericho", age : 20));
```

We have overridden the default age and made it from 10 to 20.

Next, in the optional parameter function, we have made the age optional by keeping the value inside the second bracket opened and closed.

```
//optional parameters
String optionalParameters(String name, String address, [int age] ){
return "$name and $address and $age";
}
```

Since the parameter 'age' is optional, we can either pass it or can ignore it. However, ignoring the optional parameter will return 'null'. So the output of the above code will be like this:

```
//output of code 3.22
John and Jericho and age 10
John and Form Chikago and null
JOhn and Jericho and age 20
```

In the case of the 'named parameter', we can swap the value and it has got a very high flexibility. Here sequence does not matter. Let us consider this code:

```
//code 2.10
//named parameter
int findTheVolume(int length, {int height, int breadth}){
return length * height * breadth;
}
void main(){
//sequence does not matter
var  result1 = findTheVolume(10, height: 20, breadth: 30);
var  result2 = findTheVolume(10, breadth: 30, height: 10);
print(result1);
print(result2);
}
```

In the above code, we have placed 'height and breadth' inside curly braces. So they are named parameters that we can interchange while passing the values. Interchanging the value will not affect our code. In the case of named parameters or arguments, position does not matter anymore. That is the advantage of named parameters.

Now, again we will go back to our Flutter project and watch the code 1.22, the very first code that we have written to run our first Flutter application. It is a positional argument:

```
Widget build(BuildContext context)
```

And this is a named argument:

```
return MaterialApp(home: Text('My First Flutter app from scratch...'),
```

Flutter SDK and Widget libraries have done all the heavy lifting for us. However, we should be aware of the very basic programming paradigms, otherwise, the Flutter code will not appear meaningful to us.

In the next chapter, we will try to learn a few basic rules regarding the Dart programming language. We should have a clear knowledge about how variables work, what are data types, etc. Moreover, we should know about the programming logic and basic idea about data structures and algorithm by learning the control flow.

After learning the language basic of Dart, we will again come back to our Flutter project. That will help us to understand the Flutter app logic.

For more Flutter related Articles and Resources[7]

[7] https://sanjibsinha.com

3. Dart Language Basic and its implementation in Flutter

In this chapter, we will discuss some initial key concepts of Dart that are absolutely necessary for the beginners.

First of all, like C++, or Java, Dart is also an object-oriented programming language. Everything is an object here. It means a lot to the modern day programming paradigm. Every object in Dart has a class behind it. All the objects inherit from the "Object" class.

Consider a whole number like 2. In nature, it is a positive integer. In Dart all integers are objects. Even functions and null are also objects. I know, the term "object" may fill a beginner with bewilderment. We have already discussed object-oriented programming a little bit. Besides that, we should know what are variables, what are constants and what is function in more detail. Otherwise, we could not follow the Flutter App logic in coming chapters.

Like other programming languages, Dart has also several types, such as integers, strings, boolean, etc. Although Dart is strongly typed language, it also allows you to be duck typed.

What is that? In normal circumstances, in Dart, we mention what type we are going to use. If we use integers and strings, we write it like this:

```
1  //code 3.1
2  int myAge = 12;
3  String myName = "John Smith";
```

In the above examples, we have explicitly declared the type that would be inferred. In the next example we do the same thing, but implicitly.

Therefore, you can also write the same code this way:

```
1  //code 3.2
2  var myAge = 12;
3  var myName = "John Smith";
```

Variables Store References

Explicit or implicit, we have actually created two variables and initialized them with values. Variables store references to objects. In other words, you may say, a variable is a spot on the

memory or a container that contains some references to some values. Since the name is “variable”, the reference can change.

Now, the question is: with the change of reference, does the type also change? Please read on.

In the above code snippets, variable ‘myAge’ store a value 12 and reference it to an integer object. The same way, ‘myAge’ variable store a value ‘John Smith’ and reference it to a ‘String’ object. The type of the ‘myName’ variable is inferred to string-specific but you can change it. If you don’t want a specific or restricted type, specify ‘Object’ or ‘dynamic’ type.

```
dynamic myName = “John Smith”;
```

If you don’t initialize a variable, the default value is set to be ‘null’. Let us consider the following code: int myNumber;

Although it is an integer, it is not initialized. Therefore the default value is NULL. Let us run the code and watch the output.

```
//code 3.3
main() {
print("Hello World!");
int myNumber;
print(myNumber);
}
```

The output is as expected:

```
Hello World!
null
```

Before the discussion of ‘const’ and ‘final’ let us know what are the ‘Built-in Types’ in Dart. So far we have seen some of the types, such as number and string. We have not seen the others.

Built-in Types in Dart

The Dart language has special support for the following types and you can always follow the strongly typed or duck typed pattern to initialize them:

```
1. numbers
2. strings
3. boolean
4. lists (also known as arrays)
5. sets
6. maps
7. runes (for expressing Unicode characters in a string)
8. symbols
```

You can initialize an object of any of these special types using a literal. For example, 'Hello John Smith' is a string literal, and "false" is a boolean literal. Consider this code:

```
//code 3.4
main() {
String saySomething = "Hello John Smith";
var isFalse = true;
if(saySomething == null){
    print("It is ${isFalse}");
}else print("It is not ${isFalse}");
}
```

Since the string variable is not 'null', the output should be:

```
It is not true
```

Since the string variable is not 'null', the output came out as 'not true'.

We will encounter the first four built-in types most often. We will find the usages of other built-in types also as situation demands.

Suppose, you don't like Variables

Well, in some cases, you need the value to be constant. There are two ways that you can follow where you never intend to change the value of a variable. You may use 'const' instead of 'var' or 'String, int or bool' type declaration.

You may also use 'final'; but remember, 'final' variable can be set only once. So there is a difference between these two keywords: 'const' and 'final'. We will again come back to this topic when we will discuss object-oriented programming. Because instance variable can be 'final' but not 'const'; unless we start learning object-oriented programming we are in no position to discuss what instance variable is.

Consider this code:

```
//code 3.5
main() {
const firstName = "Sanjib";
final lastName = "Sinha";
String firstName = "John";
String lastName = "Sinha";
}
```

Watch the output full of errors:

```
//output
bin/main.dart:8:10: Error: 'firstName' is already declared in this scope.
String firstName = "John";
        ^^^^^^^^^
bin/main.dart:5:9: Context: Previous declaration of 'firstName'.
const firstName = "Sanjib";
        ^^^^^^^^^
bin/main.dart:9:10: Error: 'lastName' is already declared in this scope.
String lastName = "Sinha";
        ^^^^^^^^
bin/main.dart:6:9: Context: Previous declaration of 'lastName'.
final lastName = "Sinha";
```

When you want a variable to be compile-time constants, use ‘const’; and use ‘final’ for the instance variable that you will never change.

More about built-in types

In a quick review, we will first check the numbers. Then one after another we will learn about string, boolean, and other types.

Dart numbers are of two types: integers and decimals. We write them as ‘int’ and ‘double. Integers are numbers without decimal points. Examples: 1, 2, 22, etc. Doubles do have a decimal point like this: 1.5, 3.723, etc. Both ‘int’ and ‘double’ types are sub-types of ‘num’. The ‘num’ type includes basic Arithmatic operators such as “+, -, /, and *”; and they represent ‘plus, minus. division and multiplication’ signs. We can call them arithmetic operators and it also includes modulo, that is, remainder and the sign is: ‘%’.

Let us see some interesting examples:

```
//code 3.6
main() {
var one = int.parse('1');
print(one);
if(one.isOdd){
    print("It is an odd number.");
} else print("It is an even number.");
}
```

We have converted a string into an integer, or number.

```
//output
1
It is an odd number.
```

We can also turn a string to a double number. Let us change the above code a little bit:

```
//code 3.7
main() {
var one = int.parse('1');
var doubleToString = double.parse('23.564');
print(one);
print(doubleToString);
if(one.isOdd && doubleToString.isFinite){
    print("The first number is an odd number and the second one is a double ${double\
ToString} and a finite number.");
} else print("It is an even number and the second one is not a double ${doubleToStri\
ng} and a non-finite number.");
}
```

The output is quite expected. Both statements are true so the relational operation takes to this output:

```
//output
1
23.564
```

A first number is an odd number and the second one is a double 23.564 and a finite number. We can do the vice versa too. We are going to turn an integer to string.

```
//code 3.8
main() {
int myNUmber = 542;
double myDouble = 3.42;
String numberToString = myNUmber.toString();
String doubleToString = myDouble.toString();
if ((numberToString == '542' && myNUmber.isEven) && (doubleToString == '3.42' && myD\
ouble.isFinite)){
    print("Both have been converted from an even number ${myNUmber} and a finite dou\
ble ${myDouble} to string. ");
} else print("Number and double have not been converted to string.");
}

//the output
Both have been converted from an even number 542 and a finite double 3.42 to string.
```

As we progress, we will find, Dart is extremely flexible language and the syntax are simple to remember with lots of help from the core libraries.

Understanding Strings

A Dart string is a sequence of UTF-16 code units. For absolute beginners, I am going to give a short note on UTF-8, UTF-16, and UTF-32. They all store Unicode but use different bytes. Let us first try to understand the advantages of using UTF-16 code over the other two. Let us know about UTF-8.

Where ASCII characters represent the majority of texts, UTF-8 has an advantage. Like ASCII, UTF-8 also encodes all characters into 8 bits. It is the opposite for UTF-16; where ASCII is not predominant, UTF-16 has an advantage. UTF-16 remains at just 2 bytes for most characters. However, UTF-32 tries to cover all possible characters in 4 bytes, it means, processors have extra load making it pretty bloated.

The Unicode support makes Dart more powerful and you can make your mobile and web applications in any language. Let us see one example where I have tried some Bengali script.

```
//code 3.9
main(List<String> arguments) {
//print("Hello World ${IdeaProjects.calculate()}");
String bengaliString = "□□□□□□ □□□□";
String englisgString = "This is some English text.";
print("Here is some Bengali script - ${bengaliString} and some English script ${engl\
isgString}");
}


//output
Here is some Bengali script - □□□□□□ □□□□ and some English script This is some Englis\
h text.
```

While handling strings, we should remember a few things. We can use both single quote(‘’) and double quote(“”).

```
//code 3.10
main(List<String> arguments) {
String stringWithSingleQuote = 'I\'m a single quote';
String stringWithDoubleQuote = "I'm a double quote.";
print("Using delimiter in single quote - ${stringWithSingleQuote} and using delimite\
r in double quote - ${stringWithDoubleQuote}");
}
```

We can use the delimiter in both cases, but the double quote is more helpful in such cases. Watch the output:

```
//output
Using delimiter in the single quote - I'm a single quote and using delimiter in the \
double quote - I'm a double quote
```

We have put the value of expression inside a string by using our variable in this way: ${stringWith-SingleQuote}.

If you are about to express the variable in normal circumstances, you do not have to use the curly braces {}. You can use the variable this way: print(”$stringWithSingleQuote”); or print($stringWithSingleQuote); String interpolation, concatenation and even making it multi-line is quite easy in Dart. Consider this code:

```
//code 3.11
main(List<String> arguments) {
String stringInterpolation = 'string ' + 'interpolation';
print(stringInterpolation);
String multiLIneString = """
    This is
    a multi line
    string.
""";
print(multiLIneString);
}
```

Watch the output here, we have used a triple quote with either single or double quotation marks:

```
//output
string interpolation
    This is
    a multi line
    string.
```

If you want to store some constant value inside a constant string, the value cannot be variables. Consider this code:

```
//code 3.12
main(List<String> arguments) {
const aConstantInteger = 12;
const aConstantBoolean = true;
const aConstantString = "I am a constant string.";
const aValidConstantString = "this is a constant integer: ${aConstantInteger}, a con\
stant boolean: ${aConstantBoolean}, a constant string: ${aConstantString}";
print("This is a valid constant string and the output is: $aValidConstantString");
}
```

We have created a valid constant string by storing constant value inside them. The output is perfectly OK.

```
//output
This is a valid constant string and the output is: this is a constant integer: 12, a\
 constant boolean: true, a constant string: I am a constant string.
```

It will not work if you want to hold variable data inside a constant string. We have changed the code 2.10 to this:

```
//code 3.13
main(List<String> arguments) {
var aConstantInteger = 12;
var aConstantBoolean = true;
var aConstantString = "I am a constant string.";
const aValidConstantString = "this is a constant integer: ${aConstantInteger}, a con\
stant boolean: ${aConstantBoolean}, a constant string: ${aConstantString}";
print("This is a valid constant string and the output is: $aValidConstantString");
}
```

It does not work, it will give us errors. As we progress, we will learn more about string, because understanding string is very important in the context of making Flutter applications. In the next section, we will try to understand boolean; that also plays a vital role in building algorithms.

To be True or to be False

We have already seen that Dart has a type called 'bool'. The boolean literals 'true' and 'false' have type 'bool'. They are compiled time constants. Consider this code:

```
//code 3.14
main(List<String> arguments) {
bool isTrue = true;
bool isFalse = false;
if(isFalse || isTrue){
    print("It is true.");
} else print("It is false.");
}
```

We have set two boolean literals: true and false; and after that, we try to find out between two boolean literals using 'if control logic'. Between 'true' and 'false', use 'OR' conditional operator, it always chooses the 'true'.

Hence the output is:

```
//output
It is true.
```

What happens if we use 'AND' conditional operator? Let us check the code:

```
//code 3.15
main(List<String> arguments) {
bool isTrue = true;
bool isFalse = false;
if(isFalse && isTrue){
    print("It is true.");
} else print("It is false.");
}
```

Use 'AND', it chooses 'false'.

```
//output
It is false.
```

This is an extremely important concept in computer science because, in our control structure, we always depend on whether a statement is 'true' or 'false'. At the same time,we got a hint of relational operation; which we will discuss in a minute.

Introduction to Collections: Arrays are Lists in Dart

This is the most common collection in every programming language: array or an "ordered group of objects". In Dart, arrays are List objects. We will address them as 'lists' in our future discussion. At the time of building Flutter application, we will use 'list' quite extensively.

Therefore, this is a very key concept. It is also the first step to learn data structures.

JavaScript array literals look like Dart lists. Here is a sample code we may consider to understand why this concept is important:

```
//code 3.15
main(List<String> arguments) {
List fruitCollection = ['Mango', 'Apple', 'Jack fruit'];
print(fruitCollection[0]);
}
```

Consider another piece of code:

```
//code 3.16
main(List<String> arguments) {
List fruitCollection = ['Mango', 'Apple', 'Jack fruit'];
var myIntegers = [1, 2, 3];
print(myIntegers[2]);
print(fruitCollection[0]);
}
```

What is the difference between these two code snippets? In the above code 2.14, we have explicitly mentioned that we are going to declare a collection of fruits. And we can pick any item from that collection from the key. As we know, when in an array key is not mentioned with the value pair, it automatically infers that the key starts from 0.

Therefore, the output of code 2.14 is 'Mango'. In the second instance we do not have any explicit declaration about the 'myInteger' lists. We have written: var myIntegers = [1, 2, 3]; however, Dart infers that list has type List<int>. Let us see the output of the code 2.15:

```
//output
3
Mango
```

If we try to inject non-integer objects to the 'myInteger' list, what happens?

```
//code 3.17
main(List<String> arguments) {
List fruitCollection = ['Mango', 'Apple', 'Jack fruit'];
var myIntegers = [1, 2, 3, 'non-integer object'];
print(myIntegers[3]);
print(fruitCollection[0]);
}
```

It did not raise any error. See the output:

```
//output
non-integer object
Mango
```

Only remember, Dart Lists use zero-based indexing like all other collections we have seen in other programming languages. Just think it as a key⇒value pair, where 0 is the index of the first value or element. As we progress, we will discuss Lists as there are other useful methods around that we will use when we will build our first mobile application. Dart Lists have many handy methods.

Get, Set and Go

In Dart, a Set is an unordered collection of unique items. There are small difference in syntax between List and Set. Let us see an example first to know more about the difference.

```
//code 3.18
main(List<String> arguments) {
var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(fruitCollection.lookup('Apple'));
}

//output
Apple
```

We can search the Set using the lookup() method. If we search something else, it returns 'null'.

```
//code 3.19
main(List<String> arguments) {
var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(fruitCollection.lookup('Something Else'));
}
//output of code 2.19
null
```

Remember one key point regarding Set and Map. When we write:

```
var myInteger = {};
```

It does not create a Set, but a Map. The syntax for map literals is similar to that of for set literals. Why does it happen? Because map literals came first, the literal {} is a default to the Map type. We can prove this by a simple test:

```
//code 3.20
main(List<String> arguments) {
var myInteger = {};
if(myInteger.isEmpty){
    print("It is a map that has no key, value pair.");
} else print("It is a set that has no key, value pair.");
}

//output
It is a map that has no key, value pair.
```

It means the map is empty. If it was a set, we would have got the output in that direction. We will see lots of examples of Sets in the future, while we build our mobile application. At present just remember, in general, a map is an object that associates keys and values. The set has also keys, but that are implicit. In cases of Sets, we call it indexes.

Let us see one example of Map type by map literals. While writing keys and values, it is important to note that each key occurs only once, but you can use the same value many times.

```
//code 3.21
main(List<String> arguments) {
var myProducts = {
    'first' : 'TV',
    'second' : 'Refrigerator',
    'third' : 'Mobile',
    'fourth' : 'Tablet',
    'fifth' : 'Computer'
};
print(myProducts['third']);
}
```

The output is obvious : 'Mobile'. Dart understands that the 'myProducts' has the type Map<String, String>(Map<Key, Value>); we could have made the key integers or number type, instead of a string type. In any Flutter application, the implementation of 'map' takes place very often.

```
//code 3.22
main(List<String> arguments) {
var myProducts = {
    1 : 'TV',
    2 : 'Refrigerator',
    3 : 'Mobile',
    4 : 'Tablet',
    5 : 'Computer'
};
print(myProducts[3]);
}
```

The output is the same as before – mobile. Can we add a Set type collection of value inside a Map? Yes, we can. Consider this code:

```
//code 3.23
main(List<String> arguments) {
Set mySet = {1, 2, 3};
var myProducts = {
    1 : 'TV',
    2 : 'Refrigerator',
    3 : mySet.lookup(2),
    4 : 'Tablet',
    5 : 'Computer'
};
print(myProducts[3]);
}
```

In the above code (3.23) we have injected a collection of Set type and we also have looked up for the defining value through the Map key. Here, inside the Map key, value pair we have added the set element number 2, this way: 3 : mySet.lookup(2), and later we have told our Android Studio editor to display the value of the Map type 'myProducts' . The output is quite expected: 2.

You can create the same products lists by Map constructor. For the beginners, the term "constructor" might seem difficult. We will discuss this term in detail in our object-oriented programming category. Consider this code:

```
//code 3.24
main(List<String> arguments) {
var myProducts = Map();
myProducts['first'] ='TV';
myProducts['second'] ='Mobile';
myProducts['third'] ='Refrigerator';
if(myProducts.containsValue('Mobile')){
    print("Our products' list has ${myProducts['second']}");
}
}

//output
Our products' list has Mobile
```

Since we have had an instance (in code 3.24) of Map class, the seasoned programmer might have expected 'new Map()' instead of only 'Map()'.

As of Dart 2, the new keyword is optional. We will learn about these, in detail, in the coming object-oriented programming chapter.

We will also have a separate "Collections" chapter later, where we will learn more about List, Set and Map.

Operators are Useful

In Dart, when you use operators, you actually create expressions. If you are a seasoned programmer, you may skip this section entirely. If you are completely new, then please go on reading. Here expressions mean such examples: a++, a + b, a * b, a/b, a~/b, a%b etc. There are many types of operators in Dart. Even absolute beginners probably have heard of arithmetic operators. Relational operators are extremely useful for the control structures.

We will have a look at them one after another.

Usual arithmetic operators are - add (+), subtract (-), multiply (*), divide (/), and modulo or remainder (%); a special operator divide, returning an integer is like this: ~/.

Let us see one example:

```
1  //code 3.25
2  main(List<String> arguments) {
3  int aNum = 12;
4  double aDouble = 2.25;
5  var theResult = aNum ~/ aDouble;
6  print(theResult);
7  }
8  //output of code 3.25
9  5
```

Note this special operator has displayed an integer; not a double. However, if we had divided it in a plain fashion, it would have this output:

```
1  //code 3.26
2  main(List<String> arguments) {
3  int aNum = 12;
4  double aDouble = 2.25;
5  var theResult = aNum / aDouble;
6  print(theResult);
7  }
8
9  //output of code 2.26
10 5.333333333333333
```

One key feature of Dart is it supports both prefix and postfix increment and decrement operators. Let us see an example:

```
//code 3.27
main(List<String> arguments) {
int aNum = 12;
aNum++;
++aNum;
int anotherNum = aNum + 1;
print(anotherNum);
}
```

The output is as expected: 15. Prefix and postfix, both work in case of '–' also.

Equality and relational operators

The seasoned programmers know what relational operators actually mean. It is also called equality operators because '==' means equal and other relational operators usually check the equality in various forms. Let us consider some code snippets which would show us many types of relational operators at one glance.

```
//code 3.28
main(List<String> arguments) {
int firstNum = 40;
int secondNum = 41;
if (firstNum != secondNum){
    print("$firstNum is not equal to the $secondNum");
} else print("$firstNum is equal to the $secondNum");
}
//output of code 3.28
40 is not equal to the 41
```

The output is quite expected. Let us change this code a little bit:

```
//code 3.29
main(List<String> arguments) {
int firstNum = 40;
int secondNum = 40;
if (firstNum == secondNum){
    print("$firstNum is equal to the $secondNum");
} else print("$firstNum is not equal to the $secondNum");
}
```

Quite expected, it will give us the first output. Since the condition is true. Two values are equal. Let us add some more logic to our code:

```
//code 3.30
main(List<String> arguments) {
int firstNum = 40;
int secondNum = 40;
int thirdNum = 74;
int fourthNum = 56;
if (firstNum == secondNum || thirdNum == fourthNum){
    print("If choice between 'true' or 'false', the 'true' gets the precedence.");
} else print("If choice between 'true' or 'false', the 'false' gets the precedence."\
);
}

//output of code 3.30
If choice between 'true' or 'false', the 'true' gets the precedence.
```

We have learned a key concept when one value is true and another value is false, if we use 'OR', '||' operator, the 'true' value gets preceded. It is not true for the 'AND', '&&' relational operator. Watch this code:

```
//code 3.31
main(List<String> arguments) {
int firstNum = 40;
int secondNum = 40;
int thirdNum = 74;
int fourthNum = 56;
if (firstNum == secondNum && thirdNum == fourthNum){
    print("If choice between 'true' or 'false', in this case the 'true' gets the pre\
cedence.");
} else print("If choice between 'true' or 'false', in this case the 'false' gets the\
 precedence.");
}

//output of code 3.31
If choice between 'true' or 'false', in this case the 'false' gets the precedence.
```

We have used the '&&' conditional operator and here the 'false' gets preceded. The '!' sign has many roles. Consider this code snippet:

```
//code 3.32
main(List<String> arguments) {
int aNUmber = 35;
if(!(aNUmber != 150) && aNUmber <= 150){
    print("It's true");
} else print("It's false.");
}
```

Can you guess what would be the output? The first statement is false because we have negated a true statement by using '!' sign and the second statement is true, value is less than or equal to 150. Since the logical operator is '&&' or 'AND' here, it will be false. Had we used the '||', 'OR' logical operator, the output would have come out as true.

Just to remember, the '>=' operator means greater than or equal to. It is '>' greater than, it is '<' less than. Just play around your logical or relational operators as because this is one of the main pillars of computer science.

Type test operators

The "as, is, and is!" operators are handy for checking types at runtime. Consider this code:

```
//code 3.33
main(List<String> arguments) {
int myNumber = 13;
bool isTrue = true;
print(myNumber is int);
print(myNumber is! int);
print(myNumber is! bool);
print(myNumber is bool);
}

//output of code 2.33
true
false
true
false
```

Assignment operators

While assigning a value we use '=' operator. What happens when the assigned-to value is null? We use a special type of operator - '??='. Consider this code:

```
//code 3.34
main(List<String> arguments) {
int firstNum = 10;
int secondNum;
if(firstNum == 10) print("The value of ${firstNum} is set.");
if (secondNum == null) print("It is true.");
secondNum ??= firstNum;
print(secondNum);
}

//output of code 2.34
The value of 10 is set.
It is true.
10
```

In the above code 2.34, we have assigned the value of 'firstNum' to 10 and the type is an integer. So we can say,the value of 'firstNum' is set. At the same time, we have not assigned any value to the 'secondNum', so by default, it is null. After that, we have assigned a null value to an integer value by this special operator: '??='.

Almost the same thing happens in the case of compound assignment operators. Now we are going to write the above code in this way:

```
//code 3.35
main(List<String> arguments) {
int firstNum = 10;
int secondNum;
if(firstNum == 10) print("The value of ${firstNum} is set.");
if (secondNum == null) print("It is true.");
secondNum ??= firstNum;
print(secondNum);
print("After using an assignment operator, the value changes.");
secondNum += secondNum;
print(secondNum);
print("After using an assignment operator, the value changes again.");
secondNum -= secondNum;
print(secondNum);
if (secondNum == null) print("It is true.");
else print("it is false, because the 'secondNUm' has the value of ${secondNum} now."\
);
}
```

Watch the output where it is evident that we have changed the value of 'secondNum' consecutively and finally get this output:

```
//output
The value of 10 is set.
It is true.
10
After using an assignment operator, the value changes.
20
After using an assignment operator, the value changes again.
0
it is false, because the 'secondNUm' has the value of 0 now.
```

As we progress, we will find more examples of operators.

Summary of this Part

Numbers, Strings and Boolean, all they are Literals in Dart.

Consider these Literals: 1, 2.3, “Some Strings”, true, false. We need to remember a few things, such as the following:

```
var isValid = true;
1. var□data type
2. isValid□Variable Name (or Spot in the Memory)
3. true□Literal
4. We can mention the data type of the variable as ‘int’, ‘double’, ‘String’ or ‘boo\
l’. If we don’t, we can simply refer to them as ‘var’. In that case, if not mentione\
d, the data type is inferred.
5. String Interpolation is a good practice. Don’t use ‘+’ sign to add two strings.
6. We should not use the expression for a single variable name, like this: ${name}. \
We should write $name, instead.
7. Use expression for operators such as: ${number1 + number2}.
8. What will be your choice? The ‘final’ or ‘const’? It is a difficult choice. You n\
eed to remember a few things: when you choose ‘final’, it is initialized and when it\
 is accessed, the memory is allocated for it. The ‘const’ is implicitly ‘final’; it \
means when while compilation it is initialized, the memory is allocated for it.
```

So far we have learned a few key concepts of Dart language;now, we would like to implement those lessons to our first Flutter application.

Implementing Dart concepts to Flutter

As we have said earlier, Flutter is all about Widgets. Because we have just started to learn Flutter we will concentrate on basic Widgets. We will try to learn them, master them, and after that, we will proceed to the less known facts about Flutter framework.

Widgets are of two types. One is visible, and the other is invisible. In Flutter, Text() class passes a String data, which is visible. We have already passed a String data, although it does not look good, yet it is visible. The same way, we are going to use different types of buttons, those are also visible.

In a few minute we are going to use a Widget, 'RaisedButton()'; that will be also very much visible.

Incidentally, there are Widgets that are not visible, such as Column(), Row(), and many more other Widgets that we will use in the future when we will build our application. With reference to the invisible Widgets, we would like to mind you that these invisible Widgets actually help the visible Widgets to draw every pixel on the mobile screen. Therefore, they depend on each other.

To return to the previous subject of implementing our Dart concepts to our Flutter project, let us start from that point where we had left.

Our first challenge is to change the look of our application. To do that we should think our mobile screen as the 'home'. At that 'home' we should have separate sections, such as the header part, the body part, etc. We have seen in our last code Widget build() method passes one object as an argument; the name of the object is 'context'. At the same time, Flutter returns a class constructor MaterialApp() Widget through that method.

This MaterialApp() Widget is getting its all materials from the flutter package 'material.dart'. There are hundreds of classes extending one another, weaving together and form a synchronized effect on the UI design. Now, it is our duty to code that UI design with the help of many 'named parameters'.

At the very beginning of MaterialApp() Widget, we will call another Widget Scaffold(). As the word 'scaffold' literally means, it is the base platform from where we can execute our other important commands.

First of all, we need a header section where we will display the name of the application we are going to build. There is a named parameter called 'appbar'. It directly points to the another class AppBar(). Through the AppBar() class constructor we can pass another named parameter 'title', which points to the Text() constructor. Our next code snippet will show you how we are going to organize our first Flutter project:

```
// code 3.36
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {

@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
```

```
14          appBar: AppBar(
15          title: Text(
16              'Test Your Personality...',
17              style: TextStyle(),
18              ),
19          backgroundColor: Color(0123,),
20          )
21      )
22      );
23  }
24  }
```

It will change the look of the application considerably (Figure 3.1). Although, that is just a temporary change.

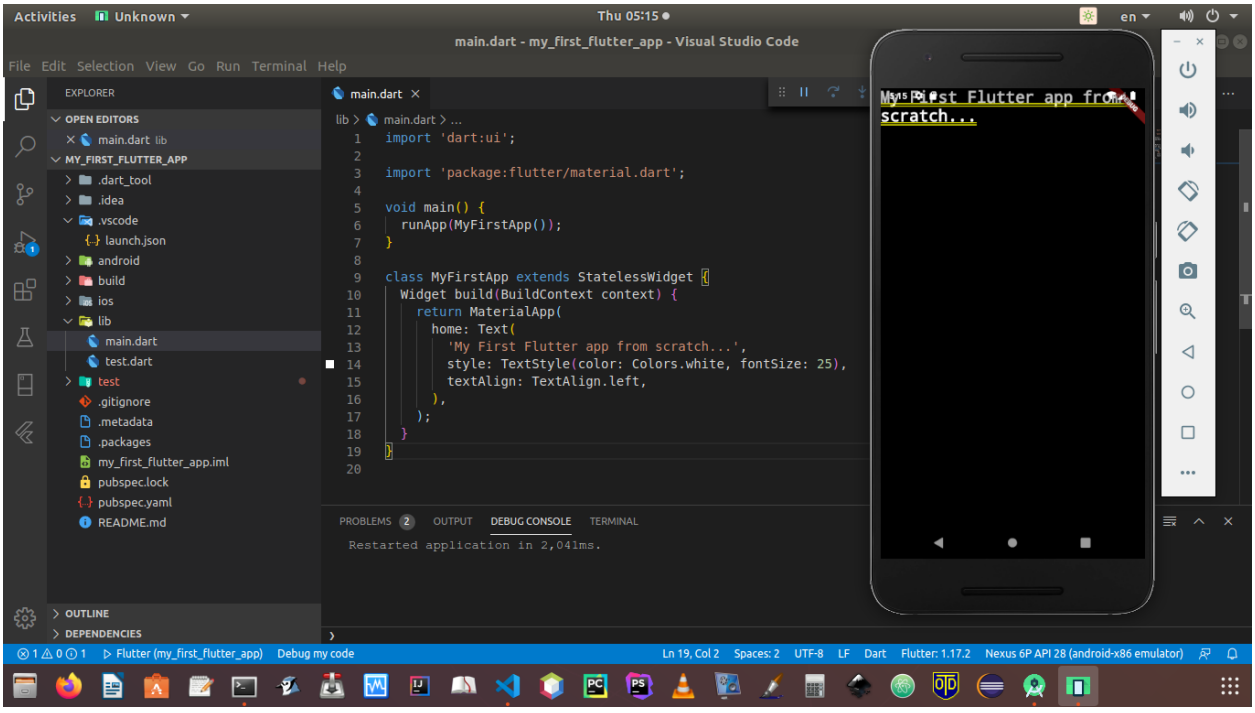

Figure 3.1 – Changing the look of the Flutter application

In any case, we need to change the look considerably better. If we had not used any Text() class constructor, and just left the Scaffold() empty, it would give us a white background (Figure 3.2).

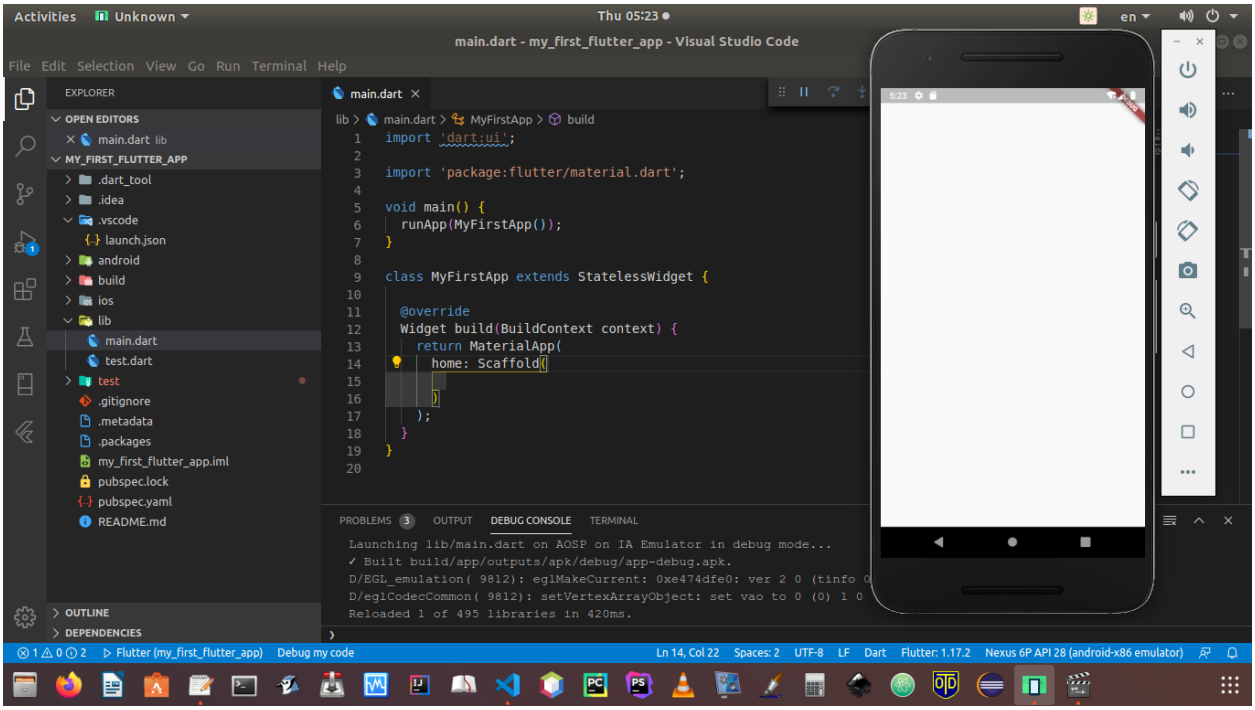

Figure 3.2 – Scaffold() class constructor empty

We could have only passed the Text() class constructor Widget with a message inside the Scaffold() Widget. That would also give us an output, where no styling would have maintained.

The following image (Figure 3.3) gives us an idea.

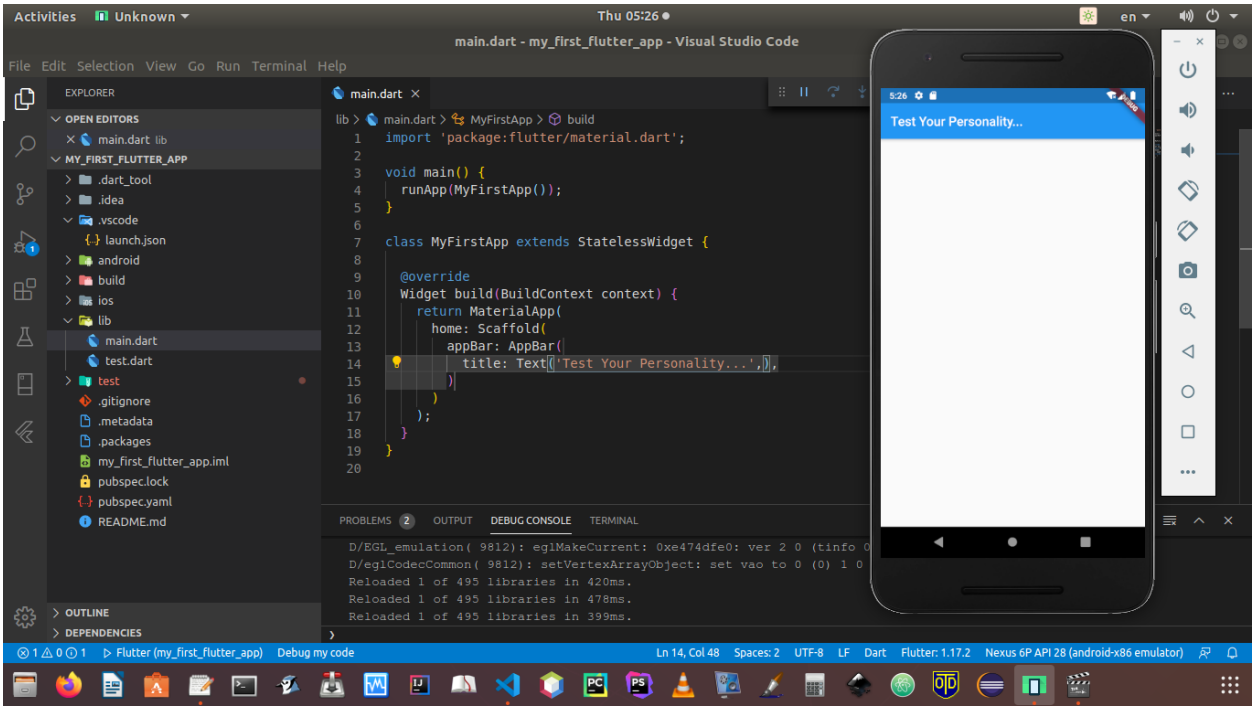

Figure 3.3 – Changing the look of the application

We have started understanding one core feature of Flutter, it mixes many Widgets, and can render a beautiful UI design. Nonetheless, it is needless to say that we need to code that design. To do that, we need to change the previous code snippet, adding many other Widgets.

```
// code 3.38
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: Text(
            'Test Your Personality...',
            style: TextStyle(
            fontSize: 36,
            ),
        ),
```

```
20          backgroundColor: Color(
21              0125,
22          ),
23          ),
24      ),
25      );
26  }
27  }
```

We have changed the background color of the 'appBar' Text, adding more styling. The font size has also been changed.

All together, many different types of Widgets have acted upon collectively. It now consecutively changes the look of the application (Figure 3.4).

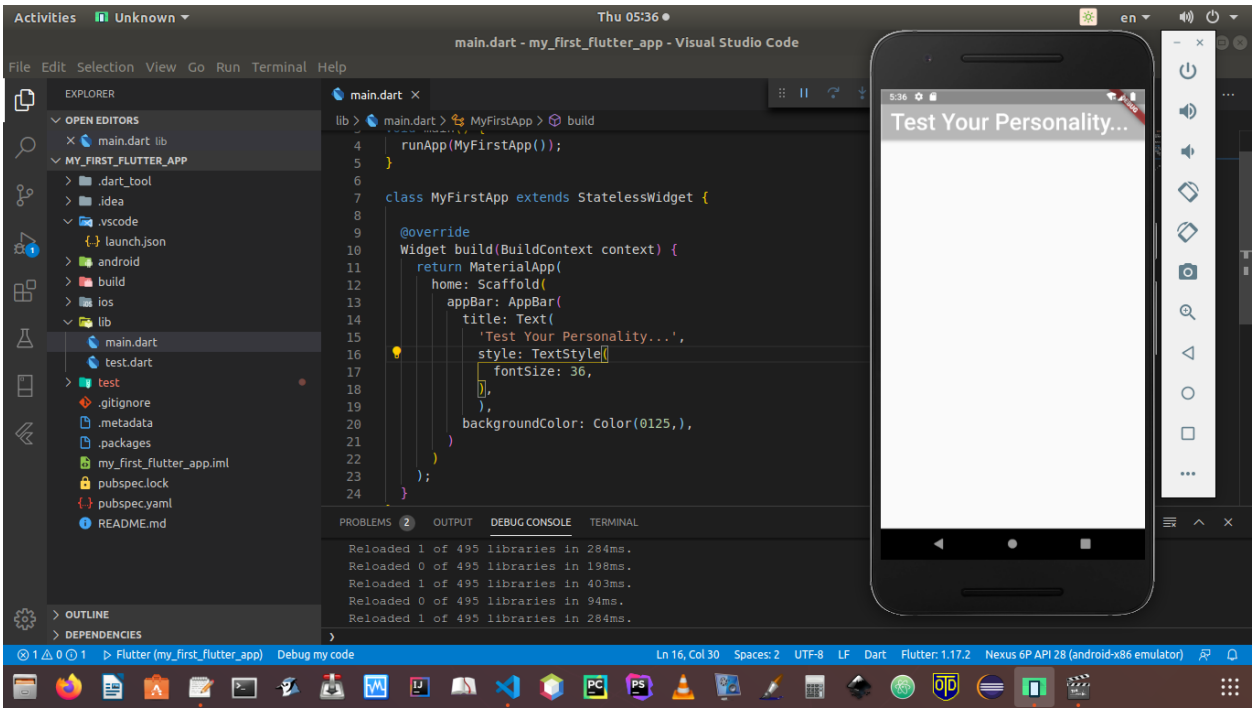

Figure 3.4 – The header text has different background color and font size

Now, we should think about the 'body' part of our application. This is the main part, where we need to do many things. We need to use many Widgets, class constructors, named parameters, etc.

If you are still not feeling quite sure about Dart programming concepts has been implemented, do not worry.

As an absolute beginner, we all have to take time to understand the core features of any programming language. We do not want to code our UI design without understanding what we are doing actually. In the next code snippet, we will introduce a Widget, which is 'list'.

We have already covered 'list' data structure before. If you do not feel very sure about it, please go back to the previous lessons, and try to understand them first.

```
// code 3.39
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: Text(
            'Test Your Personality...',
            style: TextStyle(
            fontSize: 36,
            ),
        ),
        backgroundColor: Color(
            0125,
        ),
        ),
        body: Column(children: <Widget>[],),
    ),
    );
}
}
```

We have added a new line where the named parameter 'body' points to a Column Widget. As we have learned before, Column Widget is invisible, but it helps other visible Widgets to get displayed on the screen.

Watch this line:

```
body: Column(children: <Widget>[],),
```

Column Widget constructor class passes one named parameter 'children', which directly points out to a Widget 'list'. How do we know it represents a 'list' data structure? By the symbol - []. We see the second bracket open and close.

Inside that we can keep a collection of data.

We will discuss more about 'collection' and data structure when times comes. Until then, we will remember that in a collection of data structure, we can keep more than one data.

The next code snippet will clear the picture:

```
// code 3.40
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: Text(
            'Test Your Personality...',
            style: TextStyle(
            fontSize: 36,
            ),
        ),
        backgroundColor: Color(
            0125,
        ),
        ),
        body: Column(
        children: [
            Text(
            'You need to answer a few questions',
            style: TextStyle(
                fontSize: 22,
            ),
            ),
        ],
        ),
    ),
    );
}
}
```

It will change the look of the connected virtual device. Inside the Column Widget, we have passed a list or collection. Now, we are not only able to pass more than one Text() class constructor, but also many more other Widgets.

Before doing that, let us check the different look that Flutter has drawn on the screen.

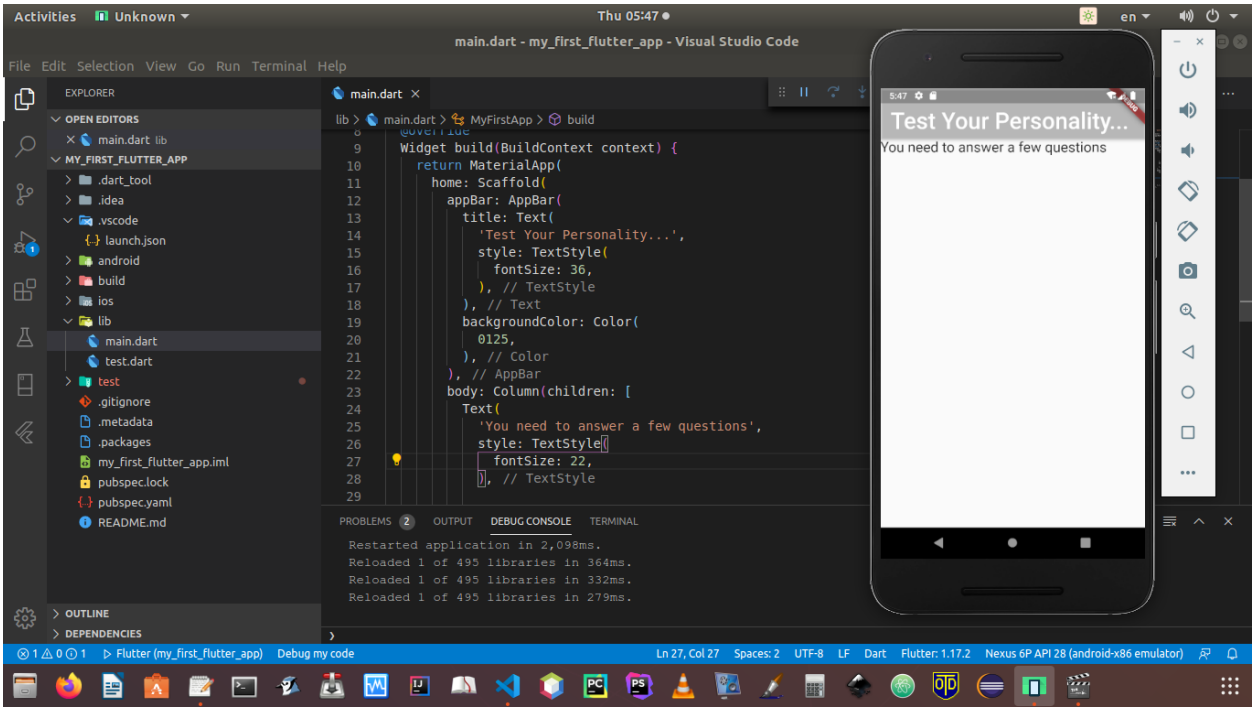

Figure 3.5 – We have successfully added more Text Widget in the body part

Watch this part of the code snippet, where we have added one element inside the list. It is a Text Widget with font size

```
children: [
            Text(
            'You need to answer a few questions',
            style: TextStyle(
                fontSize: 22,
            ),
            ),
        ],
```

Now, inside the ‘children’ list, the following part is one element.

```
Text(
            'You need to answer a few questions',
            style: TextStyle(
                fontSize: 22,
            ),
            ),
```

We can add more elements. It can be button Widget, because when you ask a question to your user, you expect an answer. Moreover, there should be more than one choices or options that the user can click.

In Flutter, there is a Widget called RaisedButton(). It is a visible Widget, and that will deliver a button to click. We can add three buttons, with three different values like the following code snippet:

```
// code 3.41
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: Text(
            'Test Your Personality...',
            style: TextStyle(
            fontSize: 36,
            ),
        ),
        backgroundColor: Color(
            0125,
        ),
        ),
        body: Column(
        children: [
            Text(
            'You need to answer a few questions',
            style: TextStyle(
                fontSize: 22,
```

```
30              ),
31              ),
32              RaisedButton(
33              child: Text('You have chosen answer 1'),
34              onPressed: null,
35              ),
36              RaisedButton(
37              child: Text('You have chosen answer 1'),
38              onPressed: null,
39              ),
40              RaisedButton(
41              child: Text('You have chosen answer 1'),
42              onPressed: null,
43              ),
44          ],
45          ),
46      ),
47      );
48  }
49  }
```

Automatically it will render three buttons with text ‘You have chosen answer 1’. It should be in order, that is 1, 2,and 3. We will correct that in our next code. Before that, let us take a look at the following image (Figure 3.6).

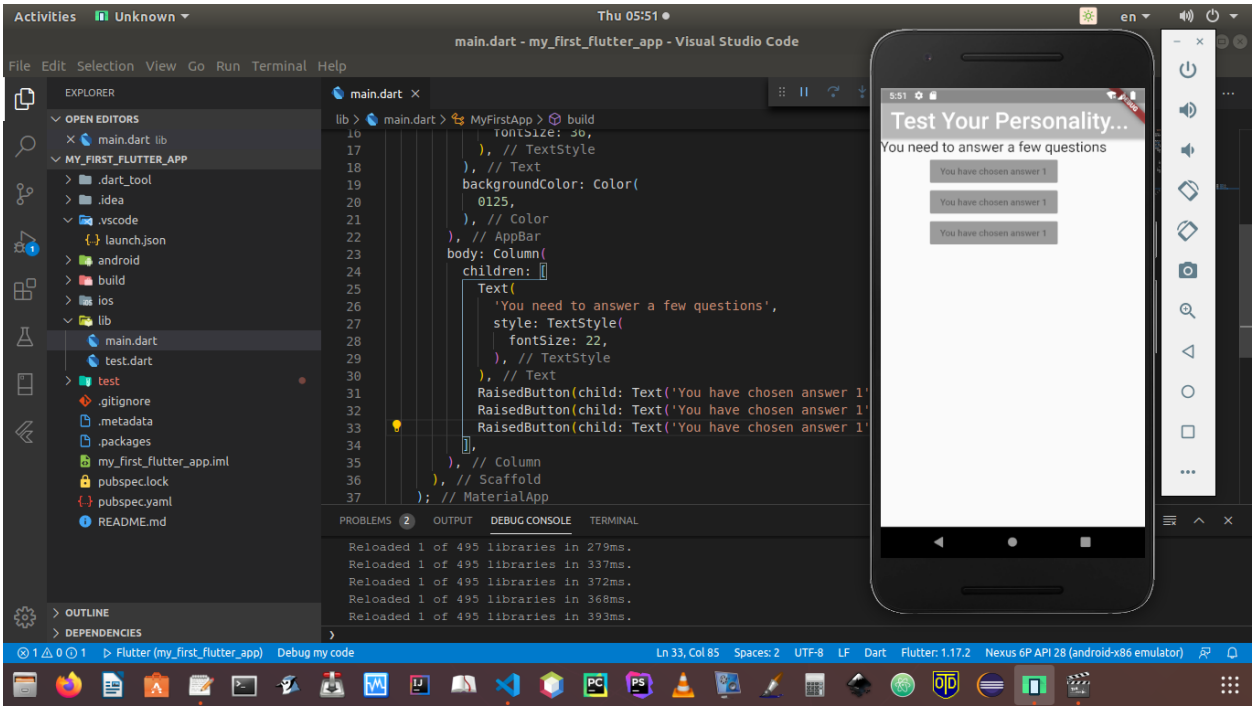

Figure 3.6 – Rendering header question and three buttons

Although we have reacted successfully to the challenge of adding more elements (here RaisedButton() Widget) to our list, yet we are not satisfied with the look. The text inside the button appears to be small. We want to make them bigger.

Now, we have learned how to add more than one Widgets inside one Widget. Therefore, it should not give us anymore trouble.

```
// code 3.42
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: Text(
            'Test Your Personality...',
            style: TextStyle(
```

```
17              fontSize: 36,
18              ),
19          ),
20          backgroundColor: Color(
21              0125,
22          ),
23          ),
24          body: Column(
25          children: [
26              Text(
27              'You need to answer a few questions',
28              style: TextStyle(
29                  fontSize: 22,
30              ),
31              ),
32              RaisedButton(
33              child: Text(
34                  'You have chosen answer 1',
35                  style: TextStyle(
36                  fontSize: 18,
37                  ),
38              ),
39              onPressed: null,
40              ),
41              RaisedButton(
42              child: Text(
43                  'You have chosen answer 2',
44                  style: TextStyle(
45                  fontSize: 18,
46                  ),
47              ),
48              onPressed: null,
49              ),
50              RaisedButton(
51              child: Text(
52                  'You have chosen answer 3',
53                  style: TextStyle(
54                  fontSize: 18,
55                  ),
56              ),
57              onPressed: null,
58              ),
59          ],
```

```
60            ),
61        ),
62        );
63  }
64  }
```

We have added TextStyle() Widget inside the RaisedButton() Widget and change the font size to 18. As we had previously changed the look of the connected virtual device, this time, the body part gets a new makeover (Figure 3.7).

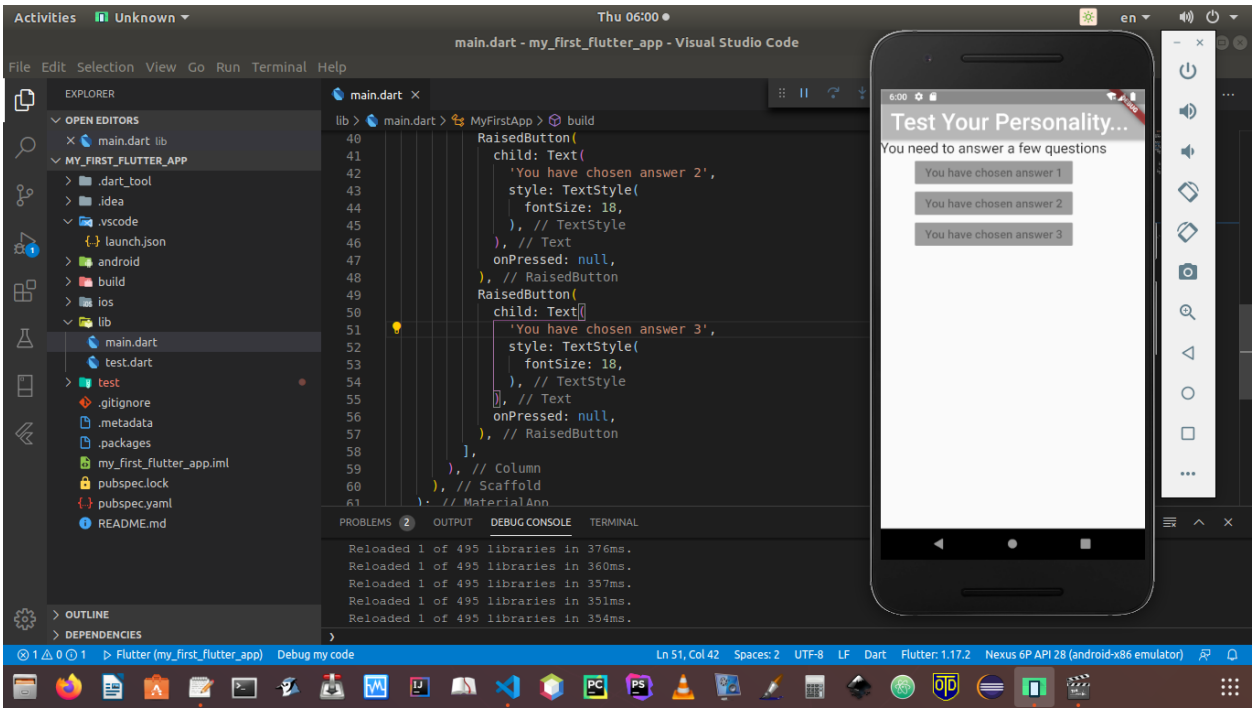

Figure 3.7 – The font size of the button has been increased

Although we have added three buttons to our question, when we click the button nothing happens.

It is happening, because we have kept the 'onPressed' named parameter to 'null'. This property is also related to the State management of any application. We will learn that in the coming chapters.

Before that, in the coming chapter, we will learn a few more features of Dart language, so that we will be able to proceed with our flutter application.

Want to read more Flutter related Articles and resources?

For more Flutter related Articles and Resources[8]

[8] https://sanjibsinha.com

4. Digging Deep into Dart to learn Flutter Logic

When we say: functions are objects in Dart, it seems confusing to the absolute beginners. The seasoned programmers may get the hint: Dart is an out and out object-oriented language. So even functions are objects and have a type called – Function. It means many things. One of the key things is you can assign a function to a variable, and even you can pass a function as arguments to other functions. In our Flutter app, we will see many implementations of this concept. We have already seen examples. You can also call an instance of a Dart class as if it were a function. However, to understand this key concepts that we are going to implement in our Flutter app, we need to know some more functionalities.

Control the flow of your code

Controlling the flow of your code is very important. Every programmer wants to control the logic for many reasons; one of the main reasons is the user of the software should have many options open to them. You do not know the conditions beforehand. You can only guess and as a developer, you should open as many avenues before the user as possible. There are several techniques adopted for controlling the flow of the code. The 'if and else' logic is very popular.

If and Else

Let us see an example where it works to control the flow of the code:

```
1  //code 4.1
2  main(List<String> arguments) {
3  bool firstButtonTouch = true;
4  bool secondButtonTouch = false;
5  bool thirdButtonTouch = true;
6  bool fourthButtonTouch = false;
7  if(firstButtonTouch) print("The giant starts running.");
8  else print("To stop the giant please touch the second button.");
9  if(secondButtonTouch) print("The giant stops.");
10 else print("You have not touched the second button.");
11 print("Touch any button to start the game.");
12 if(thirdButtonTouch) print("The giant goes to sleep.");
```

```
else print("You have not touched any button.");
if(fourthButtonTouch) print("The giant wakes up.");
else print("You have not touched any button.");
}

//output of code
The giant starts running.
You have not touched the second button.
Touch any button to start the game.
The giant goes to sleep.
You have not touched any button.
```

Now you can make this small code snippet more complicated.

```
//code 4.2
main(List<String> arguments) {
bool firstButtonTouch = true;
var firstButtonUntouch;
bool secondButtonTouch = false;
bool thirdButtonTouch = true;
bool fourthButtonTouch = false;
firstButtonUntouch ??= firstButtonTouch;
firstButtonUntouch = false;
if (firstButtonUntouch == false || firstButtonTouch == true) print("The giant is sle\
eping.");
else print("You need to wake up the giant. Touch the first button.");
if(firstButtonTouch == true && firstButtonUntouch == false) print("The giant starts \
running.");
print("To stop the giant please touch the second button.");
if((secondButtonTouch == true && thirdButtonTouch == true) || fourthButtonTouch == f\
alse) print("The giant stops.");
else print("You have not touched the second button.");
print("Touch any button to start the game.");
if(thirdButtonTouch) print("The giant goes to sleep.");
else print("You have not touched any button.");
if(fourthButtonTouch) print("The giant wakes up.");
else print("You have not touched any button.");
}
```

According to your complexity of code, you should arrange your ‘if and else’ logic. And your output varies.

```
//output
The giant is sleeping.
The giant starts running.
To stop the giant please touch the second button.
The giant stops.
Touch any button to start the game.
The giant goes to sleep.
You have not touched any button.
```

For ‘if and else’ logic always remember these golden rules. 1. 1. When both conditions are true, the result is true. 2. 2. When both conditions are false, the result is false. 3. 3. When one condition is true and the other condition is false, the result is false. 4. 4. When one condition is true or one condition is false, the result is true.

In the above code, I just try to give you an idea about how you can use ‘if and else’ logic where you really need it. However, this example is too simple. It can be complex when relational operators get added and the logic may become complex.

Finally, before leaving this section, I would like to show you another code snippet where the existing set of rules or principles has been changed.

```
//code 4.3
main(List<String> arguments) {
bool firstButtonTouch = true;
var firstButtonUntouch;
bool secondButtonTouch = false;
bool thirdButtonTouch = true;
bool fourthButtonTouch = false;
firstButtonUntouch ??= firstButtonTouch;
firstButtonUntouch = false;
if (firstButtonUntouch == false || firstButtonTouch == true) print("The giant is sle\
eping.");
else if (thirdButtonTouch) print("You need to wake up the giant. Touch the first but\
ton.");
else if(firstButtonTouch == true && firstButtonUntouch == false) print("The giant st\
arts running.");
else if (secondButtonTouch) print("To stop the giant please touch the second button.\
");
else if((secondButtonTouch == true && thirdButtonTouch == true) || fourthButtonTouch\
 == false) print("The giant stops.");
else if (thirdButtonTouch) print("You have not touched the second button.");
else if (secondButtonTouch) print("Touch any button to start the game.");
else if(thirdButtonTouch) print("The giant goes to sleep.");
else if (firstButtonUntouch) print("You have not touched any button.");
```

```
if(fourthButtonTouch) print("The giant wakes up.");
else print("You have not touched any button.");
}

//output of code
The giant is sleeping.
You have not touched any button.
You can change the pattern and see what happens.
```

Conditional Expression

Consider this code where we check only one condition :

```
//condition? exp1 : exp2;
int num1 = 20;
int num2 = 30;
int smallerNumber = num1 < num2? num1 : num2;
// it is expected that num1 will always be smaller
int smallNumber = num1 ?? "Default number $num2";

TIPS: For small operations, we can use this conditional expression; it is extremely \
handy and useful. When we know the output, we can go for the second one. When there \
are two numbers, we can go for the first one.
```

Looking at Looping

For loop is necessary for iterating any collections of data, with the standard ‘for’ loop. Here is a typical example of ‘for’ loop.

```
//code 4.4
main(List<String> arguments) {
var proverb = StringBuffer('As Dark as a Dungeon.');
for(var x = 0; x <= 10; x++){
    proverb.write("!");
    print(proverb);
}
}
```

In the above code we have used two in-built functions, in our following ‘functions’ and ‘object-oriented programming’ chapters, we will discuss it later. The output is as follows:

```
//output
As Dark as a Dungeon.!
As Dark as a Dungeon.!!
As Dark as a Dungeon.!!!
As Dark as a Dungeon.!!!!
As Dark as a Dungeon.!!!!!
As Dark as a Dungeon.!!!!!!
As Dark as a Dungeon.!!!!!!!
As Dark as a Dungeon.!!!!!!!!
As Dark as a Dungeon.!!!!!!!!!
As Dark as a Dungeon.!!!!!!!!!!
As Dark as a Dungeon.!!!!!!!!!!!
```

In our future discussions, we will use 'for loop' quite extensively, so at present, we stop here. I hope you get the concept.

I am going to tell you about a very interesting feature of iterating collections such as 'Set' and 'Map'. When the object you are going to iterate is Iterable, you can use 'forEach()' method. We are about to present two sets of collections; one is Set and the other is Map. In our Flutter app we will use the concept of Map. Without using Map data structure, we cannot add interactivity in our Flutter application.

```
//code 4.5
main(List<String> arguments) {
Set mySet = {1, 2, 3};
var myProducts = {
    1 : 'TV',
    2 : 'Refrigerator',
    3 : mySet.lookup(2),
    4 : 'Tablet',
    5 : 'Computer'
};
var userCollection = {"name": "John Smith", 'Email': 'john@sanjib.site'};
myProducts.forEach((x, y) => print("${x} : ${y}"));
userCollection.forEach((k,v) => print('${k}: ${v}'));
}

//output of code
1 : TV
2 : Refrigerator
3 : 2
4 : Tablet
5 : Computer
```

```
name: John Smith
Email: john@sanjib.site
```

When we do not know the current iteration counter, the 'forEach()' method is a good option. In usual cases, Iterable classes, such as, List and Set also support the 'for()' loop form of iteration. Consider this code:

```
//code 4.6
main(List<String> arguments) {
var myCollection = [1, 2, 3, 4];
for(var x in myCollection){
    print("${x}");
}
}

//output of code
1
2
3
4
```

While and Do-While

Be careful about handling the while loop. Since a while loop evaluates the condition before the loop, you must know to stop the loop at right time before it enters into infinity.

This is the pretty basic concept, but people often get confused about it.

```
//code 4.7
main(List<String> arguments) {
var num = 5;
var factorial = 1;
print("The value of the variable 'num' is decreasing this way:");
while(num >=1) {
    factorial = factorial * num;
    num--;
    print("'=>' ${num}");
}
print("The factorial  is ${factorial}");
}
```

In the above code, before the loop begins, the while() loop evaluates the condition. Since the value of the variable 'num' is 5 and it is greater than or equal to 1, the condition is true. So the loop begins. As the loop begins, we have also kept reducing the value of the variable 'num'; otherwise, it would have been entered an infinite loop.

The value of the variable reduces this way:

```
//output of code
The value of the variable 'num' is decreasing this way:
'=>' 4
'=>' 3
'=>' 2
'=>' 1
'=>' 0
The factorial  is 120
```

In case of do-while loop, it evaluates the condition after the loop.

```
//code 4.8
main(List<String> arguments) {
var num = 5;
var factorial = 1;
do {
    factorial = factorial * num;
    num--;
    print("The value of the variable 'num' is decreasing to : ${num}");
    print("The factorial  is ${factorial}");
}
while(num >=1);
}
```

We have slightly changed the code snippet so that it will show the reducing value of the variable and at the same time it will show you how the value of the factorial increases.

```
//output of code
The value of the variable 'num' is decreasing to : 4
The factorial  is 5
The value of the variable 'num' is decreasing to : 3
The factorial  is 20
The value of the variable 'num' is decreasing to : 2
The factorial  is 60
The value of the variable 'num' is decreasing to : 1
The factorial  is 120
The value of the variable 'num' is decreasing to : 0
The factorial  is 120
```

We can summarize the whole looping system. Actually they have a pattern. Once you understand the pattern, you can easily choose between ‘for’ , ‘while’ or ‘do-while’.

Understanding the Looping Patterns

I have met many students who feel confused about the ‘while’ loop. People often do not know that a ‘for’ loop can also turn into an infinite loop if it is not handled properly.

Actually, the concept of ‘loop’ is the same for every loop;be it ‘for’, ‘while’ or ‘do-while’. There are three things to remember:

```
    1. 1. counter variable
    2. 2. condition checking
    3. 3. according to the condition, increment or decrement.
```

Let us consider a code snippet:

```
void forLoopFunction(){
for(var i = 0; i <= 5; i ++){
    print(i);
}
}
void whileLoopFunction (){
var i = 0;
while(i <= 5){
    print(i);
    i++;
}
}
void doWhileLoop (){
```

```
var i = 0;
do{
    print(i);
    i++;
} while(i <= 5);
}
main(){
//print(smallerNumber);
//print(smallNumber);
forLoopFunction();
print("");
whileLoopFunction();
print("");
doWhileLoop();
}
```

I did not display the output, because you know what the output could have been. Let us consider the 'for' loop first.

```
for(var i = 0; i <= 5; i ++){
print(i);
}
```

We have started with the 'counter variable', here 'i = 0'. Then we have checked the condition: 'i <= 5'. After the second step, we have the third and the final step, according to the condition, we have incremented the value: 'i++'.

The steps are quite logical. We could not have decremented the value. It would have taken us to the infinite loop. Because in the 'condition checking' step, we will stop when the value of 'i' either less than or equal to 5.

If we had decremented the value of 'i', by writing 'i–', the condition checking would have never stopped until our computer's memory permits.

Now we have done the same thing in 'while' loop. Only the steps are a little bit different.

```
var i = 0;
while(i <= 5){
print(i);
i++;
}
```

In the above code, the 'counter variable' comes before the 'while loop' originally starts. The 'while loop' starts with the 'condition checking: i <= 5'; the same thing we have seen in the second step

of ‘for’ loop. After that, according to the condition, we have incremented the value of ‘i’ inside the ‘while loop’. Once the value of ‘i’ equals 5, it immediately stops.

Now check the ‘do-while loop’ code. We start with the counter variable. And then we increment or decrement the value.

```
var i = 0;
do{
print(i);
i++;
} while(i <= 5);
```

In the last stage we check the condition inside the ‘while loop’.

You may ask which one is better. Actually, it depends on the context. In some situations, ‘for loop’ is enough. In fact, in most cases, we can manage with the ‘for loop’. However, in some situations, we have to use while loop. In our Flutter application, we will face such situations. At that point, we will understand the actual mechanism of looping.

For Loop Labels

In some situations, we use nested for loops. Inside a ‘for loop’, we can run another ‘for loop’; in many cases, it is essential. In Dart, there is a concept called ‘Label’. We can handle the ‘outer loop’ and the ‘inner loop’ separately.

Let us see this code first:

```
void labelsLoop (){
outerloop: for(var x = 1; x <= 3; x++){
    print("One cycle of outerloop with $x starts and the whole innerloop runs.");
    innerloop: for(var y = 1; y <= 3; y++){
    if(x == 1 && y == 1){
        print("Since outerloop $x and innerloop $y both are 1, it gives no output.");
        break innerloop;
    }
    print(y);
    }
    print("One cycle of outerloop ends with $x");
}
}
main(List<String> arguments){
labelsLoop();
}
```

If you look at the output, you can understand how it works: One cycle of the outer loop with 1 starts and the whole inner loop runs.

```
Since outer loop 1 and inner loop 1 both are 1, it gives no output.
One cycle of the outer loop ends with 1
One cycle of the outer loop with 2 starts and the whole inner loop runs.
1
2
3
One cycle of the outer loop ends with 2
One cycle of the outer loop with 3 starts and the whole inner loop runs.
1
2
3
One cycle of the outer loop ends with 3
```

As you see in the above code the counter variable, condition checking and the increment parts are the same in both cases: the outer loop and the inner loop. So when the outer loop starts with 1, the inner loop inside the outer loop also starts with 1 and it should have completed the whole cycle. But we have injected an 'if clause' and told the program that when the value of the outer loop and the inner loop both are 1, break the 'inner loop'. We have used the 'Label': 'outer loop' and 'inner loop' to demarcate the loops. For the 'if clause' that particular cycle of 'inner loop' could not complete the whole cycle. However, after that, it goes on as usual.

The 'Label' is a very distinctive concept of Dart. Although, we have seen the same concept in Java.

Continue with For Loop

You have just seen how we have explicitly broken the inner loop and stopped one cycle of the inner loop. So 'break' is a very important concept while using 'for loop'. At the same breath, the 'continue' keyword also plays a very key role in 'for loop'. Let us consider this code snippet:

```
void loopContinue(){
for(var num = 1; num <= 5; num++){
    if(num % 2 == 0 ){
    print("These are all even numbers. $num");
    continue;
    } print("These are all odd numbers. $num");
}
}
main(List<String> arguments){
loopContinue();
}
```

Watch the output and you will understand how the keyword ‘continue’ works.

```
These are all odd numbers. 1
These are all even numbers. 2
These are all odd numbers. 3
These are all even numbers. 4
These are all odd numbers. 5
```

Let us change the above code a little bit and see how the output changes accordingly.

```
void loopContinue(){
for(var num = 1; num <= 5; num++){
    if(num % 2 == 0 ){
    //print("These are all even numbers. $num");
    continue;
    } print("These are all odd numbers. $num");
}
}

// output
These are all odd numbers. 1
These are all odd numbers. 3
These are all odd numbers. 5
```

According to the context, the keyword ‘continue’ means when the value is divisible by 2 and there is no remainder,just skip printing. Let us change the code again and see the output.

```
void loopContinue(){
for(var num = 1; num <= 5; num++){
    if(num % 2 == 0 ){
    print("These are all even numbers. $num");
    continue;
    } //print("These are all odd numbers. $num");
}
}

// output

These are all even numbers. 2
These are all even numbers. 4
```

Now our context has changed. When the value is divisible by 2 and there is no remainder, the keyword ‘continue’ tells the program to continue with printing the value as long as the ‘if clause’ stays true.

'Break and Continue' are two very important concepts not only in Dart, but in every programming language.

Decision making with Switch and case

In some cases, decision making seems to be easier, when you use 'Switch' instead of 'if and else' logic. Switch statements in Dart compare integers, string, or compile-time constants using the double equal sign '=='; it maintains a rule though, the compared objects must be instances of the same class and not of any of its sub types.

However, Switch statements in Dart are intended for limited circumstances, such as in interpreters or scanners. Let us see an example first to have an first-hand experience.

```
//code 4.9
main(List<String> arguments) {
//that could be the input value that would take inputs from users
var startingTime = 5;
switch (startingTime) {
    case 5:
    print("Printer Ready");
    break;
    case 6:
    print("Start printing");
    break;
    case 7:
    print("Stop for a second");
    break;
    case 8:
    print("Loading a tray and roll the paper.");
    break;
    case 9:
    print("Printer Ready, start priniting.");
    break;
    default:
    print("Default ${startingTime}");
}
}
```

When someone starts the printer it gives us output like this:

```
//output
Printer Ready
```

We have used a default clause to execute code when no case clause matches.

Controlling the flow of code is essential for many reasons. This is the base of any algorithms that instruct the machines to behave in a certain way. Building a mobile or web application needs a hundred and thousands of such instructions; algorithms could be complex and the set of algorithms require an understanding of a few other key concepts such as data structures, functions and object-oriented programming.

Digging Deep into Object-Oriented Programming

When we use a (.) notation, we usually refer to object properties or methods. A class may have properties and methods. After all, it is a blueprint of how an object will behave. How an object will behave in the future, depends on the class that has already been written. Whether a car object will start or stop, depends on that blueprint.

So we can say that objects have members consisting of functions and data; when you call a method you actually invoke it on an object.

Let us see some more examples to get acquainted with the idea of class and object. To start with let us assume a father bear is eating 6 fishes. To create the object of father bear, we need to have a bear class first where we should have one member variable or property ‘number of fish’ and one member method ‘eating that number of fish’. Ideally, both the property and the method should be annotated with the type ‘int’.

```
//code 4.10
class Bear {
int numberOfFish;
int eatFish(int numberOfFish){
    return numberOfFish;
}
}
main(List<String> arguments){
var fatherBear = new Bear();
print("Father bear eats ${fatherBear.eatFish(6)} number of fish.");
}
```

Very simple program. We have this output:

```
//output
Father bear eats 6 number of fish.
```

Can we take this code to the next level? As father bear eats fish and sleeps for some hours, he gains weight. Consider this code:

```
//code 4.11
class Bear {
int numberOfFish;
int hourOfSleep;
int weightGain;
int eatFish(int numberOfFish){
    return numberOfFish;
}
int sleepAfterEatingFish(int hourOfSleep){
    return hourOfSleep;
}
int weightGaining(int weightGain){
    weightGain = numberOfFish * hourOfSleep;
    return weightGain;
}
}
main(List<String> arguments){
var fatherBear = new Bear();
fatherBear.numberOfFish = 6;
fatherBear.hourOfSleep = 10;
fatherBear.weightGain = fatherBear.numberOfFish * fatherBear.hourOfSleep;
print("Father bear eats ${fatherBear.eatFish(fatherBear.numberOfFish)} number of fis\
h. And he sleeps for ${fatherBear.sleepAfterEatingFish(fatherBear.hourOfSleep)} hour\
s.");
print("Father bear has gained ${fatherBear.weightGaining(fatherBear.weightGain)} pou\
nds of weight.");
}
```

With the previous code we have added a few things, such as ‘hourOfSleep’ and ‘weightGain’; further we have added two related methods: ‘sleepAfterEatingFish(hourOfSleep)’ and ‘weightGaining(weightGain)’. As you see, we have passed two related parameters through those methods.

Father bear sleeps after eating the fish and gains weight. The value of weight he gains comes from the multiplication of ‘hourOfSleep’ and ‘weightGain’.

So we get this output while running this small program:

```
//output
Father bear eats 6 number of fish. And he sleeps for 10 hours.
Father bear has gained 60 pounds of weight.
```

Dart is extremely flexible language. You can write the same code in fewer lines. You do not have to use typical curly braces, and even you can omit the 'return' keyword to return the value automatically. You can also omit the 'new' word to create an instance. We are going to write the same code in this way now:

```
//code 4.12
class Bear {
int numberOfFish;
int hourOfSleep;
int weightGain;
//changing the styles of the methods completely
int eatFish(int numberOfFish) => numberOfFish;
int sleepAfterEatingFish(int hourOfSleep) => hourOfSleep;
int weightGaining(int weightGain) => weightGain = numberOfFish * hourOfSleep;
}
main(List<String> arguments){
var fatherBear = Bear(); //omitted the 'new' word
fatherBear.numberOfFish = 7;
fatherBear.hourOfSleep = 20;
fatherBear.weightGain = fatherBear.numberOfFish * fatherBear.hourOfSleep;
print("Father bear eats ${fatherBear.eatFish(fatherBear.numberOfFish)} fishes. And h\
e sleeps for ${fatherBear.sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");
print("Father bear has gained ${fatherBear.weightGaining(fatherBear.weightGain)} pou\
nds of weight.");
}
```

We have slightly changed the value of hours and the number of fishes. And the output also changes:

```
//output
Father bear eats 7 fishes. And he sleeps for 20 hours.
Father bear has gained 140 pounds of weight.
```

Before creating an object, we should have a clear picture of what that object is going to do. How we will use that object? According to that plan, we should have a blueprint,and write down the algorithms.

To make our life easier, in object-oriented programming, there is a concept called "constructor". Whenever you create an instance or object by using or by not using the 'new' keyword, inside the class, a method is automatically called, it is called the constructor method. In the next section, we will try to understand the concept.

More about Constructors

The first and foremost task of constructors is the construction of objects. In our Flutter app logic, we have already encountered such situations, where a Widget class constructor passes another Widget class constructor or methods.

Whenever we try to create an object the constructor is called first.

```
var fatherBear = Bear();
```

In the above code snippet, the left hand side of the equation represents the reference type of variable, which indicates to the new Bear object that has just been created in some memory place.

We actually try to arrange a spot in the memory for that object. The real work begins when we connect that spot with class properties and methods.

Using 'constructor' we can do that job more efficiently. Not only that, Dart allows to create more than one 'constructor', which is a great advantage. Let us write our 'Bear' class in a new way of using constructor:

```
//code 4.13
class Bear {
int numberOfFish;
int hourOfSleep;
int weightGain;
Bear(this.numberOfFish, this.hourOfSleep);
int eatFish(int numberOfFish) => numberOfFish;
int sleepAfterEatingFish(int hourOfSleep) => hourOfSleep;
int weightGaining(int weightGain) => weightGain = numberOfFish * hourOfSleep;
}
main(List<String> arguments){
var fatherBear = Bear(6, 10);
fatherBear.weightGain = fatherBear.numberOfFish * fatherBear.hourOfSleep;
print("Father bear eats ${fatherBear.eatFish(fatherBear.numberOfFish)} fishes. And h\
e sleeps for ${fatherBear.sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");
print("Father bear has gained ${fatherBear.weightGaining(fatherBear.weightGain)} pou\
nds of weight.");
}
```

Creating 'constructor' is extremely easy. Watch this line: Bear(this.numberOfFish, this.hourOfSleep);

The same class name works as a function or method and we have passed two arguments through that method. Once we get those values, we would calculate the third variable. Writing constructor

this way is known as “Syntactic Sugar”. In the later section of the book we will know more about the constructor.

Now it gets easier to pass the two values while creating the object. We could have done the same by creating constructor this way, which is more traditional:

```
//code 4.14
class Bear {
int numberOfFish;
int hourOfSleep;
int weightGain;
Bear(int numOfFish, int hourOfSleep){
this.numberOfFish = numOfFish;
this.hourOfSleep = hourOfSleep;
}
//Bear(this.numberOfFish, this.hourOfSleep);
int eatFish(int numberOfFish) => numberOfFish;
int sleepAfterEatingFish(int hourOfSleep) => hourOfSleep;
int weightGaining(int weightGain) => weightGain = numberOfFish * hourOfSleep;
}
main(List<String> arguments){
var fatherBear = Bear(6, 10);
fatherBear.weightGain = fatherBear.numberOfFish * fatherBear.hourOfSleep;
print("Father bear eats ${fatherBear.eatFish(fatherBear.numberOfFish)} fishes. And h\
e sleeps for ${fatherBear.sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");
print("Father bear has gained ${fatherBear.weightGaining(fatherBear.weightGain)} pou\
nds of weight.");
}
```

In both cases, the output is same as before:

```
//output
Father bear eats 6 fishes. And he sleeps for 10 hours.
Father bear has gained 60 pounds of weight.
```

In the above code, you can even get the object’s type very easily. We can change the type of value quite easily. Watch the main() function again:

```
//code 4.15
main(List<String> arguments){
var fatherBear = Bear(6, 10);
fatherBear.weightGain = fatherBear.numberOfFish * fatherBear.hourOfSleep;
print("Father bear eats ${fatherBear.eatFish(fatherBear.numberOfFish)} fishes. And h\
e sleeps for ${fatherBear.sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");
print("Father bear has gained ${fatherBear.weightGaining(fatherBear.weightGain)} pou\
nds of weight.");
print("The type of the object : ${fatherBear.weightGain.runtimeType}");
String weightGained = fatherBear.weightGain.toString();
print("The type of the same object has changed to : ${weightGained.runtimeType}");
}

//output of code
Father bear eats 6 fishes. And he sleeps for 10 hours.
Father bear has gained 60 pounds of weight.
The type of the object : int
The type of the same object has changed to : String
```

How to implement Classes

Now we have an idea of how classes and objects work together. A class is a blueprint that has some instance variables and methods. A class might have many tasks; but, it is a good practice and one of the major paradigms of object-oriented programming – a single class should have a single task. When many classes work together they should not be tightly coupled. They should be loosely coupled. It is a principle that is known as SOLID design principle. In Dart, we might implement the same principle while creating classes. We create a single class with a single task. We are going to create a class that will check whether the URL is secured or not.

```
//code 4.16
class CheckHTTPS {
String urlCheck;
CheckHTTPS(this.urlCheck);
bool checkingURL(String urlCheck){
if(this.urlCheck.contains("https")){
    return true;
} else return false;
}
}
main(List<String> arguments){
var newURL = CheckHTTPS('http://example.com');
```

```
print("The URL ${newURL.urlCheck} is not secured");
}
```

We get this output after checking the URL:

```
//output of code
The URL http://example.com is not secured
```

So we have some basic steps to follow. Whenever we want to create a class we should have a clear vision about what this class will do. What will be its task?

First, we need some variables. Next, we need one or more methods where we can play with these variables.

```
//code 4.17
class MyClass {
String myVariable; //property or instance variable, initially null
MyClass(this.myVariable); //constructor
String myMethod(){ //method declaration
    return "This is my method and this is ${myVariable}"; //returning value
}
}
main(List<String> arguments){
var myObject = MyClass("My String"); //creating new instance of class MyClass
print("${myObject.myMethod()}"); //printing the value
}
```

Watch the code: we have declared an instance variable first. It is of 'String' type. Since we have not initialized the variable, it is initially null. In the next step, we have constructed an object by declaring a constructor where we have passed the instance variable.

Our method's type is also 'String'. In the method, we have returned the instance variable.

In the 'main()' function, we have created an object declaring the class 'MyClass'; and at the same time, we passed a string value through the class name. We have done this for one reason: when we constructed the object by declaring the constructor, one instance variable had been passed through it. Finally, we have called the class method and display the output.

From the above example, one thing is certain. We need to know more about the functions. So in the next section, we will write some functions and will try to understand how the functions work. Remember, inside a class, we usually call a function by a different name – method. Methods are essential parts of any class because these are the action part. So we need to understand it properly.

More on Functions or Methods

We need to understand a few important features of functions before we dig deep into object-oriented programming again. The proper understanding of functions will help us to understand methods inside a class. First of all there are functions that just do nothing. It called: 'void'.

Let us consider this code:

```
//code 4.18
main(List<String> arguments){
print(showConnection());
}
//optional positional parameter
String myConnection(String dbName, String hostname, String username, [String optiona\
lPassword]){
if(optionalPassword == null){
    return "${dbName}, ${hostname}, $username";
} else return "${dbName}, ${hostname}, $username, $optionalPassword";
}
void showConnection(){
myConnection("MySQL", "localhost", "root", "*******");
}
```

We have declared the function 'showConnection()' as 'void' and want to return it through the 'main()' function. We have this output:

```
//output
bin/main.dart:4:9: Error: This expression has type 'void' and can't be used.
print(showConnection());
```

We cannot use the type 'void'. If we use, we cannot return something through that function. So from a function we always expect something. We want a function to return a value. So we are going to change the above code and write it this way:

```
//code 4.19
main(List<String> arguments){
var myConnect = myConnection("MySQL", "localhost", "root", "*******");
print(myConnect);
}
//optional positional parameter
String myConnection(String dbName, String hostname, String username, [String optiona\
lPassword]){
if(optionalPassword == null){
    return "${dbName}, ${hostname}, $username";
} else return "${dbName}, ${hostname}, $username, $optionalPassword";
}
```

The above code displays a simple program to express the “database connections” using parameters. In the above code, we have used a new concept called the optional parameter. You have already known that if we declare parameters or arguments in our functions, we have to pass them as it is. Otherwise, it gives us errors. However, we can use the concept of ‘optional parameters’. In the function ‘myConnections()’ we have passed four arguments. Watch this line:

```
String myConnection(String dbName, String hostname, String username, [String optiona\
lPassword]){}
```

We have written the last argument as [String optionalPassword]. It means this argument is optional. If you study the Flutter packages, and default code libraries, you will find many instances of such optional parameters. In fact, we have already seen examples of named parameters and positional parameters.

You do not have to pass it when you call the function. Here the logic is simple. If the optional parameter ‘optionalPassword’ is not defined when we pass it inside the ‘main()’ function, it is treated as ‘null’. Since it has been defined and passed it afterward, it is not null. Therefore, we have got this output:

```
//output of code
MySQL, localhost, root, *******
```

Now, we change the above code slightly and will not pass that argument anymore.

```
//code 4.20
main(List<String> arguments){
var myConnect = myConnection("MySQL", "localhost", "root");
print(myConnect);
}
//optional positional parameter
String myConnection(String dbName, String hostname, String username, [String optiona\
lPassword]){
if(optionalPassword == null){
    return "${dbName}, ${hostname}, $username";
} else return "${dbName}, ${hostname}, $username, $optionalPassword";
}

// The output also changes:

//output of code
MySQL, localhost, root
```

Compare these two lines before and after and we will only then understand why optional parameter is important:

```
//code 4.21
var myConnect = myConnection("MySQL", "localhost", "root", "*******");

//code 4.22
var myConnect = myConnection("MySQL", "localhost", "root");
```

If we did not declare it as optional, it would have given us an error. Let us change the optional parameter and see what type of error we get.

```
//code 4.23
main(List<String> arguments){
var myConnect = myConnection("MySQL", "localhost", "root");
print(myConnect);
}
//optional positional parameter is no more
String myConnection(String dbName, String hostname, String username, String optional\
Password){
if(optionalPassword == null){
    return "${dbName}, ${hostname}, $username";
} else return "${dbName}, ${hostname}, $username, $optionalPassword";
}
```

Now optional parameter is no more, and for that reason, we encounter this error in the output:

```
//output of code
bin/main.dart:3:31: Error: Too few positional arguments: 4 required, 3 given.
var myConnect = myConnection("MySQL", "localhost", "root");
                             ^
bin/main.dart:8:8: Context: Found this candidate, but the arguments don't match.
String myConnection(String dbName, String hostname, String username, String optional\
Password){
```

Since there was no optional parameter we had to pass the fourth argument. In the previous code snippets, we did not have to do that.

In Dart, a function is an object, for that reason, the same concept is true in case of methods and you will find it when we will discuss object-oriented programming again.

Lexical Scope in Function

This concept is extremely important as long as Dart functions are concerned. In building Flutter application this concept is also crucial.

In the later chapters, when we will dig more deeply into the object-oriented programming, we will see how the concepts of 'access' play vital roles in Dart, as well as Flutter.

Let us be back into functions again. Fist watch the code below and read the comments added with the lines:

```
//code 4.24
var outsideVariable = "I am an outsider.";
main(List<String> arguments){
//we can access the outside variable
print(outsideVariable);
// we cannot access the insider variable, it gives us error
//print(insiderVariable);
// it is an insider function
String insiderFunction(){
    // I can access the outside variable, no problem
    print("This is from the insider function.");
    print(outsideVariable);
    String insiderVariable = "I am an insider";
    print(insiderVariable); // it's okay to access this insider
}
insiderFunction();
}
```

First, we have declared a variable outside our 'main()' function. It is called 'outsideVariable'. We can access that variable inside the main() function as an object. Remember, everything in Dart is an object or an instance of a class.

Second, we have declared an insider function called: 'insiderFunction()' of type 'String'. Now inside that insider function, we can safely call the outsider variable. Besides, if we create another insider variable, we can also call that.

So we get this output:

```
//output of code
I am an outsider.
This is from the insider function.
I am an outsider.
I am an insider
```

As such, there is no problem regarding the output. However,it will not be the same experience if we try to call the insider variable from outside the scope of our insider function.

```
//code 4.25
var outsideVariable = "I am an outsider.";
main(List<String> arguments){
//we can access the outside variable
print(outsideVariable);
// we cannot access the insider variable, it gives us error
print(insiderVariable);
// it is an insider function
String insiderFunction(){
    // I can access the outisde variable, no problem
    print("This is from the insider function.");
    print(outsideVariable);
    String insiderVariable = "I am an insider";
    print(insiderVariable); // it's okay to access this insider
}
insiderFunction();
}

// Now, watch the output:
//output of code
bin/main.dart:11:9: Error: Getter not found: 'insiderVariable'.
print(insiderVariable);
        ^^^^^^^^^^^^^^^
```

This output takes us to another interesting concept, 'getter'. We will see it in a minute. Before that, we should understand this 'inside and outside' case. This is called Lexical scope. You can call an

outside variable inside ‘main()’ function. However, if you define an object inside a function, you cannot call it outside.

A few words about Getter and Setter

We again come back to object-oriented programming with a key concept ‘getter and setter’. Whenever we write a class,it implicitly sets the value and we can get that value by using the ‘.’ notation. We can explicitly set the value and get it in this way:

```
1  //code 4.26
2  class myClass {
3      String name;
4      String get getName => name;
5      set setName(String aValue) => name = aValue;
6  }
7  main(List<String> arguments){
8  var myObject = myClass();
9  myObject.setName = "Sanjib";
10 print(myObject.getName);
11 }
```

It gives us the output ‘Sanjib’ as usual. But how does this happen? In ‘myClass’, we have ‘set’ the ‘setName()’ method by passing a parameter ‘aValue’. Later we have accessed that value through an instance (myObject.setName) of the class ‘myClass’. The interesting thing is, the method ‘setName(String aValue)’ defined inside the ‘myClass’, now works as an attribute.

You may ask why should we use ‘getter and setter’ when every class has been associated with default ‘getter and setter’?

Actually, it is a kind of overriding the default value by explicitly defining the ‘getter and setter’. The advantage is a getter has no parameter and returns a value, and the setter has one parameter and does not return a value.

More than one Constructor

In any class, there are many types of constructors that can be used in any application. As usual, we have a default constructor. We can pass parameters through it. We also have named parameters. When we call any Flutter Widget, there are hundreds of in-built classes that have many constructors defined for our use.

Let us see them in a code snippet and try to understand how they work.

```
//code 4.27
class Bear {
//reference variable
int collarID;
//default and parameterized constructor
Bear(this.collarID);
//first named constructor
Bear.firstNamedConstructor(this.collarID);
//second named constructor
Bear.secondNamedConstructor(this.collarID);
void trackingBear() {
    String color; // local variable
    print("Tracking the bear with collar ID ${collarID}");
}
}
main(List<String> arguments){
// bear1 is reference variable
// Bear() is object
var bear1 = Bear(1);
bear1.trackingBear();
var bear2 = Bear.firstNamedConstructor(2);
bear2.trackingBear();
var bear3 = Bear.secondNamedConstructor(3);
bear3.trackingBear();
}
```

In the above code, by Dart convention, when we write a class, we might have many things in place. First of all, we have a reference variable here: int collarID;

Inside the main() function, when we create an instance, we will again have a reference variable:

```
// bear1 is reference variable
// Bear() is object
var bear1 = Bear(1);
```

According to the Dart convention, the default Bear() constructor is the object here. We have passed the class level reference variable ‘collar ID’ through this constructor.

So while defining a class and afterward creating an instance, we have two types of reference variable: the first is class level reference variable, which can be pointed out as class properties or attributes; and the second one is object level or instance level reference variable.

In the constructor part we have one default and parameterized constructor:

```
//default and parameterized constructor
Bear(this.collarID);
Besides, we have two named constructors.
//first named constructor
Bear.firstNamedConstructor(this.collarID);
//second named constructor
Bear.secondNamedConstructor(this.collarID);
```

Through the named constructors, we have created three bear instances; moreover, each instance works as if we are using the default constructors. Finally, when you run the code, you cannot distinguish between the default and the named constructors.

```
Tracking the bear with collar ID 1
Tracking the bear with collar ID 2
Tracking the bear with collar ID 3
```

In the following code snippets, we will see how we follow this central idea of Flutter. We will code our UI and change the view of the application with the help of different widgets.

With the help of widgets we will describe what the view of the application should look look like. Suppose we want to add an icon of search, or an icon of menu button, there are default widgets available for that. Suppose we want to place the title in the middle of the body section. There are widgets for that.

Primarily every widget has two key things – configuration and state. We will discuss about State later in chapter seven. In the next chapter, we will discuss about widgets in great detail, we will also try to understand how we can configure the widgets. Whenever, we change the configuration or state of the widgets, the change takes place in the description part.

Changing the UI of the Flutter projects

In this chapter, we have learned many Dart language basic concepts. These concepts are implemented through the widget trees.

We can think of our UI as a collection of hundreds and hundreds of widgets. One widget consists of many widgets, which again consist of several other widgets, and by doing this we change the view of the Flutter application. Inside the main function of the Flutter project, we have seen the runApp() function. We pass our application object as parameter or argument inside that function like this:

```
void main() {
runApp(MyFirstApp());
}
```

But, what is this 'MyFirstApp()', it actually extends another Widget 'StatelessWidget'. The widget tree is building in that way. According to our code, 'MyFirstApp()' becomes the root widget when we pass it through the runApp() function. The root widget sits on the top of the widget tree structure. Below that tree, we start connecting other widgets and our tree grows bigger and bigger.

In the coming chapter, we will discuss this widget tree structure in great detail, because this is the core concept of any Flutter application.

In this section, we are going to change the UI of the existing Flutter application by adding and altering some widgets.

Quite naturally, the code is getting bigger and bigger, however, if we can focus on widget tree structure, we will understand how one widget consists of another widget. Just use 'CTRL+SHIFT and I', it will auto-format your code and give the perfect shape of the widget tree structure. We will also inspect the changes, especially those changes that modifies our UI design.

```
// code 4.28
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {

    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        leading: IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null,
        ),
        title: Text('Test Your Knowledge...',
        style: TextStyle(fontSize: 25.00,
        fontStyle: FontStyle.normal,
        ),
        ),
```

```
25          actions: <Widget>[
26              IconButton(
27              icon: Icon(Icons.search),
28              tooltip: 'Search',
29              onPressed: null,
30              ),
31          ],
32          ),
33
34          body: Center(
35          child: Text('First Flutter Application...',
36          style: TextStyle(fontSize: 20.00,
37          fontStyle: FontStyle.italic,
38          ),
39          ),
40          ),
41          floatingActionButton: FloatingActionButton(
42          tooltip: 'Add',
43          child: Icon(Icons.add),
44          onPressed: null,
45          ),
46      ),
47      );
48  }
49  }
```

Running this code will build the new UI for us with the help of new widgets (Figure 4.1). We will change the the title of the application from 'Test your personality...' to 'Test Your Knowledge...'. The background color has also been changed.

There is a change in the look of the 'body' section. Here Scaffold() widget is the main layout of the major material components.

For the reason, under the Scaffold() widget, comes the 'appBar' widget tree.

Let us first see the new look of our Flutter application first. After that we will discuss how changing of the widgets affects the old UI design.

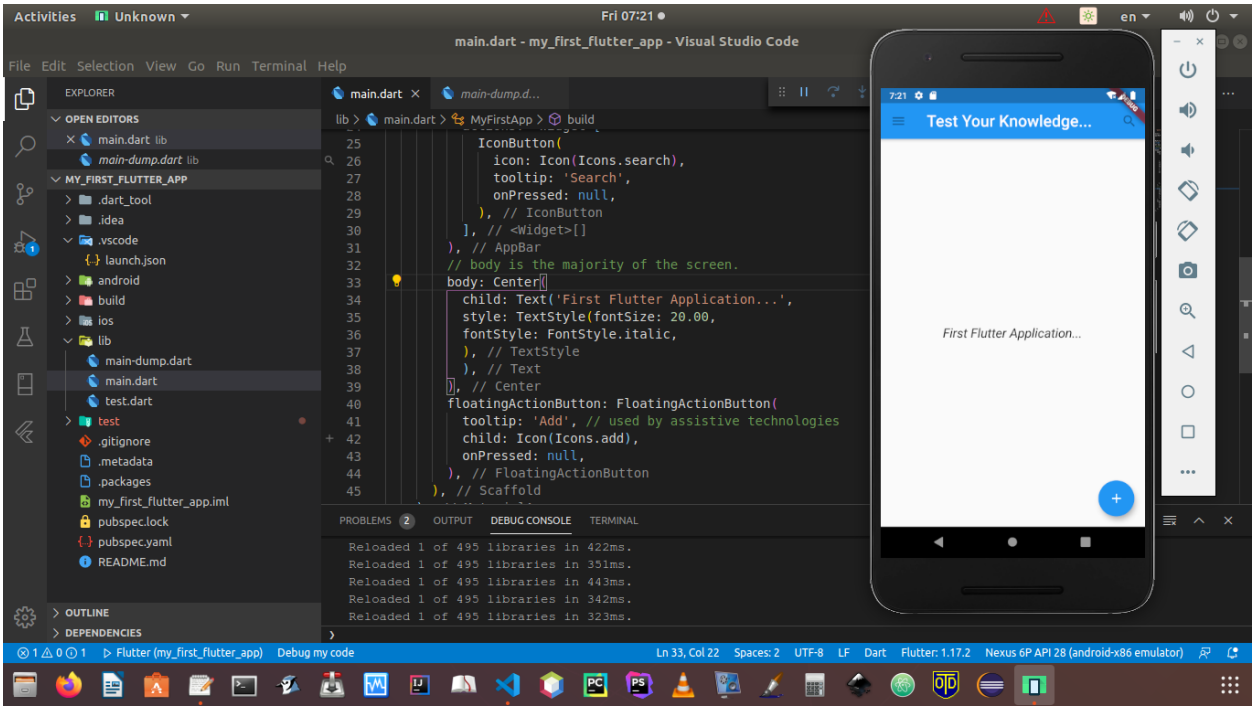

Figure 4.1 – Figure 4.1 – New widgets build a new UI for the Flutter application

Watch the 'appBar' section especially.

```
appBar: AppBar(
        leading: IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null,
        ),
        title: Text('Test Your Knowledge...',
        style: TextStyle(fontSize: 25.00,
        fontStyle: FontStyle.normal,
        ),
        ),
        actions: <Widget>[
            IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
            ),
        ],
        ),
```

Now the appearance has been completely changed. In the upper part, we have 'leading' widget that

has three sub-trees under it - 'icon', 'tooltip and 'onPressed'. Obviously, these widgets act as named parameters. They describe what will be the view of the header section. For that reason, on the left side of the text 'Test Your Knowledge...' we have a menu button, and on the right side,we have a 'search' icon.

Until now, the body part is quite straight forward. Inside the 'body' widget, we have three sub-trees - 'child', 'style', and 'fontStyle'.

In the lower part of the body section, we have this code:

```
floatingActionButton: FloatingActionButton(

        tooltip: 'Add',
        child: Icon(Icons.add),
        onPressed: null,
        ),
```

The 'floatingActionButton' widget has again three widget sub-trees - 'tooltip', 'child', and 'onPressed'.

In the next figure (Figure 4.2), we have not brought a considerable change in the UI design. We have brought back the three RaisedButton() widget, just like before. Below the figure we have our changed source code snippet.

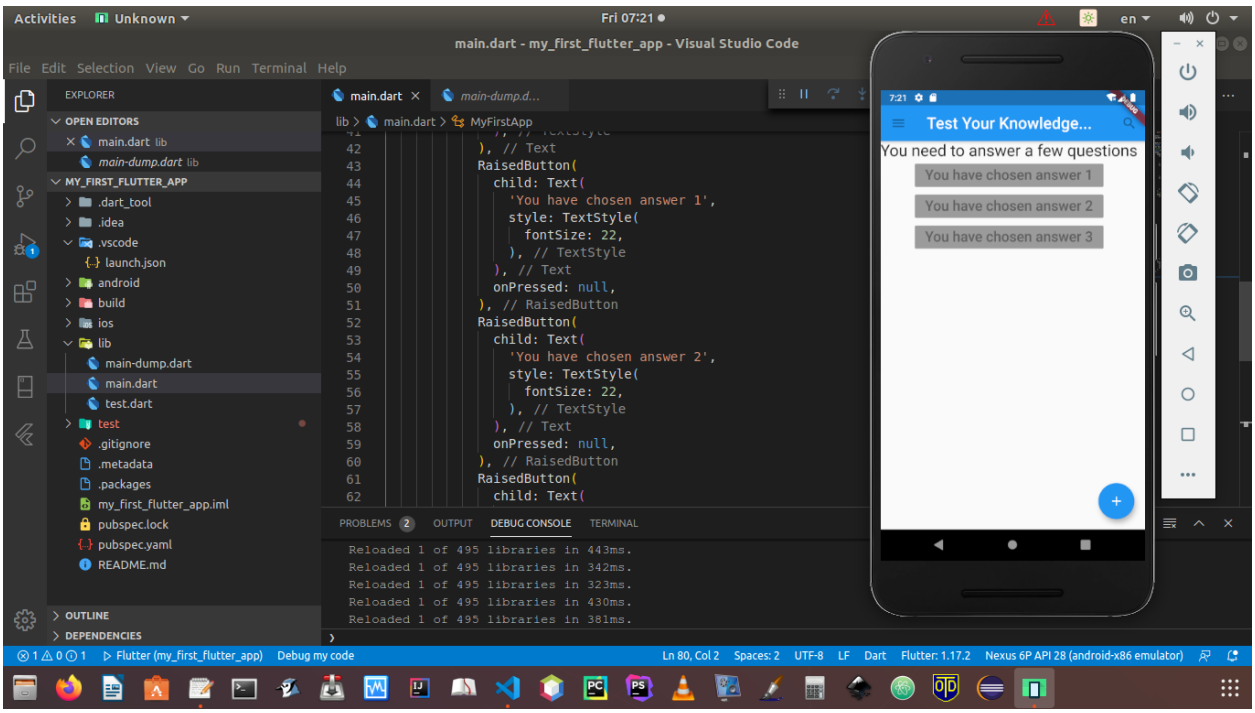

Figure 4.2 – Bringing back the RaisedButton() widgets

We have changed mainly the body part. Altering the body part gives us the above image.

```
//code 4.29
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        leading: IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null,
        ),
        title: Text(
            'Test Your Knowledge...',
            style: TextStyle(
            fontSize: 25.00,
            fontStyle: FontStyle.normal,
            ),
        ),
        actions: <Widget>[
            IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
            ),
        ],
        ),

        body: Column(
        children: [
            Text(
            'You need to answer a few questions',
            style: TextStyle(
                fontSize: 25,
            ),
            ),
            RaisedButton(
```

```
44              child: Text(
45                  'You have chosen answer 1',
46                  style: TextStyle(
47                  fontSize: 22,
48                  ),
49              ),
50              onPressed: null,
51              ),
52              RaisedButton(
53              child: Text(
54                  'You have chosen answer 2',
55                  style: TextStyle(
56                  fontSize: 22,
57                  ),
58              ),
59              onPressed: null,
60              ),
61              RaisedButton(
62              child: Text(
63                  'You have chosen answer 3',
64                  style: TextStyle(
65                  fontSize: 22,
66                  ),
67              ),
68              onPressed: null,
69              ),
70          ],
71          ),
72          floatingActionButton: FloatingActionButton(
73          tooltip: 'Add', // we can add more questions later
74          child: Icon(Icons.add),
75          onPressed: null,
76          ),
77      ),
78      );
79  }
80  }
```

Now, the tree structure is getting clearer than before. We have one widget, and under that widget, we add many other widgets. It goes on like this.

However, the next figure (Figure 4.3) will show you how we have changed the entire look by adding more widgets inside our existing tree structure.

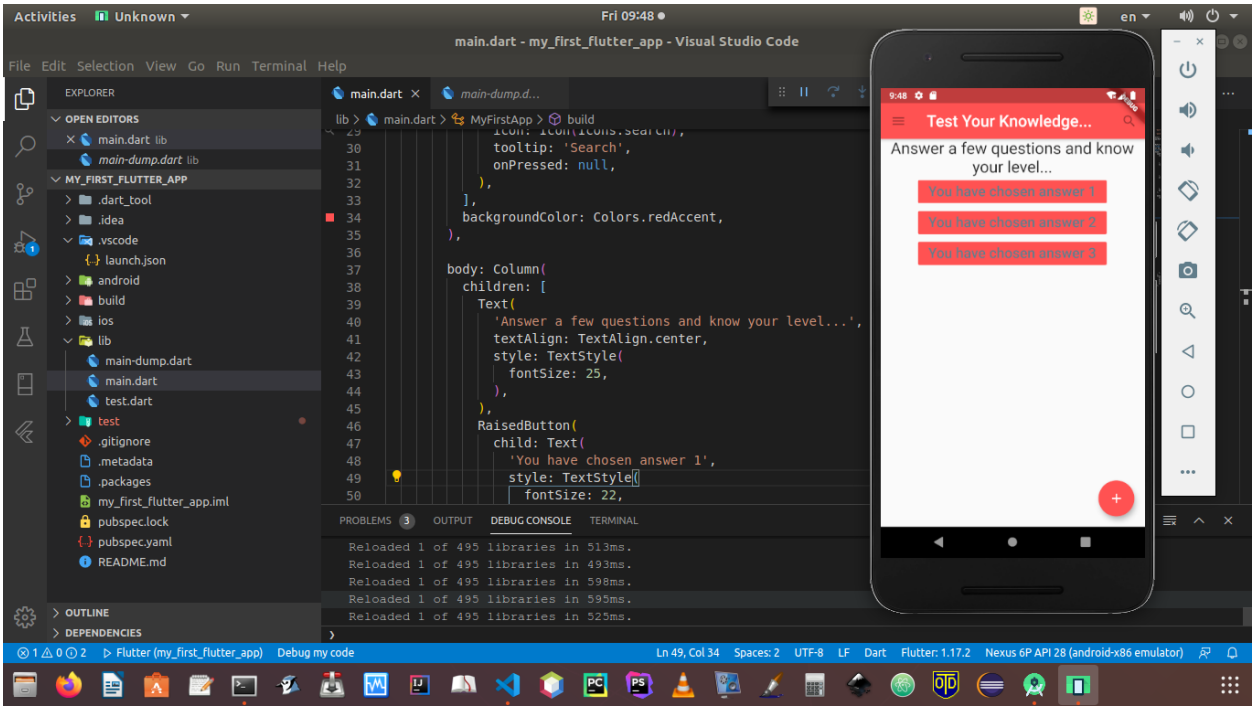

Figure 4.3 – A completely new look of our existing Flutter application

First, we will see the entire code. The previous code have slightly been changed. It makes our Flutter application more presentable.

```
//code 4.30

import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        leading: IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null,
        ),
        title: Text(
```

```
21              'Test Your Knowledge...',
22              style: TextStyle(
23              fontSize: 25.00,
24              fontStyle: FontStyle.normal,
25              ),
26          ),
27          actions: <Widget>[
28              IconButton(
29              icon: Icon(Icons.search),
30              tooltip: 'Search',
31              onPressed: null,
32              ),
33          ],
34          backgroundColor: Colors.redAccent,
35          ),
36
37          body: Column(
38          children: [
39              Text(
40              'Answer a few questions and know your level...',
41              textAlign: TextAlign.center,
42              style: TextStyle(
43                  fontSize: 25,
44              ),
45              ),
46              RaisedButton(
47              child: Text(
48                  'You have chosen answer 1',
49                  style: TextStyle(
50                  fontSize: 22,
51                  color: Colors.blueGrey,
52                  ),
53              ),
54              disabledColor: Colors.redAccent,
55              onPressed: null,
56              ),
57              RaisedButton(
58              child: Text(
59                  'You have chosen answer 2',
60                  style: TextStyle(
61                  fontSize: 22,
62                  color: Colors.blueGrey,
63                  ),
```

```
            ),
            disabledColor: Colors.redAccent,
            onPressed: null,
            ),
            RaisedButton(
            child: Text(
                'You have chosen answer 3',
                style: TextStyle(
                fontSize: 22,
                color: Colors.blueGrey,
                ),
            ),
            disabledColor: Colors.redAccent,
            onPressed: null,
            ),
        ],
        ),
        floatingActionButton: FloatingActionButton(
        tooltip: 'Add', // we can add more questions later
        backgroundColor: Colors.redAccent,
        child: Icon(Icons.add),
        onPressed: null,
        ),
    ),
    );
}
}
```

So far, we have understood that Widgets are everything in a Flutter project.

What are constraints in flutter

As a beginner at the very beginning we need to know what are constraints in Flutter .

When you plan to learn Flutter, it starts with Layout. Right? Now, you cannot learn or understand flutter layout without understanding constraints. Therefore, our flutter learning starts by answering this question first – what are constraints in flutter?

Firstly, let me warn you at the very beginning. Flutter layout is not like HTML layout.

Secondly, if you come from HTML or web development background, don't try to apply those CSS rules here. Why so? Because, HTML targets a large screen. Whereas, we're dealing with a Mobile screen. So, layout should not be same. And, there are other reasons too.

Finally, flutter is all about widgets. As a result, we need to understand Flutter layout keeping widgets in our mind.

Let's start with a Material App, and, start building a simple Container widget with a Text widget as its child.

```
import 'package:flutter/material.dart';

class ConstraintSample extends StatelessWidget {
const ConstraintSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Constraint Sample',
    debugShowCheckedModeBanner: false,
    home: ConstraintSampleHomme(),
    );
}
}
@override
Widget build(BuildContext context) {
    return Container(
    width: 150,
    height: 200,
    child: const Text('Constraint Sample'),
    );
}
```

Let's run this simple flutter app, and see what happens.

What is the problem here?

We've returned a Container widget mentioning its width and height.

```
return Container(
    width: 150,
    height: 200,
    child: const Text('Constraint Sample'),
    );
```

However, that didn't work at all.

Now we know that the constraint in Flutter is all about the size of the box, which is nothing but a widget.

A size always deals with width and height.

In the above case, the material app takes the width and height of the whole screen and directs its immediate child Container to take that size.

That is why, although we've mentioned the size of the Container widget, it doesn't work.

Therefore, we can conclude that any Widget gets its constraint or size from its immediate parent. After that, it passes that constraint to its immediate child.

And this practice goes on as the number of Widgets increases in the widget tree.

In Flutter, a parent widget always controls the immediate child's size. However, when the parent becomes grand-parent, it cannot affect the constraint or size of the grand-child.

Why?

Because, the grand child has its parent that inherits the size from its parent and decides what should be the size of its child.

We can compare this mechanism with wealth. If a person inherits some wealth from her parent, she is in a position to decide how much of that wealth she will give to her child or children.

And this process goes on.

One after another widget tells its children what their constraints or sizes are.

How do you use constraints in flutter?

Since we have got the idea, now we can apply and see how we can use constraints in flutter.

Our next code snippet is like the following.

```
@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        width: 300,
        height: 100,
        color: Colors.amber,
        alignment: Alignment.bottomCenter,
        child: const Text(
        'Constraint Sample',
        style: TextStyle(
            fontSize: 25,
            fontWeight: FontWeight.bold,
            color: Colors.black,
        ),
        ),
```

```
17
18      ),
19      );
20  }
```

Now, we've wrapped the Container widget with a Center widget. As a result, the Center widget gets the full size now. And, after that, it asks immediate child Container widget how big it wants to be.

The Container said, "I want to be 300 in width and 100 in height. Not only that, I want to place my child at the bottom Center position. And, I also want my child should be of color amber."

The Center widget says, "Okay. No problem. Get what you want because I have inherited the whole screen-size from my parent. Take what you need."

Next, we change our code to this:

```
1   @override
2   Widget build(BuildContext context) {
3       return Center(
4       child: Container(
5           width: 300,
6           height: 100,
7           color: Colors.amber,
8           alignment: Alignment.bottomCenter,
9           child: Container(
10          width: 200,
11          height: 50,
12          color: Colors.blue,
13          alignment: Alignment.center,
14          child: const Text(
15              'Constraint Sample',
16              style: TextStyle(
17              fontSize: 20,
18              fontWeight: FontWeight.bold,
19              color: Colors.white,
20              ),
21          ),
22          ),
23      ),
24      );
25  }
```

Let's see the effect on the screen first. Then we'll discuss the code.

The above image is displayed because the code says that the child Container with width 300, height 100 and color amber has a child, which is another Container and that has color blue, width 200 and

height 50. However, the parent Container decides that it will show its child in the bottom Center position.

Align the child at the bottom Center means, we would pass this child Container a tight constraint that is bigger than the child's natural size, with an alignment of Alignment.bottomCenter.

As a result, the child Container gets the width 200 and height 50 and is placed at the bottom Center alignment. It happens smoothly because the parent Container's width and height is bigger than the child Container. Therefore, it allocates the exact size that it's asked for.

But it didn't happen, if the parent Container didn't set the alignment and said that, "Okay, child container, you can place yourself anywhere you like. Even you may decide your alignment."

So the code is like the following:

```
@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        width: 300,
        height: 100,
        color: Colors.amber,
        child: Container(
        width: 200,
        height: 50,
        color: Colors.blue,
        alignment: Alignment.center,
        child: const Text(
            'Constraint Sample',
            style: TextStyle(
            fontSize: 20,
            fontWeight: FontWeight.bold,
            color: Colors.white,
            ),
        ),
        ),
    ),
    );
}
```

As a result, the child Container decides to looks upward and tries to find if its grand-parent has any alignment already allocated for it.

While looking upward, it finds that the grand-parent is a Center widget itself. Therefore, it decides to take the Center position, as it has the Center alignment itself; and not only that, while doing so, it

ignores its own width and height, and it takes the width and height of the immediate parent, which eventually is another Container.

As a result it overlaps completely the parent Container.

To sum up, constraints are basically sizes of width and height that any Widget gets from its parent. However, in case, padding is added, the constraint may change. Although we have not added that feature, still you may test that on your own and see how it affects the sizes of the child.

What are BoxConstraints in Flutter

Imagine each Widget as a box. So it has a size or constraints that defines width and height.

In the previous section we have discussed what constraints are. In Flutter, every widget is rendered by their underlying RenderBox objects. As a result, for each boxes constraints are BoxConstraints.

For a Flutter beginner we need to say one thing at the very beginning. The constraints actually represent sizes of the rendered boxes, which are nothing but widgets.

In that light of the previous discussion, we must try to understand what BoxConstraints are.

We've already seen that widgets pass their constraints, which consist of minimum and maximum width and height, to their children. Moreover, each child may vary in size.

As a result we can say that render tree actually passes a concrete geometry, which is size.

Subsequently the widget tree grows in sizes and for each boxes the constraints are BoxConstraints. And, it consists of four numbers – a minimum width minWidth, a maximum width maxWidth, a minimum height minHeight, and a maximum height maxHeight. Therefore we can set a range of width and height.

As we've said before, the geometry of boxes consists of a Size. Consequently, the Size satisfies the constraints.

Each child in the rendered widget tree, gets BoxConstraints from its parent. After that, the child picks us the size that satisfies the BoxConstraints adjusting with the parent's size.

Certainly, with the size, position changes. A child does not know its position.

Why?

Because, if the parent adds some padding the child's position changes with it.

We'll see that in a minute.

Consider a Flutter app where Scaffold widget acts as the immediate child of Material App widget.

As a result, the Scaffold takes the entire screen from its immediate parent Material App.

Next, the Scaffold passes its constraints to its immediate child Center widget. And then the Center takes the constraints or size from Scaffold. However, as the Scaffold widget allocates some space for the App Bar widget, therefore, Center doesn't get the whole screen.

Let us see the code.

```dart
import 'package:flutter/material.dart';

class BoxConstraintsSample extends StatelessWidget {
const BoxConstraintsSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'BoxConstraints Sample',
    debugShowCheckedModeBanner: false,
    home: BoxConstraintsSampleHomme(),
    );
}
}

class BoxConstraintsSampleHomme extends StatelessWidget {
const BoxConstraintsSampleHomme({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('BoxConstraints Sample'),
    ),
    body: Center(
        child: Container(
        color: Colors.redAccent,
        padding: const EdgeInsets.all(
            20,
        ),
        child: const Text(
            'Box',
            style: TextStyle(
            fontFamily: 'Allison',
            color: Colors.black38,
            fontSize: 60,
            fontWeight: FontWeight.bold,
            ),
        ),
        constraints: const BoxConstraints(
            minHeight: 70,
            minWidth: 70,
            maxHeight: 200,
```

```
            maxWidth: 200,
        ),
        ),
    ),//Center
    );
}
}
```

If we run the code, we see the following screenshot.

The above code tells us about the Container's constraints that point to BoxConstraints, this way.

```
constraints: const BoxConstraints(
            minHeight: 70,
            minWidth: 70,
            maxHeight: 200,
            maxWidth: 200,
        ),
```

The Container widget has a constraints parameter that points to the BoxConstraints widget, which simply defines the minimum and maximum width, height. And the range is between 70px to 200px.

As a result, if we try to make the Text widget bigger, that will not display the whole text, in that case.

Why?

Because, Container passes its constraints to its child Text widget and it gets that exact value. That means, the size of Text widget must remain in between 70px to 200px.

What happens if we try to pass the same constraints to another Container, which will act as the immediate child.

Let's change our code.

How do you use box constraints in Flutter?

To use box constraints in Flutter, we must understand how BoxConstraints widget acts, maintaining the range of width and height.

Let's change our above code and it will look like the following, now.

```dart
import 'package:flutter/material.dart';

class BoxConstraintsSample extends StatelessWidget {
const BoxConstraintsSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'BoxConstraints Sample',
    debugShowCheckedModeBanner: false,
    home: BoxConstraintsSampleHomme(),
    );
}
}

class BoxConstraintsSampleHomme extends StatelessWidget {
const BoxConstraintsSampleHomme({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('BoxConstraints Sample'),
    ),
    body: Center(
        child: Container(
        color: Colors.redAccent,
        padding: const EdgeInsets.all(
            20,
        ),
        constraints: const BoxConstraints(
            minHeight: 170,
            minWidth: 170,
            maxHeight: 400,
            maxWidth: 400,
        ),
        child: Container(
            color: Colors.blueAccent[200],
            padding: const EdgeInsets.all(
            20,
            ),
            child: const Text(
            'Box',
```

```
            style: TextStyle(
                fontFamily: 'Allison',
                color: Colors.white,
                fontSize: 60,
                fontWeight: FontWeight.bold,
            ),
            ),
            constraints: const BoxConstraints.expand(
            height: 100,
            width: 100,
            ),
        ),
        ), //container
    ), //Center
    );
}
}
```

In the above code, we find two Container widgets. Both have constraints defined. The first Container have constraints like the following:

```
constraints: const BoxConstraints(
            minHeight: 170,
            minWidth: 170,
            maxHeight: 400,
            maxWidth: 400,
        ),
```

And its child, the second Container has constraints like the following:

```
constraints: const BoxConstraints.expand(
            height: 100,
            width: 100,
            ),
```

The first Container's constraints define a range of width and height. However,the second Container's constraints point to BoxConstraints constructor BoxConstraints.expand.

What does this mean, as long as the size of the child Container is concerned?

We can explain it this way.

Since the child Container receives its constraints from its parent Container. Within that range it can expand its width and height up to 100px. Not more than that.

Moreover, this expansion process starts from the Center.

Understanding how these widgets or boxes handle the constraints in flutter is very important for the beginners.

In general, there are three kind of boxes that we'll encounter while we learn Flutter.

Firstly, the widgets like Center and ListView; they always try to be as big as possible.

Secondly, the widgets like Transform and Opacity always try to take the same size as their children.

And, finally, there are widgets like Image and Text that try to fit to a particular size.

As we'll progress, we'll find, how these constraints vary from widget to widget.

As example, we can remember the role of Center widget. It always maintains the maximum size. The minimum constraint does not work here.

What is widget in Flutter

In Flutter everything is Widget. But in reality it's a class and creates its instances also.

Firstly, widget in flutter is a class that describes how our flutter app should look like by creating its instances.

Secondly, the central idea behind using widgets is to build our user interface using widgets. To clarify, widgets are like boxes on the mobile, tab or desktop screen. Consequently, since each box has a size, every widget has constraints that deal with width and height.

Therefore, finally, we can define a widget as a class that builds or rebuilds its description; and, our flutter app works on that principle.

For a beginner, we need to add something more with this definition.

In Flutter everything is widget. As a result, we must think widget as a central hierarchy in a Flutter framework.

Let us see a minimalist Flutter App to understand this concept first.

```
1 import 'package:flutter/material.dart';
2 import 'widgets/first_flutter_app.dart';
3
4 void main() {
5 runApp(
6     const FirstFlutterApp(),
7 );
8 }
```

The above code shows us that the runApp() function takes the given Widget FirstFlutterApp() and makes it the root of the widget tree.

And this is our first custom widget that will sit on the top of the tree.

With reference to the main() function we must also add that to work it properly we need to import Material App library. However, we don't want to dig deep now. Just to make it simple, let's know that we need to have a material design so we can show our widget boxes. Right?

Further, we have created a sub-directory called "widgets" in our "lib" directory and keep the code of FirstFlutterApp(). Remember that also represents a class of hierarchy. Therefore we need to import that local library too.

That is why we need to import that also.

```
import 'widgets/first_flutter_app.dart';
```

Now, we can take a look at the custom widget that we've built.

```
import 'package:flutter/material.dart';

class FirstFlutterApp extends StatelessWidget {
const FirstFlutterApp({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: Center(
        child: Text(
        'Hello, Flutter!',
        ),
    ),
    );
}
}
```

The FirstFlutterApp() extends Stateless widget. Widgets have state. But, don't worry, we'll discuss it later.

The widget tree consists of three more widgets. The MaterialApp, Center widget and its child, the Text widget.

As a consequence, the framework forces the root widget to cover the screen. It places the text "Hello, Flutter" at the Center of the screen.

Since we have used MaterialApp, we don't have to worry about the text direction. The MaterialApp will take care of that.

Since each widget is a Dart class, we need to instantiate each widget and want them to show up at the particular location. This is done through the Element class. This is nothing but an instantiation of a Widget at a particular location in the tree.

What do we see?

A text "Hello, Flutter".

However, it is displayed on the particular position in the widget tree. Now, this widget tree can be a complex one.

As a result, widgets can be inflated into more elements as the tree grows in size.

Before we close down, one thing to remember. A widget is an immutable description of part of a user interface.

We'll discuss that later when we will discuss Element class in detail.

What is element in Flutter

We can locate any widget in a widget tree by its element that is an instantiation of widget.

We have been trying to understand how Flutter works from a beginner's point of view. We've already discussed how Widget, constraints and BoxConstraints are related in our previous sections. Now, we'll try to understand what's the role of Element in Flutter.

So far, we've learned that widget is a class and a root widget might have a tree of other widgets.

That's fine.

However, we haven't known so far, how the instantiation of Widget works. After all, widget is a class. Therefore, that must be instantiated.

Element is the answer. In a widget tree, we can find the instantiation of a widget in a particular location, because that is a unique element.

Suppose we reuse Text widget several times in a widget tree. As a result, Text widget is instantiated several times. But each element is unique. Moreover, we can find each Text widget by that unique Element.

We can add one more comment to the above statement. With the widget tree, an Element tree is also formed.

Since widgets are immutable, any widget can be used to configure multiple sub-trees. However, due to the presence of Element, we can easily find the widget's specific location.

Let us see a screenshot and try to understand how this concept works.

In the above screenshot we have two Text widgets, two Text Button widgets. And at the bottom, we have a Floating action button also.

Let's see the associated code.

```
import 'package:flutter/material.dart';

class UnderstandingElement extends StatelessWidget {
const UnderstandingElement({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Constraint Sample',
    debugShowCheckedModeBanner: false,
    home: UnderstandingElementHomme(),
    );
}
}

class UnderstandingElementHomme extends StatelessWidget {
const UnderstandingElementHomme({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Element Sample'),
    ),
    body: Center(
        child: Column(
        children: [
            const Text('Element One: Text Widget'),
            const Text('Element Two: : Text Widget'),
            Row(
            children: [
                TextButton(
                onPressed: () {},
                child: const Text('Element Three: TextButton Widget'),
                ),
                TextButton(
                onPressed: () {},
                child: const Text('Element Four: TextButton Widget'),
```

```
                ),
            ],
            )
        ],
        ),
    ),
    floatingActionButton: FloatingActionButton(
        onPressed: () {},
        child: const Icon(Icons.add_a_photo),
    ),
    );
}
}
```

Now, in the above widget tree, how can we find the second Text widget? It has a unique element or instantiation of that Widget, which actually points to a particular location.

Actually that element represents the rendered widget on the screen.

If the parent widget rebuilds and creates a new widget for this location, a widget associated with a given element can also change.

In the next chapter we will mainly discuss the Widget part. There we will dissect the above code and see what fires this change in the look

Want to read more Flutter related Articles and resources?

For more Flutter related Articles and Resources[9]

[9] https://sanjibsinha.com

5. How to build Flutter UI using Widgets

We have already learned that Flutter UI is built out of Widgets. In that sense, widgets are everything in Flutter. First of all, we need to memorize the basic widgets, without which we cannot start building our Flutter UI design. For another thing, we need to know how to build our widgets tree logically. Depending on how widgets sub-trees are arranged, the User Interface gets the final shape. Finally, while writing our application, we will use widgets that are sub-classes of either 'StatelessWidget or StatefulWidget'.

For more Flutter related Articles and Resources[10]

We will discuss State management, and 'StatefulWidget' in chapter seven; although in this chapter, we are going to get a glimpse of how 'StatefulWidget' works.

In general, the main job of a widget is the implementation of a build() function. This build() function returns the root widget, which in turn builds the UI with the help of lower-level widgets.

The job of Flutter framework is to build those widgets, synchronizing one widget with another, keeping the process moving on until the process reaches the low point to represent the 'RenderObject', which computes and gives the representation of the final UI design.

We need to remember that widgets are passed as arguments to other widgets. We have already seen how one widget takes a number of different widgets as named arguments. We have seen that our Flutter application 'MyFirstApp()' extends 'StatelessWidget' class and the calls the Widget build() function that passes 'BuildContext context' as its only argument.

The 'MaterialApp' widget acts as the root widget. By and large, it passes two named arguments 'title' and 'home'. The 'title' could be a simple text, the title of the application we are going to build. Both are passed as arguments to other widgets, one of them is Scaffold(), another important widget.

In turn, the Scaffold widget takes a number of different widgets as named arguments, of which 'AppBar' widget plays a major role. The 'AppBar' widget again passes different types of widgets as named arguments.

We can define our title widget here also. This pattern of calling and adding lower-level widgets one after another, goes on until your design is complete.

Common Widgets in Flutter

As we were saying, we need to use hundreds of widgets to build our UI. However, keeping all the widgets in one place, especially in 'main.dart' file, does not look clean. It also makes our code

[10] https://sanjibsinha.com

unnecessarily lengthy, hard to debug.

The advantage of object-oriented programming is that we can use our objects as different module. It keeps the modularity. The objects are not tightly coupled, they should remain loosely coupled.

For that reason, we will use break our code snippets in different dart files and finally import them in the ‘main.dart’ file.

We will see the code snippet first, after that we will see how it affects our UI design. After that we will discuss the widgets used in our code.

```
1  //code 5.1
2  // my_appbar.dart
3  import 'package:flutter/material.dart';
4
5  class MyAppBar extends StatelessWidget {
6  MyAppBar({this.title});
7
8  final Widget title;
9
10 @override
11 Widget build(BuildContext context) {
12     return Container(
13     height: 116.0,
14     decoration: BoxDecoration(color: Colors.redAccent),
15
16     child: Row(
17
18         children: <Widget>[
19         IconButton(
20             icon: Icon(Icons.menu),
21             tooltip: 'Navigation menu',
22             onPressed: null,
23         ),
24
25         Expanded(
26             child: title,
27         ),
28         IconButton(
29             icon: Icon(Icons.search),
30             tooltip: 'Search',
31             onPressed: null,
32         ),
33         ],
34     ),
```

```
    );
}
}

-----------------

//my_scaffold.dart
import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_appbar.dart';

class MyScaffold extends StatelessWidget {
@override
Widget build(BuildContext context) {

    return Material(

    child: Column(
        children: <Widget>[
        MyAppBar(
            title: Text(
            'Test Your Knowledge...',
            style: Theme.of(context).primaryTextTheme.headline6,
            ),
        ),
        Expanded(
            child: Center(
            child: Text('Here we will place our body widget...',
            style: TextStyle(fontSize: 25),
            ),
            ),
        ),
        ],
    ),
    );
}
}

--------------------
//main.dart
import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_scaffold.dart';

void main() {
```

```
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'My app',
    home: MyScaffold(),
    );
}
}
```

We have used three different Dart files. While talking about this, we need to remember that either we can keep these files in 'lib' folder along with the 'main.dart' file, or we can create separate folders inside the 'lib' folder, and keep those files there. Wherever, we keep this file, while importing we have to mention the full path. We will come to that point in a minute, before we need to see how our widgets build the UI.

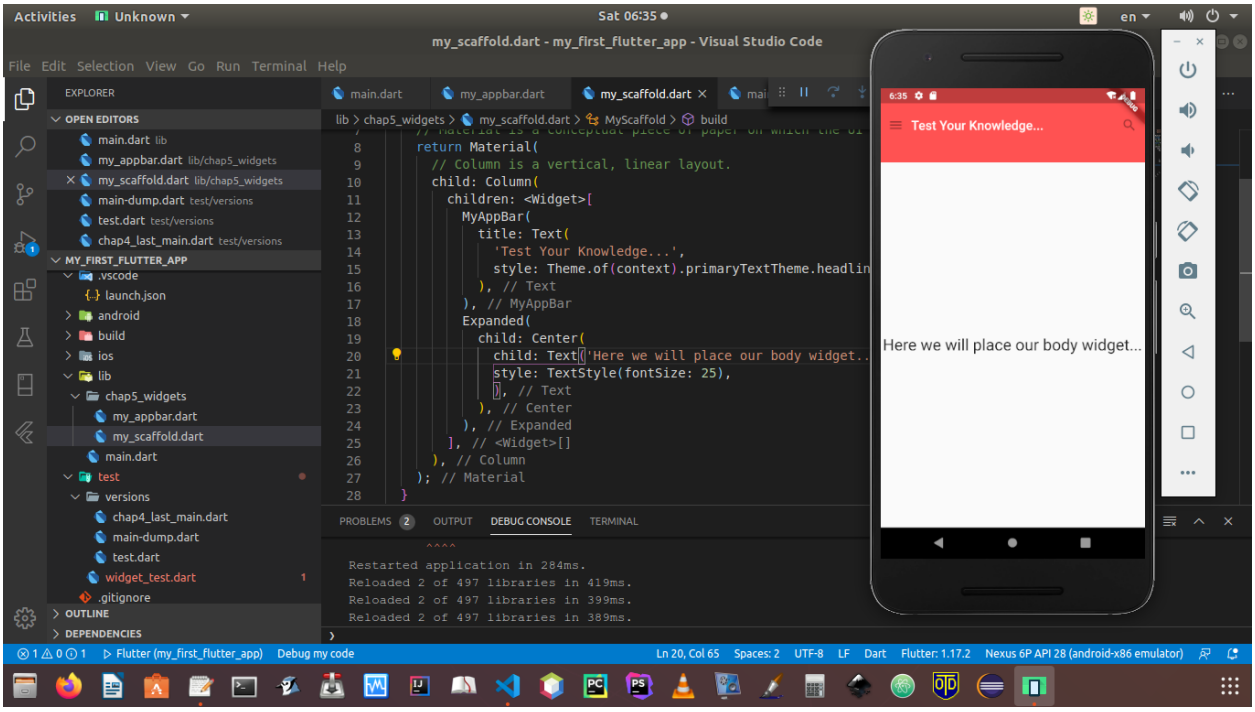

Figure 5.1 – How widgets build the UI of the application

It represents a simple UI design. To make it happen we have used three different Dart files to make our code clean.

Now, let us watch the widgets used in those files. Let us start with 'my_appbar.dart' file. It describes the 'AppBar' widget. We could have called it directly, while building the UI. However, we have

decided to keep it in a generic class, where we have passed one named argument ‘tile’ through its constructor.

```
MyAppBar({this.title});

final Widget title;
```

Fields (here ‘title’) in a Widget subclass (here ‘MyAppBar’) are always marked “final”. It is a convention, and we will maintain always.

Here as a lower-level widget we first use the ‘Container’. That widget, in turn, passes many named arguments as lower-level widgets. As the ‘child’ it passes the ‘Row’ widget and inside passes a list of ‘children’ widgets.

```
children: <Widget>[]
```

The above line describes what type of Widget we should pass as a list. We have learned the syntax of list data structure in Dart.

Whenever we use the lower-level widgets we always maintain the structure of hierarchies. A mobile screen will finally render this UI design. We have to build our widget trees keeping that in our mind.

Powerful Basic Widgets

Flutter tools come with hundreds of powerful widgets. We will start with some most commonly used basic widgets that we have already seen. Text widget helps us to build a load of styled text within our application. We will see their implementation as we progress.

Then come Row and Column. Both have extreme flexibility to create our layouts. When we want to build layout in horizontal, linear directions, we use Row. If we need vertically aligned layout, we use Column widget. Container widget is another very important widget that we will use in our application. It creates a rectangular visual element. We can skew the whole Container widget using matrix, we can also decorate with a ‘BoxDecoration’ that helps to build background, border, or a shadow. The size of the Container can be controlled by margins, padding, and other constraints.

AppBar class produces one of the most commonly used widgets. In the previous code, we have seen its implementation. It mainly displays the toolbar widgets, leading, title, icons, and other actions. For rendering different types of icons, it uses IconButton widget, that also has many sub-trees of widgets that we will see in a minute.

AppBar is a material design app bar that we may treat as the header section. It comes under the Scaffold widget, which is also one of the most commonly used widgets. Let us see our next code snippet and the figure of our changed application. After that, we will discuss the Scaffold widget.

```dart
//code 5.2
// testing_my_first_app.dart
import 'package:flutter/material.dart';

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {

    return Scaffold(
    appBar: AppBar(
        leading: IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null,
        ),
        title: Text(
            'Test Your Knowledge...',
            style: TextStyle(
            fontSize: 25.00,
            fontStyle: FontStyle.normal,
            ),
        ),
        actions: <Widget>[
            IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
            ),
        ],
        backgroundColor: Colors.redAccent,
        ),

    body: Column(
        children: [
            Text(
            'Answer a few questions and know your level...',
            textAlign: TextAlign.center,
            style: TextStyle(
                fontSize: 25,
            ),
            ),
            RaisedButton(
            child: Text(
```

```
                'You have chosen answer 1',
                style: TextStyle(
                fontSize: 22,
                color: Colors.blueGrey,
                ),
            ),
            disabledColor: Colors.redAccent,
            onPressed: null,
            ),
            RaisedButton(
            child: Text(
                'You have chosen answer 2',
                style: TextStyle(
                fontSize: 22,
                color: Colors.blueGrey,
                ),
            ),
            disabledColor: Colors.redAccent,
            onPressed: null,
            ),
            RaisedButton(
            child: Text(
                'You have chosen answer 3',
                style: TextStyle(
                fontSize: 22,
                color: Colors.blueGrey,
                ),
            ),
            disabledColor: Colors.redAccent,
            onPressed: null,
            ),
        ],
        ),
    floatingActionButton: FloatingActionButton(
        tooltip: 'Add',
        child: Icon(Icons.add),
        onPressed: null,
    ),
    );
}
}

// Scaffold is a layout for the major Material Components.
```

```
// body is the majority of the screen.

//main.dart
import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_first_app.dart';

void main(List<String> args) => runApp(MyFirstApp());
```

We have used two Dart files, 'testing_my_first_app.dart' and the 'main.dart'. We have made our 'main.dart' file just a one-line code:

```
void main(List<String> args) => runApp(MyFirstApp());
```

We have seen the fat arrow notation '⇒' before in our Dart practices. It can reduce any function to one line of code, when it calls one function.

The 'testing_my_first_app.dart' file starts with the Scaffold widget that has two main sub-trees widgets – appbar, and body. Let us see some code snippets from the appbar widget first.

```
appBar: AppBar(
        leading: IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null,
        ),
        title: Text(
            'Test Your Knowledge...',
            style: TextStyle(
            fontSize: 25.00,
            fontStyle: FontStyle.normal,
            ),
        ),
        actions: <Widget>[
            IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
            ),
        ],
        backgroundColor: Colors.redAccent,
        ),
```

The 'appbar' named argument points to AppBar Class constructor that passes three main widgets, leading, title and actions. Between these three widgets, 'actions' is a list widget. Although, at present,

we have only one element, IconButton class constructor that passes several other widgets as named parameters.

We are not repeating the 'body' widget, that as a named argument points to the one of the most commonly used widgets, Column. The Column has many other widgets that are being pointed by the named parameter 'children' which represents a 'list' data structure (Figure 5.2).

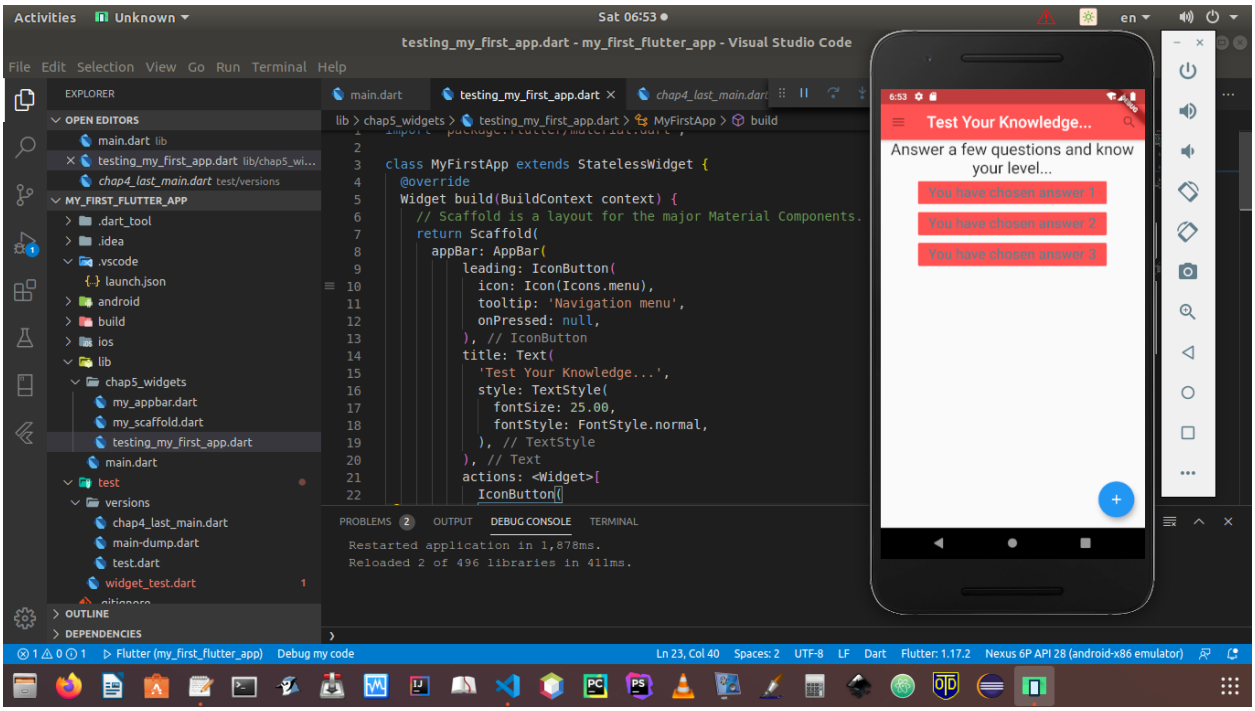

Figure 5.2 – The Scaffold widget and its widget sub-trees

So far, we have tried to maintain our code design as it was. Now we are going to add more functionalities to our application. So far we have used the 'StatelessWidget'. For the first time, we are going to change the state of our application. As we have found before, 'state' is used to maintain the data or information used by the application. When a user logs into our application, we should try to maintain the state as long as the user gets logged in.

We will discuss this elaborately in the coming chapter seven. We will also implement that in our application in due course. For the time being, we must remember that the 'Widget state' holds the current user input. Suppose we click a button, or icon, it automatically gives us a Text output and maintains that state for a few seconds. For the 'App state', this duration could be longer, because a user may want to gets logged in to our application for hours.

To get an idea, we have created three separate Dart files, 'my_stateless_scaffold.dart', 'my_first_-app.dart', and the 'main.dart' file, as usual.

We have also created a folder 'chap5_widgets' inside the 'lib' folder, where we put our code snippets. We need to import the full path so that the application works in a synchronized way.

```dart
//code 5.3
//my_stateless_scaffold.dart
import 'package:flutter/material.dart';

final GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>();
final SnackBar snackBarOne = const SnackBar(
    content: Text(
'Alert has been pressed!',
style: TextStyle(fontSize: 30),
));
final SnackBar snackBarTwo = const SnackBar(
    content: Text(
'Search has been pressed!',
style: TextStyle(fontSize: 30),
));
final SnackBar snackBarThree = const SnackBar(
    content: Text(
'Navigation has been pressed!',
style: TextStyle(fontSize: 30),
));

void clickNextPage(BuildContext context) {
Navigator.push(context, MaterialPageRoute(
    builder: (BuildContext context) {
    return Scaffold(
        appBar: AppBar(
        title: const Text('Know Yourself...'),
        ),
        body: const Center(
        child: Text(
            'Dig deep into every layer of your mind to find yourself...',
            style: TextStyle(fontSize: 24),
            textAlign: TextAlign.center,
        ),
        ),
    );
    },
));
}

class MyStatelessScaffoldWidget extends StatelessWidget {

MyStatelessScaffoldWidget({Key key}) : super(key: key);
```

```
44
45  @override
46  Widget build(BuildContext context) {
47      return Scaffold(
48      key: scaffoldKey,
49      appBar: AppBar(
50          actions: <Widget>[
51          IconButton(
52              icon: const Icon(Icons.add_alert),
53              tooltip: 'Show Snackbar',
54              onPressed: () {
55              scaffoldKey.currentState.showSnackBar(snackBarOne);
56              },
57          ),
58          IconButton(
59              icon: Icon(Icons.search),
60              tooltip: 'Search',
61              onPressed: () {
62              scaffoldKey.currentState.showSnackBar(snackBarTwo);
63              },
64          ),
65          IconButton(
66              icon: const Icon(Icons.navigate_next),
67              tooltip: 'Next page',
68              onPressed: () {
69              clickNextPage(context);
70              },
71          ),
72          ],
73          leading: IconButton(
74          icon: Icon(Icons.menu),
75          tooltip: 'Navigation menu',
76          onPressed: () {
77              scaffoldKey.currentState.showSnackBar(snackBarThree);
78          },
79          ),
80          title: Text(
81          'Knowledge Test',
82          style: TextStyle(
83              fontSize: 25.00,
84              fontStyle: FontStyle.normal,
85          ),
86          ),
```

```
87          backgroundColor: Colors.redAccent,
88      ),
89      body: Column(
90          children: [
91          Text(
92              'Answer a few questions and know your level...',
93              textAlign: TextAlign.center,
94              style: TextStyle(
95              fontSize: 25,
96              ),
97          ),
98          RaisedButton(
99              child: Text(
100             'You have chosen answer 1',
101             style: TextStyle(
102                 fontSize: 22,
103                 color: Colors.blueGrey,
104             ),
105             ),
106             disabledColor: Colors.redAccent,
107             onPressed: null,
108         ),
109         RaisedButton(
110             child: Text(
111             'You have chosen answer 2',
112             style: TextStyle(
113                 fontSize: 22,
114                 color: Colors.blueGrey,
115             ),
116             ),
117             disabledColor: Colors.redAccent,
118             onPressed: null,
119         ),
120         RaisedButton(
121             child: Text(
122             'You have chosen answer 3',
123             style: TextStyle(
124                 fontSize: 22,
125                 color: Colors.blueGrey,
126             ),
127             ),
128             disabledColor: Colors.redAccent,
129             onPressed: null,
```

```
130         ),
131         ],
132     ),
133     floatingActionButton: FloatingActionButton(
134         tooltip: 'Add', // we can add more questions later
135         backgroundColor: Colors.redAccent,
136         child: Icon(Icons.add),
137         onPressed: null,
138     ),
139     );
140 }
141 }
142 
143 //my_first_app.dart
144 import 'package:flutter/material.dart';
145 import 'package:my_first_flutter_app/chap5_widgets/my_stateless_scaffold.dart';
146 
147 class MyFirstApp extends StatelessWidget {
148 
149 @override
150 Widget build(BuildContext context) {
151     return MaterialApp(
152     home: MyStatelessScaffoldWidget()
153     );
154 }
155 }
156 
157 //main.dart
158 import 'package:flutter/material.dart';
159 import 'package:my_first_flutter_app/chap5_widgets/my_first_app.dart';
160 
161 void main(List<String> args) => runApp(MyFirstApp());
```

Although the whole code snippets look pretty lengthy, most parts are repeating the old code, especially the 'body' part.

In the 'appbar' section, we have some few new features, like 'alert' icon and the 'arrow' that points to a completely new page.

Let us first see the display. Now the 'appbar' section has a complete new look with more icons and the arrow symbol.

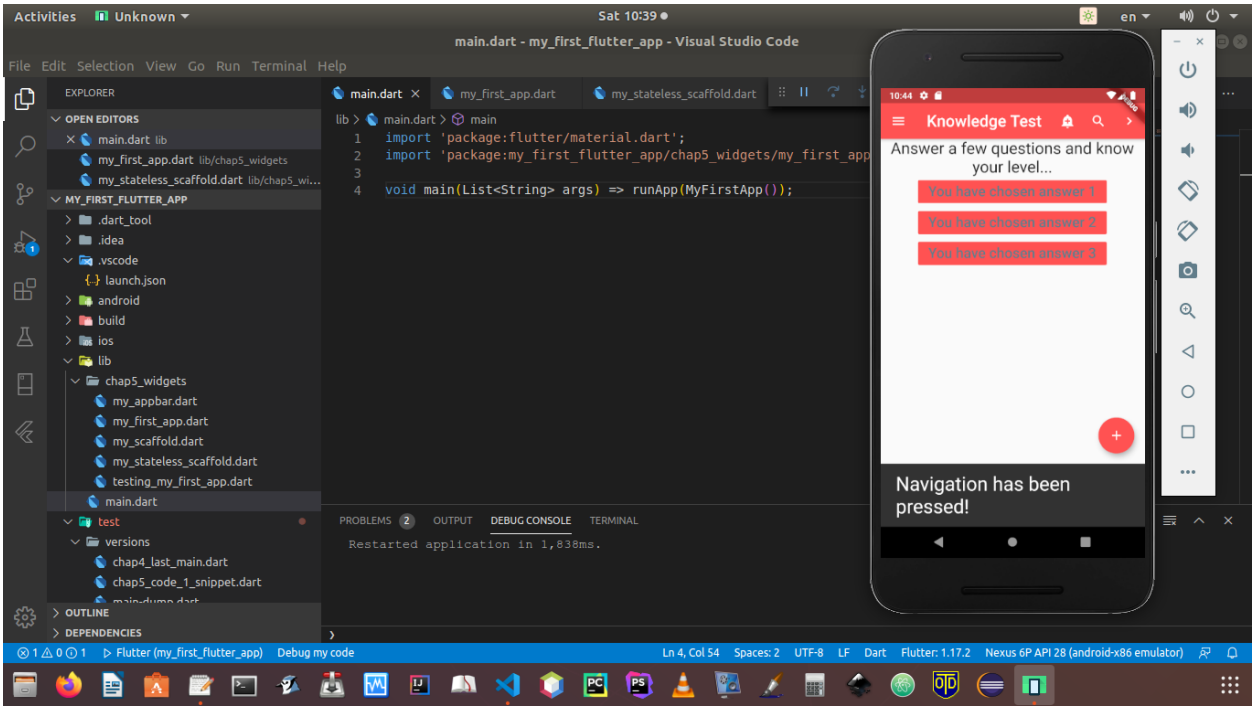

Figure 5.3 – A complete new look of 'appbar' widget, where widget state has been managed

Now the named parameter 'appbar' points to the AppBar class constructor that passes many named parameters, of which 'actions' points to a widget 'list'.

```
actions: <Widget>[
        IconButton(
            icon: const Icon(Icons.add_alert),
            tooltip: 'Show Snackbar',
            onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarOne);
            },
        ),
        IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarTwo);
            },
        ),
        IconButton(
            icon: const Icon(Icons.navigate_next),
            tooltip: 'Next page',
            onPressed: () {
            clickNextPage(context);
```

```
            },
        ),
        ],
        leading: IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarThree);
        },
        ),
```

So far we have maintained the 'onPressed' to 'null'. For the first time we have used an anonymous function that returns a chained method that connects different widget methods. In the later sections of this chapter, we have discussed anonymous or lambda function of Dart. If you are a beginner, please go through it.

Let us take a look at the last one:

```
leading: IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarThree);
        },
        ),
```

It leads us to the Navigation menu. Here according to the 'tooltip' name, we have chosen the 'Icons.menu' field in the Icon class constructor.

Let us see what happens when we press the 'alert' icon (Figure 5.4).

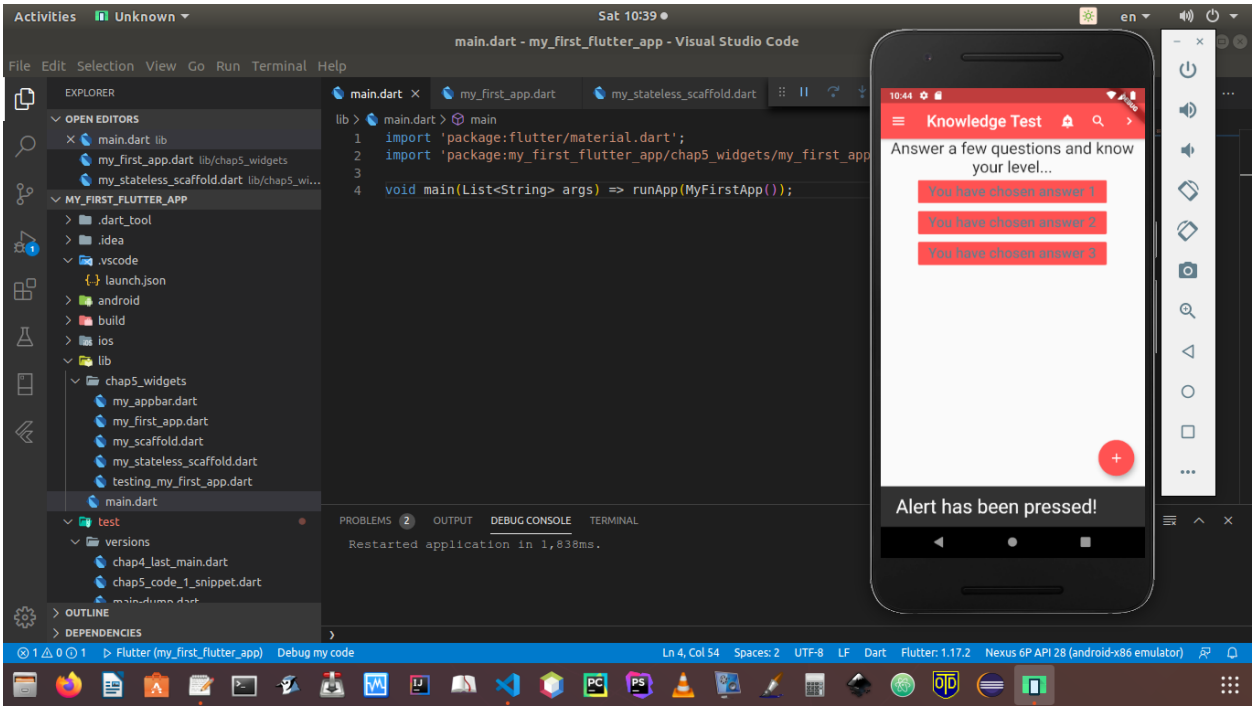

Figure 5.4 – The message 'Alert has been pressed' pops up at the bottom

Pressing the 'alert' icon gives us a message and that takes place due to the change of state. It stays for a few seconds.

Same thing happens, when we press the 'search' icon (Figure 5.5).

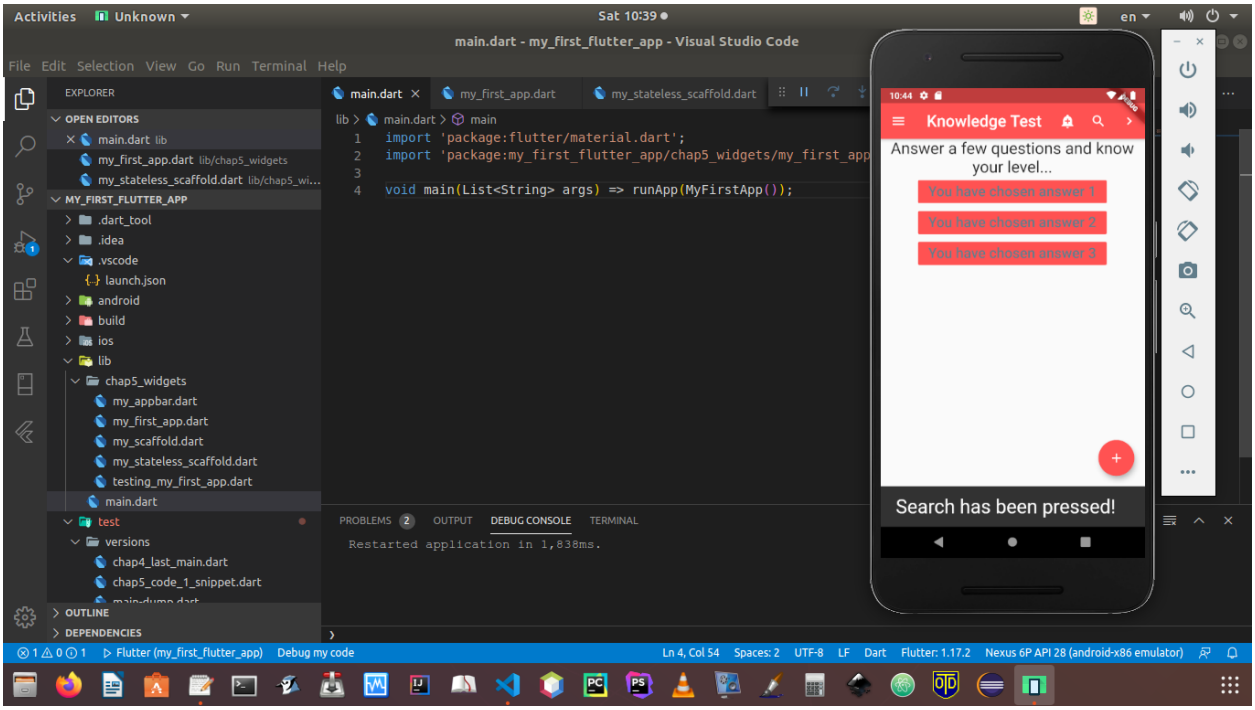

Figure 5.5 – The 'search' icon has been pressed

Now the time has come to go to the next page. We have created a special method for that purpose only.

```
void clickNextPage(BuildContext context) {
Navigator.push(context, MaterialPageRoute(
    builder: (BuildContext context) {
    return Scaffold(
        appBar: AppBar(
        title: const Text('Know Yourself...'),
        ),
        body: const Center(
        child: Text(
            'Dig deep into every layer of your mind to find yourself...',
            style: TextStyle(fontSize: 24),
            textAlign: TextAlign.center,
        ),
        ),
    );
    },
));
}
```

After that, we have called that function or method, in this part of our code:

```
IconButton(
            icon: const Icon(Icons.navigate_next),
            tooltip: 'Next page',
            onPressed: () {
            clickNextPage(context);
            },
        ),
```

Here as a named parameter, ‘onPressed’ passes an anonymous function that calls the ‘clickNextPage(context)’ method. The ‘actions’ widget list has that ‘IconButton’ widget. Clicking the navigation arrow takes us to the next page(Figure 5.6).

We have also defined a simple page inside that function just to get an idea how Flutter widgets manage these UI designs out of the box.

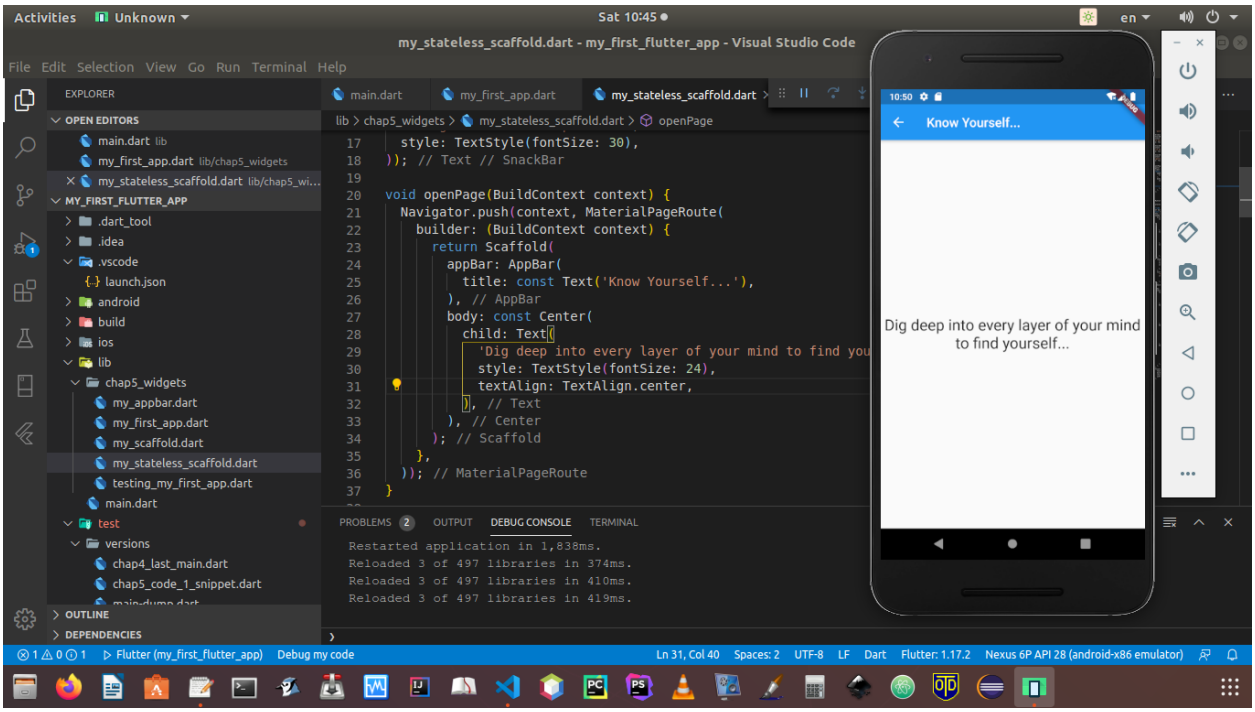

Figure 5.6 – We have navigated to the next page

Now our application becomes more interactive, maintaining the state also.

However, we want to see the Container widget separately, because in the later part of our application, we will use that widget for other purpose.

The next code snippet will take us to a different type of display where we will only show the Container widget.

```dart
//code 5.4
//my_container.dart
import 'package:flutter/material.dart';

class MyContainerWidget extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text(
        'Knowledge Test',
        style: TextStyle(
            fontSize: 25.00,
            fontStyle: FontStyle.normal,
        ),
        ),
        backgroundColor: Colors.redAccent,
    ),
    body: Container(
        constraints: BoxConstraints.expand(
        height: Theme.of(context).textTheme.headline4.fontSize * 1.1 + 200.0,
        ),
        padding: const EdgeInsets.all(8.0),
        color: Colors.blue[600],
        alignment: Alignment.center,
        child: Text('This is Container Widget',
            style: Theme.of(context)
                .textTheme
                .headline4
                .copyWith(color: Colors.white)),
        transform: Matrix4.rotationZ(-0.2),
    ),

    floatingActionButton: FloatingActionButton(
        tooltip: 'Add', // we can add more questions later
        backgroundColor: Colors.redAccent,
        child: Icon(Icons.add),
        onPressed: null,
    ),
    );
}
}

```

```
//my_first_app.dart
import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_container.dart';
//import 'package:my_first_flutter_app/chap5_widgets/my_stateless_scaffold.dart';

class MyFirstApp extends StatelessWidget {

@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: MyContainerWidget()
    );
}
}

//main.dart
import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_first_app.dart';

void main(List<String> args) => runApp(MyFirstApp());
```

The only new file is 'my_container.dart' where in the 'body' part, we have used the Container widget to get an idea, how we can use the 'matrix'. Watch this part:

```
body: Container(
        constraints: BoxConstraints.expand(
        height: Theme.of(context).textTheme.headline4.fontSize * 1.1 + 200.0,
        ),
        padding: const EdgeInsets.all(8.0),
        color: Colors.blue[600],
        alignment: Alignment.center,
        child: Text('This is Container Widget',
            style: Theme.of(context)
                .textTheme
                .headline4
                .copyWith(color: Colors.white)),
        transform: Matrix4.rotationZ(-0.2),
    ),
```

We can control many things of this Text element. With the Text widget, we cannot do those staff (Figure 5.7). Container widget is extremely handy tool that combines many other widgets, which in turn, controls padding, positioning and sizing.

The most amazing part of the Container class is it transforms the Text in different angles that can be controlled by changing the parameter values. transform: Matrix4.rotationZ(-0.2),

We can also control the background and foreground colors. If we do not use 'children', the Container can be as big as we want to make it look like, as well as we can also make it very small.

Moreover, with children, the Container controls its size according to the size of the children.

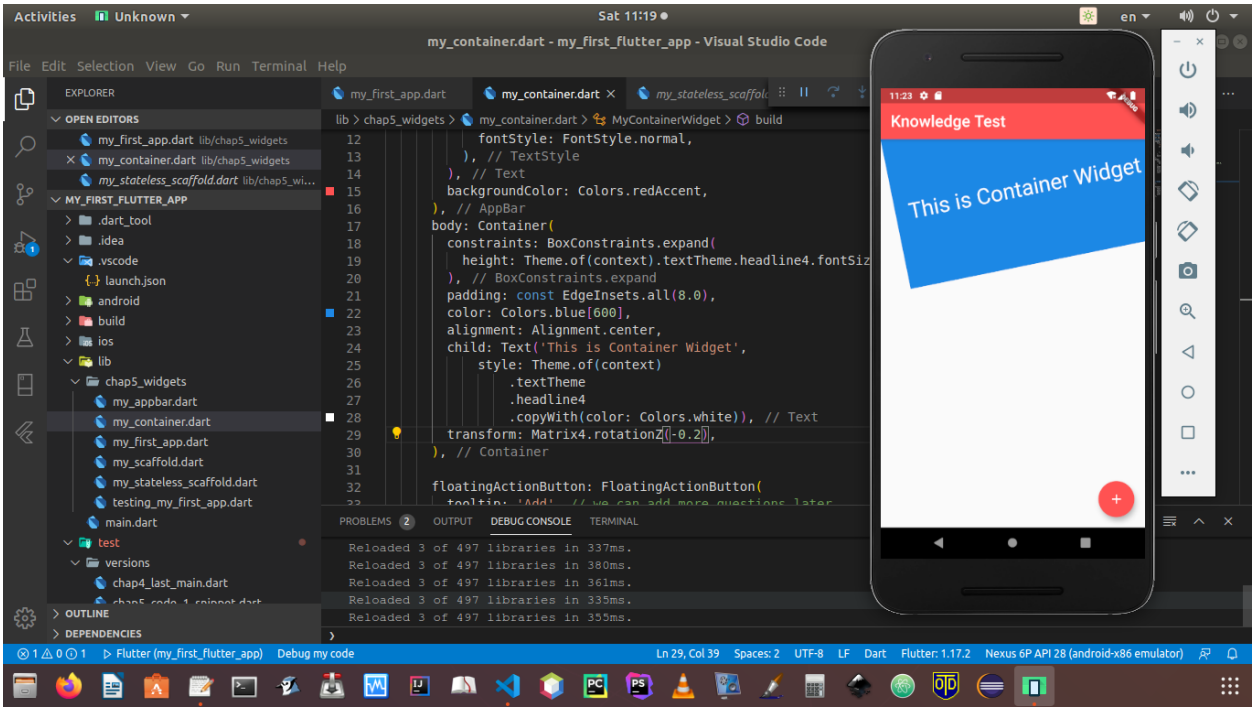

Figure 5.7 – A sample of Container Widget

Just like Container class, the Icon widget or class also plays a major role to build the design of our application UI. By default Icon class is not interactive. If we want to make it interactive we need to use material's 'IconButton' widget.

We will see to that later in our application. Here we again come back to our original Flutter application, the only difference is we have used Icon widget inside the 'RaisedButton' widget.

We will see this code snippet where many things have been repeated. For brevity, we are not going to repeat other snippets, such as 'main.dart', etc. We want to see only the 'MyStatelessScaffoldWidget' class, where we have used the Icon class widget.

```dart
//code 5.5
//my_stateless_scaffold.dart
import 'package:flutter/material.dart';


class MyStatelessScaffoldWidget extends StatelessWidget {
MyStatelessScaffoldWidget({Key key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    key: scaffoldKey,
    appBar: AppBar(
        actions: <Widget>[
        IconButton(
            icon: const Icon(Icons.add_alert),
            tooltip: 'Show Snackbar',
            onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarOne);
            },
        ),
        IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarTwo);
            },
        ),
        IconButton(
            icon: const Icon(Icons.navigate_next),
            tooltip: 'Next page',
            onPressed: () {
            clickNextPage(context);
            },
        ),
        ],
        leading: IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarThree);
        },
        ),
```

```
44          title: Text(
45          'Knowledge Test',
46          style: TextStyle(
47              fontSize: 25.00,
48              fontStyle: FontStyle.normal,
49          ),
50          ),
51          backgroundColor: Colors.redAccent,
52      ),
53      body: Column(
54          children: [
55          Text(
56              'Answer a few questions and know your level...',
57              textAlign: TextAlign.center,
58              style: TextStyle(
59              fontSize: 25,
60              ),
61          ),
62          RaisedButton(
63              child: Text(
64              'You have chosen answer 1',
65              style: TextStyle(
66                  fontSize: 22,
67                  color: Colors.blueGrey,
68              ),
69              ),
70              disabledColor: Colors.redAccent,
71              onPressed: null,
72          ),
73          Icon(
74              Icons.favorite,
75              color: Colors.pink,
76              size: 24.0,
77              semanticLabel: 'Text to announce in accessibility modes',
78          ),
79          RaisedButton(
80              child: Text(
81              'You have chosen answer 2',
82              style: TextStyle(
83                  fontSize: 22,
84                  color: Colors.blueGrey,
85              ),
86              ),
```

```
87              disabledColor: Colors.redAccent,
88              onPressed: null,
89          ),
90          Icon(
91              Icons.audiotrack,
92              color: Colors.green,
93              size: 30.0,
94          ),
95          RaisedButton(
96              child: Text(
97              'You have chosen answer 3',
98              style: TextStyle(
99                  fontSize: 22,
100                 color: Colors.blueGrey,
101             ),
102             ),
103             disabledColor: Colors.redAccent,
104             onPressed: null,
105         ),
106         Icon(
107             Icons.beach_access,
108             color: Colors.blue,
109             size: 36.0,
110         ),
111         ],
112     ),
113     floatingActionButton: FloatingActionButton(
114         tooltip: 'Add', // we can add more questions later
115         backgroundColor: Colors.redAccent,
116         child: Icon(Icons.add),
117         onPressed: null,
118     ),
119     );
120 }
121 }
```

The change in the above code snippet automatically changes the look of our application that we have been building (Figure 5.8).

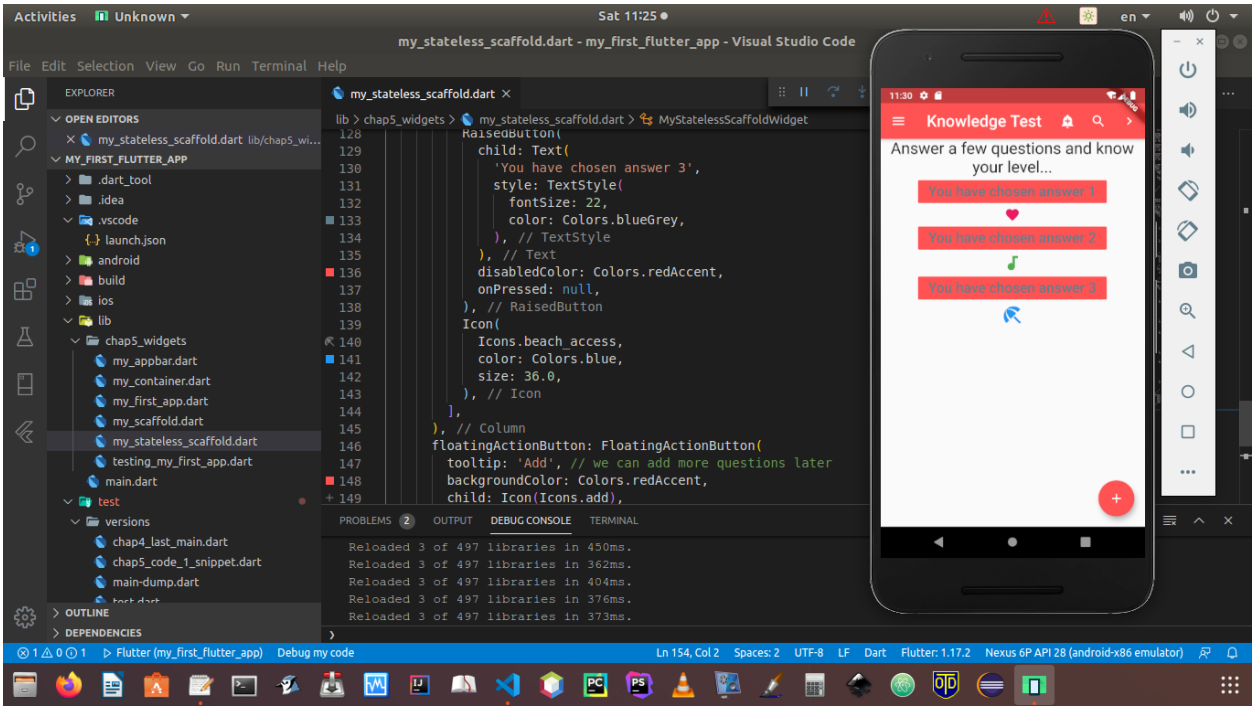

Figure 5.8 – Icon widgets have changed the look of the application

We will see to more widgets later, as we will keep building our Flutter application.

Before closing this chapter, we will take a look at the anonymous or lambda functions of Dart. Besides that, we will also try to understand some core features of object-oriented programming in Dart. It includes, parent-child relationship, extending a class, and many more.

Anonymous Functions: Lambda, Higher Order Functions, and Lexical Closures

Lambda, Higher Order functions, and Lexical Closures have some similarities. In their namelessness and anonymity, these features of Dart are very interesting. Let us start with Lambda. Then we will discuss Higher Order functions and Closures. In reality, you will find that Lambda actually implements Higher-Order Functions. This features have been widely used in our Flutter project.

As the name suggests, Lambda is a nameless function and we can use it in two ways. We can use it in a traditional method and also we can use the 'Fat Arrow'. Consider the first code snippet:

```
//code 5.6
class LambdaCode{
// here addingTwonumbers is a nameless function
Function addingTwonumbers = (int x, int y){
    var sum = x + y;
    return sum;
};
}
main(List<String> arguments){
var lambdaShow = LambdaCode();
print(lambdaShow.addingTwonumbers(12, 47));
}
```

We will give the output after adding the 'Fat Arrow' method. The whole code snippet looks like this:

```
//code 5.7
class LambdaCode{
// here addingTwonumbers is a nameless function
Function addingTwonumbers = (int x, int y){
    var sum = x + y;
    return sum;
};
Function divideByFour = (int num) => num ~/ 4;
}
main(List<String> arguments){
var lambdaShow = LambdaCode();
print(lambdaShow.addingTwonumbers(12, 47));
print(lambdaShow.divideByFour(56));
}
```

The output is quite expected, in the first anonymous function 'Function addingTwonumbers' we have passed two parameters and added them. And using the 'Fat Arrow' method, we have passed a number through another nameless function 'Function divideByFour' and divided it by 4.

```
//output
59
14
```

While building a native iOS or Android app, we have seen how these nameless functions come to your help. Remember the 'onPress' named parameter that points to the anonymous function.

Exploring Higher-Order Functions

The specialty of Higher Order functions is it can accept a function as a parameter. That is why it is named the Higher Order Function. It not only can accept a function as a parameter, it can also return it; actually, it can do both. This concept also has widely used in Flutter tools. We will see a very simple code snippet to get accustomed to the idea.

```
1  //code 5.8
2  //returning a function
3  Function DividingByFour(){
4  Function LetUsDivide = (int x) => x ~/ 4;
5  return LetUsDivide;
6  }
7  main(List<String> arguments){
8  var result = DividingByFour();
9  print(result(56));
10 }
11
12 The output is 14.
```

So we have passed a nameless function 'Function LetUsDivide' as a parameter and returned the value of the division through a higher order function 'Function DividingByFour()'.

Inheritance and Mixins in Dart

One of the key features of object-oriented programming is being able to extend your classes. We extend to create a class and the extended class is known as a subclass. The subclass inherits reference variables and class methods from the parent class, which is known as a superclass. In our Flutter project we have seen a lot extensions. One Widget class extends other class that also extends other, and the process goes on.

Consider this simple example where we have extended an Animal class to a Cat class.

```
//code 5.9
class Animal {
String name = "Animal";
Animal(){
    print("I am Animal class constructor.");
}
Animal.namedConstructor(){
    print("This is parent animal named constructor.");
}
void showName(){
    print(this.name);
}
void eat(){
    print("Animals eat everything depending on whay type it is.");
}
}
class Cat extends Animal {
//overriding parent constructor
//although constructors are not inherited
Cat() : super(){
    print("I am child cat class overriding super Animal class.");
}
Cat.namedCatConstructor() : super.namedConstructor(){
    print("The child cat named constructor overrides the parent animal named constru\
ctor.");
}
@override
void showName(){
    print("Hi from cat.");
}
@override
void eat(){
    super.eat();
    print("Cat doesn't eat vegetables..");
}
}
main(List<String> arguments){
var cat = Cat();
cat.name = "Meaow";
cat.showName();
cat.eat();
var anotherCat = Cat.namedCatConstructor();
}
```

Watching these code snippets, automatically helps us to remember one Widget class extends either 'StatelessWidget',or 'StatefulWidget'.

Let us first see the output and after that we will discuss the features of subclass and superclass.

```
//output
I am Animal class constructor.
I am child cat class overriding super Animal class.
Hi from cat.
Animals eat everything depending on what type it is.
Cat doesn't eat vegetables..
This is parent animal named constructor.
The child cat named constructor overrides the parent animal named constructor.
```

The code is quite simple to follow; the superclass 'Animal' has two constructors: default and named constructor. The subclass 'Cat' overrides both the constructors. Superclass constructors are not inherited. Therefore, a subclass always has its own constructor; either default parameterized or named one. However, a subclass can always override the superclass constructors. That is what we have done here.

Mixins: Adding more Features to a Class

Dart has lot to offer when re-usability of classes are needed; there is a very important concept called 'mixins'. It is a way of reusing any class' code in multiple class hierarchies. We have seen the same features while building our Flutter project.

We can rewrite the above code using 'mixins'. All we need to do is use the keyword 'with'. Suppose we have another class 'Dog' that has a method 'canRun()'. A cat object can also run, isn't it? Let us try the same code in a slight different way.

```
//code 5.10
class Animal {
String name = "Animal";
Animal(){
    print("I am Animal class constructor.");
}
Animal.namedConstructor(){
    print("This is parent animal named constructor.");
}
void showName(){
    print(this.name);
}
void eat(){
```

```
    print("Animals eat everything depending on what type it is.");
}
}
class Dog {
void canRun(){
    print("I can run.");
}
}
class Cat extends Animal with Dog {
//overriding parent constructor
//although constructors are not inherited
Cat() : super(){
    print("I am child cat class overriding super Animal class.");
}
Cat.namedCatConstructor() : super.namedConstructor(){
    print("The child cat named constructor overrides the parent animal named constru\
ctor.");
}
@override
void showName(){
    print("Hi from cat.");
}
@override
void eat(){
    super.eat();
    print("Cat doesn't eat vegetables..");
}
}
main(List<String> arguments){
var cat = Cat();
cat.name = "Meaow";
cat.showName();
cat.eat();
var anotherCat = Cat.namedCatConstructor();
anotherCat.canRun();
}
```

The subclass ‘Cat’ has been extended and at the same it has used ‘mixins’ by reusing the ‘Dog’ class’ code. Watch this line:

```
class Cat extends Animal with Dog {…}
```

And in the main() function the ‘Cat’ object uses the ‘Dog’ class’ method this way:

```
anotherCat.canRun();
```

The output has not been changed except the last line:

```
//output
I am Animal class constructor.
I am child cat class overriding super Animal class.
Hi from cat.
Animals eat everything depending on what type it is.
Cat doesn't eat vegetables..
This is parent animal named constructor.
The child cat named constructor overrides the parent animal named constructor.
I can run.
```

Remember, for 'mixin' we need to use the 'with' keyword followed by one or more 'mixin' names. Support for the 'mixin' keyword was introduced in Dart 2.1. Before that, in such cases, the abstract class was used.

We will learn how to use abstract class in the coming chapters.

Want to read more Flutter related Articles and resources?

For more Flutter related Articles and Resources[11]

[11] https://sanjibsinha.com

6. Layouts in Flutter, Tips and Tricks

Any type of UI design is always concerned with layouts. Flutter's layout mechanism, in its core, has nothing but widgets. We have found it earlier, in Flutter, almost everything is widgets.

For more Flutter related Articles and Resources[12]

While designing a layout, we need mainly images, icons, text, etc. These are all visible widgets. However, there are a few invisible widgets, too. These inconspicuous widgets play crucial roles in building a proper layout by placing all the visible widgets in right places. Widgets like Column, Row, Grid, and many others have the quality of not being perceivable by the eye.

We create a layout by using different types of widgets to create more complex widgets.

For example, the first screenshot of this chapter below shows two icons with a label under each one (Figure 6.1).

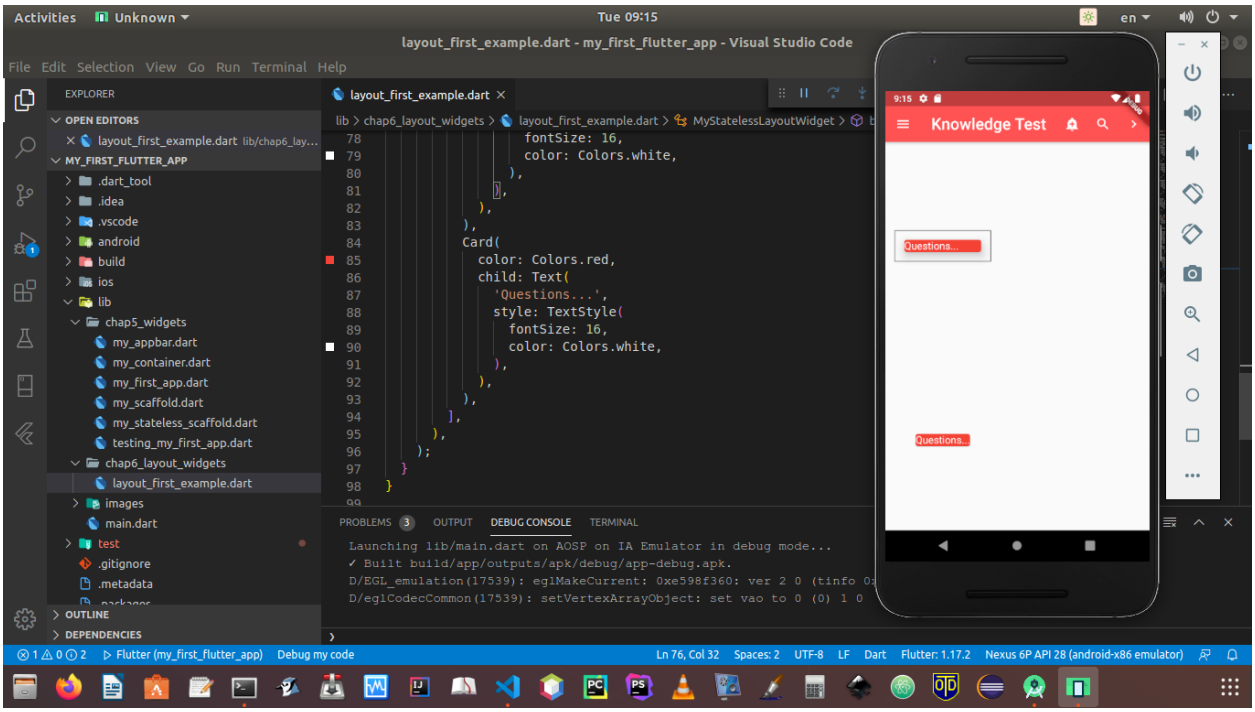

Figure 6.1 – creating a layout by using widgets

Next, the below code snippets show us how we have used column, container and card widgets to create this simple basic layout.

//code 6.1 // layout_first_example.dart

[12] https://sanjibsinha.com

import ‘package:flutter/material.dart’;

class MyStatelessLayoutWidget extends StatelessWidget { @override Widget build(Object context) {

```
  return Scaffold(
    appBar: AppBar(
      actions: <Widget>[
        IconButton(
          icon: const Icon(Icons.add_alert),
          tooltip: 'Show Snackbar',
          onPressed: () {
            Text(
              'Nothing',
            );
          },
        ),
        IconButton(
          icon: Icon(Icons.search),
          tooltip: 'Search',
          onPressed: () {
            Text(
              'Nothing',
            );
          },
        ),
        IconButton(
          icon: const Icon(Icons.navigate_next),
          tooltip: 'Next page',
          onPressed: () {
            Text(
              'Nothing',
            );
          },
        ),
      ],
      leading: IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: () {
          Text(
            'Nothing',
          );
```

```
        },
      ),
      title: Text(
        'Knowledge Test',
        style: TextStyle(
          fontSize: 25.00,
          fontStyle: FontStyle.normal,
        ),
      ),
      backgroundColor: Colors.redAccent,
    ),
    body: Column(
      mainAxisAlignment: MainAxisAlignment.spaceAround,
      children: <Widget>[
        Container(
          margin: EdgeInsets.symmetric(
            vertical: 10,
            horizontal: 15,
          ),
          decoration: BoxDecoration(
            border: Border.all(
              color: Colors.black,
              width: 1,
            ),
          ),
          padding: EdgeInsets.all(10),
          width: 150,
          child: Card(
            elevation: 12,
            color: Colors.red,
            child: Text(
              'Questions...',
              style: TextStyle(
                fontSize: 16,
                color: Colors.white,
              ),
            ),
          ),
        ),
        Card(
          color: Colors.red,
          child: Text(
            'Questions...',
```

```
            style: TextStyle(
              fontSize: 16,
              color: Colors.white,
            ),
          ),
        ),
      ],
    ),
  );
}
```

}

// my_first_app.dart

import ‘package:flutter/material.dart’; import ‘package:my_first_flutter_app/chap6_layout_widgets/layout_first_example.dart’;

class MyFirstApp extends StatelessWidget {

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    home: MyStatelessLayoutWidget()
  );
}
```

}

As we progress, these code snippets will grow bigger and bigger. We will try to maintain the modularity so that we can understand what is happening under the hood.

In the above code, we will take a look at the ‘body’ section, where the actual layout part is getting built. The ‘body’ named parameter primarily points to column widget.

body: Column(mainAxisAlignment: MainAxisAlignment.spaceAround, children: <Widget>[Container(...

First of all, we can adjust the alignment of the column through another widgets. Then comes the ‘children’, a named parameter, and a widget that holds a list of other widgets. At the very beginning we see container widget.

Inside the container widget, we have a ‘child’ widget ‘Card’.

child: Card(elevation: 12,
color: Colors.red, child: Text(‘Questions...’, style: TextStyle(fontSize: 16, color: Colors.white,),),),

After that, we come out of the container widget, and again use another card widget. We can move forward to see whether, we can create more complex layout, step by step, in this chapter.

The next screenshot shows you how we try to organize the quiz app (Figure 6.2) using other layout widgets.

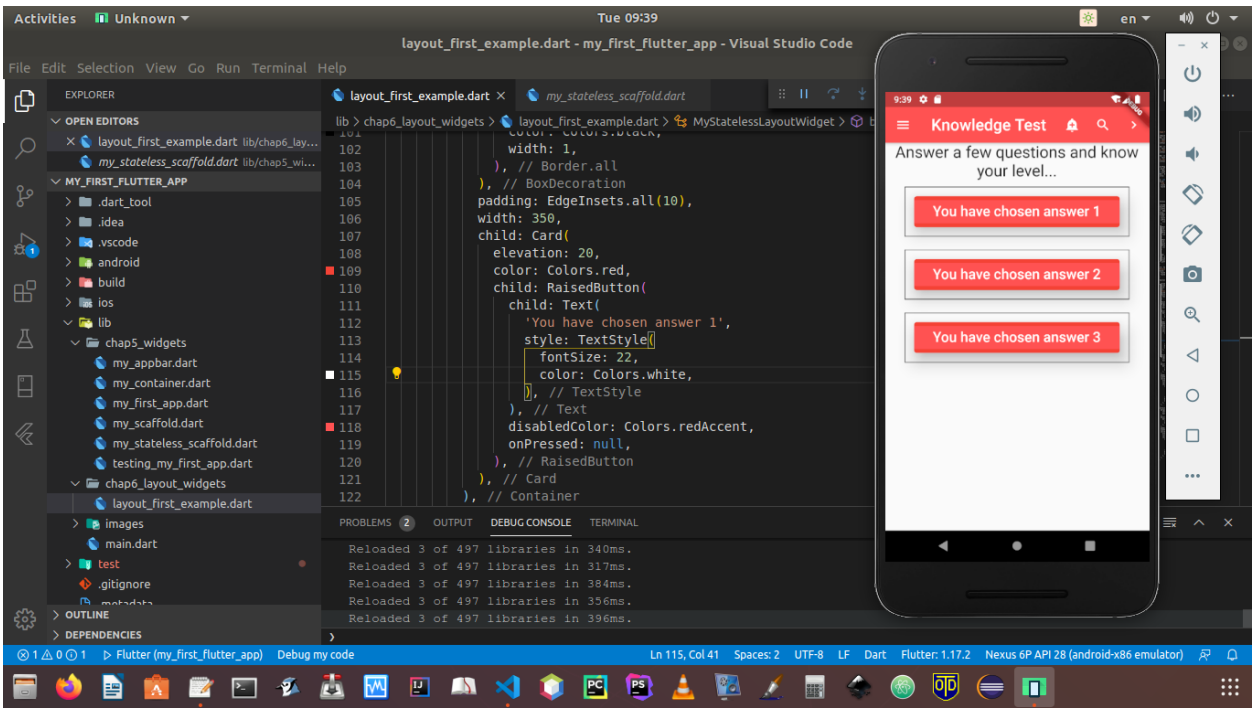

Figure 6.2 – creating complex layout step-by-step

We have also changed our code adding more layout related widgets.

Before we go to the next code snippets, let us take a close look at the Container widget. It is one of the most common and important layout widgets that we will use again and again.

Customize child Widgets

The Container widget class allows you to customize its child widgets. Therefore, it is very handy to create a complex layout using the Container as the root widget of the body sections. There are other advantages of using the Container widget. We can control padding, margins, borders, background colors and many more other capabilities.

```dart
//code 6.2
// layout_first_example.dart

import 'package:flutter/material.dart';

final GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>();
final SnackBar snackBarOne = const SnackBar(
    content: Text(
    'Alert has been pressed!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarTwo = const SnackBar(
    content: Text(
    'Search has been pressed!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarThree = const SnackBar(
    content: Text(
    'Navigation has been pressed!',
    style: TextStyle(fontSize: 30),
));

void clickNextPage(BuildContext context) {
    Navigator.push(context, MaterialPageRoute(
    builder: (BuildContext context) {
        return Scaffold(
        appBar: AppBar(
            title: const Text('Know Yourself...'),
        ),
        body: const Center(
            child: Text(
            'Dig deep into every layer of your mind to find yourself...',
            style: TextStyle(fontSize: 24),
            textAlign: TextAlign.center,
            ),
        ),
        );
    },
    ));
}

class MyStatelessLayoutWidget extends StatelessWidget {
    @override
```

```
44      Widget build(Object context) {
45      return Scaffold(
46          key: scaffoldKey,
47          appBar: AppBar(
48          actions: <Widget>[
49              IconButton(
50              icon: const Icon(Icons.add_alert),
51              tooltip: 'Show Snackbar',
52              onPressed: () {
53                  scaffoldKey.currentState.showSnackBar(snackBarOne);
54              },
55              ),
56              IconButton(
57              icon: Icon(Icons.search),
58              tooltip: 'Search',
59              onPressed: () {
60                  scaffoldKey.currentState.showSnackBar(snackBarTwo);
61              },
62              ),
63              IconButton(
64              icon: const Icon(Icons.navigate_next),
65              tooltip: 'Next page',
66              onPressed: () {
67                  clickNextPage(context);
68              },
69              ),
70          ],
71          leading: IconButton(
72              icon: Icon(Icons.menu),
73              tooltip: 'Navigation menu',
74              onPressed: () {
75              scaffoldKey.currentState.showSnackBar(snackBarThree);
76              },
77          ),
78          title: Text(
79              'Knowledge Test',
80              style: TextStyle(
81              fontSize: 25.00,
82              fontStyle: FontStyle.normal,
83              ),
84          ),
85          backgroundColor: Colors.redAccent,
86          ),
```

```
87          body: Column(
88          mainAxisAlignment: MainAxisAlignment.start,
89          children: <Widget>[
90              Text(
91              'Answer a few questions and know your level...',
92              textAlign: TextAlign.center,
93              style: TextStyle(
94                  fontSize: 25,
95              ),
96              ),
97              Container(
98              margin: EdgeInsets.symmetric(
99                  vertical: 10,
100                 horizontal: 15,
101             ),
102             decoration: BoxDecoration(
103                 border: Border.all(
104                 color: Colors.black,
105                 width: 1,
106                 ),
107             ),
108             padding: EdgeInsets.all(10),
109             width: 350,
110             child: Card(
111                 elevation: 20,
112                 color: Colors.red,
113                 child: RaisedButton(
114                 child: Text(
115                     'You have chosen answer 1',
116                     style: TextStyle(
117                     fontSize: 22,
118                     color: Colors.white,
119                     ),
120                 ),
121                 disabledColor: Colors.redAccent,
122                 onPressed: null,
123                 ),
124             ),
125             ),
126             Container(
127             margin: EdgeInsets.symmetric(
128                 vertical: 10,
129                 horizontal: 15,
```

```
            ),
            decoration: BoxDecoration(
                border: Border.all(
                color: Colors.black,
                width: 1,
                ),
            ),
            padding: EdgeInsets.all(10),
            width: 350,
            child: Card(
                elevation: 20,
                color: Colors.red,
                child: RaisedButton(
                child: Text(
                    'You have chosen answer 2',
                    style: TextStyle(
                    fontSize: 22,
                    color: Colors.white,
                    ),
                ),
                disabledColor: Colors.redAccent,
                onPressed: null,
                ),
            ),
            ),
            Container(
            margin: EdgeInsets.symmetric(
                vertical: 10,
                horizontal: 15,
            ),
            decoration: BoxDecoration(
                border: Border.all(
                color: Colors.black,
                width: 1,
                ),
            ),
            padding: EdgeInsets.all(10),
            width: 350,
            child: Card(
                elevation: 20,
                color: Colors.red,
                child: RaisedButton(
                child: Text(
```

```
                    'You have chosen answer 3',
                    style: TextStyle(
                    fontSize: 22,
                    color: Colors.white,
                    ),
                ),
                disabledColor: Colors.redAccent,
                onPressed: null,
                ),
            ),
            ),
        ],
        ),
    );
    }
}
```

Let us examine the the Container part especially, where we have added such capabilities that we have just discussed.

We take one instance of Container class widget. We have used three to get the display.

Container(margin: EdgeInsets.symmetric(vertical: 10, horizontal: 15,), decoration: BoxDecoration(border: Border.all(color: Colors.black, width: 1,),), padding: EdgeInsets.all(10), width: 350, child: Card(elevation: 20, color: Colors.red, child: RaisedButton(child: Text('You have chosen answer 3', style: TextStyle(fontSize: 22, color: Colors.white,),), disabledColor: Colors.redAccent, onPressed: null,),),),

We have added margins, padding, width, etc. Inside the Container class widget we have used different types of 'child' widgets, such as Card. Inside Card class widget we have used another 'child' widget 'RaisedButton'.

Layout mechanism of Flutter

Flutter's layout mechanism is controlled by the widgets. In the above example, each Text widget is placed in a RaisedButton widget, which is inside the Card widget, and the Card widget is again placed inside the Container widget.

While adding these widget trees, we have controlled many layout capabilities, such as the padding, margins, width, etc. And by this way, we have designed our UI.

Let us modify our code so that we could make our UI more interactive with the help of 'SnackBar' and 'ScaffoldState' widget classes. We have also added an extra Dart file 'questions.dart', where we have defined a 'Questions' class and through the constructor passed a named parameter.

```dart
//code 6.3
// layout_first_example.dart

import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/questions.dart';

final GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>();
final SnackBar snackBarOne = const SnackBar(
    content: Text(
    'Alert has been pressed!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarTwo = const SnackBar(
    content: Text(
    'Search has been pressed!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarThree = const SnackBar(
    content: Text(
    'Navigation has been pressed!',
    style: TextStyle(fontSize: 30),
));

void clickNextPage(BuildContext context) {
    Navigator.push(context, MaterialPageRoute(
    builder: (BuildContext context) {
        return Scaffold(
        appBar: AppBar(
            title: const Text('Know Yourself...'),
        ),
        body: const Center(
            child: Text(
            'Dig deep into every layer of your mind to find yourself...',
            style: TextStyle(fontSize: 24),
            textAlign: TextAlign.center,
            ),
        ),
        );
    },
    ));
}

class MyStatelessLayoutWidget extends StatelessWidget {
```

```
44      final questions = [
45      Questions(questions: 'Are you impulsive?'),
46      Questions(questions: 'Do you get angry easily?'),
47      Questions(questions: 'Are you sloth?'),
48      Questions(questions: 'Do you cheat others?'),
49      ];
50      @override
51      Widget build(Object context) {
52      return Scaffold(
53          key: scaffoldKey,
54          appBar: AppBar(
55          actions: <Widget>[
56              IconButton(
57              icon: const Icon(Icons.add_alert),
58              tooltip: 'Show Snackbar',
59              onPressed: () {
60                  scaffoldKey.currentState.showSnackBar(snackBarOne);
61              },
62              ),
63              IconButton(
64              icon: Icon(Icons.search),
65              tooltip: 'Search',
66              onPressed: () {
67                  scaffoldKey.currentState.showSnackBar(snackBarTwo);
68              },
69              ),
70              IconButton(
71              icon: const Icon(Icons.navigate_next),
72              tooltip: 'Next page',
73              onPressed: () {
74                  clickNextPage(context);
75              },
76              ),
77          ],
78          leading: IconButton(
79              icon: Icon(Icons.menu),
80              tooltip: 'Navigation menu',
81              onPressed: () {
82              scaffoldKey.currentState.showSnackBar(snackBarThree);
83              },
84          ),
85          title: Text(
86              'Knowledge Test',
```

```
            style: TextStyle(
            fontSize: 25.00,
            fontStyle: FontStyle.normal,
            ),
        ),
        backgroundColor: Colors.redAccent,
        ),
        body: Column(
        mainAxisAlignment: MainAxisAlignment.start,
        children: <Widget>[
            Text(
            '${questions[0].questions}',
            textAlign: TextAlign.center,
            style: TextStyle(
                fontSize: 25,
            ),
            ),
            Container(
            margin: EdgeInsets.symmetric(
                vertical: 10,
                horizontal: 15,
            ),
            decoration: BoxDecoration(
                border: Border.all(
                color: Colors.black,
                width: 1,
                ),
            ),
            padding: EdgeInsets.all(10),
            width: 350,
            child: Card(
                elevation: 20,
                color: Colors.red,
                child: RaisedButton(
                child: Text(
                    'No. Not at all...',
                    style: TextStyle(
                    fontSize: 22,
                    color: Colors.white,
                    ),
                ),
                disabledColor: Colors.redAccent,
                onPressed: null,
```

```
130                 ),
131             ),
132             ),
133             Container(
134             margin: EdgeInsets.symmetric(
135                 vertical: 10,
136                 horizontal: 15,
137             ),
138             decoration: BoxDecoration(
139                 border: Border.all(
140                 color: Colors.black,
141                 width: 1,
142                 ),
143             ),
144             padding: EdgeInsets.all(10),
145             width: 350,
146             child: Card(
147                 elevation: 20,
148                 color: Colors.red,
149                 child: RaisedButton(
150                 child: Text(
151                     'I try to control it...',
152                     style: TextStyle(
153                     fontSize: 22,
154                     color: Colors.white,
155                     ),
156                 ),
157                 disabledColor: Colors.redAccent,
158                 onPressed: null,
159                 ),
160             ),
161             ),
162             Container(
163             margin: EdgeInsets.symmetric(
164                 vertical: 10,
165                 horizontal: 15,
166             ),
167             decoration: BoxDecoration(
168                 border: Border.all(
169                 color: Colors.black,
170                 width: 1,
171                 ),
172             ),
```

```
            padding: EdgeInsets.all(10),
            width: 350,
            child: Card(
                elevation: 20,
                color: Colors.red,
                child: RaisedButton(
                child: Text(
                    'I am very impulsive.',
                    style: TextStyle(
                    fontSize: 22,
                    color: Colors.white,
                    ),
                ),
                disabledColor: Colors.redAccent,
                onPressed: null,
                ),
            ),
            ),
        ],
        ),
    );
    }
}

//questions.dart

class Questions{
    final String questions;
    Questions({this.questions});
}
```

The next screenshot shows you how our quiz application looks like.

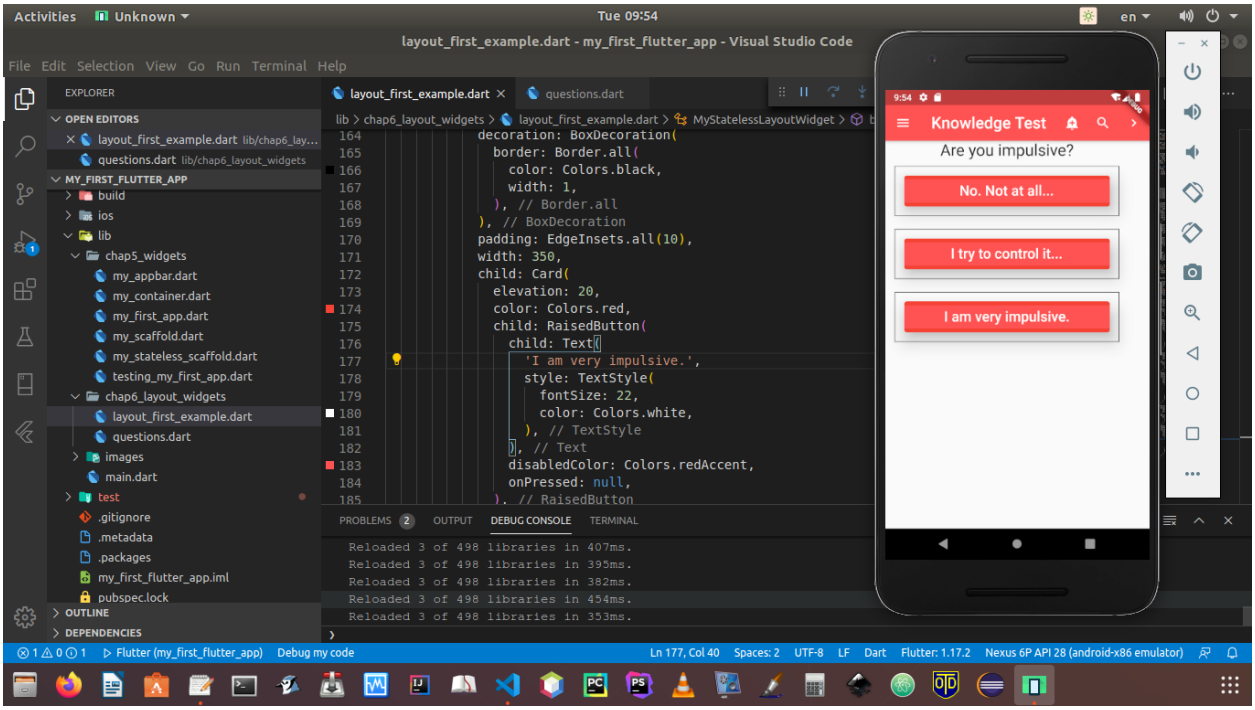

Figure 6.3 – Taking the application to the next level

What happens if we click the buttons? It is now interactive, so at the bottom, it will keep displaying the answers. We are testing our knowledge about ourselves. Therefore, it will give the correct output (Figure 6.4).

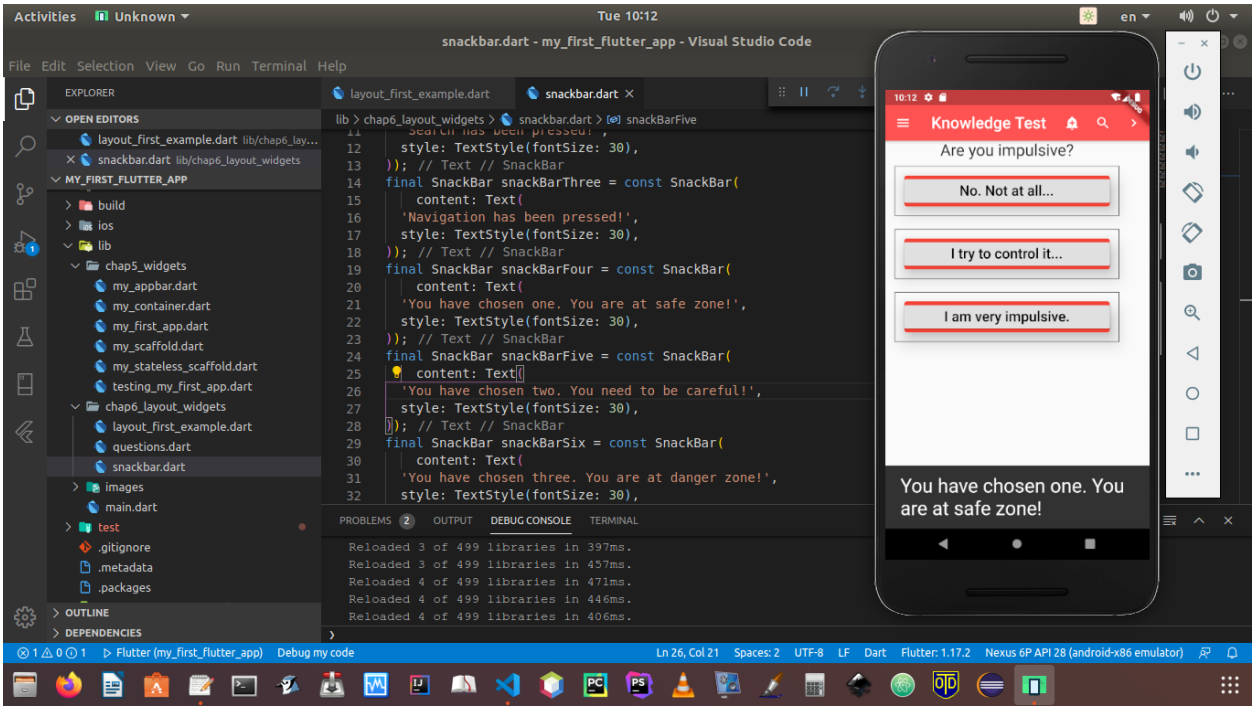

Figure 6.4 – clicking the buttons gives us the related answers

Now we can add more modularity to our code snippets by taking the 'SnackBar' class widget to a different file and import it to our layout first example code, like this.

```
//code 6.4
//snackbar.dart

import 'package:flutter/material.dart';

final SnackBar snackBarOne = const SnackBar(
    content: Text(
    'Alert has been pressed!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarTwo = const SnackBar(
    content: Text(
    'Search has been pressed!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarThree = const SnackBar(
    content: Text(
    'Navigation has been pressed!',
    style: TextStyle(fontSize: 30),
));
```

```
final SnackBar snackBarFour = const SnackBar(
    content: Text(
    'You have chosen one. You are at safe zone!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarFive = const SnackBar(
    content: Text(
    'You have chosen two. You need to be careful!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarSix = const SnackBar(
    content: Text(
    'You have chosen three. You are at danger zone!',
    style: TextStyle(fontSize: 30),
));
//// layout_first_example.dart

import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/questions.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/snackbar.dart';

final GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>();
…
```

The code snippets have been shortened for brevity. However, in the next code snippets, we will use the full code so that we can go the next question.

```
//code 6.5
// layout_first_example.dart

import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/next_page.dart'
    as next_page;
import 'package:my_first_flutter_app/chap6_layout_widgets/questions.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/snackbar.dart';

final GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>();

class MyStatelessLayoutWidget extends StatelessWidget {
    final questions = [
    Questions(questions: 'Are you impulsive?'),
    Questions(questions: 'Do you get angry easily?'),
    Questions(questions: 'Are you sloth?'),
```

```
17      Questions(questions: 'Do you cheat others?'),
18      ];
19      @override
20      Widget build(Object context) {
21      return Scaffold(
22          key: scaffoldKey,
23          appBar: AppBar(
24          actions: <Widget>[
25              IconButton(
26              icon: const Icon(Icons.add_alert),
27              tooltip: 'Show Snackbar',
28              onPressed: () {
29                  scaffoldKey.currentState.showSnackBar(snackBarOne);
30              },
31              ),
32              IconButton(
33              icon: Icon(Icons.search),
34              tooltip: 'Search',
35              onPressed: () {
36                  scaffoldKey.currentState.showSnackBar(snackBarTwo);
37              },
38              ),
39              IconButton(
40              icon: const Icon(Icons.navigate_next),
41              tooltip: 'Next page',
42              onPressed: () {
43                  next_page.clickNextPage(context);
44              },
45              ),
46          ],
47          leading: IconButton(
48              icon: Icon(Icons.menu),
49              tooltip: 'Navigation menu',
50              onPressed: () {
51              scaffoldKey.currentState.showSnackBar(snackBarThree);
52              },
53          ),
54          title: Text(
55              'Knowledge Test',
56              style: TextStyle(
57              fontSize: 25.00,
58              fontStyle: FontStyle.normal,
59              ),
```

```
60          ),
61          backgroundColor: Colors.redAccent,
62          ),
63          body: Column(
64          mainAxisAlignment: MainAxisAlignment.start,
65          children: <Widget>[
66              Text(
67              '${questions[0].questions}',
68              textAlign: TextAlign.center,
69              style: TextStyle(
70                  fontSize: 25,
71              ),
72              ),
73              Container(
74              margin: EdgeInsets.symmetric(
75                  vertical: 10,
76                  horizontal: 15,
77              ),
78              decoration: BoxDecoration(
79                  border: Border.all(
80                  color: Colors.black,
81                  width: 1,
82                  ),
83              ),
84              padding: EdgeInsets.all(10),
85              width: 350,
86              child: Card(
87                  elevation: 20,
88                  color: Colors.red,
89                  child: RaisedButton(
90                  child: Text(
91                      'No. Not at all...',
92                      style: TextStyle(
93                      fontSize: 22,
94                      color: Colors.black,
95                      ),
96                  ),
97                  disabledColor: Colors.redAccent,
98                  onPressed: () {
99                      scaffoldKey.currentState.showSnackBar(snackBarFour);
100                 },
101                 ),
102             ),
```

```
103             ),
104             Container(
105             margin: EdgeInsets.symmetric(
106                 vertical: 10,
107                 horizontal: 15,
108             ),
109             decoration: BoxDecoration(
110                 border: Border.all(
111                 color: Colors.black,
112                 width: 1,
113                 ),
114             ),
115             padding: EdgeInsets.all(10),
116             width: 350,
117             child: Card(
118                 elevation: 20,
119                 color: Colors.red,
120                 child: RaisedButton(
121                 child: Text(
122                     'I try to control it...',
123                     style: TextStyle(
124                     fontSize: 22,
125                     color: Colors.black,
126                     ),
127                 ),
128                 disabledColor: Colors.redAccent,
129                 onPressed: () {
130                     scaffoldKey.currentState.showSnackBar(snackBarFive);
131                 },
132                 ),
133             ),
134             ),
135             Container(
136             margin: EdgeInsets.symmetric(
137                 vertical: 10,
138                 horizontal: 15,
139             ),
140             decoration: BoxDecoration(
141                 border: Border.all(
142                 color: Colors.black,
143                 width: 1,
144                 ),
145             ),
```

```
146             padding: EdgeInsets.all(10),
147             width: 350,
148             child: Card(
149                 elevation: 20,
150                 color: Colors.red,
151                 child: RaisedButton(
152                 child: Text(
153                     'I am very impulsive.',
154                     style: TextStyle(
155                     fontSize: 22,
156                     color: Colors.black,
157                     ),
158                 ),
159                 disabledColor: Colors.redAccent,
160                 onPressed: () {
161                     scaffoldKey.currentState.showSnackBar(snackBarSix);
162                 },
163                 ),
164             ),
165             ),
166             RaisedButton(
167             child: Text(
168                 'Next Question',
169                 style: TextStyle(
170                 fontSize: 22,
171                 color: Colors.blueGrey,
172                 ),
173             ),
174             onPressed: () {
175                 next_page.clickNextQuestion(context);
176             },
177             ),
178             IconButton(
179             icon: const Icon(Icons.navigate_next),
180             tooltip: 'Next Question',
181             onPressed: () {
182                 next_page.clickNextQuestion(context);
183             },
184             ),
185         ],
186         ),
187     );
188     }
```

```
}


//next_page.dart

import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_stateless_scaffold.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/questions.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/snackbar.dart';

void clickNextPage(BuildContext context) {
    Navigator.push(context, MaterialPageRoute(
    builder: (BuildContext context) {
        return Scaffold(
        appBar: AppBar(
            title: const Text('Know Yourself...'),
        ),
        body: const Center(
            child: Text(
            'Dig deep into every layer of your mind to find yourself...',
            style: TextStyle(fontSize: 24),
            textAlign: TextAlign.center,
            ),
        ),
        );
    },
    ));
}

void clickNextQuestion(BuildContext context) {
    Navigator.push(context, MaterialPageRoute(
    builder: (BuildContext context) {

        final questions = [
        Questions(questions: 'Are you impulsive?'),
        Questions(questions: 'Do you get angry easily?'),
        Questions(questions: 'Are you sloth?'),
        Questions(questions: 'Do you cheat others?'),
        ];

        return Scaffold(
        key: scaffoldKey,
        appBar: AppBar(
```

```
232             title: const Text('Know Yourself...'),
233         ),
234         body: Column(
235             mainAxisAlignment: MainAxisAlignment.start,
236             children: <Widget>[
237             Text(
238                 '${questions[1].questions}',
239                 textAlign: TextAlign.center,
240                 style: TextStyle(
241                 fontSize: 25,
242                 ),
243             ),
244             Container(
245                 margin: EdgeInsets.symmetric(
246                 vertical: 10,
247                 horizontal: 15,
248                 ),
249                 decoration: BoxDecoration(
250                 border: Border.all(
251                     color: Colors.black,
252                     width: 1,
253                 ),
254                 ),
255                 padding: EdgeInsets.all(10),
256                 width: 350,
257                 child: Card(
258                 elevation: 20,
259                 color: Colors.red,
260                 child: RaisedButton(
261                     child: Text(
262                     'No. Not at all...',
263                     style: TextStyle(
264                         fontSize: 22,
265                         color: Colors.black,
266                     ),
267                     ),
268                     disabledColor: Colors.redAccent,
269                     onPressed: () {
270                     scaffoldKey.currentState.showSnackBar(snackBarFour);
271                     },
272                 ),
273                 ),
274             ),
```

```
275             Container(
276                 margin: EdgeInsets.symmetric(
277                 vertical: 10,
278                 horizontal: 15,
279                 ),
280                 decoration: BoxDecoration(
281                 border: Border.all(
282                     color: Colors.black,
283                     width: 1,
284                 ),
285                 ),
286                 padding: EdgeInsets.all(10),
287                 width: 350,
288                 child: Card(
289                 elevation: 20,
290                 color: Colors.red,
291                 child: RaisedButton(
292                     child: Text(
293                     'I try to control it...',
294                     style: TextStyle(
295                         fontSize: 22,
296                         color: Colors.black,
297                     ),
298                     ),
299                     disabledColor: Colors.redAccent,
300                     onPressed: () {
301                     scaffoldKey.currentState.showSnackBar(snackBarFive);
302                     },
303                 ),
304                 ),
305             ),
306             Container(
307                 margin: EdgeInsets.symmetric(
308                 vertical: 10,
309                 horizontal: 15,
310                 ),
311                 decoration: BoxDecoration(
312                 border: Border.all(
313                     color: Colors.black,
314                     width: 1,
315                 ),
316                 ),
317                 padding: EdgeInsets.all(10),
```

```
                width: 350,
                child: Card(
                elevation: 20,
                color: Colors.red,
                child: RaisedButton(
                    child: Text(
                    'I cannot control it.',
                    style: TextStyle(
                        fontSize: 22,
                        color: Colors.black,
                    ),
                    ),
                    disabledColor: Colors.redAccent,
                    onPressed: () {
                    scaffoldKey.currentState.showSnackBar(snackBarSix);
                    },
                ),
                ),
            ),
            RaisedButton(
                child: Text(
                'Next Question',
                style: TextStyle(
                    fontSize: 22,
                    color: Colors.blueGrey,
                ),
                ),
                onPressed: () {
                clickNextQuestion(context);
                },
            ),
            IconButton(
                icon: const Icon(Icons.navigate_next),
                tooltip: 'Next Question',
                onPressed: () {
                clickNextQuestion(context);
                },
            ),
            ],
        ),
        );
    },
    ));
```

```
361 }
```

To go to the next question, we need these widgets. We could have used any one of them – either RaisedButton or Icon. We have used both, to get an idea how it works.

RaisedButton(child: Text('Next Question', style: TextStyle(fontSize: 22, color: Colors.blueGrey,),), onPressed: () { next_page.clickNextQuestion(context); },), IconButton(icon: const Icon(Icons.navigate_-next), tooltip: 'Next Question', onPressed: () { next_page.clickNextQuestion(context); },),

Now, after answering the first question, if we press the 'Next Question' button, or press the arrow Icon below, we reach to the next page, where we get the second question. In the 'next_page.dart' file this part of the 'body' widget is important.

final questions = [Questions(questions: 'Are you impulsive?'), Questions(questions: 'Do you get angry easily?'), Questions(questions: 'Are you sloth?'), Questions(questions: 'Do you cheat others?'),];

```
    return Scaffold(
      key: scaffoldKey,
      appBar: AppBar(
        title: const Text('Know Yourself...'),
      ),
      body: Column(
        mainAxisAlignment: MainAxisAlignment.start,
        children: <Widget>[
          Text(
            '${questions[1].questions}',
            textAlign: TextAlign.center,
            style: TextStyle(
              fontSize: 25,
            ),
          ),
```

...

The code is incomplete for brevity. However, we have got the second question for these two lines:

Questions(questions: 'Do you get angry easily?'), '${questions[1].questions}',

Any list index starts with 0. We have tried to get the second element, that is why we have passed the index number 1.

Incidentally, it takes us to the next question, in the next page. The layout mechanism of the next page, or the second question is a little bit simple. We have not worked on the AppBar class widget, this time.

Because it gives us an idea, we have skipped that part. While practicing, and modifying the code, we can of course add more layout capabilities.

The next screenshot will show us how it looks like.

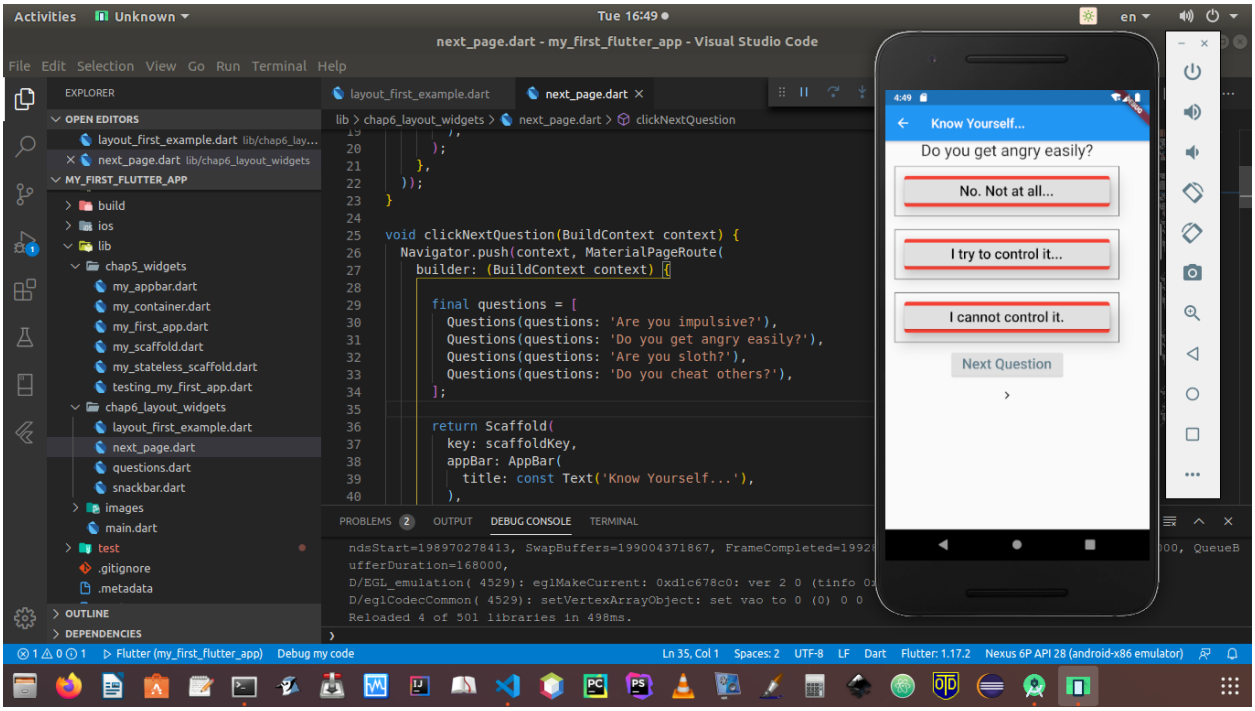

Figure 6.6 – Reaching to the second question

In the next code snippets, we are going to use another widget, Card class widget.

Library of layout widgets

There is a rich library of layout widgets in Flutter. A few of them are commonly used, such as Container, Card, Stack, GridView, ListView, etc.

In Flutter's official documentation the Widget catalog we will get the complete list. There are two categories of layout widgets. We can get the standard widgets from the widgets library, and the specialized widgets from the Material library. One of the most commonly used is Container class widget, where besides using padding, margins, borders or width, we can change the device's background by changing the background color or image. Because we are going to use the Stack class widget in the next code snippets, we will try to understand how this layout widget functions.

When we use Stack widget, the first one is base widget. It could be an image, or any type of colored shapes, where we can use icons, etc. The second one always overlaps the first one. The third one overlaps the second one. Things go on like this, just like a Stack.

The Stack is a list of children of which the first one is the base widget. We keep on adding the subsequent widgets as children on top of the base widget.

Let us see the final code snippets of this chapter. It will give us an idea of how we can manipulate

the layout mechanism with different types of layout widgets. Let us first see the full code first, after that we will see the screenshot of the UI that has been produced by this code snippets.

//code 6.6 //my_new_layout.dart

import 'package:flutter/material.dart'; import 'package:my_first_flutter_app/chap6_layout_widgets/snackbar_and_page.dart';

class MyNewLayout extends StatelessWidget { MyNewLayout({Key key}) : super(key: key);

```
Widget build(BuildContext context) {
  return Scaffold(
    key: scaffoldKey,
    appBar: AppBar(
      actions: <Widget>[
        IconButton(
          icon: const Icon(Icons.add_alert),
          tooltip: 'Show Snackbar',
          onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarOne);
          },
        ),
        IconButton(
          icon: Icon(Icons.search),
          tooltip: 'Search',
          onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarTwo);
          },
        ),
        IconButton(
          icon: const Icon(Icons.navigate_next),
          tooltip: 'Next page',
          onPressed: () {
            clickNextPage(context);
          },
        ),
      ],
      leading: IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: () {
          scaffoldKey.currentState.showSnackBar(snackBarThree);
        },
      ),
      title: Text(
```

```
36          'War Quiz App',
37          style: TextStyle(
38            fontSize: 25.00,
39            fontStyle: FontStyle.normal,
40          ),
41        ),
42        backgroundColor: Colors.redAccent,
43      ),
44      body: Center(
45        child: Column(
46          mainAxisAlignment: MainAxisAlignment.start,
47          children: <Widget>[
48            Stack(
49              alignment: Alignment.topCenter,
50              children: <Widget>[
51                Container(
52                  margin: EdgeInsets.only(top: 25.00),
53                  height: 60,
54                  width: 60,
55                  decoration: BoxDecoration(
56                    borderRadius: BorderRadius.circular(100.00),
57                    color: Colors.redAccent,
58                  ),
59                  child: Icon(Icons.landscape, color: Colors.brown),
60                ),
61                Container(
62                  margin: EdgeInsets.only(top: 70.00, right: 50.00),
63                  height: 60,
64                  width: 60,
65                  decoration: BoxDecoration(
66                    borderRadius: BorderRadius.circular(100.00),
67                    color: Colors.green,
68                  ),
69                  child: Icon(Icons.keyboard_arrow_down, color: Colors.black),
70                ),
71                Container(
72                  margin: EdgeInsets.only(left: 50.00, top: 70.00),
73                  height: 60,
74                  width: 60,
75                  decoration: BoxDecoration(
76                    borderRadius: BorderRadius.circular(100.00),
77                    color: Colors.blueAccent,
78                  ),
```

```
79                 child: Icon(Icons.keyboard_arrow_up, color: Colors.black),
80               ),
81             ],
82           ),
83           Row(
84             mainAxisAlignment: MainAxisAlignment.center,
85             children: <Widget>[
86               Text(
87                 'Take a Quick War Quiz!',
88                 style: TextStyle(
89                   fontSize: 35.00,
90                   fontStyle: FontStyle.normal,
91                 ),
92               ),
93             ],
94           ),
95           Column(
96             children: [
97               Text(
98                 'Answer a few Questions to test your Knowledge, Scores will decide..\
99 .',
100                textAlign: TextAlign.center,
101                style: TextStyle(
102                  fontSize: 25,
103                ),
104              ),
105              Text(
106                '...EITHER...',
107                style: TextStyle(
108                  fontSize: 22,
109                  fontStyle: FontStyle.italic,
110                  color: Colors.deepOrangeAccent,
111                ),
112              ),
113              RaisedButton(
114                child: Text(
115                  'You are a War Expert!',
116                  style: TextStyle(
117                    fontSize: 22,
118                    color: Colors.white,
119                  ),
120                ),
121                disabledColor: Colors.redAccent,
```

```
122                 onPressed: null,
123               ),
124               Icon(
125                 Icons.favorite,
126                 color: Colors.pink,
127                 size: 24.0,
128               ),
129               Text(
130                 '...OR ...',
131                 style: TextStyle(
132                   fontSize: 22,
133                   fontStyle: FontStyle.italic,
134                   color: Colors.deepOrangeAccent,
135                 ),
136               ),
137               RaisedButton(
138                 child: Text(
139                   'You are a Learned Person!',
140                   style: TextStyle(
141                     fontSize: 22,
142                     color: Colors.white,
143                   ),
144                 ),
145                 disabledColor: Colors.redAccent,
146                 onPressed: null,
147               ),
148               Icon(
149                 Icons.audiotrack,
150                 color: Colors.green,
151                 size: 30.0,
152               ),
153               Text(
154                 '...FINALLY...',
155                 style: TextStyle(
156                   fontSize: 22,
157                   fontStyle: FontStyle.italic,
158                   color: Colors.deepOrangeAccent,
159                 ),
160               ),
161               RaisedButton(
162                 child: Text(
163                   'You need to Study More!',
164                   style: TextStyle(
```

```
                    fontSize: 22,
                    color: Colors.white,
                  ),
                ),
                disabledColor: Colors.redAccent,
                onPressed: null,
              ),
              Row(
                mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                  Expanded(
                    child: Padding(
                      padding: EdgeInsets.all(20.00),
                      child: Container(
                        alignment: Alignment.center,
                        height: 40.00,
                        decoration: BoxDecoration(
                          color: Colors.blueGrey,
                          borderRadius: BorderRadius.circular(30.00),
                        ),
                        child: RaisedButton(
                          child: Text(
                            'Let\'s Start...',
                            style: TextStyle(
                              fontSize: 22,
                              color: Colors.blue,
                            ),
                          ),
                          //disabledColor: Colors.redAccent,
                          onPressed: (){
                            clickNextPage(context);
                          },
                        ),
                      ),
                    ),
                  ),
                ],
              ),
            ],
          ),
        ),
      ),
```

```
208     );
209   }

}
```

// snackbar_and_page.dart

import ‘package:flutter/material.dart’;

final GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>(); final SnackBar snackBarOne = const SnackBar(content: Text(‘Alert has been pressed!’, style: TextStyle(fontSize: 30),)); final SnackBar snackBarTwo = const SnackBar(content: Text(‘Search has been pressed!’, style: TextStyle(fontSize: 30),)); final SnackBar snackBarThree = const SnackBar(content: Text(‘Navigation has been pressed!’, style: TextStyle(fontSize: 30),));

void clickNextPage(BuildContext context) { Navigator.push(context, MaterialPageRoute(builder: (BuildContext context) { return Scaffold(appBar: AppBar(title: const Text(‘Know Yourself…’),), body: const Center(child: Text(‘Dig deep into every layer of your mind to find yourself…’, style: TextStyle(fontSize: 24), textAlign: TextAlign.center,),),); },)); }

The next screenshot shows us how this layout widgets work (Figure 6.7). We have not used any image. We created the logo of the page simply by using Stack and Icons widget classes.

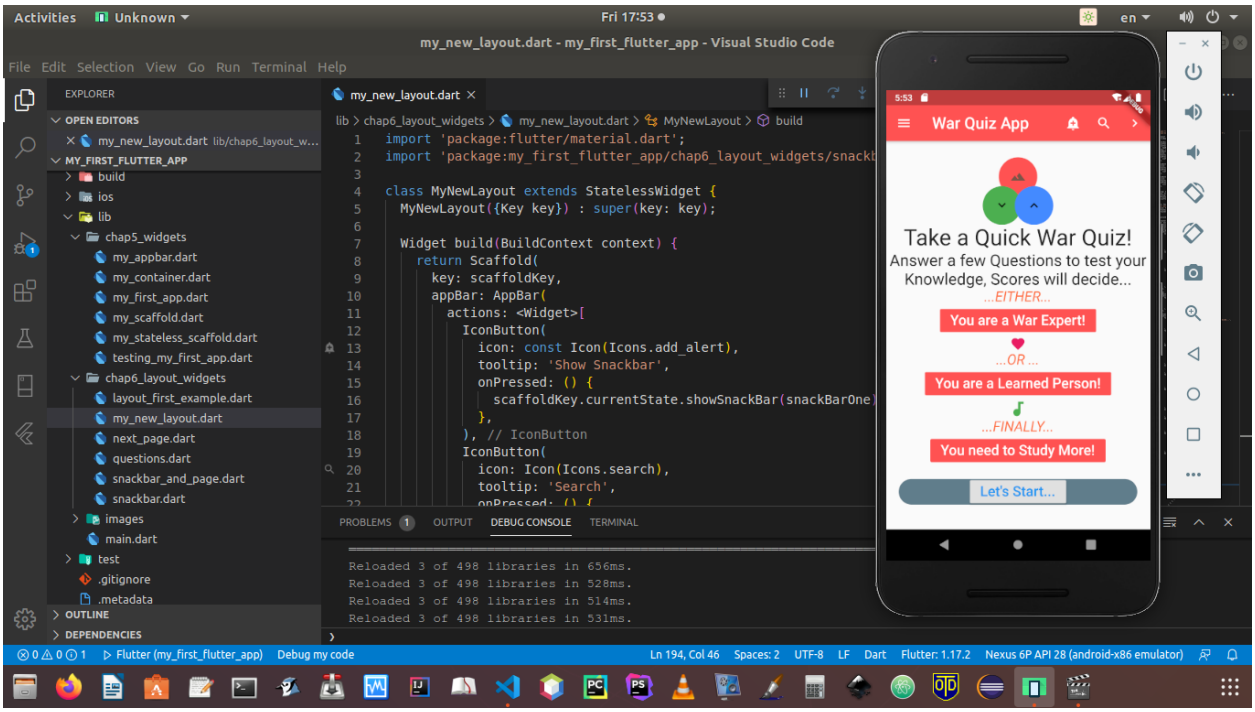

Figure 6.7 – A complete new look of our application

We will take a look at some particular spots in our ‘body’ widget, where we have designed the logo, and after that in a series of Row and Column widget classes we have added more layout capabilities.

body: Center(child: Column(mainAxisAlignment: MainAxisAlignment.start, children: <Widget>[Stack(alignment: Alignment.topCenter, children: <Widget>[Container(margin: EdgeInsets.only(top: 25.00), height: 60, width: 60, decoration: BoxDecoration(borderRadius: BorderRadius.circular(100.00), color: Colors.redAccent,), child: Icon(Icons.landscape, color: Colors.brown),), Container(margin: EdgeInsets.only(top: 70.00, right: 50.00), height: 60, width: 60, decoration: BoxDecoration(borderRadius: BorderRadius.circular(100.00), color: Colors.green,), child: Icon(Icons.keyboard_arrow_down, color: Colors.black),), Container(margin: EdgeInsets.only(left: 50.00, top: 70.00), height: 60, width: 60, decoration: BoxDecoration(borderRadius: BorderRadius.circular(100.00), color: Colors.blueAccent,), child: Icon(Icons.keyboard_arrow_up, color: Colors.black),),],), Row(mainAxisAlignment: MainAxisAlignment.center, children: <Widget>[Text('Take a Quick War Quiz!', style: TextStyle(fontSize: 35.00, fontStyle: FontStyle.normal,),),],), Column(children: [Text('Answer a few Questions to test your Knowledge, Scores will decide…', textAlign: TextAlign.center, style: TextStyle(fontSize: 25,),), Text('…EITHER…', style: TextStyle(fontSize: 22, fontStyle: FontStyle.italic, color: Colors.deepOrangeAccent,),), RaisedButton(child: Text('You are a War Expert!', style: TextStyle(fontSize: 22, color: Colors.white,),), disabledColor: Colors.redAccent, onPressed: null,), Icon(Icons.favorite, color: Colors.pink, size: 24.0,), …

The code has been shortened as we need not repeat the same snippets again and again. Especially watch the Stack class widget, which uses children widget of Container class widgets. We have used three Containers, and by modifying the padding, margins, etc.

Take a close look at one Container class widget that has been placed inside the Stack class widget.

Container(margin: EdgeInsets.only(left: 50.00, top: 70.00), height: 60, width: 60, decoration: BoxDecoration(borderRadius: BorderRadius.circular(100.00), color: Colors.blueAccent,), child: Icon(Icons.keyboard_arrow_up, color: Colors.black),),

By changing the 'EdgeInsets' class widget we can control the position of the Icons. Although we are not going to use this layout when we will create the final application, yet hopefully this layout mechanism gives us an idea of how we can add or modify different type of layout capabilities.

Abstract Class and Methods

Let us look back to some Dart programming concepts again to understand a few more capabilities of Flutter layout mechanism.

An abstract class is something where we define an interface but leaving its implementation up to other classes. Quite naturally, abstract methods can only exist in abstract classes. In abstract methods, we just leave a semicolon (;) at the end of the method name. We don't define the method body.

```
//code 6.7
//we cannot instantiate any abstract class
abstract class volume{
//we can declare instance variable
int age;
void increase();
void decrease();
// a normal function
void anyNormalFunction(int age){
    print("This is a normal function to know the $age.");
}
}
class soundSystem extends volume{
void increase(){
    print("Sound is up.");
}
void decrease(){
    print("Sound is down.");
}
//it is optional to override the normal function
void anyNormalFunction(int age){
    print("This is a normal function to know how old the sound system is: $age.");
}
}
main(List<String> arguments){
var newSystem = soundSystem();
newSystem.increase();
newSystem.decrease();
newSystem.anyNormalFunction(10);
}
And here is the output of code 5.1.
Sound is up.
Sound is down.
This is a normal function to know how old the sound system is: 10.
```

We have used the abstract modifier to define an abstract class and it cannot be instantiated. So we can say, the abstract class and methods summarize the main ideas; and we can extend that idea. There are a few things to remember:

1. In abstract class, we can use normal properties and methods.
2. It is optional that we would override the method.
3. We can also define instance variables in the abstract class.

Advantage of Interfaces

In some cases, we need to use reference variables and methods of many classes at the same time. 'Mixins' may come to our help. No doubt. But there is another good feature in Dart, we can also use – interfaces. Let us see the code first then we will discuss it in detail.

```
1  //code 6.8
2  // interface in dart is class, but we don't extend, we implement it
3  class Vehicle{
4  void steerTheVehicle() {
5      print("The vehicle is moving.");
6  }
7  }
8  class Engine{
9  //in the interface, but only visible when used publicly
10 final _name;
11 //not in the interface, since it is a constructor
12 Engine(this._name);
13 String lessOilConsumption(){
14     return "It consumes less oil.";
15 }
16 }
17 class Car implements Vehicle, Engine{
18 get _name => "";
19 void carName(String name) => print("$name");
20 void steerTheVehicle() {
21     print("The car is moving.");
22 }
23 String lessOilConsumption(){
24     print("This model of car consumes less oil.");
25 }
26 void ridingExperience(Engine engine) => print("This car gives good ride, because the\
27  engine is ${engine._name}");
28 }
29 main(List<String> arguments){
30 var car = Car();
31 car.carName("Opel");
32 car.steerTheVehicle();
33 car.lessOilConsumption();
34 car.ridingExperience(Engine("Suzuki"));
35 }
36 Here is the output of code 6.1:
```

```
Opel
The car is moving.
This model of car consumes less oil.
This car gives a good ride because the engine is Suzuki
```

When a class implements an interface, it implicitly defines all the instance members of the implemented class. A class implements one or more than one interfaces at a time by declaring the 'implement' clause.

Considering the above code, we see that class Car supports class Vehicle and class Engine's API and for that requirement, the class Car implements class Vehicle and class Engine's interfaces.

A class cannot extend more than one classes. But it can implement more than one interface by declaring the implement clause.

Therefore, a few things to remember:

1. The biggest advantage of the interface is that we can implement multiple classes.
2. We cannot inherit multiple classes through inheritance.

Static Variables and Methods

To implement class-wide variables and methods, we use the static keyword. Static variables are also called class variables. Let us first see a code snippet and after that, we will discuss the advantages and disadvantages of static variables and methods.

```
//code 6.9
// static variables and methods consume less memory
// they are lazily initialized
class Circle{
static const pi = 3.14;
static Function drawACircle(){
    //from static method you cannot call a normal function
    print(pi);
}
Function aNonStaticFunction(){
    //from a normal function ou can call a static meethod
    Circle.drawACircle();
    print("This is normal function.");
}
}
main(List<String> arguments){
var circle = Circle();
```

```
circle.aNonStaticFunction();
Circle.drawACircle();
}
And here is the output:
3.14
This is normal function.
3.14
```

As you see, static variables are useful for class-wide state and constants. So in the main() method we can add this line at the end:

```
main(List<String> arguments){
var circle = Circle();
circle.aNonStaticFunction();
Circle.drawACircle();
print(Circle.pi);
}
```

And get the value of constant 'pi' again. Here, 'Circle.pi' is the class variable. And the class method is: 'Circle.drawACircle();'. The biggest advantage of using static variables and methods is it consumes less memory. An instance variable once instantiated, consumes memory whether it is being used or not. The static variables and methods are not initialized until they are used in the program. It consumes memory when they are used.

A few things to remember:

1. From a normal function, you can call a static method.
2. From a static method, you cannot call a normal function.
3. In a static method, you cannot use the 'this' keyword. It is because the static methods do not operate on an instance and thus do not have access to this.

So, in the end, we can conclude that using static variables and methods depend on the context and situations. In the next part of the book, where we will build native iOS and Android mobile apps with the help of Flutter framework, you will see how and when we use static variables and methods.

The 'Closure' is a Special Function

We can define Closure in two ways. According to the first definition, we can say that Closure is the only function that has access to the parent scope, even after the scope is closed.

To understand this definition, let us see a very short code snippet:

```
//code 6.10
//a closure can modify the parent scope
String message = "Any Parent String";
Function overridingParentScope = (){
String message = "Overriding the parent scope";
print(message);
};
main(List<String> arguments){
print(message);
overridingParentScope();
}
The output is:
Any Parent String
Overriding the parent scope
```

By the second definition, we can say that a Closure is a function object that has access to the variables in its lexical scope, even when the function is used outside of its original scope.

```
//code 6.11
Function show = (){
String pathToImage = "This is an old path.";
Function gettingImage(){
    String path = "This is a new path to image.";
    print(path);
}
return gettingImage;
};
main(List<String> arguments){
var showing = show();
showing();
}
```

Here is the output: This is a new path to image.

It actually returns a function object 'gettingImage' that has accessed the variable in its lexical scope. So at the end of this section, we can summarize the points about Closure.

1. In several other languages, you are not allowed to modify the parent variable.
2. However, within a closure, you can mutate or modify the values of variables present in the parent scope.

Now we will conclude our whole journey to study the nameless functions in one single code base, and we will also see the output.

```
//code 6.12
//Lambda is an anonymous function
class AboutLambdas{
//first way of expressing Lambda or anonymous function
Function addingNumbers = (int a, int b){
    var sum = a + b;
    //print(sum);
    return sum;
};
Function multiplyWithEight = (int num){
    return num * 8;
};
//second way of expressing Lambda by Fat Arrow
Function showName = (String name) => name;
//higher order functions pass function as parameter
int higherOrderFunction(Function myFunction){
    int a = 10;
    int b = 20;
    print(myFunction(a, b));
}
//returning a function
Function returningAFunction(){
    Function showAge = (int age) => age;
    return showAge;
}
//a closure can modify the parent scope
String anyString = "Any Parent String";
Function overridingParentScope = (){
    String message = "Overriding the parent scope";
    print(message);
};
Function show = (){
    String pathToImage = "This is an old path.";
    Function gettingImage(){
    String path = "This is a new path to image.";
    print(path);
    }
    return gettingImage;
};
}
main(List<String> arguments){
var add = AboutLambdas();
var addition = add.addingNumbers(5, 10);
```

```
print(addition);
var mul = AboutLambdas();
var result = mul.multiplyWithEight(4);
print(result);
var name = AboutLambdas();
var myName = name.showName("Sanjib");
print(myName);
var higher = AboutLambdas();
var higherOrder = higher.higherOrderFunction(add.addingNumbers);
higherOrder;
var showAge = AboutLambdas();
var showingAge = showAge.returningAFunction();
print(showingAge(25));
var sayMessage = AboutLambdas();
sayMessage.overridingParentScope();
var image = AboutLambdas();
var imagePath = image.show();
imagePath();
}
```

And in the output we will see how the nameless functions work in different ways.

```
//output
15
32
Sanjib
30
25
Overriding the parent scope
This is a new path to image.
```

Data Structures and Collections

Understanding the concepts of data structures and collections, as a whole, plays a crucial role in your future Dart programming. We will see in a minute that there are four types of data structures in Dart: List, Set,Map, and Queue. In my opinion, Lists and Maps will cover almost everything, so you hardly need the other two types in your programming life, except a few occasions.

Anyway, although we have seen an introduction to collections before, we will learn these data structures concepts exclusively in this chapter. We will cover all the concepts of Dart collections in detail.

In a nutshell, data structures are something that helps you to organize information for storage and later retrieval.

Although Sets are not necessary for day-to-day programming like Lists and Maps, it is good on some occasions, especially, when your data elements are unique. We will see them in a minute. Remember, learning data structures properly will definitely make you a good Dart programmer in the future.

So let us start with Lists.

Lists: Fixed Length and Growable

If you are a complete beginner, you may not have heard about 'array'. Of course, you have heard it, if you have already had a programming background. You won't find the terms like 'array' or 'associative array' in Dart. But Dart collections can be used to duplicate the data structures like an array.

Now, what is List?

The list is a simple ordered group of objects. Creating a List seems easy because Dart core libraries have the necessary support and a List class. There are two types of Lists.

1. Fixed Length List
2. Growable List

In the Fixed Length List, the length of Lists cannot change at run-time; however, in the second type, Growable List, the length can change at run-time. We will see them separately.

```
1  //code 6.13
2  int listFunction(){
3  List<int> nameOfTest = List(3);
4  nameOfTest[0] = 1;
5  nameOfTest[1] = 2;
6  nameOfTest[2] = 3;
7  //there are three methods to capture the list
8  //1. method
9  for(int element in nameOfTest){
10     print(element);
11 }
12 print("-----------");
13 //2. method
14 nameOfTest.forEach((v) => print('${v}'));
15 print("-----------");
16 //3. method
```

```
for(int i = 0; i < nameOfTest.length; i++){
    print(nameOfTest[i]);
}
}
main(List<String> arguments){
listFunction();
}
```

As you see this is an ordered list of 3 numbers. We are getting the output by using three methods, very simple and straight forward.

```
//output
1
2
3
-----------
1
2
3
-----------
1
2
3
```

Now we will see an example of Growable List.

```
//code 6.14
Function growableList(){
//1. method
List<String> names = List();
names.add("Mana");
names.add("Babu");
names.add("Gopal");
names.add("Pota");
//there are two methods to capture the list
print("-----------");
//1. method
names.forEach((v) => print('${v}'));
print("-----------");
//2. method
for(int i = 0; i < names.length; i++){
    print(names[i]);
```

```
}
}
main(List<String> arguments){
growableList();
}
```

It is also very straight forward, we have not passed any number through the List() and keeping it open lets us add any number of elements into it. Here we have added a few names. And we can capture the List elements through two methods, instead of three. The output is quite expected.

```
//output
-----------
Mana
Babu
Gopal
Pota
-----------
Mana
Babu
Gopal
Pota
```

So it is evident from the output and the code that Growable Lists are dynamic in nature. We can dynamically add any number of elements and we can also remove it by a simple method: ‘ names.remove(“any name”)’. We can also use the key; as this ordered list starts from 0. So we can remove the first name just by passing this key value: ‘ names.removeAt(0)’. We use the ‘removeAt(key)’ method for that operation. We can also clear the Lists just by typing: ‘ names.clear()’.

Set: An Unordered Collections of Unique Items

The headline says everything. A Set represents a collection of objects in which each object can occur only once; it literally stands for the uniqueness of the items. In the dart core library, there is a Set class that supports to achieve this criterion.

Since Set is an unordered collection of unique items, you cannot get element1s by the INDEX. There is a concept called ‘HashSet’ that actually implements the unordered Set and it is based on hash-table based Set implementation. We will look into those features in a minute.

```
//code 6.15
void setFunction(){
//set is an unordered collections of unique items
//cannot get elements by INDEX since the items are unordered
//1. method of creating Set
Set<String> countries = Set.from(['India', 'England', 'US']);
Set<int> numbers = Set.from([1, 45, 58]);
Set<int> moreNumbers = Set();
moreNumbers.add(178);
moreNumbers.add(568);
moreNumbers.add(569);
//1. method
for(int element in numbers){
    print(element);
}
print("-----------");
//2. method
countries.forEach((v) => print('${v}'));
print("-----------");
for(int element in moreNumbers){
    if(moreNumbers.lookup(178) == 178){
    print(moreNumbers);
    break;
    }
}
//set
var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(fruitCollection.lookup('Something Else'));
//it gives null
//lists
List fruitCollections = ['Mango', 'Apple', 'Jack fruit'];
var myIntegers = [1, 2, 3, 'non-integer object'];
print(myIntegers[3]);
print(fruitCollections[0]);
}
main(List<String> arguments){
setFunction();
}
```

Let us see the output first, then we will be able to understand what happens.

```
//output
1
45
58
-----------
India
England
US
-----------
{178, 568, 569}
null
non-integer object
Mango
```

We have created Set of ‘countries’, ‘numbers’ and ‘morenumbers’; finally we have created a List at the end to distinguish between the characters of Lists and Sets. These three methods have created Lists:

```
Set<String> countries = Set.from(['India', 'England', 'US']);
Set<int> numbers = Set.from([1, 45, 58]);
Set<int> moreNumbers = Set();
```

We get the output of the first one we have by this method:

```
countries.forEach((v) => print('${v}'));
The second List has been retrieved by this method:
for(int element in numbers){
print(element);
}
```

And the values of the third Set we have captured using this method:

```
for(int element in moreNumbers){
if(moreNumbers.lookup(178) == 178){
    print(moreNumbers);
    break;
}
}
```

To manipulate a Set there are lots of methods available in Dart core libraries. You can use ‘moreNumbers.contains(value)’, ‘moreNumbers.remove(value)’ or ‘moreNumbers.isEmpty’ etc.

In this code snippet, the return value is ‘null’, since there is no such value present in the Set.

```
//set
var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(fruitCollection.lookup('Something Else'));
```

We need to remember one thing, when the Set type is integer, it is easier to use 'for' loop to loop over the elements. Otherwise, it will be wise to use 'foreach' as we have used in the above code:

```
countries.forEach((v) => print('${v}'));
```

In the next section we will see how Map in Dart works.

Maps: the Key, Value Pair

An unordered collection of Key-Value pair is known as Map in Dart. The main advantage of Map is the Key-Value pair can be of any type. In the next chapter, where we will discuss state management in Flutter, we will use Map, and the key-value pair.

To begin with, let us start with some points that we should remember while working with Map.

1. Each Key in a Map should be unique.
2. The value can be repeated.
3. The Map can commonly be called hash or dictionary.
4. Size of a Map is not fixed, it can either increase or decrease as per the number of elements. In other words, Maps can grow or shrink at runtime.
5. HashMap is an implementation of a Map and it is based on a Hash table.

Let us see a code snippet to understand how a Map works in Dart.

```
//code 6.16
void mapFunction(){
//unordered collection of key=>value pair
Map<String, String> countries = Map();
countries['India'] = "Asia";
countries["German"] = "Europe";
countries["France"] = "Europe";
countries["Brazil"] = "South America";
//1. method we can obtain key or value
for(var key in countries.keys){
    print("Countries' name: $key");
}
print("-----------");
```

```
for(String value in countries.values){
    print("Continents' name: $value");
}
//2. method
countries.forEach((key, value) => print("Country: $key and Continent: $value"));
//we can update any map very easily
if(countries.containsKey("German")){
    countries.update("German", (value) => "European Union");
    print("Updated country German.");
    countries.forEach((key, value) => print("Country: $key and Continent: $value"));
}
//we can remove any country
countries.remove("Brazil");
countries.forEach((key, value) => print("Country: $key and Continent: $value"));
print("Barzil has been removed successfully.");
print("-----------");
//3. method of creating a map
Map<String, int> telephoneNumbersOfCustomers = {
    "John" : 1234,
    "Mac" : 7534,
    "Molly" : 8934,
    "Plywod" : 1275,
    "Hagudu" : 2534
};
telephoneNumbersOfCustomers.forEach((key, value) => print("Customer: $key and Contac\
t NUmber: $value"));
}
main(List<String> arguments){
mapFunction();
}

And here is the output
Countries' name: India
Countries' name: German
Countries' name: France
Countries' name: Brazil
-----------
Continents' name: Asia
Continents' name: Europe
Continents' name: Europe
Continents' name: South America
Country: India and Continent: Asia
Country: German and Continent: Europe
```

```
Country: France and Continent: Europe
Country: Brazil and Continent: South America
Updated country German.
Country: India and Continent: Asia
Country: German and Continent: European Union
Country: France and Continent: Europe
Country: Brazil and Continent: South America
Country: India and Continent: Asia
Country: German and Continent: European Union
Country: France and Continent: Europe
Barzil has been removed successfully.
-----------
Customer: John and Contact NUmber: 1234
Customer: Mac and Contact NUmber: 7534
Customer: Molly and Contact NUmber: 8934
Customer: Plywod and Contact NUmber: 1275
Customer: Hagudu and Contact NUmber: 2534
```

There are three methods that we use to retrieve the values of a Map.

```
//1. method we can obtain key or value
for(var key in countries.keys){
print("Countries' name: $key");
}
print("-----------");
//2. Method
for(String value in countries.values){
print("Continents' name: $value");
}
//3. method
countries.forEach((key, value) => print("Country: $key and Continent: $value"));
```

Besides, there are several methods to add, update or remove the elements in a Map. As we progress and build apps in native iOS or Android, we will see more features of Map. Lastly we will see another collection feature in Map, which is called Queue.

Queue is Open-Ended

The queue is useful when you try to build a collection that can be added from one end and can be deleted from another end. The values are removed or read in the order of their insertion.

Consider this code:

```
//code 6.16
import 'dart:collection';
main(List<String> arguments){
Queue myQueue = new Queue();
print("Default implementation ${myQueue.runtimeType}");
myQueue.add("Sanjib");
myQueue.add(54);
myQueue.add("Howrah");
myQueue.add("sanjib12sinha@gmail.com");
for(var allTheQueues in myQueue){
    print(allTheQueues);
}
print("----------");
print("We are removing the first element ${myQueue.elementAt(0)}.");
myQueue.removeFirst();
for(var allTheQueues in myQueue){
    print(allTheQueues);
}
print("----------");
print("We are removing the last element ${myQueue.elementAt(2)}.");
myQueue.removeLast();
for(var allTheQueues in myQueue){
    print(allTheQueues);
}
}
```

The output is as expected; it gives us the full lists of what we have added in the Queue. After that we have removed the first and the last element1.

```
//output
Default implementation ListQueue<dynamic>
Sanjib
54
Howrah
sanjib12sinha@gmail.com
----------
We are removing the first element Sanjib.
54
Howrah
sanjib12sinha@gmail.com
----------
We are removing the last element sanjib12sinha@gmail.com.
```

```
54
Howrah
```

In most cases, as I have said earlier at the beginning of Data Structures chapter, we can handle with Lists and Maps. So Queue is an option that you may need some time; but not very often.

Callable Classes

It is a very interesting feature in Dart, where we can call a Class like a function. All we need to do is just implement the call() function.

```
//code 6.17
//when dart class is callable like a function, use call() function
class Person{
String name;
String call(String message, [name]){
    return "This message: '$message', has been passed to the person $name.";
}
}
main(List<String> arguments){
var John = Person();
John.name = "John Smith";
String name = John.name;
String msgAndName = John("Hi John how are you?", name);
print(msgAndName);
}
And here is the output:
This message: 'Hi John how are you?', has been passed to the person John Smith.
```

Here, 'John' is the instance variable and the 'Person()' is the class object. The class 'Person' is called like a function because we have implemented the call() function, through which we have passed two parameters:'String message' and the optional parameter 'name'. Finally, we have passed both and captured the value through 'msgAndName'.

Exception Handling

During the execution of any program, if Exception occurs, the program is disrupted. The normal flow of the program gets disturbed.

For the complete beginners, these concepts may seem a little bit tough. Seasoned programmers will understand how to catch the exceptions and display them in a nice formatted way. From the complete

beginners' perspective, we can say, there are some errors that usually disrupt the flow of program automatically.

Suppose you want to divide a number by zero.

It is an impossible task and will disrupt the flow resulting in some errors. However, you cannot control a user's behavior, so you need to take every precaution to avoid getting such ugly errors.

Dart programmers have thought about it and they have included many built-in exceptions. One of them is: 'IntegerDivisionByZeroException'; it is thrown when a number is divided by zero. Likewise, when a scheduled timeout happens while waiting for an 'async' result, the 'Timeout' exception occurs. If deferred libraries fail to load, there is 'DeferredLoadException' happens.

Suppose a string data cannot be parsed because it does not have the proper format. In that case, 'FormatException' exception occurs. Any input and output related exceptions are captured through 'IOException' class.

In Dart, everything is an Object and behind an object, there must be a class. In the exception handling cases, the class 'Exception' plays the main role to prevent the application from terminating abruptly.

Let us see some code snippets so that we can understand easily how we can catch the exceptions.

```
1  //code 6.18
2  main(List<String> arguments){
3  try{
4      int result = 10 ~/ 0;
5      print("The result is $result");
6  } on IntegerDivisionByZeroException{
7      print("We cannot divide by zero");
8  }
9  try{
10     int result = 10 ~/ 0;
11     print("The result is $result");
12 } catch(e){
13     print(e);
14 }
15 try{
16     int result = 10 ~/ 0;
17     print("The result is $result");
18 } catch(e){
19     print("The exception is : $e");
20 } finally{
21     print("This is Finally and it always is executed.");
22 }
23 }
24
```

```
//Here is the output:
We cannot divide by zero
IntegerDivisionByZeroException
The exception is : IntegerDivisionByZeroException
This is Finally and it always is executed.
```

As you see in the output, there are several methods through which we can catch the exceptions. If we know the type of exception, we can use try/on. As we have used in the above code:

```
try{
int result = 10 ~/ 0;
print("The result is $result");
} on IntegerDivisionByZeroException{
print("We cannot divide by zero");
}
```

In this case, we did know what type of exception could be generated. So we have used “try/on”. But what happens, when we do not know the exception? The syntax of handling exception is as given below:

```
try{
int result = 10 ~/ 0;
print("The result is $result");
} catch(e){
print(e);
}
```

The catch block is used when the handler needs the exception object.

The try block may be followed by finally block after the catch block. We have used the same thing in the above code:

```
try{
int result = 10 ~/ 0;
print("The result is $result");
} catch(e){
print("The exception is : $e");
} finally{
print("This is Finally and it always is executed.");
}
```

The final block will be executed at the end, whatever be the outcome:

```
The exception is : IntegerDivisionByZeroException
This is Finally and it always is executed.
```

In case, an exception occurs in the try block, the control goes to the catch block; and at the end, the final block gives the output.

Dart Packages and Libraries

Dart programmings very heavily rely on libraries. There are several common libraries that serve many purposes while we build any Dart application. So far you have seen many built-in functions that we have used in many user-defined functions; such as 'dart:core' libraries provide assistance for numbers, string-specific operations or collections. With the help of 'dart:math' we can do many types of mathematical operations quite easily.

We can also build our own libraries. In fact, as you progress you will feel the necessity of creating your own libraries. Besides, you can get additional libraries by importing them from packages.

We should also know why we need libraries? To create a modular and shareable code base, we need a good organization of the code base. It is an essential part of object-oriented programming. Libraries not only provide support for modular, object-oriented programming, it also gives you a kind of privacy in your own code.

Identifiers, starting with (_) underscore, are only visible in your libraries.

Libraries also give you good support to avoid name conflicts which is an essential part of coding. How it does so? Let us see an example to clarify those points. First, we have created a 'RelationalOperators.dart' file inside the 'lib' folder.

```
//code 6.19
//lib/ RelationalOperators.dart
class TrueOrFalse{
int firstNum = 40;
int secondNum = 40;
int thirdNum = 74;
int fourthNum = 56;
void BetweenTrueOrFalse(){
    if (firstNum == secondNum || thirdNum == fourthNum){
    print("If choice between 'true' or 'false', in this case the 'TRUE' gets the pre\
cedence. $firstNum is equal to $secondNum");
    } else print("Nothing happens.");
}
void BetweenTrueAndFalse(){
    if (firstNum == secondNum && thirdNum == fourthNum){
    print("It will go to else clause");
```

```
    } else print("If choice between 'true' and 'false', in this case the 'FALSE' get\
s the precedence. $thirdNum is not equal to $fourthNum");
}
}
```

Next, we create a file 'PowProject.dart' inside the 'lib' folder.

```
//code 6.20
//lib/PowProject.dart
class PowProject{
void MultiplyByAGivenNumber(int fixedNumber, int givenNumber){
    int result = fixedNumber * givenNumber;
    print(result);
}
void pow(int x, int y){
    int addition = x + y;
    print(addition);
}
}
```

Now take a look at the 'main()' function body:

```
//code 6.21
import 'dart:math' as math;
import 'package:IdeaProjects/PowProject.dart';
import 'package:IdeaProjects/RelationalOperators.dart' as Relation;
main(List<String> arguments){
print("Prinitng 2 to the power 5 using Dart's built-in 'dart:math' library.");
var int = math.pow(2, 5);
print(int);
print("Now we are going to use another 'pow()' function from our own library.");
var anotherPowObject = PowProject();
anotherPowObject.MultiplyByAGivenNumber(4, 3);
anotherPowObject.pow(2, 12);
print("Now we are going to use another library to test the relational operators.");
var trueOrFalse = Relation.TrueOrFalse();
trueOrFalse.BetweenTrueOrFalse();
trueOrFalse.BetweenTrueAndFalse();
}
```

In the 'lib' or libraries folder we have created two classes. One of them has a function called: 'pow()'. But we know that the built-in 'dart:math' library has a function of the same name: 'pow()'. By any

way, we cannot use those same-name functions consecutively. It will give us errors. So to avoid the name conflict what we have done,we created our own library and defined it inside the class. Quite naturally, for the book's sake, our created 'pow()' function is doing something different than calculating the power of a number.

Look at the top of the main() function.

```
import 'dart:math' as math;
import 'package:IdeaProjects/PowProject.dart';
import 'package:IdeaProjects/RelationalOperators.dart' as Relation;
```

We have used the keyword 'import' to specify how our libraries, besides the core libraries can be used. After the 'import' we need to pass an argument which is nothing but a URI (Uniform Resource Identifier) specifying the libraries. For any built-in libraries, the URI has the special 'dart:...' scheme. For other libraries, you can use the file system path or the 'package:...' scheme. We have used the 'package:...' scheme; it is easy and it is provided b the package manager such as the 'pub' tool. When we directly use the libraries, we use a normal line like this:

```
import 'package:IdeaProjects/PowProject.dart';
```

In that case, we can directly create the class object that belongs to that particular library, such as:

```
var anotherPowObject = PowProject();
```

However, there is another good method; we can call any library by a name, like this:

```
import 'package:IdeaProjects/RelationalOperators.dart' as Relation;
```

The advantage is, now we can create any class object belonging to that library, such as:

```
var trueOrFalse = Relation.TrueOrFalse();
```

These prefixes basically are used to avoid name conflicts. You can write same-name classes in libraries and you can use them by giving them any name.

There are a few good built-in libraries that come with Dart; you need not write them again. Here are some of them:

1. 'dart:core' : It gives us many core functionalities. It is automatically imported into every Dart program.
2. 'dart:math' : You have seen how we have used the core mathematical libraries in our program. We can do many types of mathematical operations using that library; we can generate random numbers.

3. ‘dart:convert’ : Converting between different data representations is made easy through this library; this conversion includes JSON and UTF-8.
4. ‘dart:async’ : This library is beyond the scope of this book. I have written a separate book on Dart advanced programming. Please consult that book to know how Dart helps asynchronous programming, with classes such as Future and Stream.

Before concluding this book, we will have a look at the user-defined libraries again, that time for Flutter libraries. Usually, you can create any package libraries inside the ‘/lib’ directory and ‘import’ them, as we have done. Besides, you can also create sub-folders inside ‘/lib’ and create the hierarchies as needed.

Want to read more Flutter related Articles and resources?

For more Flutter related Articles and Resources[13]

[13] https://sanjibsinha.com

7. Introduction to State Management and Form Validation in Flutter and Dart

Some form of user interaction with the system is needed in any application, be it a mobile application or web application. If you already have some web development experience, you may have managed state in one way or other. Because of 'state', we have persistent data all along any application. Therefore, state is some kind of app data, which is persistent all over its life cycle.

For more Flutter related Articles and Resources[14]

State of any application can be managed by various ways; in Flutter also, it is true. Although there are several state management patterns in flutter, we cannot dive deep into every pattern here, for many reasons. Because this book is aimed for absolute beginners, we will try to stick around the basic. We want to understand what state management is, and how it works in flutter.

As we have already said that state represents some kind of user interactions. A user presses a button and it gives her some output. She presses again, it gives another output; it might keep going on until the output list is exhausted.

It cannot be achieved by a 'StateLess' widget. By using a 'StateLess' widget we can represent a 'RaisedButton', however, it won't work or won't give some output, until we 'set state'. Setting state can only be done by the 'StateFul' widget. With the help of 'Stateful' widget we can build from simple to complex user interactions. Probably we have noticed that whenever we create a flutter application it comes up with a 'StateFul' widget counter, where pressing the 'plus' button increases the number.

This is Google's BLoC pattern. The advantage of this pattern is we do not need any type of outside libraries. Therefore, we can use this pattern for any type of application, simple or complex.

Flutter renders the widgets by building the 'element' trees. Just picture this: a Container widget has a child element, which has a Padding widget, which has a child element that has a Row widget, which has a children widget,and so on.

Now inside a 'StateLess' widget, we can have a RaisedButton widget, which passes a parameter called 'onPress'; it is an anonymous void function that could call another named function that returns a value.

Now in that named function, we can set the state. When we create a 'StateFulWidget', its variables are immutable just like any 'StateLessWidget'; you can only declare any 'final' property, which is immutable. However, 'StateFulWidget' creates an associated state object by 'createState()' method.

[14] https://sanjibsinha.com

State is mutable

Let us first see an image, which represents a simple button.

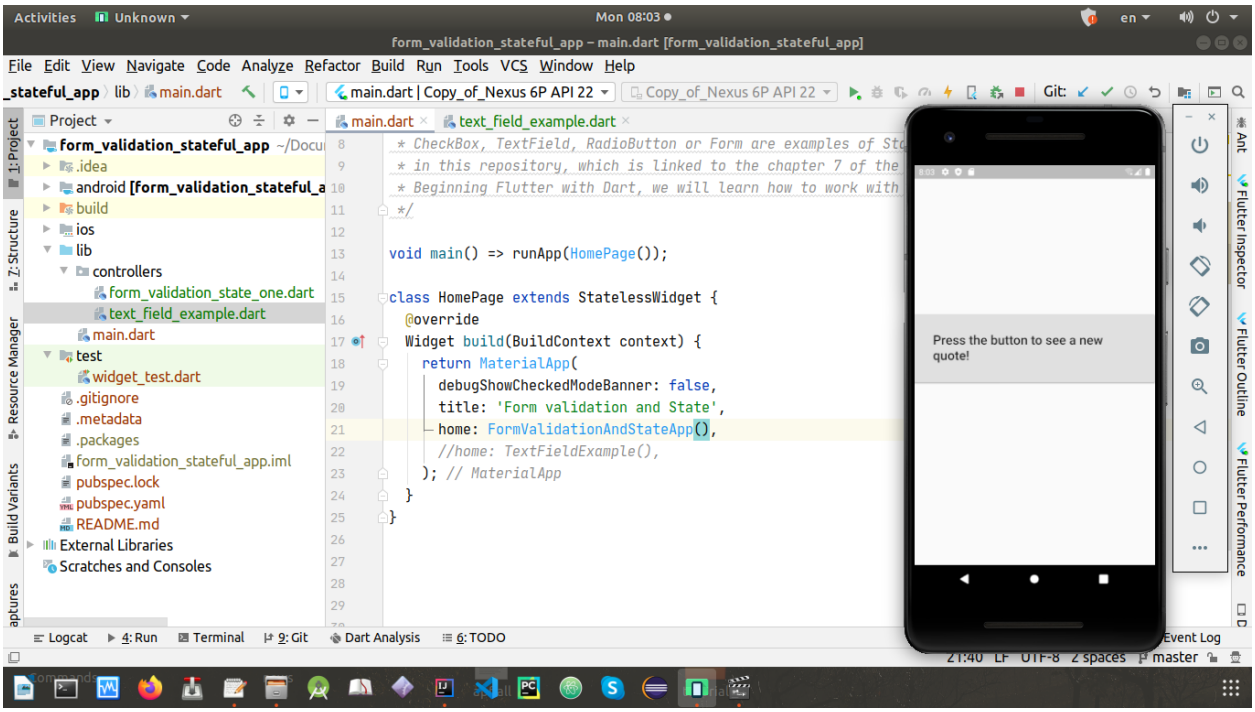

Figure 7.1 – A simple RaisedButton

If we press this button, it will give us different quotes, until the list is exhausted. The state object holds to the app data persistently throughout its life-cycle. Let us see the full code and after that we will dissect the code and try to understand how it works.

```
1  //code 7.1
2  import 'package:flutter/material.dart';
3
4  /**
5  * inheriting StatefulWidget makes a class immutable
6  */
7
8  class FormValidationAndStateApp extends StatefulWidget {
9  /**
10 * the widget returns the state in createState() method
11 */
12 @override
13 _FormValidationAndStateAppState createState() => _FormValidationAndStateAppState();
14 }
```

```
/**
* a simple example of state where user presses a button to see a list of quotes
*/
class _FormValidationAndStateAppState extends State<FormValidationAndStateApp> {

var _quotes = [
    '',
    'Life is a Tragedy',
    'Life is Beautiful',
    'Life consists of problems to solve',
];

int _questionIndex = 0;

int _answerQuestions() {
    /**
    * calling the setState() method makes the change and redraw the widget
    */
    setState(() {
    _questionIndex = _questionIndex + 1;
    });
    if(_questionIndex == 4) {
    _questionIndex = 0;
    }
    return _questionIndex;
}

@override
Widget build(BuildContext context) {
    return Scaffold(
    body: Container(
        alignment: Alignment.center,
        child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        //crossAxisAlignment: CrossAxisAlignment.start,
        children: <Widget>[
            RaisedButton(
            padding: EdgeInsets.all(32.0),
            child: Text(
                'Press the button to see a new quote!',
                style: TextStyle(
                fontSize: 22,
                //color: Colors.blue,
```

```
                ),
            ),
            onPressed: () {
                _answerQuestions();
            },
            disabledColor: Colors.blueAccent,
            ),
            SizedBox(height: 10.0,),
            Text(
            '${_quotes[_questionIndex]}',
            style: TextStyle(
                fontSize: 40.0,
            ),
            textAlign: TextAlign.center,
            ),
        ],
        ),
    ),
    );
}
}


import 'package:flutter/material.dart';
import 'package:form_validation_stateful_app/controllers/form_validation_state_one.d\
art';
import 'package:form_validation_stateful_app/controllers/text_field_example.dart';

/**
* the state is mutable and might change during the lifetime of the widget
* each time it redraws the widget whenever the state is changed
* CheckBox, TextField, RadioButton or Form are examples of Stateful widgets
* in this repository, which is linked to the chapter 7 of the book
* Beginning Flutter with Dart, we will learn how to work with these widgets
*/

void main() => runApp(HomePage());

class HomePage extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    debugShowCheckedModeBanner: false,
```

```
    title: 'Form validation and State',
    home: FormValidationAndStateApp(),
    );
}
}
```

Now we are going to study the above code part by part, so that we could understand how it works.

```
/**
* inheriting StatefulWidget makes a class immutable
*/

class FormValidationAndStateApp extends StatefulWidget {
/**
* the widget returns the state in createState() method
*/
@override
_FormValidationAndStateAppState createState() => _FormValidationAndStateAppState();
}
```

Our 'FormValidationAndStateApp' extends a 'StateFulWidget'; and it makes the class immutable. However, at the same time it creates a state object. The next step is to use that state object.

```
/**
* a simple example of state where user presses a button to see a list of quotes
*/
class _FormValidationAndStateAppState extends State<FormValidationAndStateApp> { }
```

In between the curly braces, we will now write our code. To hold the state and iterate the list values using its index, we need to use the 'setState()' method.

```
var _quotes = [
    '',
    'Life is a Tragedy',
    'Life is Beautiful',
    'Life consists of problems to solve',
];

int _questionIndex = 0;

int _answerQuestions() {
    /**
```

```
    * calling the setState() method makes the change and redraw the widget
    */
    setState(() {
    _questionIndex = _questionIndex + 1;
    });
    if(_questionIndex == 4) {
    _questionIndex = 0;
    }
    return _questionIndex;
}
```

We have increased the question index value by 1, and finally when the index value reaches 4, it comes back to 0 again. Using a simple trick we have used a blank string to start with. We could have used a 'reset' button. We will see such examples later in this chapter.

With the help of two 'StateLessWidget' - the RaisedButton and Text, we press the button and catch the value.

```
RaisedButton(
            padding: EdgeInsets.all(32.0),
            child: Text(
                'Press the button to see a new quote!',
                style: TextStyle(
                fontSize: 22,
                //color: Colors.blue,
                ),
            ),
            onPressed: () {
                _answerQuestions();
            },
            disabledColor: Colors.blueAccent,
            ),
            SizedBox(height: 10.0,),
            Text(
            '${_quotes[_questionIndex]}',
            style: TextStyle(
                fontSize: 40.0,
            ),
            textAlign: TextAlign.center,
            ),
```

The next image shows the above example, where a user presses the button and gets the quote.

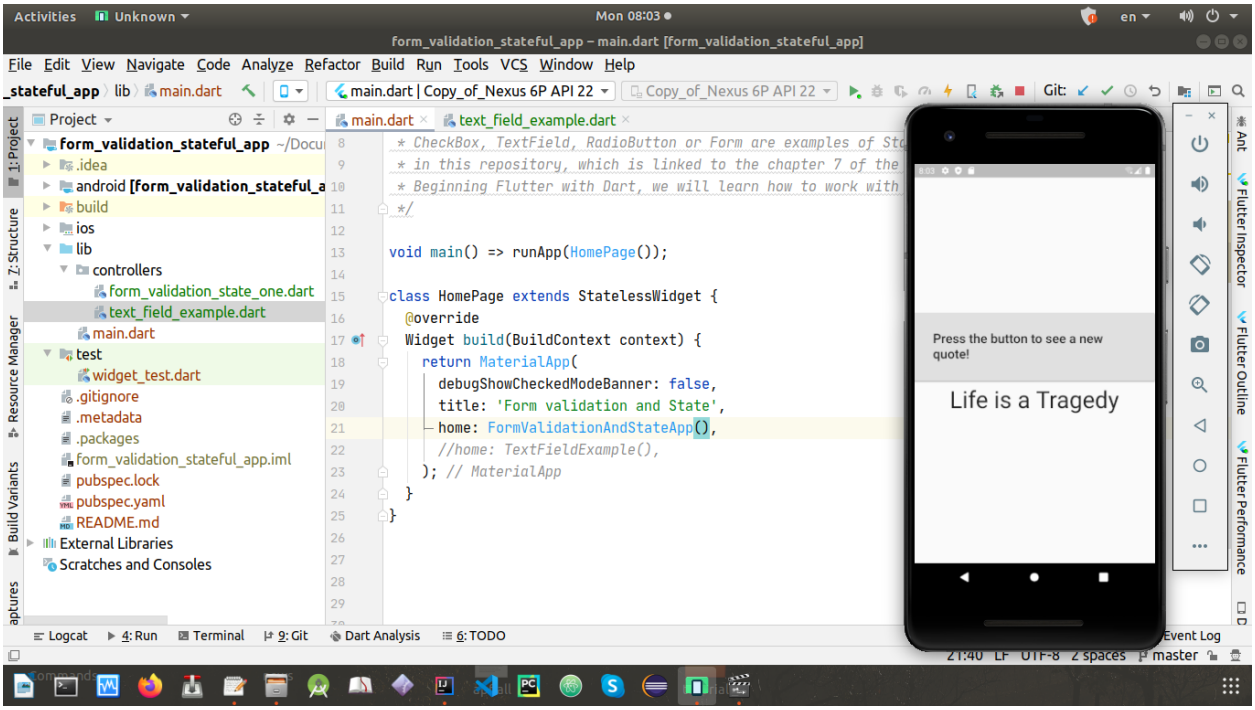

Figure 7.2 – Pressing the button gives an output

The state object is associated with the StateFulWidget's life-cycle. Flutter manages this complex process internally and with each sequence it re-draws the screen.

That never happens with the Stateless widget.

Life cycle of State

The state is mutable and it changes with the life-cycle of the widget. Each time the state changes, Flutter re-draws the widget.

The next examples will make the abstraction clearer than before. The TexField widget is by default a Stateful widget. It has a property 'onChanged'; the named parameter points to the anonymous function that passes the string value which the user types on the mobile screen. Whenever some text is is being typed through the 'TextField', the text is reflected on the screen.

```
/// it reflects the text input on the screen while typing
            onChanged: (String name) {
                setState(() {
                yourName = name;
                });
            },
            ),
```

However, it has also another property ‘onSubmitted’, which takes the input and gives the output on the screen.

```
onSubmitted: (String name) {
                setState(() {
                yourName = name;
                });
            },
```

Let us see how it looks like. The next image shows us how it looks like on the screen.

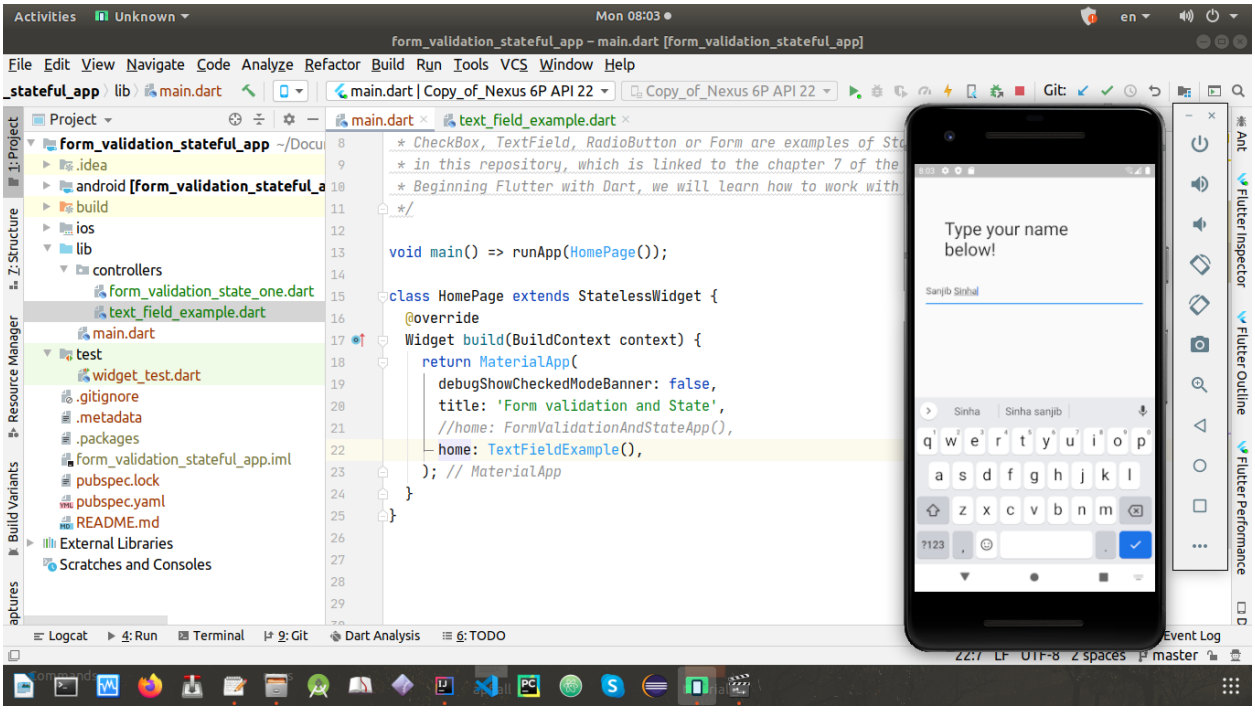

Figure 7.3 – The TextField Stateful widget

There is nothing fancy in the following code snippets. A simple TextField widget, where we would write any name and pressing the blue button below on the mobile keypad gives us an output.

```dart
//code 7.2

import 'package:flutter/material.dart';

class TextFieldExample extends StatefulWidget {
@override
_TextFieldExampleState createState() => _TextFieldExampleState();
}
/**
* a simple example of TextField, where one can type the name
* and see the output below; each time user types a String data type or text, a state\
ful widget
* is created, however, after that when she re-type another text, the set state is ca\
lled, which
* tells the framework to redraw the TextFieldExampleState widget and it's created ag\
ain
*/

class _TextFieldExampleState extends State<TextFieldExample> {

String yourName = '';

@override
Widget build(BuildContext context) {
    return Scaffold(
    body: Container(
        margin: EdgeInsets.all(20.0),
        child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        crossAxisAlignment: CrossAxisAlignment.start,
        children: <Widget>[
            Padding(
            padding: const EdgeInsets.all(32.0),
            child: Text(
                'Type your name below!',
                style: TextStyle(
                fontSize: 30.0,
                ),
            ),
            ),
            TextField(
            /*
            onSubmitted: (String name) {
```

```
                setState(() {
                yourName = name;
                });
            },
            */
            /// it reflects the text input on the screen while typing
            onChanged: (String name) {
                setState(() {
                yourName = name;
                });
            },
            ),
            Padding(
            padding: const EdgeInsets.all(32.0),
            child: Text(
                yourName,
                style: TextStyle(
                fontSize: 30.0,
                ),
            ),
            ),
        ],
        ),
    ),

    );
}
}


import 'package:flutter/material.dart';
import 'package:form_validation_stateful_app/controllers/form_validation_state_one.d\
art';
import 'package:form_validation_stateful_app/controllers/text_field_example.dart';

void main() => runApp(HomePage());

class HomePage extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    debugShowCheckedModeBanner: false,
    title: 'Form validation and State',
```

```
87        home: TextFieldExample(),
88        );
89  }
90  }
```

In the comments section we have written what is happening under the hood. Whenever we type some text and press the button, it gives us the output. Moreover, when we re-type any text, the set state is called, which tells the framework to redraw the assigned widget and it's created again.

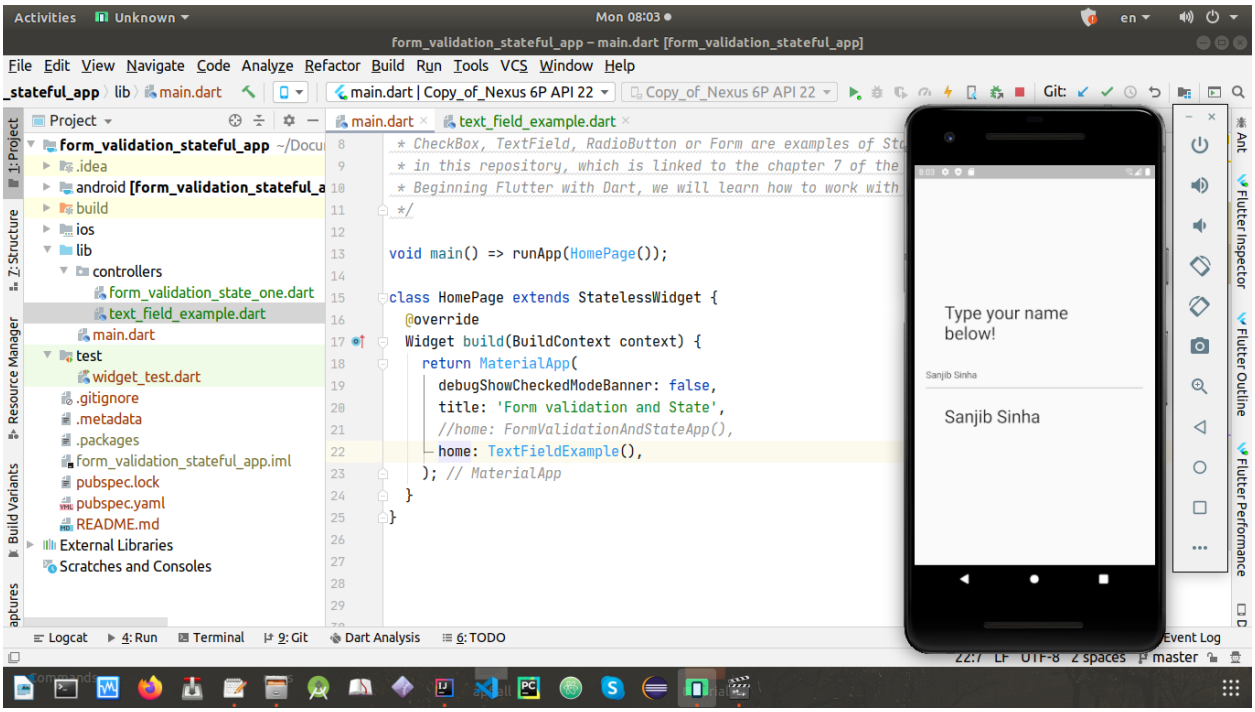

Figure 7.4 – The TextField widget gives us an output

As we have learned before, changing the 'onSubmitted' property to 'onChanged' gives us a visual representation of what we are typing on the screen. The next image shows us how this magical activity takes place directly on the screen.

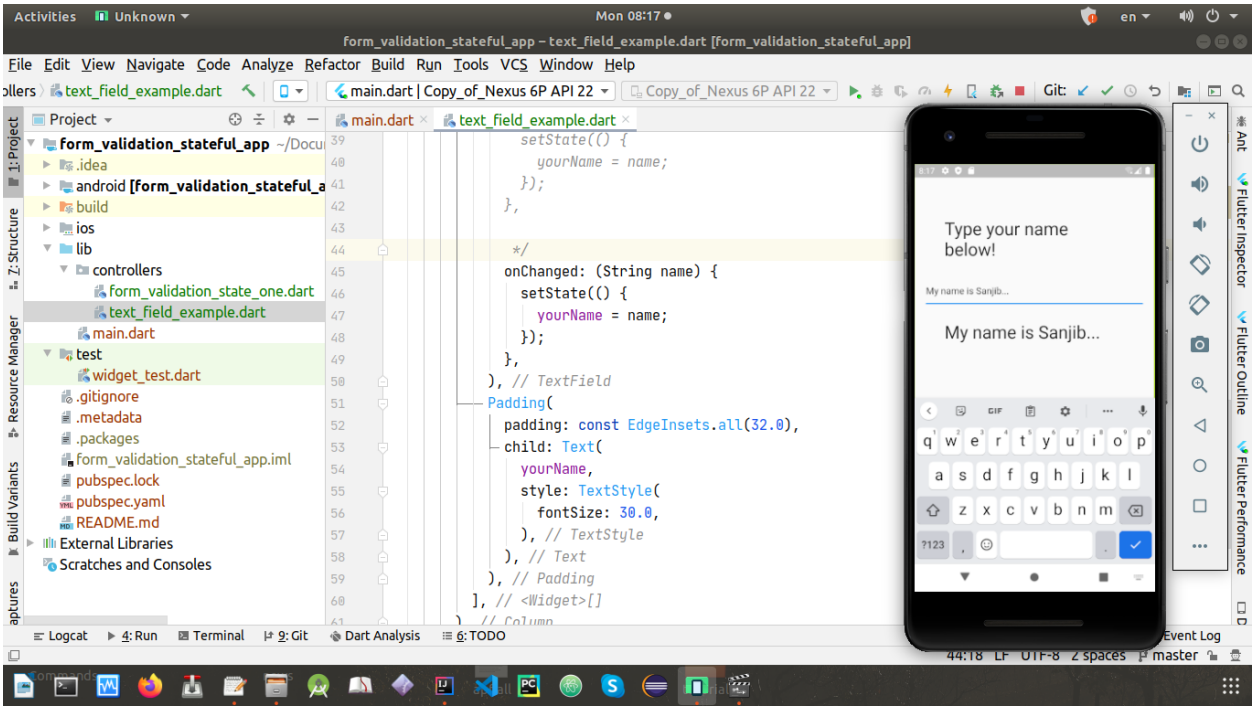

Figure 7.5 – While typing using TextField widget the text is being shown on the screen

If we want to use any button click to get the output, we can use the TextField widget as well. The next code snippet will give you an idea.

```
//code 7.3

import 'package:flutter/material.dart';

class TextFieldApp extends StatefulWidget {
@override
_TextFieldAppState createState() => _TextFieldAppState();
}

class _TextFieldAppState extends State<TextFieldApp> {

TextEditingController textOfName = TextEditingController();

String _displayText = '';

String displayAllSelectedValue() {
    String name = textOfName.text;
    String result = 'Name is: ${name}';
    return result;
}
```

```
21
22
23  @override
24  Widget build(BuildContext context) {
25      return Scaffold(
26      body: Container(
27          alignment: Alignment.center,
28          child: Padding(
29          padding: EdgeInsets.all(20.0),
30          child: ListView(
31              children: <Widget>[
32              TextField(
33                  keyboardType: TextInputType.text,
34                  controller: textOfName,
35                  style: TextStyle(
36                  fontSize: 16.0,
37                  color: Colors.blue,
38                  ),
39                  decoration: InputDecoration(
40                  labelText: 'Your Nmae',
41                  hintText: 'In text...',
42                  labelStyle: TextStyle(
43                      fontSize: 17.0,
44                      color: Colors.red,
45                  ),
46                  border: OutlineInputBorder(
47                      borderRadius: BorderRadius.circular(5.0),
48                  ),
49                  ),
50              ),
51              SizedBox(height: 10.0,),
52              Row(
53                  children: <Widget>[
54                  Container(
55                      width: 150.0,
56                      child: RaisedButton(
57                      color: Colors.white24,
58                      textColor: Colors.redAccent,
59                      child: Text('Press'),
60                      onPressed: () {
61                          setState(() {
62                          this._displayText = displayAllSelectedValue();
63                          });
```

```
                    },
                    ),
                ),
                ],
            ),
            SizedBox(height: 10.0,),
            Text(
                '${_displayText}',
                style: TextStyle(
                fontSize: 20.0,
                color: Colors.redAccent,
                ),
            ),
            ],
        ),
        ),
    ),
    );
}
}
```

Role of Controller in TextField Widget

However, there is a big change in the TextField widget; as we have to use a new property, a named parameter called ‘controller’. Now this controller will check what type of input we type using the TextField widget.

```
TextField(
                keyboardType: TextInputType.text,
                controller: textOfName,
                style: TextStyle(
                fontSize: 16.0,
                color: Colors.blue,
                ),
                decoration: InputDecoration(
                labelText: 'Your Nmae',
                hintText: 'In text...',
                labelStyle: TextStyle(
                    fontSize: 17.0,
                    color: Colors.red,
                ),
```

```
                border: OutlineInputBorder(
                    borderRadius: BorderRadius.circular(5.0),
                ),
                ),
            ),
```

Visibly there is a lot of change when we have described the TextField widget. We have added a controller name and we have added some styling. The function that we have used to display the text, used that controller object. TextEditingController textOfName = TextEditingController();

```
String _displayText = '';

String displayAllSelectedValue() {
    String name = textOfName.text;
    String result = 'Name is: ${name}';
    return result;
}
```

The above code shows us that because we use the controller object, we are in a better position to control the nature of user inputs. Instead only text, we can handle integer data type also.

The next code snippets and the associated images will give us a good idea of how it is being done. We have also used inside the code a new Stateful widget 'DropdownButton' that uses String value to give users a chance to select the correct input. Let us see the code and the associated images; after that we will discuss the code snippets in detail.

Let us first see the image, where the user is asked to give some inputs, such as name, age using the TextField widget. Besides, the user will choose the name of the city where she lives currently. For that we have used the 'DropdownButton'.

Above each TextField there are label text and inside it, there is hint text also.

As we have clicked the button, the user will be given the required output.

Beside the submit button, we have also placed a button called 'reset'; whenever it is pressed, every output will disappear from the screen and Flutter re-draws every widget. It is a little bit complex examples of state management, where we have not only used the 'setState()' but also override the 'initState()' function for the first time to clean the DropdownButton widget.

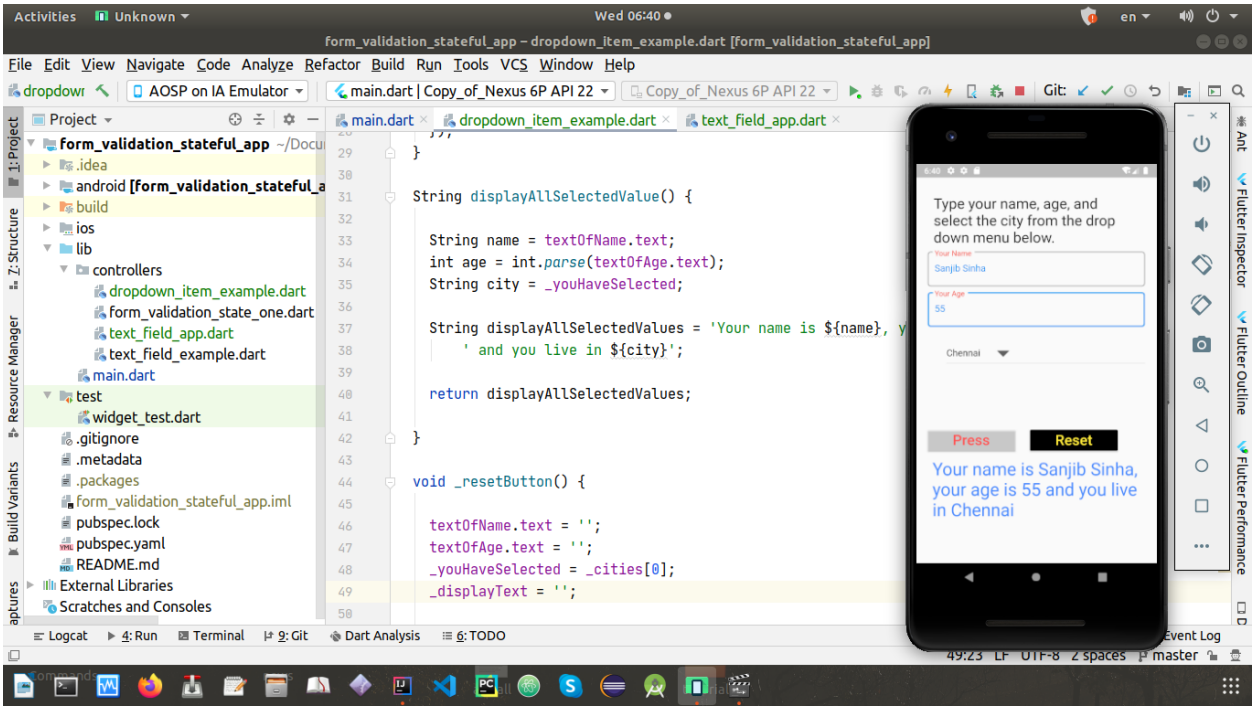

Figure 7.6 - A complex state management example

Let us see the code snippets first, and afterward we will see the next image where the reset button has been pressed.

```
//code 7.4

import 'package:flutter/material.dart';

class DropDownExample extends StatefulWidget {
@override
_DropDownExampleState createState() => _DropDownExampleState();
}

class _DropDownExampleState extends State<DropDownExample> {

String yourName = '';
String yourAge = '';
var _cities = ["Calcutta", "Delhi", "Mumbai", "Chennai", "Bangalore"];
String _youHaveSelected = '';
String _displayText = '';

@override
void initState() {
    super.initState();
```

```
21      _youHaveSelected = _cities[0];
22  }
23
24  TextEditingController textOfAge  = TextEditingController();
25  TextEditingController textOfName  = TextEditingController();
26
27  void selectedDropDownItem(String theValueSelected) {
28      setState(() {
29      this._youHaveSelected = theValueSelected;
30      });
31  }
32
33  String displayAllSelectedValue() {
34
35      String name = textOfName.text;
36      int age = int.parse(textOfAge.text);
37      String city = _youHaveSelected;
38
39      String displayAllSelectedValues = 'Your name is ${name}, your age is ${age}'
40          ' and you live in ${city}';
41
42      return displayAllSelectedValues;
43
44  }
45
46  void _resetButton() {
47
48      textOfName.text = '';
49      textOfAge.text = '';
50      _youHaveSelected = _cities[0];
51      _displayText = '';
52
53  }
54
55  @override
56  Widget build(BuildContext context) {
57      return Scaffold(
58      //resizeToAvoidBottomPadding: false,
59      body: Container(
60          margin: EdgeInsets.all(20.0),
61          child: ListView(
62          children: <Widget>[
63              Padding(
```

```
64              padding: const EdgeInsets.all(10.0),
65              child: Text(
66                  'Type your name, age, and select the city from the drop down menu be\
67  low.',
68                  style: TextStyle(
69                  fontSize: 25.0,
70                  ),
71              ),
72              ),
73              TextField(
74              keyboardType: TextInputType.text,
75              controller: textOfName,
76              style: TextStyle(
77                  fontSize: 16.0,
78                  color: Colors.blue,
79              ),
80              decoration: InputDecoration(
81                  labelText: 'Your Name',
82                  hintText: 'In text...',
83                  labelStyle: TextStyle(
84                  fontSize: 17.0,
85                  color: Colors.red,
86                  ),
87                  border: OutlineInputBorder(
88                  borderRadius: BorderRadius.circular(5.0),
89                  ),
90              ),
91              ),
92              SizedBox(height: 10.0,),
93              TextField(
94              keyboardType: TextInputType.text,
95              controller: textOfAge,
96              style: TextStyle(
97                  fontSize: 16.0,
98                  color: Colors.blue,
99              ),
100             decoration: InputDecoration(
101                 labelText: 'Your Age',
102                 hintText: 'In number...',
103                 labelStyle: TextStyle(
104                 fontSize: 17.0,
105                 color: Colors.red,
106                 ),
```

```
107                 border: OutlineInputBorder(
108                 borderRadius: BorderRadius.circular(5.0),
109                 ),
110             ),
111             ),
112             SizedBox(height: 10.0,),
113             Padding(
114             padding: const EdgeInsets.only(left: 32.0, top: 10.0),
115             child: DropdownButton<String>(
116                 items: _cities.map((String nameOfCities) {
117                 return DropdownMenuItem<String>(
118                     value: nameOfCities,
119                     child: Text(nameOfCities),
120                 );
121                 }).toList(),
122                 onChanged: (String theValueSelected) {
123                 selectedDropDownItem(theValueSelected);
124                 },
125                 value: _youHaveSelected,
126                 iconSize: 50.0,
127             ),
128             ),
129             SizedBox(height: 100.0,),
130             Row(
131             children: <Widget>[
132                 Container(
133                 width: 150.0,
134                 child: RaisedButton(
135                     color: Colors.white24,
136                     textColor: Colors.redAccent,
137                     child: Text('Press', style: TextStyle(fontSize: 25.0),),
138                     onPressed: () {
139                     setState(() {
140                         this._displayText = displayAllSelectedValue();
141                     });
142                     },
143                 ),
144                 ),
145                 Container(width: 25.0,),
146                 Container(
147                 width: 150.0,
148                 child: RaisedButton(
149                     color: Colors.black,
```

```
                    textColor: Colors.yellow,
                    child: Text('Reset', style: TextStyle(fontSize: 25.0),),
                    onPressed: () {
                    setState(() {
                        _resetButton();
                    });
                    },
                ),
                ),
            ],
            ),
            Padding(
            padding: const EdgeInsets.all(8.0),
            child: Text(
                '${_displayText}',
                style: TextStyle(
                fontSize: 30.0,
                color: Colors.blueAccent,
                ),
            ),
            ),
        ],
        ),
    ),
    );
}
}
```

Let us see the initial state when the reset button has been pressed.

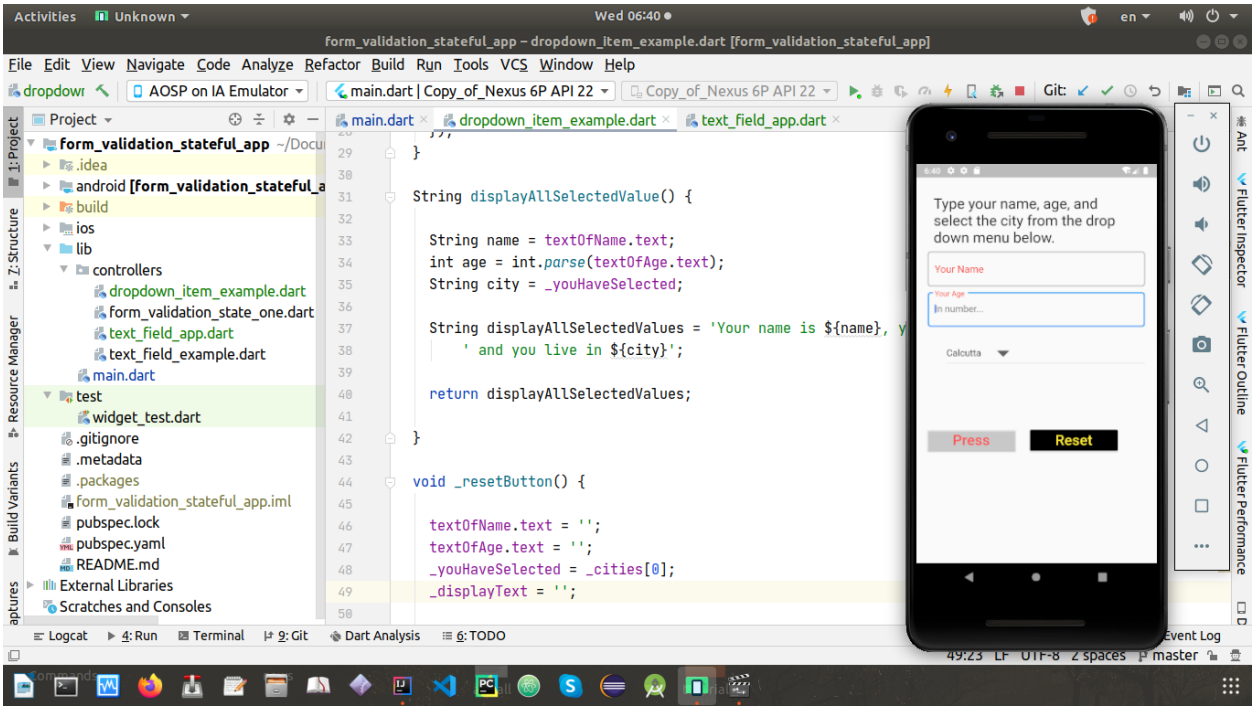

Figure 7.7 – After the reset button has been pressed

The following part of the code snippets has helped us to maintain this complex state management successfully. We have used the concepts of list and later in the

How List and Map used in StateFul DropdownButton Widget

DropdownButton widget used the concepts of map to get the Drop down Menu Item.

```
String yourName = '';
String yourAge = '';
var _cities = ["Calcutta", "Delhi", "Mumbai", "Chennai", "Bangalore"];
String _youHaveSelected = '';
String _displayText = '';

@override
void initState() {
    super.initState();
    _youHaveSelected = _cities[0];
}

TextEditingController textOfAge  = TextEditingController();
```

```
14  TextEditingController textOfName  = TextEditingController();
15
16  void selectedDropDownItem(String theValueSelected) {
17      setState(() {
18      this._youHaveSelected = theValueSelected;
19      });
20  }
21
22  String displayAllSelectedValue() {
23
24      String name = textOfName.text;
25      int age = int.parse(textOfAge.text);
26      String city = _youHaveSelected;
27
28      String displayAllSelectedValues = 'Your name is ${name}, your age is ${age}'
29          ' and you live in ${city}';
30
31      return displayAllSelectedValues;
32
33  }
34
35  void _resetButton() {
36
37      textOfName.text = '';
38      textOfAge.text = '';
39      _youHaveSelected = _cities[0];
40      _displayText = '';
41
42  }
43  …
44  child: DropdownButton<String>(
45                  items: _cities.map((String nameOfCities) {
46                  return DropdownMenuItem<String>(
47                      value: nameOfCities,
48                      child: Text(nameOfCities),
49                  );
50                  }).toList(),
51                  onChanged: (String theValueSelected) {
52                  selectedDropDownItem(theValueSelected);
53                  },
54                  value: _youHaveSelected,
55                  iconSize: 50.0,
56              ),
```

```
57              ),
```

As we have seen, managing state is not difficult, but we need to be careful to follow the correct design patterns. Google's bloc pattern is simple and user friendly. We can easily set the state and when necessary, intiState() method can be overridden to make some complex operations possible.

Finally, in this chapter, we will learn how to use validation, so the user will be prompted to fill the required fields properly.

How to Valiadate a Form using State Management

Let us see the code snippets, after that we will see the associated image; once that is done, we can discuss the code in parts.

```
1  //code 7.5
2  //form_validation_app.dart
3
4  import 'package:flutter/material.dart';
5
6  class FormValidationApp extends StatefulWidget {
7  @override
8  _FormValidationAppState createState() => _FormValidationAppState();
9  }
10
11 class _FormValidationAppState extends State<FormValidationApp> {
12
13 //we have initialized the form key with super class FormState
14 //in future, this key will be used to identify the form instance
15 var _formKey = GlobalKey<FormState>();
16
17 String yourName = '';
18 String yourAge = '';
19 var _cities = ["Calcutta", "Delhi", "Mumbai", "Chennai", "Bangalore"];
20 String _youHaveSelected = '';
21 String _displayText = '';
22
23 @override
24 void initState() {
25     super.initState();
26     _youHaveSelected = _cities[0];
27 }
28
```

```
TextEditingController textOfAge  = TextEditingController();
TextEditingController textOfName  = TextEditingController();

void selectedDropDownItem(String theValueSelected) {
    setState(() {
    this._youHaveSelected = theValueSelected;
    });
}

String displayAllSelectedValue() {

    String name = textOfName.text;
    int age = int.parse(textOfAge.text);
    String city = _youHaveSelected;

    String displayAllSelectedValues = 'Your name is ${name}, your age is ${age}'
        ' and you live in ${city}';

    return displayAllSelectedValues;

}

void _resetButton() {

    textOfName.text = '';
    textOfAge.text = '';
    _youHaveSelected = _cities[0];
    _displayText = '';

}

@override
Widget build(BuildContext context) {
    return Scaffold(
    //resizeToAvoidBottomPadding: false,
    //we have changed the previous container widget to Form
    //since Form does not allow margin, we need to add some padding around ListView
    body: Form(
        //later this key will act as an identifier
        //and it will let us know the current status of the form
        key: _formKey,
        child: Padding(
        padding: const EdgeInsets.all(8.0),
```

```
72          child: ListView(
73              children: <Widget>[
74              Padding(
75                  padding: const EdgeInsets.all(10.0),
76                  child: Text(
77                  'Type your name, age, and select the city from the drop down menu be\
78 low.',
79                  style: TextStyle(
80                      fontSize: 25.0,
81                  ),
82                  ),
83              ),
84              //we will change the TextField to TextFormField so that we can use the v\
85 alidator
86              TextFormField(
87                  keyboardType: TextInputType.text,
88                  controller: textOfName,
89                  validator: (String validationValue) {
90                  if (validationValue.isEmpty) {
91                      return 'Please fill up the form with correct input!';
92                  }
93                  },
94                  style: TextStyle(
95                  fontSize: 16.0,
96                  color: Colors.blue,
97                  ),
98                  //because we are using TextFormField, and use the validation
99                  // we can use customize the error style
100                 decoration: InputDecoration(
101                 labelText: 'Your Name',
102                 hintText: 'In text...',
103                 labelStyle: TextStyle(
104                     fontSize: 17.0,
105                     color: Colors.red,
106                 ),
107                 border: OutlineInputBorder(
108                     borderRadius: BorderRadius.circular(5.0),
109                 ),
110                 errorStyle: TextStyle(
111                     color: Colors.deepPurple,
112                     fontSize: 20.0,
113                 ),
114                 ),
```

```
115             ),
116             SizedBox(height: 10.0,),
117             TextFormField(
118                 keyboardType: TextInputType.text,
119                 controller: textOfAge,
120                 validator: (String validationValue) {
121                 if (validationValue.isEmpty) {
122                     return 'Please fill up the form with correct input!';
123                 }
124                 },
125                 style: TextStyle(
126                 fontSize: 16.0,
127                 color: Colors.blue,
128                 ),
129                 decoration: InputDecoration(
130                 labelText: 'Your Age',
131                 hintText: 'In number...',
132                 labelStyle: TextStyle(
133                     fontSize: 17.0,
134                     color: Colors.red,
135                 ),
136                 border: OutlineInputBorder(
137                     borderRadius: BorderRadius.circular(5.0),
138                 ),
139                 errorStyle: TextStyle(
140                     color: Colors.deepPurple,
141                     fontSize: 20.0,
142                 ),
143                 ),
144             ),
145             SizedBox(height: 10.0,),
146             Padding(
147                 padding: const EdgeInsets.only(left: 32.0, top: 10.0),
148                 child: DropdownButton<String>(
149                 items: _cities.map((String nameOfCities) {
150                     return DropdownMenuItem<String>(
151                     value: nameOfCities,
152                     child: Text(nameOfCities),
153                     );
154                 }).toList(),
155                 onChanged: (String theValueSelected) {
156                     selectedDropDownItem(theValueSelected);
157                 },
```

```
158                 value: _youHaveSelected,
159                 iconSize: 50.0,
160                 ),
161             ),
162             SizedBox(height: 100.0,),
163             Row(
164                 children: <Widget>[
165                 Container(
166                     width: 150.0,
167                     child: RaisedButton(
168                     color: Colors.white24,
169                     textColor: Colors.redAccent,
170                     child: Text('Press', style: TextStyle(fontSize: 25.0),),
171                     onPressed: () {
172                         setState(() {
173                         //if the form's current state validates, only then proceed
174                         if (_formKey.currentState.validate()) {
175                             this._displayText = displayAllSelectedValue();
176                         }
177                         });
178                     },
179                     ),
180                 ),
181                 Container(width: 25.0,),
182                 Container(
183                     width: 150.0,
184                     child: RaisedButton(
185                     color: Colors.black,
186                     textColor: Colors.yellow,
187                     child: Text('Reset', style: TextStyle(fontSize: 25.0),),
188                     onPressed: () {
189                         setState(() {
190                         _resetButton();
191                         });
192                     },
193                     ),
194                 ),
195                 ],
196             ),
197             Padding(
198                 padding: const EdgeInsets.all(8.0),
199                 child: Text(
200                 '${_displayText}',
```

```
                style: TextStyle(
                    fontSize: 30.0,
                    color: Colors.blueAccent,
                ),
                ),
            ),
            ],
        ),
        ),
    ),
    );
}
}
```

We have changed our old code (7.4) a little bit; all we have done is we have added some some extra functionalities and changed the Container widget to the Form widget. This Form widget has many other features, without which we could not have achieved what we wanted to do.

Let us see the image, and we will have an idea how this code works.

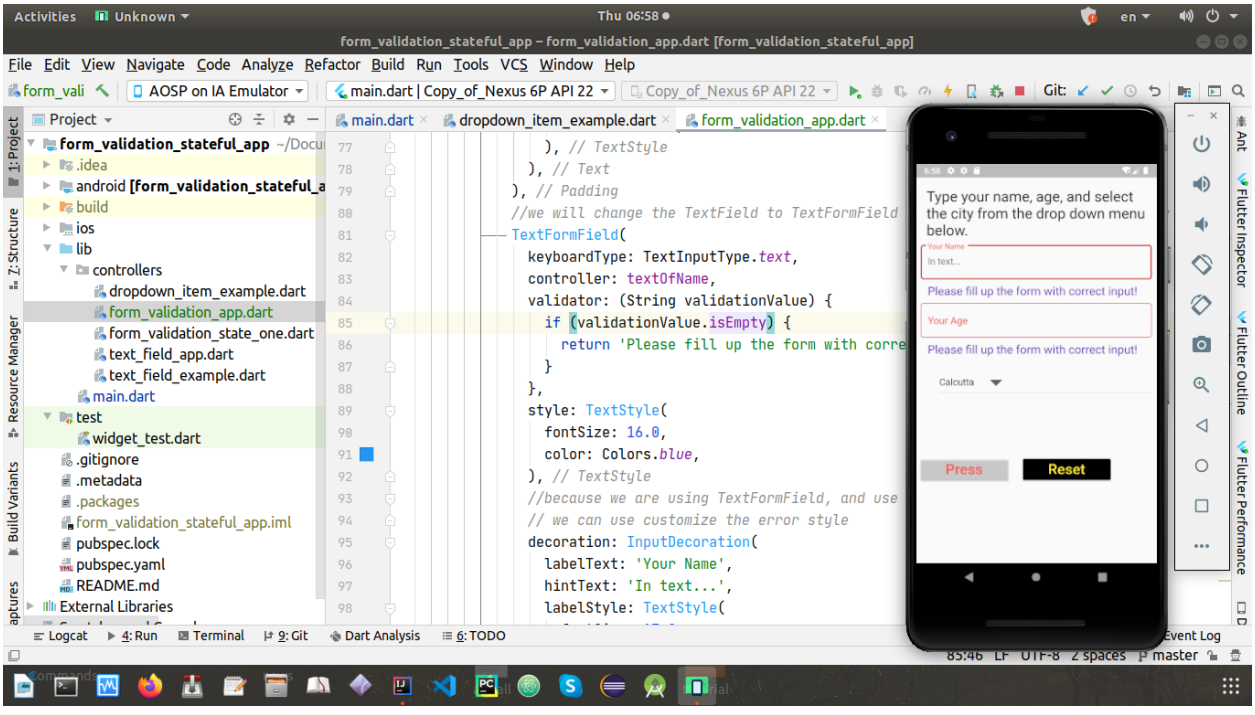

Figure 7.8 – How form validation works in state management

The major change is in the following section:

```
//we have changed the previous container widget to Form
    //since Form does not allow margin, we need to add some padding around ListView
    body: Form(
        //later this key will act as an identifier
        //and it will let us know the current status of the form
        key: _formKey,
…
```

The next big change in the TextField section.

```
//we will change the TextField to TextFormField so that we can use the validator
            TextFormField(
                keyboardType: TextInputType.text,
                controller: textOfName,
                validator: (String validationValue) {
                if (validationValue.isEmpty) {
                    return 'Please fill up the form with correct input!';
                }
                },
                style: TextStyle(
                fontSize: 16.0,
                color: Colors.blue,
                ),
                //because we are using TextFormField, and use the validation
                // we can use customize the error style
                decoration: InputDecoration(
                labelText: 'Your Name',
                hintText: 'In text...',
                labelStyle: TextStyle(
                    fontSize: 17.0,
                    color: Colors.red,
                ),
                border: OutlineInputBorder(
                    borderRadius: BorderRadius.circular(5.0),
                ),
                errorStyle: TextStyle(
                    color: Colors.deepPurple,
                    fontSize: 20.0,
                ),
                ),
            ),
….
```

In the comments we have mentioned why we are doing these changes. Every change is purposeful here, because we want to validate the Form properly.

The Form key plays a major role here. It holds on to the state of the form; for that reason, this following part is extremely important.

```
Form(
        //later this key will act as an identifier
        //and it will let us know the current status of the form
        key: _formKey,
….
```

It is clearly stated why we are using the key. Later, inside the RaisedButton widget, it plays the key role. Watch this code snippets:

```
child: RaisedButton(
                    color: Colors.white24,
                    textColor: Colors.redAccent,
                    child: Text('Press', style: TextStyle(fontSize: 25.0),),
                    onPressed: () {
                        setState(() {
                        //if the form's current state validates, only then proceed
                        if (_formKey.currentState.validate()) {
                            this._displayText = displayAllSelectedValue();
                        }
                        });
                    },
                    ),
                ),
...
```

Here the logic is also quite clear. If the form key's current state validates, only then the output will be given on the screen. Otherwise, it will give us errors that we have customized.

However, Google announced at Google I/O '19 that Provider is now its preferred package for state management. Provider is a package written in 2018 by Remi Rousselet. Because it is simple and flexible, we will also learn how to use Provider to manage state without rebuilding the whole UI widget tree.

Of course, you can still use others, because there are others, ans some of them is really good. But, keep in mind that Google recommends going with Provider.

In the next chapter, we will find the simplicity and flexibility of using Provider to manage state in a better way. We will also learn how to use Model-View-Controller design pattern to implement Provider.

Want to read more Flutter related Articles and resources?

For more Flutter related Articles and Resources[15]

[15] https://sanjibsinha.com

8. Provider: A recommended approach to manage State and Model-View-Controller Pattern

When an app is running, if we want something to exist in memory, we can call it 'state'. In the previous chapter, we have already discussed 'state' and learned a few tricks to manage it. However, that is an introduction. We need to understand the concept of 'state', because it is extremely important to build any type of complex app, that handles multiple screens, different variables, user sessions, etc.

For more Flutter related Articles and Resources[16]

State can include anything – the app's assets, as we said, all the variables that the Flutter framework keeps about the UI, user sessions that can be shared in different parts of the app, etc.

Whenever we design an app, and start building it, we don't have to manage every state. Flutter framework takes care of a large sections, like textures. Despite that, we need some data to rebuild our UI at any moment in time. The simplest example is we press a button and the text changes on the screen. Again we press the restore button, and the text disappears. We need to provide the business logic so that it happens.

Consider a complex example, where a user adds an item to cart and that item remains at that cart as long as user is logged in. Notwithstanding, state is of two types – ephemeral and app state.

We know the meaning of the word ephemeral, it means short-lived. Some kind of state is very short-lived. We may contain it in a single widget. That is why it is also called local state. Suppose we want to show the current progress of a complex animation. Once it is done, the UI is rebuilt, and we don't want it anymore.

For that reason, we don't have to need any specialized state management techniques like 'Provider' for that. There are many other techniques as well, but in this chapter we will only learn Provider, because Google recommends it.

For ephemeral state management using setState() and a field inside the StatefulWidget's State class is enough, because, a single widget needs it, no other part of the device can access its single private variable. An app state or application state is not like that. We want to share the app state across many parts of our app, not only that, we may want it to keep between user sessions. In like manner, we can call it shared state.

To manage app state we can opt for several options. Nevertheless, Google recommends Provider, we will have a brief look at other options as well.

[16] https://sanjibsinha.com

Different approaches to state management

As we have said before, Provider is the recommended approach. Provider helps you to manage state efficiently, in a very simple and it has great flexibility. We will learn that techniques in a minute.

Before that, let us see other approaches to manage state. Using setState() and a field inside the StatefulWidget's State class is another approach; yet that is good and recommended for the ephemeral state. This lower-level approach is made up when we create a new Flutter application.

InheritedWidget & InheritedModel approach is another lower-level approach that communicates between ancestors and children in the widget tree.

Redux is another approach that is familiar to the web developers. It is a state container approach, which is also very popular among Flutter developers.

BLoC is another stream and observable based patterns, in fact before Provider has stepped in, BLoC was very popular. Still the flutter community adores BLoC.

Otherwise we might use MobX or GetX approach; the first one is a popular library based conceptualization on observables and reactions, and the second one is a simplified reactive state management solution.

There are plenty of open source resources available to learn any one of them, thoroughly. In this chapter, we will learn only Provider, the state management recommended by Google, creator of Dart programming language and Flutter framework.

A Step by Step guide to use Provider

First thing first, we have add the dependency on provider to our 'pubspec.yaml' file.

```
// pubspec.yaml
# ...

dependencies:
flutter:
    sdk: flutter

provider: ^4.0.0
```

At the time of writing this book, provider package is 4 and above. We will always check the latest version.

The app state is something that we need to modify from many different places, and to do that we have to pass around a lot of callbacks; for a complex widget tree, it will be suicidal to replace several widgets again and again. To understand this mechanism we need to find out a solution that will not

disturb the widget tree as a whole, yet the app state will modify a few widgets deep down the tree. Suppose we need to modify one widget that has hundred widgets on top of it. Without disturbing top hundred widgets, we can successfully handle the app state using Provider.

Flutter has in-built mechanisms for widgets to provide data and services to their distant descendants, it means not just the immediate children, but any widgets below them.

Provider makes it possible to forget the callbacks and InheritedWidgets. We need to understand three primary concepts:

```
ChangeNotifier
ChangeNotifierProvider
Consumer
```

ChangeNotifier is an in-built class included in the Flutter SDK, this class notifies the listeners when any change in the state of the ChangeNotifier class takes place. Any widget having hundreds widgets on the top, can subscribe to its changes.

ChangeNotifierProvider, unlike ChangeNotifier, comes from the Provider package and it provides an instance of a ChangeNotifier to the widgets, which have already subscribed to it.

Where we should place the ChangeNotifierProvider? Just above the widgets that need to access it.

```
void main() {
runApp(
    ChangeNotifierProvider(
    create: (context) => AnyModel(),
    child: HomeApp(),
    ),
);
}
```

Or we can even use MultiProvider, if we want to use multiple classes.

```
void main() {
runApp(
    MultiProvider(
    providers: [
        ChangeNotifierProvider(create: (context) => FirstModel()),
        Provider(create: (context) => SecondClass()),
    ],
    child: HomeApp(),
    ),
);
}
```

Once our designed Model is provided to the desired widgets in our app through the ChangeNotifier-Provider declaration at the top, the Consumer widgets that have subscribed to the notifications can use it.

```
return Consumer<FirstModel>(
builder: (context, value, child) {
    return Text("The value : ${value.firstModelVariable}");
},
);
```

The first rule of using Consumer widget is we need to be specific about the type of the model that we want to access. Suppose, we want 'FirstModel', so we write Consumer<FirstModel>. If the generic type <FirstModel> is not specified, the Provider package cannot help us. The Provider package is based on 'type'. Therefore, we must mention the type.

The second most important rule is we must supply the 'builder' argument of the Consumer widget. This is the only required argument of the Consumer widget. Whenever in the model class ChangeNotifier changes, the builder argument is called. Let us try to understand what is happening. Whenever the ChangeNotifier changes, the method notifyListeners() is called, and at the same time, all the builder methods of all the corresponding Consumer widgets are called.

The 'builder' is called with three arguments, the first one is quite familiar, 'context'; we get it in every build method. The second argument 'value' is the instance of the ChangeNotifier. Using that instance we can define the app state, and along with it, we can also use the data in the model according to our requirement.

The role of the third argument 'child' is quite interesting. Suppose we have a large widget subtree under our Consumer that does not change when our model changes. We can also get it through the builder argument 'child'. We have done enough talking, tried to understand the interaction between Provider, and Consumer. Nonetheless, we won't understand this concepts unless we try to implement them.

Let us start with a very simple counter model. Through Provider, we will change the counter number. We have two buttons – Increase and Decrease (Figure 8.1). Imagine a number line, using these buttons, we can either move towards the right side (positive), or towards the left side (negative).

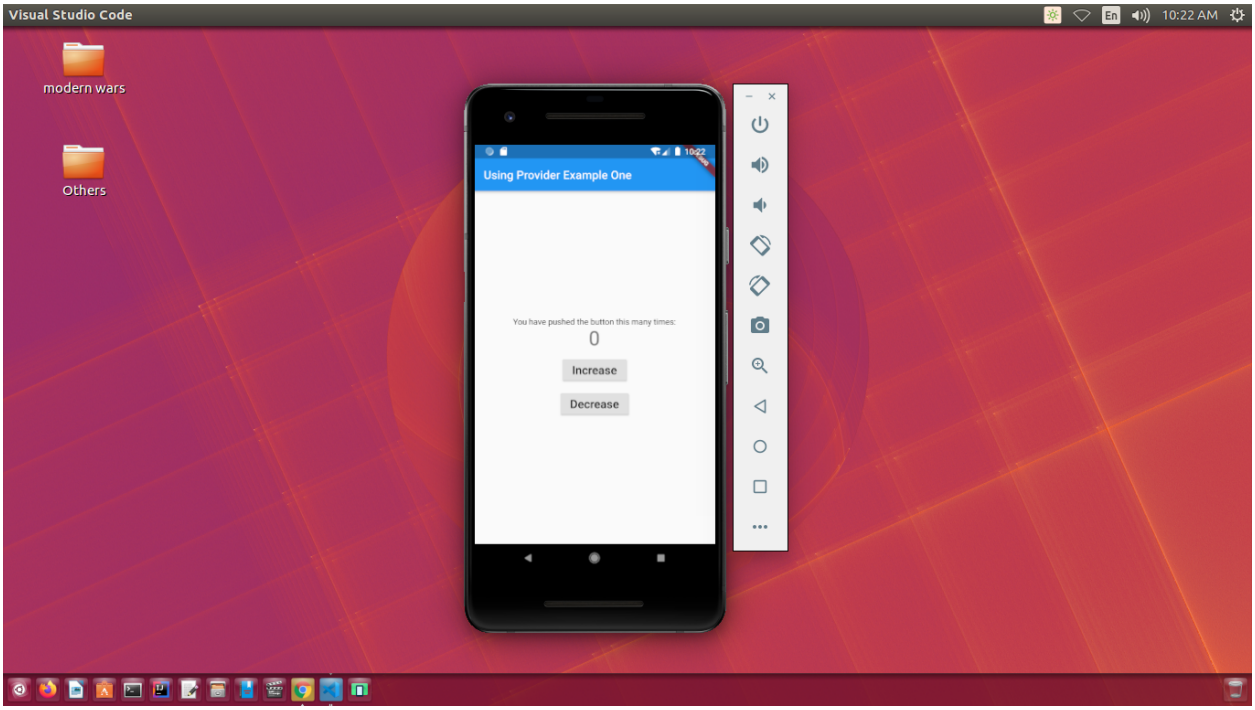

Figure 8.1 – Simple Provider example

The next two images will show you how we have increased the value and decreased the value tapping these two buttons respectively.

But before that we need to see the code and try to understand how we have used the Provider package.

```
// code 8.1

import 'package:flutter/widgets.dart';

/// using the mixin concept of dart that we have discussed
/// in our previous chapter
class CountingTheNumber with ChangeNotifier {
int value = 0;
void incrementTheValue() {
    value++;
    notifyListeners();
}

void decreaseValue() {
    value--;
    notifyListeners();
}
```

```
}
```

The above code snippets is quite simple. This is our model class through which we want to manage the state of the counter in a ChangeNotifier.

Next, we need to use the ChangeNotifierProvider in the right place.

Because we need to call two methods, using Consumer is wasteful. We don't want to change the whole UI with the help of our model data.

That is why we will use another concept - 'Provider.of', instead of using Consumer.

```
// code 8.2

import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import 'counter_class.dart';

class MyApp extends StatelessWidget {
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
    ),
    home: ChangeNotifierProvider<CountingTheNumber>(
        // it will not redraw the whole widget tree anymore
        create: (BuildContext context) => CountingTheNumber(),
        child: MyHomePage()),
    );
}
}

class MyHomePage extends StatelessWidget {
/*
MyHomePage({Key key, this.title}) : super(key: key);

final String title;
*/
```

```
33
34 @override
35 Widget build(BuildContext context) {
36     final counter = Provider.of<CountingTheNumber>(context);
37     return Scaffold(
38     appBar: AppBar(
39         title: Text('Using Provider Example One'),
40     ),
41     body: Center(
42         child: Column(
43         mainAxisAlignment: MainAxisAlignment.center,
44         children: <Widget>[
45             Text(
46             'You have pushed the button this many times:',
47             ),
48             // only Text widget listens to the notification
49             Text(
50             '${counter.value}',
51             style: Theme.of(context).textTheme.headline4,
52             ),
53             SizedBox(
54             height: 10.0,
55             ),
56             RaisedButton(
57             onPressed: () => counter.incrementTheValue(),
58             child: Text(
59                 'Increase',
60                 style: TextStyle(
61                 fontSize: 20.0,
62                 ),
63             ),
64             ),
65             SizedBox(
66             height: 10.0,
67             ),
68             RaisedButton(
69             onPressed: () => counter.decreaseValue(),
70             child: Text(
71                 'Decrease',
72                 style: TextStyle(
73                 fontSize: 20.0,
74                 ),
75             ),
```

```
            ),
        ],
        ),
    ),
    // This trailing comma makes auto-formatting nicer for build methods.
    );
}
}
```

Now, we can run the app and by tapping two buttons change the value. Before that, let us have a close look at some parts of the above code.

```
final counter = Provider.of<CountingTheNumber>(context);
```

'Provider.of', just like Consumer needs to know the type of the model. We need to specify the model 'CountingTheNumber'. Now using the 'counter' we have accessed the model data.

```
Text(
            '${counter.value}',
            style: Theme.of(context).textTheme.headline4,
            ),
…
RaisedButton(
            onPressed: () => counter.incrementTheValue(),
            child: Text(
                'Increase',
                style: TextStyle(
                fontSize: 20.0,
                ),
            ),
            ),
…
RaisedButton(
            onPressed: () => counter.decreaseValue(),
            child: Text(
                'Decrease',
                style: TextStyle(
                fontSize: 20.0,
                ),
            ),
            ),
```

The next step is running the app.

```
// code 8.3

import 'package:flutter/material.dart';
import 'utilities/first_provider_example.dart';

void main() {
runApp(MyApp());
}
```

Now we can tap the increase button (Figure 8.2).

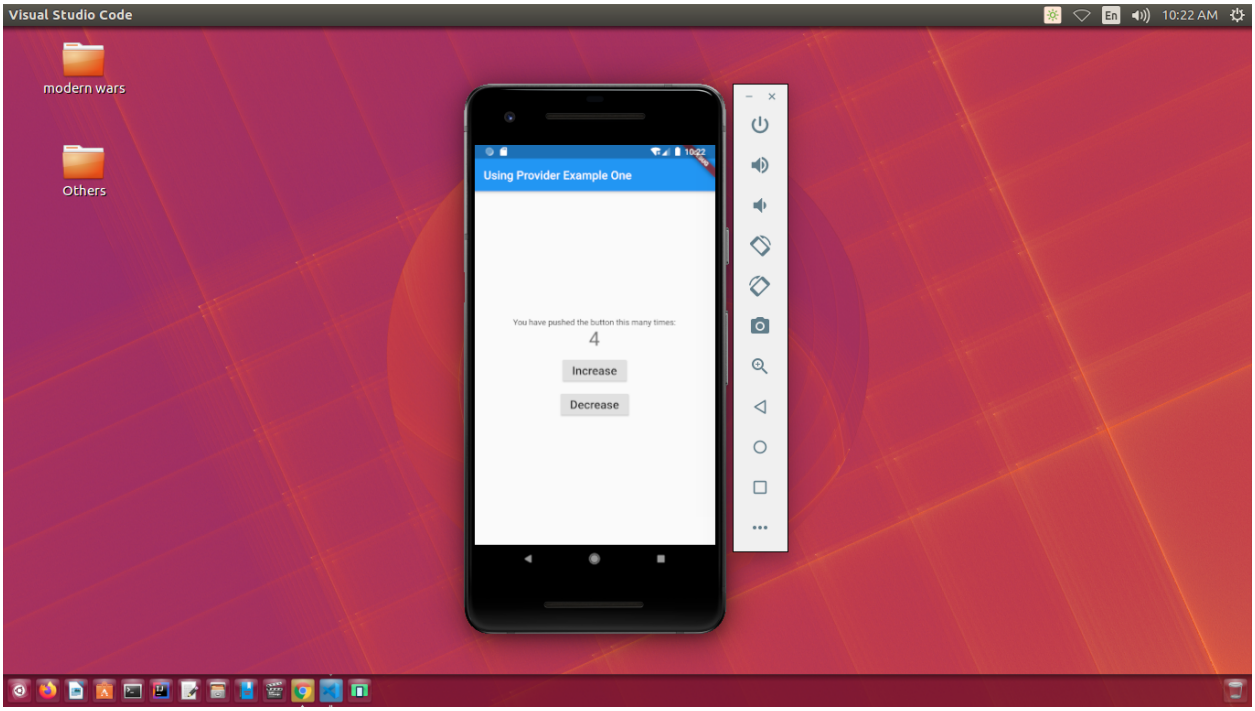

Figure 8.2 – We have tapped the increase button 4 times

After that, we can run the app once again, and it turns the counter value to 0. Now, we can test the decrease button (Figure 8.3).

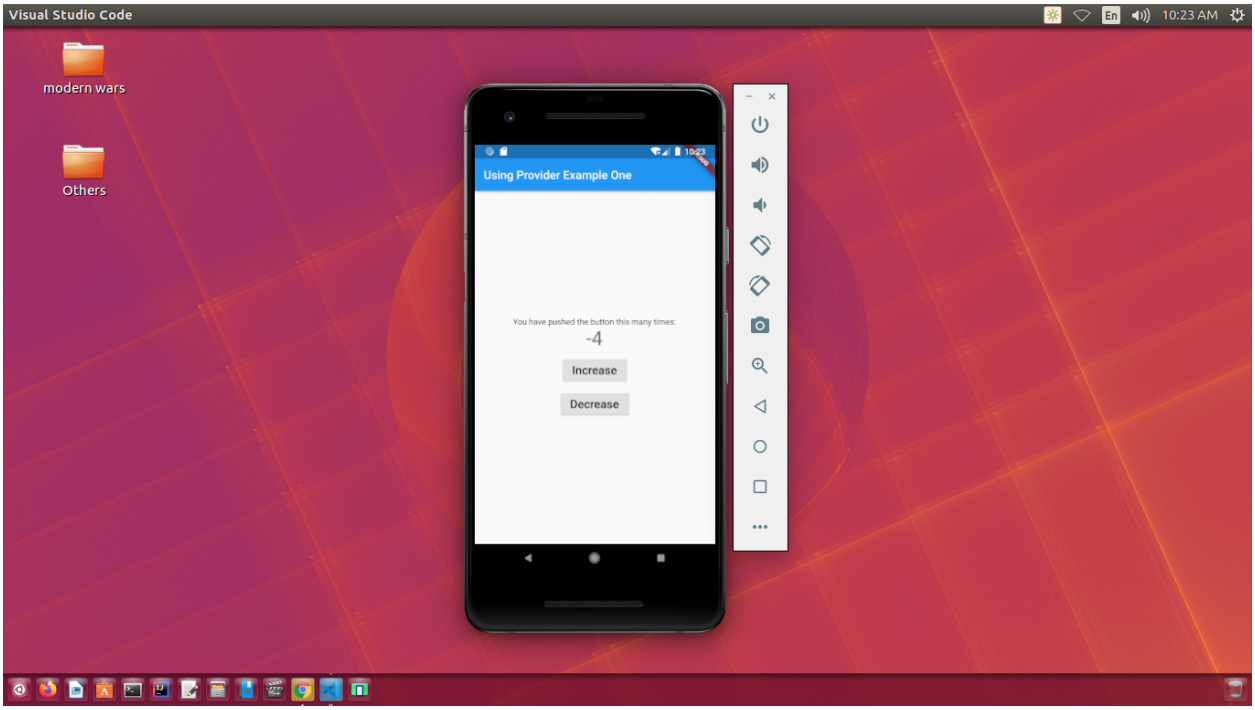

Figure 8.3 – Tapping the decrease button

The above code snippets give us an idea of how Provider package works.

Now we will take closer looks and use multi Providers and multi models to understand this process. This time we will use Consumer.

Let us start with an image. We have extended our old code added a few more generic models.

Now we can press the counter button, and besides, we will press a button to change the text below. After that, we can also press the restore button to clear that data and give an output of that.

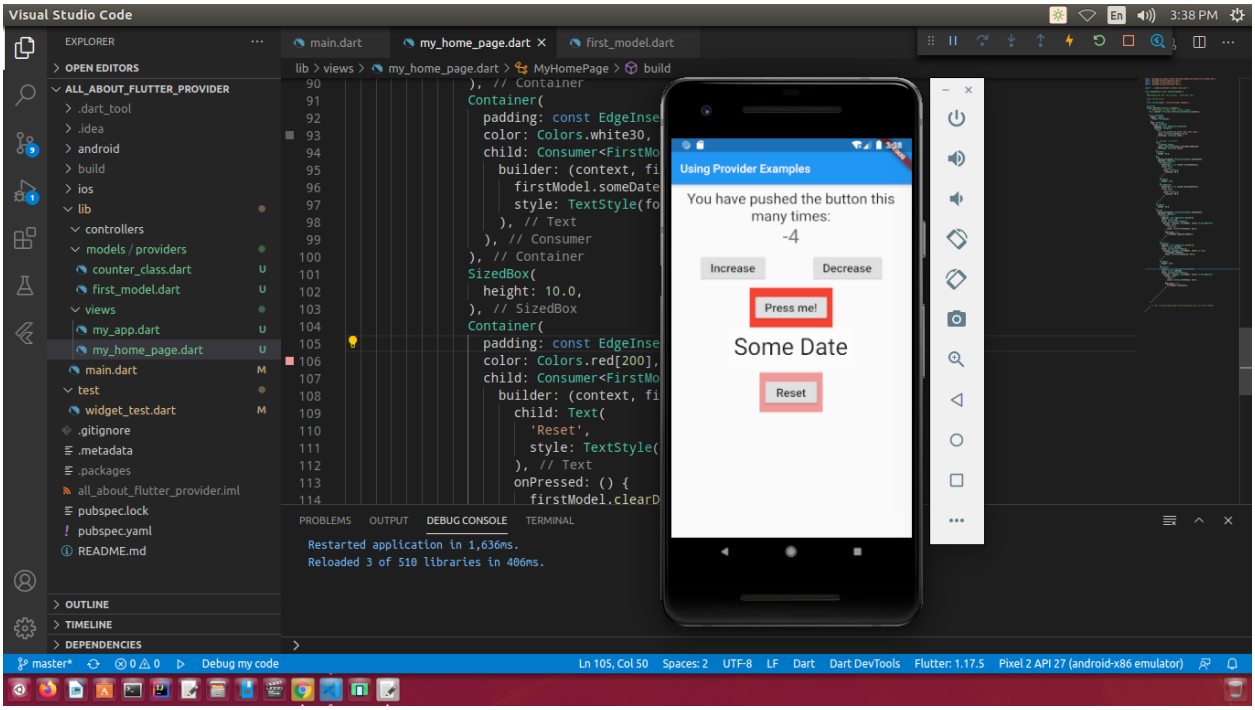

Figure 8.4 – Provider example with many models

In the above image, it is evident that we have pressed the decrease button 4 times, however, the default text data - 'Some Data', has not been affected.

Let us see the code:

```
//code 8.4

//main.dart

import 'models/providers/first_model_provider.dart';

import 'models/providers/counter_model_provider.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'models/providers/second_model_provider.dart';
import 'views/my_app.dart';

void main() {
runApp(MultiProvider(
    providers: [
    ChangeNotifierProvider(
        create: (context) => CountingTheNumber(),
    ),
```

```
    ChangeNotifierProvider(
        create: (context) => FirstModelProvider(),
    ),
    ],
    child: MyApp(),
));
}


// first_model_provider.dart

import 'package:flutter/widgets.dart';

class FirstModelProvider with ChangeNotifier {
String someDate = 'Some Date';

void supplyFirstData() {
    someDate = 'Data Changed!';
    print(someDate);
    notifyListeners();
}

void clearData() {
    someDate = 'Data Cleared!';
    print(someDate);
    notifyListeners();
}
}


// my_home_page.dart

import 'package:all_about_flutter_provider/models/providers/first_model_provider.dar\
t';
import 'package:all_about_flutter_provider/models/providers/second_model_provider.da\
rt';
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import '../models/providers/counter_model_provider.dart';

class MyHomePage extends StatelessWidget {
```

```
/*
MyHomePage({Key key, this.title}) : super(key: key);

final String title;
*/
final String title = 'Using Provider Examples';

@override
Widget build(BuildContext context) {
    /// MyHomePage is rebuilt when counter changes
    final counter = Provider.of<CountingTheNumber>(context);

    return Scaffold(
    appBar: AppBar(
        title: Text(title),
    ),
    body: SafeArea(
        child: ListView(
        padding: const EdgeInsets.all(10.0),
        children: <Widget>[
            Text(
            'You have pushed the button this many times:',
            style: TextStyle(fontSize: 25.0),
            textAlign: TextAlign.center,
            ),

            /// consumer or selector
            Text(
            '${counter.value}',
            style: Theme.of(context).textTheme.headline4,
            textAlign: TextAlign.center,
            ),
            SizedBox(
            height: 10.0,
            ),
            Row(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: <Widget>[
                RaisedButton(
                onPressed: () => counter.increaseValue(),
                child: Text(
                    'Increase',
                    style: TextStyle(
```

```
                    fontSize: 20.0,
                    ),
                ),
                ),
                SizedBox(
                height: 10.0,
                ),
                RaisedButton(
                onPressed: () => counter.decreaseValue(),
                child: Text(
                    'Decrease',
                    style: TextStyle(
                    fontSize: 20.0,
                    ),
                ),
                ),
            ],
            ),
            SizedBox(
            height: 10.0,
            ),
            Column(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: <Widget>[
                Container(
                padding: const EdgeInsets.all(10.0),
                color: Colors.red,
                child: Consumer<FirstModelProvider>(
                    builder: (context, firstModelProvider, child) =>
                        RaisedButton(
                    child: Text(
                        'Press me!',
                        style: TextStyle(fontSize: 20.0),
                    ),
                    onPressed: () {
                        firstModelProvider.supplyFirstData();
                    },
                    ),
                ),
                ),
                Container(
                padding: const EdgeInsets.all(10.0),
                color: Colors.white30,
```

```
                child: Consumer<FirstModelProvider>(
                    builder: (context, firstModelProvider, child) => Text(
                    firstModelProvider.someDate,
                    style: TextStyle(fontSize: 40.0),
                    ),
                ),
                ),
                SizedBox(
                height: 10.0,
                ),
                Container(
                padding: const EdgeInsets.all(10.0),
                color: Colors.red[200],
                child: Consumer<FirstModelProvider>(
                    builder: (context, firstModelProvider, child) =>
                        RaisedButton(
                    child: Text(
                        'Reset',
                        style: TextStyle(fontSize: 20.0),
                    ),
                    onPressed: () {
                        firstModelProvider.clearData();
                    },
                    ),
                ),
                ),
            ],
            ),
        ],
        ),
    ),

    /// This trailing comma makes auto-formatting nicer for build methods.
    );
}
}
```

In the above code, we have used two Providers, inside the main() function.

```
runApp(MultiProvider(
    providers: [
    ChangeNotifierProvider(
        create: (context) => CountingTheNumber(),
    ),
    ChangeNotifierProvider(
        create: (context) => FirstModelProvider(),
    ),
    ],
    child: MyApp(),
));
```

Along with the 'CountingTheNumber' model, we have used a new 'ChangeNotifier' model - 'FirstModelProvider' class. And finally, inside the 'MyHomePage' widget, we have used the Consumer concepts.

```
child: Consumer<FirstModelProvider>(
                    builder: (context, firstModelProvider, child) =>
                        RaisedButton(
                    child: Text(
                        'Press me!',
                        style: TextStyle(fontSize: 20.0),
                    ),
                    onPressed: () {
                        firstModelProvider.supplyFirstData();
                    },
                    ),
                ),
```

Because this Consumer's builder argument returns a RaisedButton() widget, we have used the onPressed() argument to call one of model methods. It gives us the next figure (Figure 8.5).

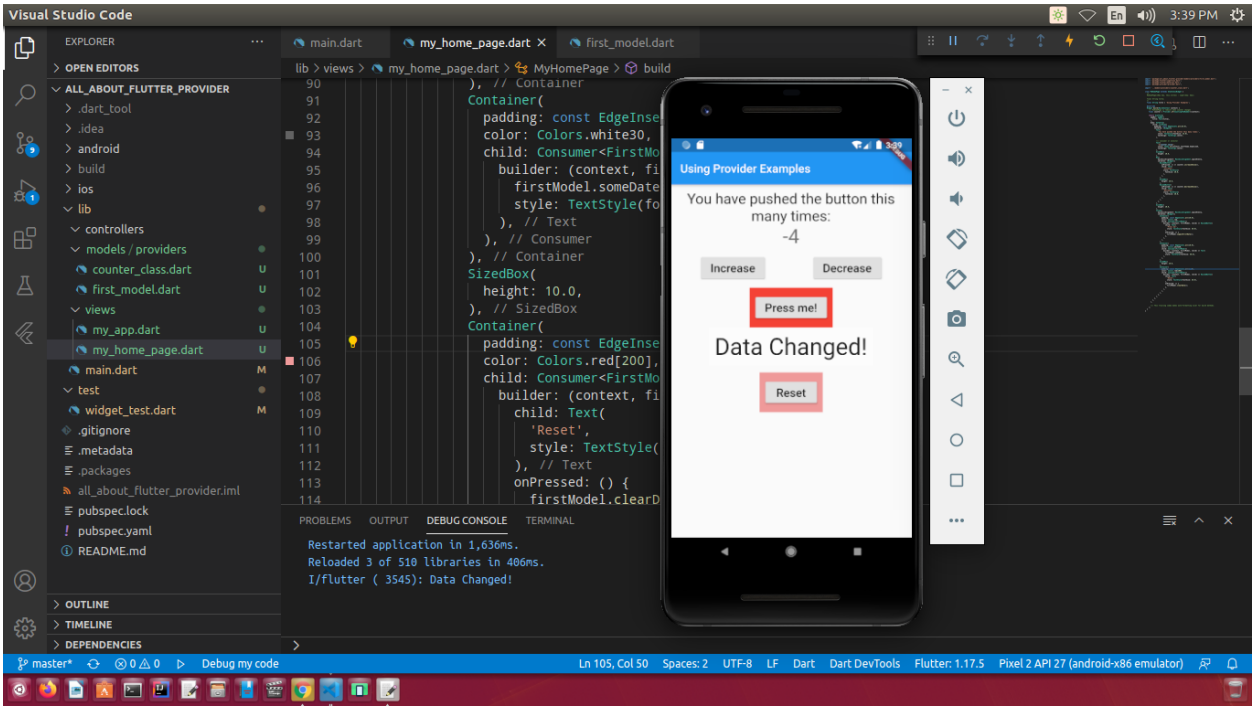

Figure 8.5 – The 'Press me' button has been pressed and the value of the model class has also been changed

If we click the 'Reset' button, the data has been cleared. The following figure (Figure 8.5) shows that display of the screen.

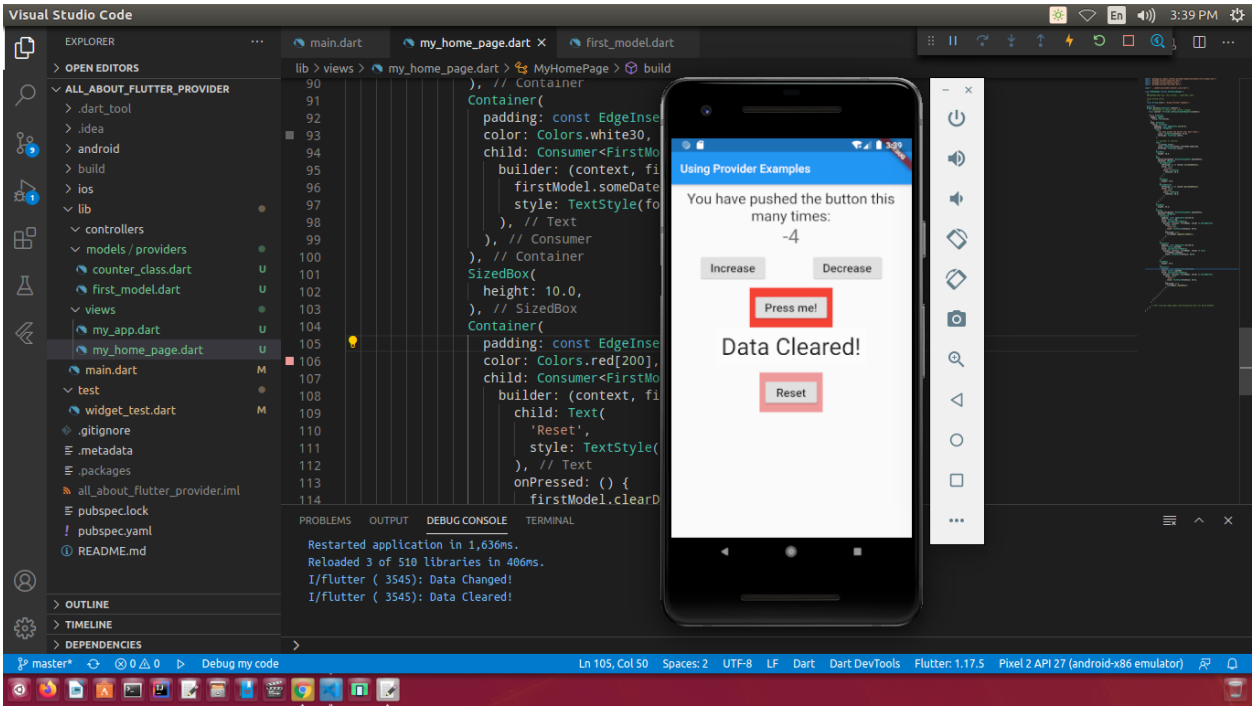

Figure 8.6 – We have pressed the 'Reset' button, and the corresponding value of model class is displayed

Now we are going to add another model class in the next code snippets. It will add another button that will display the first name.

```
//code 8.5

// main.dart

import 'models/providers/first_model_provider.dart';

import 'models/providers/counter_model_provider.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'models/providers/second_model_provider.dart';
import 'views/my_app.dart';

void main() {
runApp(MultiProvider(
    providers: [
    ChangeNotifierProvider(
        create: (context) => CountingTheNumber(),
    ),
    ChangeNotifierProvider(
        create: (context) => FirstModelProvider(),
```

```
    ),
    ChangeNotifierProvider(
        create: (context) => SecondModelProvider(),
    ),
    ],
    child: MyApp(),
));
}


// second_model_provider.dart

import 'package:flutter/widgets.dart';

class SecondModelProvider with ChangeNotifier {
String name = 'Some Name';
int age = 0;

void getFirstName() {
    name = 'Json';
    print(name);
    notifyListeners();
}
}


// my_home_page.dart

import 'package:all_about_flutter_provider/models/providers/first_model_provider.dar\
t';
import 'package:all_about_flutter_provider/models/providers/second_model_provider.da\
rt';
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import '../models/providers/counter_model_provider.dart';

class MyHomePage extends StatelessWidget {
/*
MyHomePage({Key key, this.title}) : super(key: key);

final String title;
```

```
64  */
65  final String title = 'Using Provider Examples';
66
67  @override
68  Widget build(BuildContext context) {
69      /// MyHomePage is rebuilt when counter changes
70      final counter = Provider.of<CountingTheNumber>(context);
71
72      return Scaffold(
73      appBar: AppBar(
74          title: Text(title),
75      ),
76      body: SafeArea(
77          child: ListView(
78          padding: const EdgeInsets.all(10.0),
79          children: <Widget>[
80              Text(
81              'You have pushed the button this many times:',
82              style: TextStyle(fontSize: 25.0),
83              textAlign: TextAlign.center,
84              ),
85
86              /// consumer or selector
87              Text(
88              '${counter.value}',
89              style: Theme.of(context).textTheme.headline4,
90              textAlign: TextAlign.center,
91              ),
92              SizedBox(
93              height: 10.0,
94              ),
95              Row(
96              mainAxisAlignment: MainAxisAlignment.spaceEvenly,
97              children: <Widget>[
98                  RaisedButton(
99                  onPressed: () => counter.increaseValue(),
100                 child: Text(
101                     'Increase',
102                     style: TextStyle(
103                     fontSize: 20.0,
104                     ),
105                 ),
106                 ),
```

```
                SizedBox(
                height: 10.0,
                ),
                RaisedButton(
                onPressed: () => counter.decreaseValue(),
                child: Text(
                    'Decrease',
                    style: TextStyle(
                    fontSize: 20.0,
                    ),
                ),
                ),
            ],
            ),
            SizedBox(
            height: 10.0,
            ),
            Column(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: <Widget>[
                Container(
                padding: const EdgeInsets.all(10.0),
                color: Colors.red,
                child: Consumer<FirstModelProvider>(
                    builder: (context, firstModelProvider, child) =>
                        RaisedButton(
                    child: Text(
                        'Press me!',
                        style: TextStyle(fontSize: 20.0),
                    ),
                    onPressed: () {
                        firstModelProvider.supplyFirstData();
                    },
                    ),
                ),
                ),
                Container(
                padding: const EdgeInsets.all(10.0),
                color: Colors.white30,
                child: Consumer<FirstModelProvider>(
                    builder: (context, firstModelProvider, child) => Text(
                    firstModelProvider.someDate,
                    style: TextStyle(fontSize: 40.0),
```

```
150                     ),
151                 ),
152                 ),
153                 SizedBox(
154                 height: 10.0,
155                 ),
156                 Container(
157                 padding: const EdgeInsets.all(10.0),
158                 color: Colors.red[200],
159                 child: Consumer<FirstModelProvider>(
160                     builder: (context, firstModelProvider, child) =>
161                         RaisedButton(
162                     child: Text(
163                         'Reset',
164                         style: TextStyle(fontSize: 20.0),
165                     ),
166                     onPressed: () {
167                         firstModelProvider.clearData();
168                     },
169                     ),
170                 ),
171                 ),
172                 SizedBox(
173                 height: 10.0,
174                 ),
175                 Container(
176                 padding: const EdgeInsets.all(10.0),
177                 color: Colors.white30,
178                 child: Consumer<SecondModelProvider>(
179                     builder: (context, secondModel, child) => Text(
180                     secondModel.name,
181                     style: TextStyle(fontSize: 40.0),
182                     ),
183                 ),
184                 ),
185                 SizedBox(
186                 height: 10.0,
187                 ),
188                 Container(
189                 padding: const EdgeInsets.all(10.0),
190                 color: Colors.red[200],
191                 child: Consumer<SecondModelProvider>(
192                     builder: (context, secondModel, child) => RaisedButton(
```

```
                    child: Text(
                        'Get First Name',
                        style: TextStyle(fontSize: 20.0),
                    ),
                    onPressed: () {
                        secondModel.getFirstName();
                    },
                    ),
                ),
                ),
            ],
            ),
        ],
        ),
    ),

    /// This trailing comma makes auto-formatting nicer for build methods.
    );
}
}
```

This part of the code has handled the Consumer section. Therefore, let us check that part first.

```
Container(
                padding: const EdgeInsets.all(10.0),
                color: Colors.white30,
                child: Consumer<SecondModelProvider>(
                    builder: (context, secondModel, child) => Text(
                    secondModel.name,
                    style: TextStyle(fontSize: 40.0),
                    ),
                ),
                ),
                SizedBox(
                height: 10.0,
                ),
                Container(
                padding: const EdgeInsets.all(10.0),
                color: Colors.red[200],
                child: Consumer<SecondModelProvider>(
                    builder: (context, secondModel, child) => RaisedButton(
                    child: Text(
                        'Get First Name',
```

```
                        style: TextStyle(fontSize: 20.0),
                    ),
                    onPressed: () {
                        secondModel.getFirstName();
                    },
                    ),
                ),
                ),
```

We are able to add another feature of state management through Provider. The second model Provider is a simple class.

```
// second_model_provider.dart

import 'package:flutter/widgets.dart';

class SecondModelProvider with ChangeNotifier {
String name = 'Some Name';
int age = 0;

void getFirstName() {
    name = 'Json';
    print(name);
    notifyListeners();
}
}
```

The next figure (Figure 8.7) will show how Provider and Consumer work together. First, we have pressed the decrease button for 3 times. Next, we have pressed the 'Press me' button, and the 'Data Changed'. After that, finally, we have pressed the 'Get First Name' button, and the name appears on the screen. Each Consumer widget has persisted its state, one button-press does not affect the other. The changed-data stays on the screen.

Before concluding this chapter, we will learn how we can separate business logic, application logic and screen-view.

To do that, we will keep our models inside the 'model' folder and keep our business logic there. We will keep our application logic inside the 'controller' folder, and finally we get the screen-view inside the 'view' folder.

Riverpod, another state management package, Riverpod migration, WidgetRef ref, and What is new in Riverpod

To use Riverpod state management package in Flutter, we need to assure ourselves a few things first.

Firstly, we need to use Riverpod latest package: ^0.14.0.

Secondly, we must upgrade our Flutter and Dart SDK.

Finally, we need to migrate to latest Riverpod from $^{0}.^{13}.^{0}$. ^{to}0.14.0.

In this chapter, we're going to see how we can successfully migrate from old Riverpod version to the newest version.

As regards to this tutorial, you may get the full code in this GitHub repository mentioned in the last chapter. And besides, if you want to dig deep into the flutter state management, you may read all updated Flutter articles in my website. For more Flutter related Articles and Resources[17]

By the way, the above mentioned flutter app using Riverpod was built using Riverpod ^0.13.0. and that is not working anymore.

Therefore the only solution left to us is Riverpod migration.

To migrate successfully from the old Riverpod to the newest version, we need to install the migration tool, that is Riverpod command line interface.

```
dart pub global activate riverpod_cli
```

However, we need to change the path of the executables.

```
export PATH="$PATH":"$HOME/.pub-cache/bin"
```

Now, as a result, we can use Riverpod Command line interface.

```
riverpod --help
```

And it gives us the following output.

[17] https://sanjibsinha.com

```
Usage: riverpod <command> [arguments]

Global options:
-h, --help    Print this usage information.

Available commands:
migrate   Analyse a project using Riverpod and migrate it to the latest version avai\
lable

Run "riverpod help <command>" for more information about a command.
...
```

All right, now we are ready to migrate and it looks like the following screenshot.

Figure 8.7 – Riverpod Command line interface

Next, we need to migrate from old Riverpod to the new, by issuing the following command.

```
riverpod migrate
```

Consequently, it might display the following message depending on your project and ask your permission to go ahead.

```
Widget build(BuildContext context, ScopedReader watch) {
-   StateProviderModel state = watch(provider.state);
+   StateProviderModel state = watch(provider);
}

Accept change (y = yes, n = no [default], A = yes to all, q = quit)?
Let us accept the change and it looks like the following screenshot.
```

Figure 8.8 – Riverpod migration taking place

As we have successfully migrated from the old version of Riverpod to the newest version, now we can run the Riverpod flutter app without any worry.

Figure 8.9 – Riverpod Flutter App after migration without any error

Incidentally, we don't have to change anything in our previous code. Certainly, we've only changed the theme primary swatch colour from blue to orange, so that we can differentiate between the old and the new.

Subsequently, we can click any button to change the provider value or you may say the state of the app.

Figure 8.9 – Riverpod flutter app working after migration

Watch the change on the screen, and it works like a charm!

Why we need the latest Flutter and Dart SDK?

To use the latest Riverpod state management package we need to upgrade Flutter first. Therefore, for Riverpod latest version ^0.14.0, or flutter_riverpod:1.0.0-dev.11, our Flutter SDK should also be the latest; that is Flutter 2.5.3 • channel stable • and a Dart SDK >= 2.14.0.

If we want to use the latest Riverpod first upgrade Flutter and Dart SDK. Otherwise, we get the error: Object.hash not found.

To be more precise, the error-output is quite long.

```
../../packages/riverpod/lib/src/common.dart:173:30: Error: Method not found: 'Object\
.hash'.
int get hashCode => Object.hash(runtimeType, value);
                            ^^^^
../../packages/riverpod/lib/src/common.dart:374:30: Error: Method not found: 'Object\
.hash'.
int get hashCode => Object.hash(runtimeType, previous);
                            ^^^^
../../packages/riverpod/lib/src/common.dart:415:30: Error: Method not found: 'Object\
.hash'.
int get hashCode => Object.hash(runtimeType, error, stackTrace);
                            ^^^^


FAILURE: Build failed with an exception.

* Where:
Script '/home/nobu/snap/flutter/common/flutter/packages/flutter_tools/gradle/flutter\
.gradle' line: 1035

```

```
* What went wrong:
Execution failed for task ':app:compileFlutterBuildDebug'.
> Process 'command '/home/nobu/snap/flutter/common/flutter/bin/flutter'' finished wi\
th non-zero exit value 1

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug opti\
on to get more log output. Run with --scan to get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 38s
Running Gradle task 'assembleDebug'...
Running Gradle task 'assembleDebug'... Done                         42.4s
Exception: Gradle task assembleDebug failed with exit code 1
```

With reference to the above error, let us know a few important facts about hash code.

A hash code is a single integer which represents the state of the object. Not only that, it also affects operator == comparisons.

Now, each Object has unique identity, and the single integer hash code implements that. Now equality of two objects depends on default operator == implementation. Two objects are equal if and only if they are identical.

While using Riverpod state management, we must keep one thing in mind.

Why?

Because, it's an important concept. If the default operator == is overridden to use the object state the hash code must also be changed to represent that state.

Unequal objects may have the same hash code, but if it happens too often, clashes occur and the efficiency reduces affecting the data structures, such as HashSet or HashMap.

Is Riverpod better than Provider?

In the flutter community, many developers think Riverpod is better than Provider, which is an earlier-released-state-management package. However, flutter creators still recommend Provider as the stable state management mechanism at the time of writing this post.

According to the creator of both packages, Remi Rousselet, and other flutter developers as well, Provider has had some limitations. Riverpod has overcome those limitations.

Certainly, provider is a wrapper class or rather a kind of simplification of InheritedWidgets. On the contrary, Remi Rousselet has re-implemented InheritedWidget from the scratch and made Riverpod.

Both the state management packages have similarity and also has many differences.

As a state management package, provider has become enormously popular. Then why do we need another mechanism in place?

Provider can create, observe and dispose state without rebuilding widgets. It makes objects visible in Flutter's devtool. Testing and composing is not difficult. Since the data flow is unidirectional, the app is scalable.

These are all advantages that has made provider package so popular.

However, Riverpod has extended these benefits in a great way. It is compile-safe. It doesn't throw any run time exception. With riverpod we can have multiple providers of same type. It can make a provider private.

Above all, Riverpod is flutter independent as we can achieve the above mentioned features by not using InheritedWidgets any more. Riverpod implements its own mechanism.

What is new in Riverpod?

Firstly, we need to add the latest dependency.

```
dependencies:
flutter:
    sdk: flutter
provider: ^6.0.0
flutter_riverpod: ^1.0.0-dev.11
```

Secondly, we can wrap our root with ProviderScope.

```
void main() {
runApp(
    const ProviderScope(
    child: ProviderAppSample(),
    ),
}
```

From now on, all the providers we'll create, the ProviderScope will store the state of them.

The new concept in the latest Riverpod is the ref object of type WidgetRef.

What is a WidgetRef?

In the latest Riverpod, we can watch or read provider's value by using this ref object.

When we use Consumer or ConsumerState, the WidgetRef object is available as an argument, and it can interact with any provider.

As the BuildContext allows us to access the ancestor widgets in the widget tree, the WidgetRef does the same, allowing us to interact with any provider.

How does that happen?

Because all Riverpod providers are global so that we can access them easily. As a result, we can handle state management logic outside the widget tree.

We need to remember that Provider does not let us change the value. To do that we need to create a StateProvider. It will be a global value.

```
final referenceValue = StateProvider((ref) => 0);
```

Now, we can watch and read the provider's value quite easily.

```
import 'package:flutter/material.dart';

import 'package:flutter_riverpod/flutter_riverpod.dart';

class ProviderAppSample extends StatelessWidget {
const ProviderAppSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: ProviderHome(),
    );
}
}

final referenceValue = StateProvider((ref) => 0);

class ProviderHome extends ConsumerWidget {
const ProviderHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context, WidgetRef ref) {
    final counterWatch = ref.watch(referenceValue);
    final counterRead = ref.read(referenceValue);
    return Scaffold(
    body: Center(
        child: Column(
```

```
        children: [
            Container(
            margin: const EdgeInsets.all(20),
            padding: const EdgeInsets.all(20),
            child: Text(
                '${counterWatch.state}',
                style: const TextStyle(
                fontSize: 100,
                fontFamily: 'Allison',
                fontWeight: FontWeight.bold,
                color: Colors.red,
                ),
            ),
            ),
            const SizedBox(
            height: 20,
            ),
            ElevatedButton(
            onPressed: () => counterRead.state++,
            child: const Text(
                'Press to Increment',
                style: TextStyle(
                fontSize: 30,
                color: Colors.white,
                ),
            ),
            ),
        ],
        ),
    ),
    );
}
}
```

And we can access the provider's state quite easily.

```
Widget build(BuildContext context, WidgetRef ref) {
    final counterWatch = ref.watch(referenceValue);
    final counterRead = ref.read(referenceValue);
...
```

And after that, we can either watch the change, and implement the change.

```
child: Text(
                '${counterWatch.state}',
...
ElevatedButton(
            onPressed: () => counterRead.state++,
...
```

As a result, we can press the button and the number grows by 1.

Model class with StateNotifierProvider in new Riverpod

Before we learn how to use Riverpod StateNotifierProvider, let us know a few facts about state management in flutter.

To handle state management in a more systematic way in my website, we've created a separate category named "State Management" and that category has two sub-categories named "Riverpod" and "Provider".

Therefore you may read the updated flutter articles there. For more Flutter related Articles and Resources[18]

Although we've started with Riverpod StateNotifierProvider, the next article will be on a gentle introduction on state management itself. And after that, we'll delve into Provider and Riverpod state management package simultaneously, so that a beginner can understand the core concepts behind these beautiful state management system.

Usually for simple counter we don't need Riverpod StateNotifierProvider. It helps us to manage more complex data model. However, let's use a simple use case like a counter app that changes its state and the number increases as the user presses the button below.

With reference to either Provider or Riverpod package, we can run the below code sample firstly. Secondly, we'll discuss code.

The above counter displays a number 0. From where that number has come, we'll see in our code in a minute.

Now, as we press the floating action button below, the state of the app changes and we get the next number 1.

Broadly speaking, it's a very simple app and usually comes by default when we create a flutter app. However, that's a stateful widget that manages state of the app automatically.

To tell the truth, we don't have any stateful widget in place. But we're managing state with the help of Riverpod package.

We have to add the dependency first in our pubspec.yaml file.

[18] https://sanjibsinha.com

```
dependencies:
flutter:
    sdk: flutter
provider: ^6.0.0
flutter_riverpod: ^1.0.0-dev.11
```

Then we have a simple counter data model class like the following one.

```
import 'package:flutter_riverpod/flutter_riverpod.dart';

class Countering extends StateNotifier<int> {
Countering() : super(0);

void increment() => state++;
}
```

The above class sets the initial state in the constructor.

```
Countering() : super(0);
```

The number 0 in the above screenshot comes from the initial state that we've mentioned in the constructor.

Next, we can create a new provider like the following.

```
final counterProvider =
    StateNotifierProvider<Countering, int>((ref) => Countering());
```

Now, we can use the following code to get the state of the app.

```
final counter = ref.watch(counterProvider);
```

And, to change the state we use read method.

```
onPressed: () {
        ref.read(counterProvider.notifier).increment();
        },
```

In the Countering model, we have used StateNotifier as an extension that acts as the place where our business logic goes. The Riverpod service reads and writes using Countering model that extends StateNotifier and talks to the relevant widgets.

Let's see the full code.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_artisan/models/countering.dart';
import 'package:flutter_riverpod/flutter_riverpod.dart';

class StateNotifierProviderAppSample extends StatelessWidget {
const StateNotifierProviderAppSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: StateNotifierProviderHome(),
    );
}
}

final counterProvider =
    StateNotifierProvider<Countering, int>((ref) => Countering());

class StateNotifierProviderHome extends ConsumerWidget {
const StateNotifierProviderHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context, WidgetRef ref) {
    final counter = ref.watch(counterProvider);
    return Scaffold(
    body: Center(
        child: Container(
        margin: const EdgeInsets.all(20),
        padding: const EdgeInsets.all(20),
        child: Text(
            '$counter',
            style: const TextStyle(
            fontSize: 60,
            color: Colors.red,
            ),
        ),
        ),
    ),
    floatingActionButton: FloatingActionButton(
        backgroundColor: Colors.deepOrange,
        tooltip: 'Press to Increment',
        onPressed: () {
        ref.read(counterProvider.notifier).increment();
```

```
        },
        child: const Icon(Icons.add),
    ),
    );
}
}
```

Most importantly we have used ProviderScope so every child gets the service properly.

```
void main() {
runApp(
    const ProviderScope(
    child: StateNotifierProviderAppSample(),
    ),
}
```

From onward we'll get updated articles on State Management in my website, so feel free to read those articles and if you want to know anything, please drop a line in the comment section.

For more Flutter related Articles and Resources[19]

Model-View-Controller Patterns

First of all, we need to update pubspec.yaml, because we want some special fonts to be displayed.

```
//code 8.6

dependencies:
flutter:
    sdk: flutter
provider: ^4.3.2

# To add assets to your application, add an assets section, like this:
assets: [images/]

fonts:
#   - family: Schyler
#     fonts:
#       - asset: fonts/Schyler.ttf
#       - asset: fonts/Schyler-Italic.ttf
```

[19] https://sanjibsinha.com

```
#         style: italic
    - family: Trajan Pro
    fonts:
        - asset: fonts/Trajan Pro Regular.ttf
#       - asset: fonts/TrajanPro_Bold.ttf
#         weight: 700
    - family: Sacramento
    fonts:
- asset: fonts/Sacramento-Regular.ttf
```

Next, we need two different models, ChangeNotifier, in our models folder. The first one is the following 'FirstModel' class.

```
//code 8.7

model/first_model.dart

import 'package:flutter/widgets.dart';

class FirstModel with ChangeNotifier {
String name = 'name';
void changeName() {
    name = 'Name Changed!';
    print(name);
    notifyListeners();
}

void clearName() {
    name = ' ';
    print(name);
    notifyListeners();
}
}
```

And the second model class is the 'MobileModel' that has a list of selected colors of which we will choose one for the background, and another for the mobile. We will display the mobile color on the foreground, and the background will be different. Pressing the icon of the respective mobile will change the color of both – foreground and background. At the same time a text will be displayed to make us aware that foreground and background colors have been changed.

```dart
//code 8.8

model/mobile_model.dart

import 'package:flutter/material.dart';
import 'package:flutter/widgets.dart';

class MobileModel with ChangeNotifier {
String backgroundColorOfFirst = 'Background';
String mobileColorOfFirst = 'Mobile';
String backgroundColorOfSecond = 'Background';
String mobileColorOfSecond = 'Mobile';
List<Color> selection = [
    Colors.yellow,
    Colors.blue,
    Colors.orange,
    Colors.pinkAccent,
    Colors.green,
    Colors.limeAccent,
];

void changeColorToPurple() {
    backgroundColorOfFirst = 'Background \n Purle';
    mobileColorOfFirst = 'Mobile \n White.';
    selection[0] = Colors.purple;
    selection[4] = Colors.white;
    notifyListeners();
}

void changeColorToRed() {
    backgroundColorOfSecond = 'Background \n Black';
    mobileColorOfSecond = 'Mobile \n Red.';
    selection[1] = Colors.black;
    selection[5] = Colors.red;
    notifyListeners();
}

void restoreOldColorOfFirstMobile() {
    backgroundColorOfFirst = 'Background \n Yellow';
    mobileColorOfFirst = 'Mobile \n Green.';
    selection[0] = Colors.yellow;
    selection[4] = Colors.green;
    notifyListeners();
```

```
}

void restoreOldColorOfSecondMobile() {
    backgroundColorOfSecond = 'Background \n Blue';
    mobileColorOfSecond = 'Mobile \n Limeaccent.';
    selection[1] = Colors.blue;
    selection[5] = Colors.limeAccent;
    notifyListeners();
}
}
```

The model classes are the sources of date. That data should be displayed on the screen-view. Not only that, that data must be changed on the tap of the icon.

Therefore, we need some subscribers or Consumers who will get that data and pass them to the view accordingly. Who will control that? The controllers. The controller will stay between model and view; the controllers' job is simple, it will play the role of the communicator who will manage the communication between model and view.

The data-source or model does not know where its data are going. The view does not know where from the data are coming. The controller knows everything. It controls every operation.

We have many controller widgets that will control different types of operations, such as one will control the foreground color, another will change background color,one controller will manage the text display, another will restore the value again, etc. Even we have some controllers that will also decide what type of text style we will follow.

We have kept those controllers in two separate files inside 'controller' folder. One controller file is mobile specific. Another is page specific. The mobile specific controllers are as follows:

```
// code 8.9

// controller/mobile_controller.dart

import 'package:first_flutter_app/model/mobile_model.dart';
import 'package:flutter/material.dart';
import 'package:flutter/widgets.dart';
import 'package:provider/provider.dart';

Widget changeColorButtonToPurple() => Column(
    children: [
        Container(
        padding: const EdgeInsets.all(10.0),
        child: Consumer<MobileModel>(
            builder: (context, value, child) => Container(
```

```
            padding: const EdgeInsets.all(15.0),
            child: FloatingActionButton(
                backgroundColor: value.selection[0],
                onPressed: () {
                value.changeColorToPurple();
                },
                child: Icon(
                Icons.mobile_screen_share,
                color: value.selection[4],
                ),
            ),
            ),
        ),
        ),
        Divider(
        thickness: 2.0,
        ),
        Consumer<MobileModel>(
        builder: (context, value, _) => Text(
            value.backgroundColorOfFirst,
            style: TextStyle(
            fontFamily: 'Trajan Pro',
            fontSize: 20.0,
            fontWeight: FontWeight.bold,
            ),
        ),
        ),
        Divider(
        thickness: 2.0,
        ),
        Consumer<MobileModel>(
        builder: (context, value, _) => Text(value.mobileColorOfFirst,
            style: TextStyle(
                fontFamily: 'Trajan Pro',
                fontSize: 20.0,
                fontWeight: FontWeight.bold,
            )),
        ),
    ],
    );

Widget changeColorButtonToRed() => Column(
    children: [
```

```
59          Container(
60          padding: const EdgeInsets.all(10.0),
61          child: Consumer<MobileModel>(
62              builder: (context, value, child) => Container(
63              padding: const EdgeInsets.all(15.0),
64              child: FloatingActionButton(
65                  backgroundColor: value.selection[1],
66                  onPressed: () {
67                  value.changeColorToRed();
68                  },
69                  child: Icon(
70                  Icons.mobile_screen_share,
71                  color: value.selection[5],
72                  ),
73              ),
74              ),
75          ),
76          ),
77          Divider(
78          thickness: 2.0,
79          ),
80          Consumer<MobileModel>(
81          builder: (context, value, _) => Text(
82              value.backgroundColorOfSecond,
83              style: TextStyle(
84              fontFamily: 'Trajan Pro',
85              fontSize: 20.0,
86              fontWeight: FontWeight.bold,
87              ),
88          ),
89          ),
90          Divider(
91          thickness: 2.0,
92          ),
93          Consumer<MobileModel>(
94          builder: (context, value, _) => Text(value.mobileColorOfSecond,
95              style: TextStyle(
96                  fontFamily: 'Trajan Pro',
97                  fontSize: 20.0,
98                  fontWeight: FontWeight.bold,
99              )),
100         ),
101     ],
```

```
    );

Widget restoreOldColorOfFirstMobile() => Container(
    padding: const EdgeInsets.all(10.0),
    child: Consumer<MobileModel>(
        builder: (context, value, child) => Container(
        padding: const EdgeInsets.all(10.0),
        child: RaisedButton(
            onPressed: () => value.restoreOldColorOfFirstMobile(),
            child: Text(
            'Restore',
            style: TextStyle(
                fontFamily: 'Sacramento',
                fontSize: 25.0,
                fontWeight: FontWeight.bold,
            ),
            ),
        ),
        ),
    ),
    );

Widget restoreOldColorOfSecondMobile() => Container(
    padding: const EdgeInsets.all(10.0),
    child: Consumer<MobileModel>(
        builder: (context, value, child) => Container(
        padding: const EdgeInsets.all(10.0),
        child: RaisedButton(
            onPressed: () => value.restoreOldColorOfSecondMobile(),
            child: Text(
            'Restore',
            style: TextStyle(
                fontFamily: 'Sacramento',
                fontSize: 25.0,
                fontWeight: FontWeight.bold,
            ),
            ),
        ),
        ),
    ),
    );
```

If we go through the above code, we will see several Consumers that have subscribed to those model

classes. The role of these controllers are simple. They will pass those data to the screen-view pages, which we will see in a minute.

Next goes the page-specific controller file:

```
// code 8.10

// controller/second_home_page_controller.dart

import 'package:first_flutter_app/model/first_model.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

Widget textStyleTrajanPro(String trajan) => Text(
    trajan,
    style: TextStyle(
        fontFamily: 'Trajan Pro',
        fontSize: 35.0,
        fontWeight: FontWeight.bold,
    ),
    textAlign: TextAlign.center,
    );

Widget textStyleSacramento(String sacramento) => Text(
    sacramento,
    style: TextStyle(
        fontFamily: 'Sacramento',
        fontSize: 55.0,
    ),
    textAlign: TextAlign.center,
    );

Widget changeNameButton() => Container(
    padding: const EdgeInsets.all(30.0),
    child: Consumer<FirstModel>(
        builder: (context, value, child) => Container(
        padding: const EdgeInsets.all(25.0),
        child: RaisedButton(
            child: Text(
            'Change Name',
            style: TextStyle(
                fontSize: 35.0,
                fontWeight: FontWeight.bold,
```

```
            ),
            ),
            onPressed: () {
            value.changeName();
            },
        ),
        ),
    ),
    );

Widget clearNameButton() => Container(
    padding: const EdgeInsets.all(30.0),
    child: Consumer<FirstModel>(
        builder: (context, value, child) => Container(
        padding: const EdgeInsets.all(25.0),
        child: RaisedButton(
            child: Text(
            'Clear Name',
            style: TextStyle(
                fontSize: 35.0,
                fontWeight: FontWeight.bold,
            ),
            ),
            onPressed: () {
            value.clearName();
            },
        ),
        ),
    ),
    );
```

In the above code, there are one or two Consumers, not as much as the mobile-specific controllers. Before going to read the screen-view code, we will take a look at how our flutter application looks like:

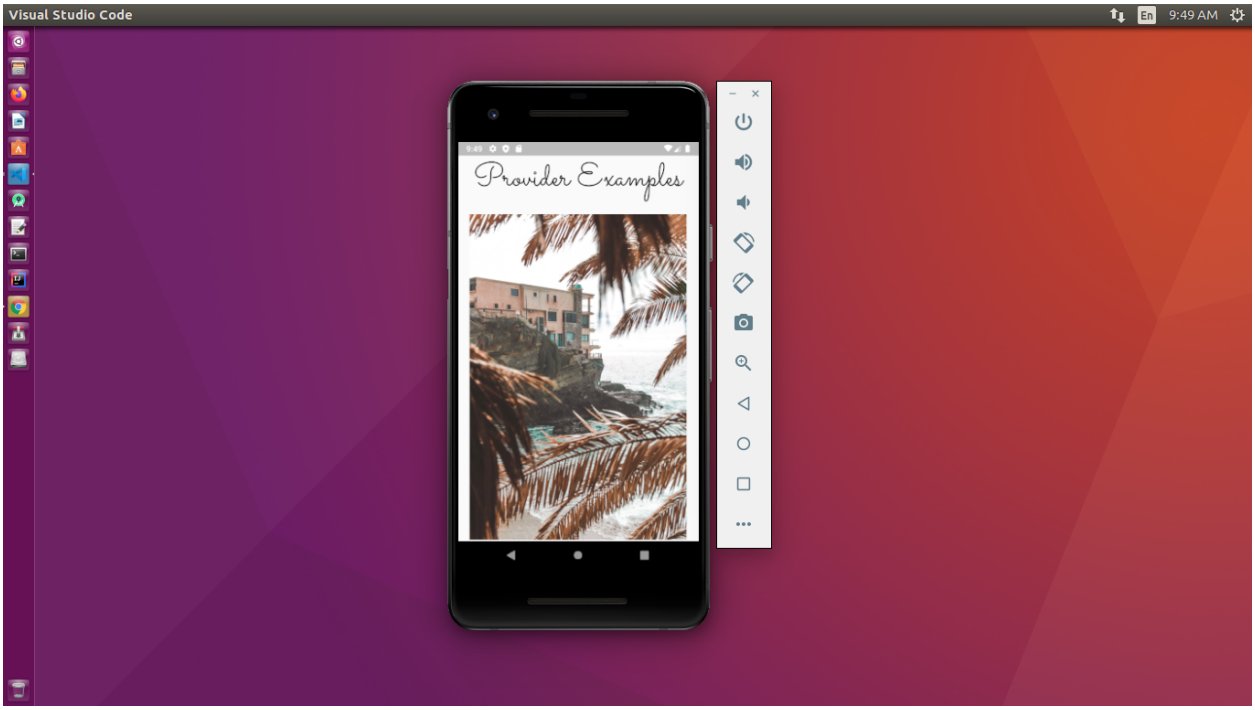

Figure 8.10 – The first look of the application that we are going to build

Now we can scroll down to the bottom and see what are waiting for us. At the bottom part, we have two buttons, and below those buttons, we have two mobile icons and respective text that tells us about the foreground and background colors.

If we click the 'Change Name' button, it will display a text 'Name Changed'. Just below that text we have the 'Clear name' button. Pressing that button will clear the text (Figure 8.9).

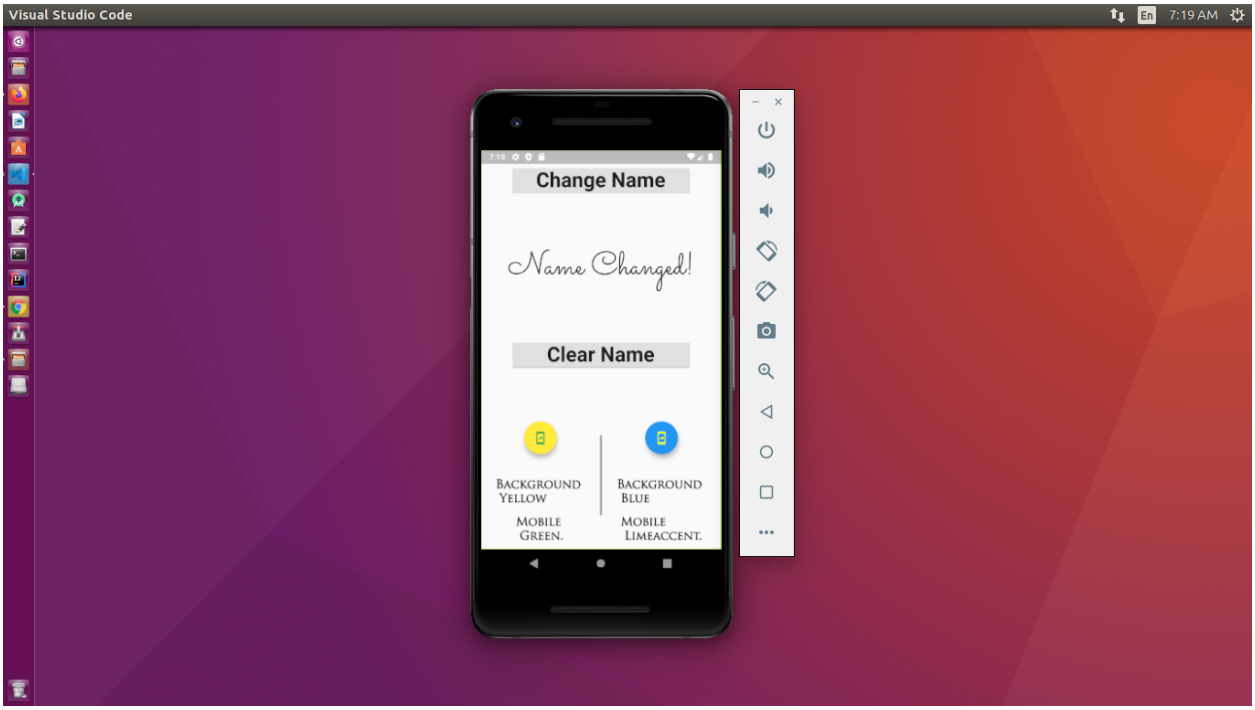

Figure 8.11 – The bottom part of our application

At the very bottom two mobile icons are visible. We can also see the name of the foreground and background color in text. Tapping any icon will change the foreground and background color, and at the same time, the description of color displayed in text will also change.

Finally, let us see the screen-view page and the main method.

```
// code 8.11

// view/second_home_app.dart


import 'package:first_flutter_app/controller/mobile_controller.dart';
import 'package:first_flutter_app/controller/second_home_page_controller.dart';
import 'package:first_flutter_app/model/first_model.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

class SecondHomeAppPage extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    debugShowCheckedModeBanner: false,
    title: 'Second Provider Example',
```

```
18      home: Scaffold(
19          body: SafeArea(
20          child: ListView(
21              children: [
22              textStyleSacramento('Provider Examples'),
23              Container(
24                  padding: const EdgeInsets.all(20.0),
25                  child: Image.asset(
26                  'images/sea1.jpg',
27                  width: 300,
28                  ),
29              ),
30              textStyleTrajanPro('We can add humongous widget tree below...'),
31              changeNameButton(),
32              Container(
33                  padding: const EdgeInsets.all(30.0),
34                  child: textStyleSacramento(
35                      Provider.of<FirstModel>(context, listen: true).name),
36              ),
37              clearNameButton(),
38              SizedBox(
39                  height: 10.0,
40              ),
41              Row(
42                  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
43                  children: [
44                  changeColorButtonToPurple(),
45                  VerticalLine(),
46                  changeColorButtonToRed(),
47                  ],
48              ),
49              SizedBox(
50                  height: 10.0,
51              ),
52              Row(
53                  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
54                  children: [
55                  restoreOldColorOfFirstMobile(),
56                  VerticalLine(),
57                  restoreOldColorOfSecondMobile(),
58                  ],
59              ),
60              ],
```

```
        ),
        ),
    ),
    );
}
}

class VerticalLine extends StatelessWidget {
const VerticalLine({
    Key key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        height: MediaQuery.of(context).size.height * 0.2,
        width: 3,
        color: Colors.black45,
    ),
    );
}
}

class HorizontalLine extends StatelessWidget {
const HorizontalLine({
    Key key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        width: MediaQuery.of(context).size.width * 0.2,
        height: 3,
        color: Colors.black45,
    ),
    );
}
}
```

And the main method is as the following where we have used multi Provider :

```
// code 8.12

// main.dart

import 'package:first_flutter_app/model/first_model.dart';
import 'package:first_flutter_app/view/second_home_app.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import 'model/mobile_model.dart';

void main() {
runApp(
    MultiProvider(
    providers: [
        ChangeNotifierProvider(create: (context) => FirstModel()),
        ChangeNotifierProvider(create: (context) => MobileModel()),
    ],
    child: SecondHomeAppPage(),
    ),
);
}
```

Now, we can press the ‘Change Name’ button, and get the text. Let us do that, and take a look at the lower bottom part.

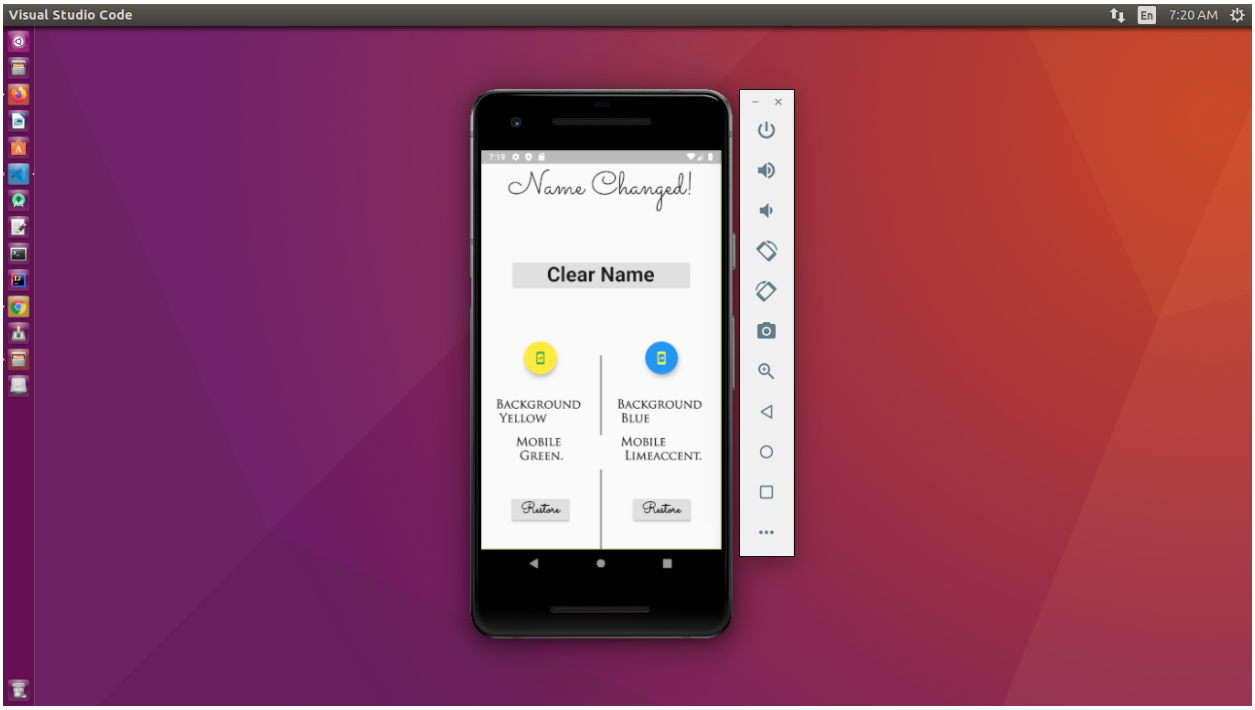

Figure 8.12 – The lower bottom part of our application

Next, we will start operating at the lower bottom part. Remember, we have already pressed the 'Change Name' button,and got the text displayed on the top of the screen-view. Now we are going to change the first mobile icon color, foreground and background, both.

Let us see the image first, after that, we will discuss the code.

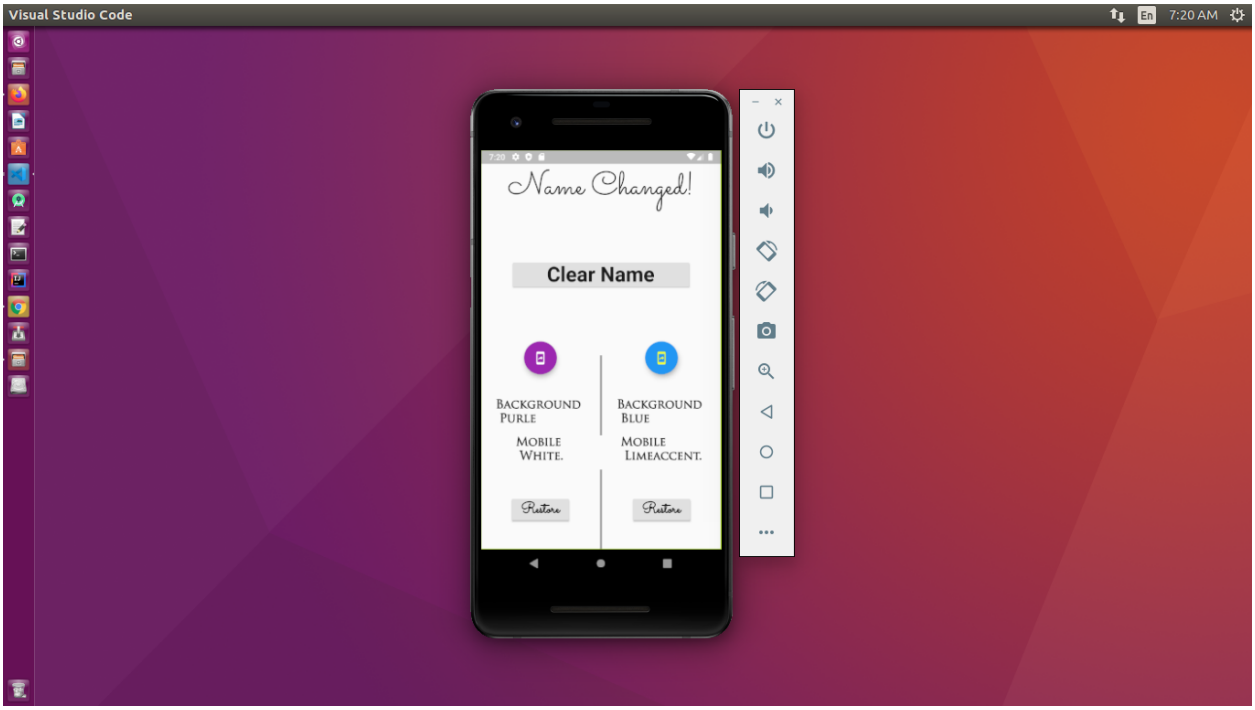

Figure 8.13 – The first mobile icon's foreground and background color have been changed and it has been reflected on the below text

We can clearly watch that the first mobile icon's foreground has been changed to white from green; at the same time the background color has been changed from yellow to purple.

Let us see this coding part in the mobile model class.

```
void changeColorToPurple() {
    backgroundColorOfFirst = 'Background \n Purle';
    mobileColorOfFirst = 'Mobile \n White.';
    selection[0] = Colors.purple;
    selection[4] = Colors.white;
    notifyListeners();
}
```

After that, we will take a look at the related mobile controller's coding part.

```
Widget changeColorButtonToPurple() => Column(
    children: [
        Container(
        padding: const EdgeInsets.all(10.0),
        child: Consumer<MobileModel>(
            builder: (context, value, child) => Container(
            padding: const EdgeInsets.all(15.0),
            child: FloatingActionButton(
                backgroundColor: value.selection[0],
                onPressed: () {
                value.changeColorToPurple();
                },
                child: Icon(
                Icons.mobile_screen_share,
                color: value.selection[4],
                ),
            ),
            ),
        ),
        ),
        Divider(
        thickness: 2.0,
        ),
        Consumer<MobileModel>(
        builder: (context, value, _) => Text(
            value.backgroundColorOfFirst,
            style: TextStyle(
            fontFamily: 'Trajan Pro',
            fontSize: 20.0,
            fontWeight: FontWeight.bold,
            ),
        ),
        ),
        Divider(
        thickness: 2.0,
        ),
        Consumer<MobileModel>(
        builder: (context, value, _) => Text(value.mobileColorOfFirst,
            style: TextStyle(
                fontFamily: 'Trajan Pro',
                fontSize: 20.0,
                fontWeight: FontWeight.bold,
            )),
```

```
44          ),
45      ],
46      );
```

And, finally we can watch the screen-view page part, where we have called this controller.

```
1  Row(
2                  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
3                  children: [
4                  changeColorButtonToPurple(),
5                  VerticalLine(),
6                  changeColorButtonToRed(),
7                  ],
8              ),
```

In the same row, we can call both controllers that will change the foreground and background color.

Therefore, in the next image, we will see that the second mobile icon's foreground and background color have also been changed, because we have tapped the second icon.

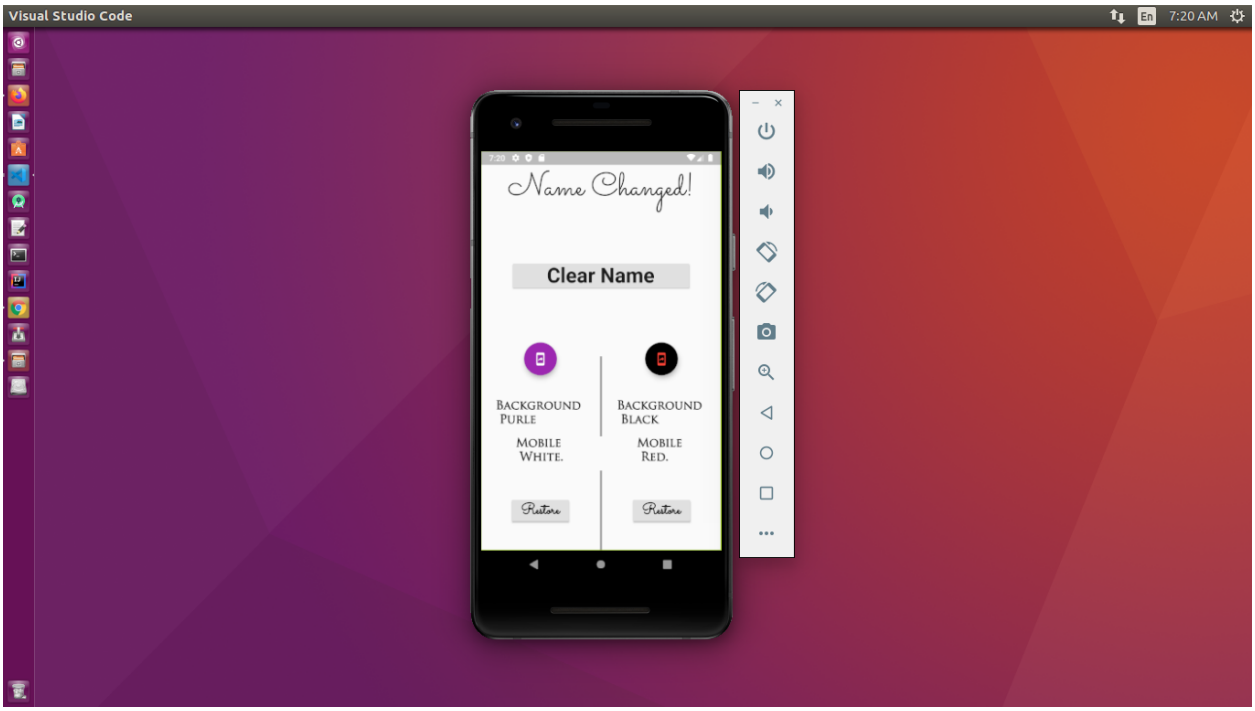

Figure 8.14 – The second mobile icon's foreground and background color have been changed

We can clearly see that the second mobile icon's foreground changed to red, and the background changed to black. The below text has also displayed the name of the color respectively.

One thing is also evident, although two controllers belong to the same Row widget, one change does not affect the other.

Our next step will be to restore the old data. First, we will click the restore button below the first mobile icon. Secondly, we will click the second restore button below the second mobile icon.

And finally, we will click the 'Clear name' button on the upper half of the screen. It will first restore the old color of the first mobile icon, next, it will change the second mobile icon; and finally it will clear the 'name' that was stuck on the upper half of the screen.

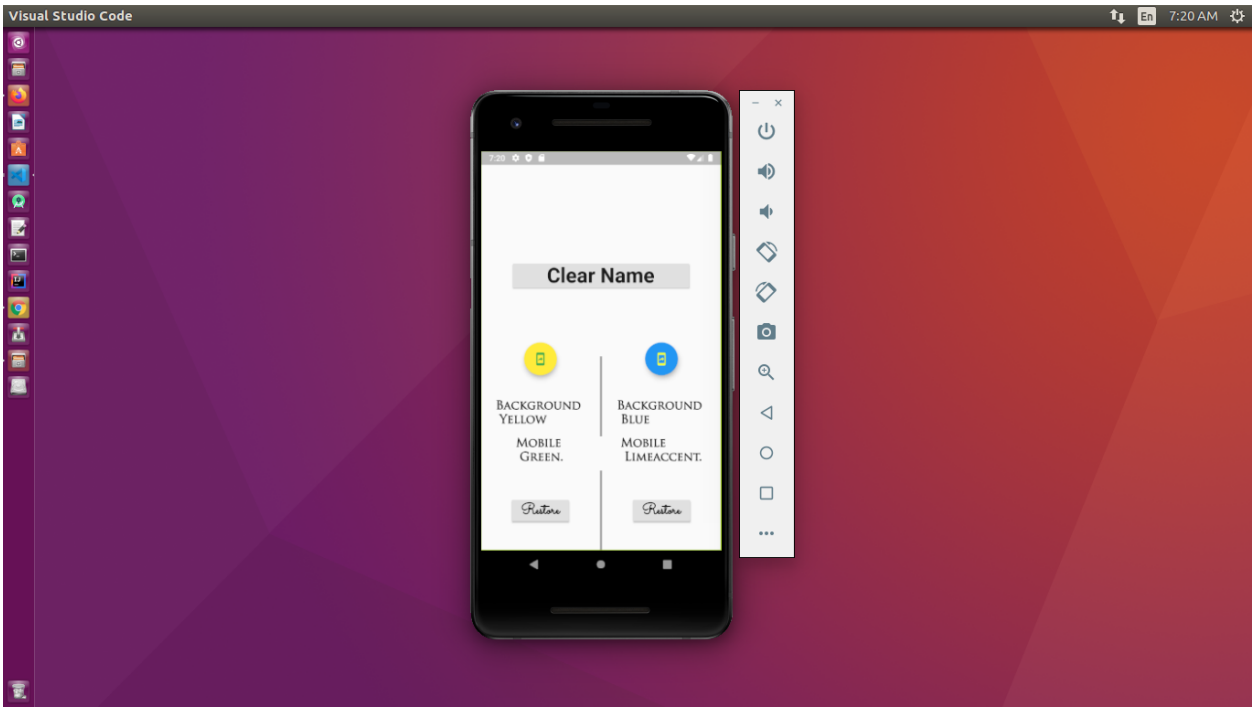

Figure 8.15 – The final screen shot of our application

We have learned how we can use Provider package, and Consumer widget to manage state efficiently. We have also learned how without rebuilding the whole widget tree, we can change and persist state of our application.

In the next chapter, we will learn how to navigate from one screen to others and come back.

Want to read more Flutter related Articles and resources?

For more Flutter related Articles and Resources[20]

[20] https://sanjibsinha.com

9. Everything about Flutter Navigation and Route

In this chapter we'll learn how we can navigate from one screen or page to another screen. How we can pass data from one screen to another.

We'll learn about Route and Navigator widgets. What are there functions and how Flutter provides many features that we can use to handle navigation.

The full code snippet is available in the respective GitHub repository.

The full code repository for this chapter[21]

Moreover, for updated flutter tutorials don't forget to visit:

Updated Flutter Tutorials[22]

Why do you use onGenerateRoute in flutter?

Why do we need onGenerateRoute navigation in Flutter?

In some cases, we want the user to log in to enter the app. Otherwise the new user can also register on that same page.

As a result, on opening the app, the route takes the user to a certain page.

Good news is, Flutter has a built-in feature for achieving such a feat. Moreover, we can control the navigation at the very beginning of our Flutter app.

To do this, in flutter MaterialApp widget, we use onGenerateRoute property.

Material design and material components play a key role in building a Flutter app. In addition, the material design system unites style, branding, interaction and motion using a set of material components.

And these material components work under material design's principles. That makes the usage of onGenerateRoute property in flutter.

[21] https://github.com/sanjibsinha/better_flutter_chapter_seven
[22] https://flutter.sanjibsinha.com

How do you use onGenerateRoute in Flutter?

To show how we use onGenerateRoute in Flutter, we must have two pages. The first page and the second page.

Consequently, when the app opens up, it takes us to the second page. If we want to get back to the first page, we just tick the cross mark and it navigates back to the first page.

However, everything starts with MaterialApp. This convenience widget builds upon a WidgetsApp and it adds material design specific functionalities.

How to use a dynamic initial route?

The MaterialApp design maintains an order to configure the top navigator. Of course, Flutter uses the home property to decide where to go first.

If we use the home property, it automatically navigates to that page.

Order-wise, next, the route tables are used. It checks whether there is any entry for the route.

Otherwise, it calls onGenerateRoute. In that case, we need to provide the route.

```
import 'dart:ui';

import 'package:flutter/material.dart';

class MaterialDesign extends StatelessWidget {
const MaterialDesign({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
return MaterialApp(
title: 'Better Flutter - Essential Widgets',
home: MDFirstPage(),
initialRoute: '/second',
onGenerateRoute: _getSecondPageFirst,
);
}

Route<dynamic>? _getSecondPageFirst(RouteSettings settings) {
if (settings.name != '/second') {
return null;
}

```

```
return MaterialPageRoute<void>(
settings: settings,
builder: (BuildContext context) => MDSecondPage(),
fullscreenDialog: true,
);
}
}
```

Watch the above code. Although the home property indicates to the first page, the initialRoute property navigates to the second page.

However, we need to be careful about one thing. It should return a non-null value.

The rest is quite simple. Now we can design our first page and second page.

In any case, the app will open the second page first.

If we want to make this page a log in and registration page, we can design that too.

In addition, if the user doesn't want to log in or register, she can touch the cross icon. In that case, the home property comes into effect, opening the first page as usual.

What is Flutter Navigation and how does Flutter Navigator work?

What is Flutter Navigation? Moreover, what is Flutter Navigator? Are they same? Or, they are different? Above all, the answer lies in a widget.

Route.

Although Route is a widget, still it actually represents a new screen. Or, a new page.

For example, we navigate to the second route with the help of Navigator.

However, we need to understand the context.

What is the context?

Moreover, why we need it?

Most apps use different screens. And, for that we need router widget.

What is a navigator and routes in Flutter?

To begin with, we'll see how a navigator and routes work in Flutter.

Firstly, we want a simple example. So the beginners can understand. Moreover, a better flutter developer must understand how routes work.

Secondly, to keep it simple, we navigate from first screen to the second.

Further, we'll learn how we can handle multiple routes. Above all, how we can pass data while navigating to a second page.

As we have said, route is a widget. Navigator is also another widget.

How do you deal with navigation in Flutter?

To deal with navigation we need a first page. Right?

As a result, we can move to the second page. And come back. Navigation deals with that.

Let us see our first chunk of code first:

```
import 'package:flutter/material.dart';

void main() {
runApp(const MyApp());
}

class MyApp extends StatelessWidget {
const MyApp({Key? key}) : super(key: key);

// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    routes: <String, WidgetBuilder>{
        '/': (BuildContext context) => const HomePage(),
    },
    title: 'Flutter Demo',
    theme: ThemeData(
        primaryColor: const Color(0xFF3EBACE),
        backgroundColor: const Color(0xFFF3F5F7),
        primarySwatch: Colors.indigo,
    ),
    );
}
}
```

As we can see, the route widget takes us to the Home Page. However, it also carries the context.

Next, we need to go to the second page.

How Does Flutter Navigator work?

Therefore, our first page must have that mechanism.

```
class HomePage extends StatelessWidget {
const HomePage({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('First Page'),
    ),
    body: GestureDetector(
        onTap: () {
        Navigator.push(
            context,
            MaterialPageRoute(builder: (context) => const SecondPage()),
        );
        },
        child: Container(
        margin: const EdgeInsets.all(10.0),
        padding: const EdgeInsets.all(10.0),
        child: ClipRRect(
            borderRadius: const BorderRadius.only(
            topLeft: Radius.circular(15.0),
            topRight: Radius.circular(15.0),
            bottomLeft: Radius.circular(15.0),
            bottomRight: Radius.circular(15.0),
            ),
            child: Image.network(
                'https://sanjibsinha.com/wp-content/uploads/2021/07/Can-you-code-in-\
WordPress-How-do-I-learn-WordPress-coding-.jpg'),
        ),
        ),
    ),
    );
}
}
```

The on tap method in the above code shows an exemplary behavior.

Why?

Because, it uses the Navigator widget push method. And after that, it passes the same context.

How Does Flutter Navigator push work?

At the first page, if we we can use the Gesture Detector, we can now tap the image. And that takes us to the second page.

Although we've not used Navigator pop method in the second page, yet we can come back to the first page using the back button in the App Bar.

```
class SecondPage extends StatelessWidget {
const SecondPage({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Second Page'),
    ),
    body: Center(
        child: Container(
        margin: const EdgeInsets.all(10.0),
        padding: const EdgeInsets.all(10.0),
        child: ClipRRect(
            borderRadius: const BorderRadius.only(
            topLeft: Radius.circular(15.0),
            topRight: Radius.circular(15.0),
            bottomLeft: Radius.circular(15.0),
            bottomRight: Radius.circular(15.0),
            ),
            child: Image.network(
                'https://sanjibsinha.com/wp-content/uploads/2021/06/What-is-toList-f\
lutter-What-is-map-in-Dart-.jpg'),
        ),
        ),
    ),
    );
}
}
```

How do you pass data from one class to another in flutter?

Passing data from one class to another in Flutter is not difficult. However, we need to understand the basic concepts of route and navigator first.

Route is a widget. So the Navigator. Moreover, for passing data we need to use these two widgets.

With the help of these two widgets we can pass data using class constructors in Flutter.

To use class constructors we first need model classes and dummy data. Because the data flow from models and we can use them to display on the screen.

Certainly, to do that we also need the help of controller.

How do you use route in flutter?

To use route widget we have to return material app widget where we define the screens or pages.

Suppose on the first page we have a bunch of categories. And clicking those categories take us to the detail page.

Therefore, at a first glance, it gives us all categories.

The above image displays categories of different type of news. If we click any one of the category, it will take us to the detail page.

To achieve this, in model folder we have category class.

```
import 'package:flutter/material.dart';

class Category {
final String id;
final String title;
final Color color;

const Category({
    required this.id,
    required this.title,
    this.color = Colors.orangeAccent,
});
}
And with the category class, we need some dummy category data like the following cod\
e.

import 'package:flutter/material.dart';
```

```
18
19  import 'category.dart';
20
21  const DUMMY_CATEGORIES = const [
22  Category(
23      id: 'c1',
24      title: 'Health',
25      color: Colors.red,
26  ),
27  Category(
28      id: 'c2',
29      title: 'Wellness',
30      color: Colors.deepOrange,
31  ),
32  Category(
33      id: 'c3',
34      title: 'Politics',
35      color: Colors.black54,
36  ),
37  Category(
38      id: 'c4',
39      title: 'Travel',
40      color: Colors.green,
41  ),
42  Category(
43      id: 'c5',
44      title: 'Internet',
45      color: Colors.yellow,
46  ),
47  Category(
48      id: 'c6',
49      title: 'Lifestyle',
50      color: Colors.indigo,
51  ),
52  Category(
53      id: 'c7',
54      title: 'Headlines',
55      color: Colors.pink,
56  ),
57  Category(
58      id: 'c8',
59      title: 'Sports',
60      color: Colors.orange,
```

```
),
Category(
    id: 'c9',
    title: 'Science',
    color: Colors.blueAccent,
),
Category(
    id: 'c10',
    title: 'Environemnt',
    color: Colors.redAccent,
),
];
```

To display these categories in the opening page, we need route widget which defines the navigation pattern.

How do you get the current route in Flutter?

To get the current route we need to return material app widget.

```
class FirstPage extends StatelessWidget {
const FirstPage({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Routing test',
    debugShowCheckedModeBanner: false,
    initialRoute: '/',
    routes: {
        '/': (context) => const FirstPageBody(),
        '/categories': (context) => SecondPage(),
    },
    );
}
}
```

Since the related code is too long, please visit the respective GitHub repository. Above all this repository also connects you to the book Better Flutter in Leanpub.

This repository will also give an idea how from the current route we move to the detail page.

As we can see that the current route takes us to the detail page where the data are coming from the dummy News class.

If we take a look at the News class, we will understand how it works.

```
enum Nature {
hard,
soft,
}

class News {
final String id;
final List<String> categories;
final String title;
final String detail;
final String imageURL;
final Nature nature;

const News({
    required this.id,
    required this.categories,
    required this.title,
    required this.detail,
    required this.imageURL,
    required this.nature,
});
}
```

At the same time, we pass the data from page one to the second page like this:

```
body: GridView(
        gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(
        maxCrossAxisExtent: 200,
        crossAxisSpacing: 20.0,
        mainAxisSpacing: 20.0,
        ),
        children: DUMMY_CATEGORIES.map(
        (e) {
            return AllCategories(
            id: e.id,
            title: e.title,
            color: e.color,
            );
        },
        ).toList(),
    ),

// code is incomplete for brevity, please consult the GitHub repository
```

How do you pass context in flutter?

We pass the whole data as a part of context. Because we have defined route where the anonymous function takes the parameter context

```
routes: {
        '/': (context) => const FirstPageBody(),
        '/categories': (context) => SecondPage(),
    },
```

As a result it becomes easier for us to catch the data in the second page.

```
Widget build(BuildContext context) {
    final Map arguments = ModalRoute.of(context)!.settings.arguments as Map;
    final id = arguments['id'];
    final title = arguments['title'];
    final color = arguments['color'];
    final categoryBooks = DUMMY_NEWS.where((book) {
    return book.categories.contains(id);
    }).toList();

// code is incomplete, please consult respective GitHub repository
```

To get the data the modal route class uses a static method “of” that passes the context. And the context also defines the location of this widget in the widget tree.

In the coming flutter tutorials we’ll discuss more about passing data through route and context.

What is enum in Dart flutter? How to use enum in Flutter?

To use enum in Dart or Flutter is not a difficult task. However, before we use Enumerated Types or enum, we need to understand it.

Dart added enum as a feature in 1.8 release. As a result, Flutter also starts using enum.

Since we have been developing a News app, in the previous post we’ve learned how to pass data through class using route and navigator widgets.

Before that, we’ve also seen the basic route and navigation.

Subsequently, we’re going to build the News app and this time we’ll use enum as a special type. We’ll also learn how to pass enum along with other data while we navigate to another page.

How do you use Enums in Navigation?

As we have just said, Enums are a list of constant values that we can use either as a numeric data, or String data.

If you've already read the previous posts you're aware that the full code is available in my GitHub repository.

As we've progressed in building the News app, we have seen that the front page displays categories of News Items.

Clicking any News item takes us to the detail screen or page. To accomplish this, we have two different classes. Category and News. Also we have a set of dummy data.

However, there is no enum then. Up to that part in our class we have not added any Enums. As a result it has not displayed what kind of story is this.

This story could be an event based hard news story. Or, it could be a soft News story that people might save and read later.

Our Enums will serve that purpose. So we add that to the News class first.

```
enum Nature {
hard,
soft,
}

class News {
final String id;
final List<String> categories;
final String title;
final String detail;
final String imageURL;
final Nature nature;

const News({
    required this.id,
    required this.categories,
    required this.title,
    required this.detail,
    required this.imageURL,
    required this.nature,
});
}
```

Next, we'll use the Enums in the display page as string data. To use Enums as string data we need to use switch case.

```
class SecondPage extends StatelessWidget {
final int id;
final String title;
final Color color;
final Nature nature;

const SecondPage({
    Key? key,
    required this.id,
    required this.title,
    required this.color,
    required this.nature,
}) : super(key: key);

String get natureText {
    switch (nature) {
    case Nature.hard:
        return 'Event based Latest Hard News >>';
    case Nature.soft:
        return 'Take time and read Soft News Story >>';
    default:
        return 'Unknown';
    }
}

// code is incomplete, for full code please visit GitHub repository
```

Now, at the top of the display page, we can use that Enums like the following images.

How do you find the enum value in flutter?

Finding enum value is not difficult either. As we've seen the switch case, each news story displays the nature. So the reader will at a glimpse know whether she should read it now or save it to read later.

As you can see, according to the id of the News, it's an event based latest hard news. However, for any other story this nature might change.

Let's take a look at the above image that belongs to the Enums of soft news story.

Certainly, to get the nature of the story we've used enum. But, to display it as a String data we need to pass it.

Moreover, we need to place it over the image as Text widget.

```
body: ListView.builder(
        itemBuilder: (context, index) {
        return Column(
            children: [
            Container(
                margin: const EdgeInsets.all(10.0),
                padding: const EdgeInsets.all(10.0),
                child: Text(
                natureText,
                style: const TextStyle(
                    fontSize: 20.0,
                ),
                ),
            ),

// code is incomplete for brevity
```

Hopefully, this makes sense. Moreover, we'll not stop here. We'll build the News app so that it looks more attractive than this.

That means, we'll design our Flutter News app better than this one.

And at the same time, to pass data we'll use route and navigation in a different way.

How do you change the theme on Flutter?

To change the theme on Flutter we must do it at the root level. The root level means when we create the Material App, we need to define the custom theme.

However, it's not easy. Because Flutter takes care of the theme. Since in Flutter everything is widget, the custom theme also belongs to the same concept.

To change the theme, we need to reflect it and display the new look on the screen.

With reference to that , we've already written about the route and navigation before. We've also shown how we can pass data from one class to another without changing its theme.

Finally, to maintain the same theme throughout the whole Flutter app, we've used enum to add more characteristic to it.

And our Flutter app is almost ready.

Can we change this look further? Certainly, to change the look we must adhere to a custom theme.

As a result, in our main dart file, we need to declare the custom theme while creating the Material App.

```
theme: ThemeData(
        primarySwatch: Colors.pink,
        primaryColor: Colors.amber,
        canvasColor: const Color.fromRGBO(255, 254, 229, 1),
        fontFamily: 'Raleway',
        textTheme: ThemeData.light().textTheme.copyWith(
            bodyText2: const TextStyle(
            color: Color.fromRGBO(20, 51, 51, 1),
            ),
            bodyText1: const TextStyle(
            color: Color.fromRGBO(20, 51, 51, 1),
            ),
            headline6: const TextStyle(
            fontSize: 20,
            fontFamily: 'RobotoCondensed',
            fontWeight: FontWeight.bold,
            )),
    ),
```

What is theme Flutter?

As we know, themes are an integral part of User Interface or UI of any flutter app. To make the app more presentable, we need to design the fonts and colors.

Above all, we need to define them at the Theme Data widget. As we've done in the above code.

Now the change reflects on our flutter app's look.

Our News app looks different due to the change in the custom theme. In the next article we'll see how we can develop our flutter app with the help of this new theme.

How do you name a route in Flutter?

To name a route in Flutter we need to have two screens. Moreover, we need to navigate from one screen to another.

That is the basic design of navigation in Flutter. To go from one screen to other, we use Navigator widget. And we also use a static method push Named.

Subsequently, to come back to the first screen, we use the Navigator widget again. To make it complete we use pop method.

However, we can name a route in a different way too. To do that, we need to declare it as a constant static property inside that class.

For brevity, we cannot display the full code snippet that involves route name and passing arguments. Please visit the respective code repository in GitHub.

```
class CategoryNewsScreen extends StatelessWidget {
static const routeName = '/category-news';

// code is incomplete
```

How do we can use this route name? To do that we need to use the route widget in our Material App widget.

```
initialRoute: '/', // default is '/'
    routes: {
        '/': (ctx) => const CategoriesScreen(),
        CategoryNewsScreen.routeName: (ctx) => const CategoryNewsScreen(),
        NewsDetailScreen.routeName: (ctx) => const NewsDetailScreen(),
    },
```

In case, our route name doesn't work, we can also use a fallback.

```
onUnknownRoute: (settings) {
        return MaterialPageRoute(
        builder: (ctx) => const CategoriesScreen(),
        );
    },
```

As a result, if we cannot go the Categories Detail Screen page, and our route name fails, it always stays in the Home page.

In our News App we can display every category using a dummy data.

How do you pass arguments to named route in Flutter?

To pass arguments to named route in Flutter is not difficult either. Because, we can get the arguments at the page where we have declared our static route name property.

```
class CategoryNewsScreen extends StatelessWidget {
static const routeName = '/category-news';

const CategoryNewsScreen({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    final routeArgs =
        ModalRoute.of(context)!.settings.arguments as Map<String, String>;
    final categoryTitle = routeArgs['title'];
    final categoryId = routeArgs['id'];
    final categoryNews = dummyNews.where((news) {
    return news.categories.contains(categoryId);
    }).toList();
    return Scaffold(
    appBar: AppBar(
        title: Text(categoryTitle!),
    ),

// code is incomplete for brevity
```

In the updated Flutter version, the null checking is mandatory. So keep that in mind. We need to add an exclamatory sign after the Modal route of method that passes the context.

As a result, whenever we click any category, it takes us to static name of route that we've defined inside the class.

As we can see, in the Health category, we have two news items. If we have a look at the dummy data, we'll find that.

```
const dummyNews = [
News(
    id: 'b1',
    categories: [
    'c1',
    'c4',
    ],
    title: 'Global Worming fuels disaster',

....

News(
    id: 'b5',
    categories: [
```

```
    'c1',
    'c5',
    ],
    title: 'Take more outdoor walks',

// code is incomplete
```

As in two news items we find this category, therefore, it displays them on the category news screen.

However, to display the route items in the right manner, we take help from a controller.

```
class CategoryItem extends StatelessWidget {
final String id;
final String title;
final Color color;

const CategoryItem({
    required this.id,
    required this.title,
    required this.color,
});

void selectCategory(BuildContext ctx) {
    Navigator.of(ctx).pushNamed(
    CategoryNewsScreen.routeName,
    arguments: {
        'id': id,
        'title': title,
    },
    );
}

// code is incomplete
```

As a matter of fact, we define the select category method here and push the route name and arguments here.

To make the whole News App complete we need to do many more things. Consequently that also makes the route name and passing arguments process complete.

How do you pass data from one screen to another in flutter?

How we can pass data from one screen to another in Flutter? How we can manage this complex process?

Moreover, what concepts we should understand before we proceed?

Firstly, we need route widget. Secondly we need Navigator widget. And finally, we need to know how list and map work in Flutter.

Therefore let us make these points clear at the beginning.

```
Route
Navigator
List and Map
```

Where should we mention the route? At our Material App widget. The route indicates where we want to go from the home screen or page.

Not only that, when we move from one screen to another, we'll also pass the data. So, in the second screen, we display that data.

Above all, when we go from the home screen, we must use a method that will send us along with the data. And based on that data we can display other data on the second screen.

To explain the whole process of passing data from one screen to another, we need to show some code. However, for brevity, we cannot show the full code.

For the full code snippets, please visit the respective GitHub repository.

```
void main() => runApp(const NewsApp());
```

In the News App stateless widget we define our Material App widget where we use the route widget.

```
class NewsApp extends StatelessWidget {
const NewsApp({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Daily News',
    theme: ThemeData(
        primarySwatch: Colors.pink,
        primaryColor: Colors.amber,
```

```
        canvasColor: const Color.fromRGBO(255, 254, 229, 1),
        textTheme: ThemeData.light().textTheme.copyWith(
            bodyText2: const TextStyle(
            color: Color.fromRGBO(20, 51, 51, 1),
            ),
            bodyText1: const TextStyle(
            color: Color.fromRGBO(20, 51, 51, 1),
            ),
            headline6: const TextStyle(
            fontSize: 20,
            fontWeight: FontWeight.bold,
            )),
    ),
    // home: CategoriesScreen(),
    initialRoute: '/', // default is '/'
    routes: {
        '/': (ctx) => const CategoriesScreen(),
        CategoryNewsScreen.routeName: (ctx) => const CategoryNewsScreen(),
        NewsDetailScreen.routeName: (ctx) => const NewsDetailScreen(),
    },

    onUnknownRoute: (settings) {
        return MaterialPageRoute(
        builder: (ctx) => const CategoriesScreen(),
        );
    },
    );
}
}
```

As we can see, the initial route is Categories Screen stateless widget. This widget must return some data. Here all the categories of the News Items like Health, Politics, Internet and many more.

To get that data we need Category class and dummy data in our model folder. Please visit the GitHub repository to have an idea.

How do you pass data through navigation?

Based on that dummy categories data, now we can return another widget in the Categories Screen widget.

```
body: GridView(
        padding: const EdgeInsets.all(25),

        /// the first page displays CategoryItem controller
        children: dummyCategories
            .map(
            (catData) => CategoryItem(
                id: catData.id,
                title: catData.title,
                color: catData.color,
            ),
            )
            .toList(),

...
// code is not complete
```

The dummy categories are a few constant data where we've defined the id, title and the color of the category.

```
const dummyCategories = [
Category(
    id: 'c1',
    title: 'Health',
    color: Colors.red,
),
Category(
    id: 'c2',
    title: 'Wellness',
    color: Colors.deepOrange,
),
....
// code is not complete
```

Actually, the Category Item widget will help to display all the categories and at the same time it will let us allow to click each category to see what kind of news items belong to that category.

```
class CategoryItem extends StatelessWidget {
final String id;
final String title;
final Color color;

const CategoryItem({
    Key? key,
    required this.id,
    required this.title,
    required this.color,
}) : super(key: key);

void selectCategory(BuildContext ctx) {
    Navigator.of(ctx).pushNamed(
    CategoryNewsScreen.routeName,
    arguments: {
        'id': id,
        'title': title,
    },
    );
}
...
// code is not complete
```

We've passed all data through the Class Constructor and then we define a method select Category.

Inside that method, Navigator widget uses a chain of static methods through which we pass the context and a list of arguments.

Because of this widget we can now see all the categories.

However, if we click any category the method fires and push the Navigator to another stateless widget Category News Screen.

Let us see a few part of Category News Screen widget code.

```
Widget build(BuildContext context) {
    final routeArgs =
        ModalRoute.of(context)!.settings.arguments as Map<String, String>;
    final categoryTitle = routeArgs['title'];
    final categoryId = routeArgs['id'];
    final categoryNews = dummyNews.where((news) {
    return news.categories.contains(categoryId);
    }).toList();

...
// code is not complete
```

We've already sent the arguments as a List. Where we have sent ID and Title of Category class and dummy category data.

Now according to the News class and dummy data we can now show the News item that belongs to that category.

Now we can click the title or the special enum to view the News in detail.

How do you make a Flutter app from scratch?

To make a flutter app from scratch is easy. But it depends on the perspective. As long as we build a small app it's really easy.

However, it becomes difficult if we want more features.

What kind of features need more skill? In fact, there are plenty.

Firstly, the design part. The User Interface should look good. Moreover, it must be user friendly.

Secondly, we need to maintain state. The state remains active between different screen. It involves navigation and route widgets.

However, we cannot allow more widget rebuilds. So we should maintain state in an economic way.

So the Flutter app speeds up while it runs.

Furthermore, there is Flutter data structures. List and Map.

In my opinion, this is the most difficult part of any Flutter app that we build from scratch.

In our previous post we've seen how we can build a News App that runs locally. We provide dummy data and pass them through Class Constructor.

For building such Flutter App please visit the respective GitHub repository.

Next, we click any category and reach the News Item that belongs to that Category.

Now we wan to see the News detail page.

How do you start learning Flutter from scratch?

To learn Flutter from scratch is another thing. But building an app like this involves some Flutter skill and knowledge about Dart object oriented programming.

Moreover, we need take a close look at the Flutter data structures that mostly use List and Map.

To sum up, in this News App, we see this page because we have a stateless widget Category News Screen.

```
class CategoryNewsScreen extends StatelessWidget {
static const routeName = '/category-news';

const CategoryNewsScreen({Key? key}) : super(key: key);

/// returns NewsItem controller
@override
Widget build(BuildContext context) {
    final routeArgs =
        ModalRoute.of(context)!.settings.arguments as Map<String, String>;
    final categoryTitle = routeArgs['title'];
    final categoryId = routeArgs['id'];
    final categoryNews = dummyNews.where((news) {
    return news.categories.contains(categoryId);
    }).toList();
    return Scaffold(
    appBar: AppBar(
        title: Text(categoryTitle!),
    ),
    body: ListView.builder(
        itemBuilder: (ctx, index) {
        return NewsItem(
            id: categoryNews[index].id,
            title: categoryNews[index].title,
            imageUrl: categoryNews[index].imageURL,
            nature: categoryNews[index].nature,
        );
        },
        itemCount: categoryNews.length,
    ),
    );
}
}
```

The above widget again returns a controller called News Item.

Now this controller or widget again displays the second image inside a Column widget.

As a children we get the respective image, title and the enum that we've defined already in the dummy data.

```
child: Column(
        children: <Widget>[
            Stack(
            children: <Widget>[
                ClipRRect(
                borderRadius: const BorderRadius.only(
                    topLeft: Radius.circular(15),
                    topRight: Radius.circular(15),
                ),
                child: Image.network(
                    imageUrl,
                    height: 250,
                    width: double.infinity,
                    fit: BoxFit.cover,
                ),
                ),
                Positioned(
                bottom: 20,
                right: 10,
                child: Container(
                    width: 300,
                    color: Colors.black54,
                    padding: const EdgeInsets.symmetric(
                    vertical: 5,
                    horizontal: 20,
                    ),
                    child: Text(
                    title,
                    style: const TextStyle(
                        fontSize: 26,
                        color: Colors.white,
                    ),
                    softWrap: true,
                    overflow: TextOverflow.fade,
                    ),
                ),
                )
```

```
38              ],
39              ),
40              Padding(
41              padding: const EdgeInsets.all(20),
42              child: Row(
43                  mainAxisAlignment: MainAxisAlignment.spaceAround,
44                  children: <Widget>[
45                  Row(
46                      children: <Widget>[
47                      const Icon(
48                          Icons.work,
49                      ),
50                      const SizedBox(
51                          width: 6,
52                      ),
53                      Text(natureText),
54                      ],
55                  ),
56                  ],
57              ),
58              ),
59          ],
60          ),
```

However, there is still a feature lacks in this Flutter app.

What is that?

We must be able to click this News Item to get the detail.

One can manage that only with a method that would push the arguments to a new screen, where we can display the detail of the News.

Moreover, we need to know the role of route and Navigator widget.

Because that deals with the List and Map and finally sends the correct data.

```
void selectNews(BuildContext context) {
    Navigator.of(context).pushNamed(
    NewsDetailScreen.routeName,
    arguments: id,
    );
}

@override
Widget build(BuildContext context) {
    return InkWell(
    onTap: () => selectNews(context),

...
```

The above method with the help of “on Tap” void function return the “select News” method with the context as its parameter.

Is Flutter Easy to Learn?

What do you think now? Is Flutter easy to learn?

Like any software toolkit Flutter also needs constant practice and a fair grasp of basic conceptions.

The above select News method in the News Item widget, actually sends us to another screen. News Detail Screen.

In that screen, we catch the data and display the detail of the News Item correctly.

```
class NewsDetailScreen extends StatelessWidget {
static const routeName = '/news-detail';

const NewsDetailScreen({Key? key}) : super(key: key);

Widget buildSectionTitle(BuildContext context, String text) {
    return Container(
    margin: const EdgeInsets.symmetric(vertical: 10),
    child: Text(
        text,
        style: Theme.of(context).textTheme.headline6,
    ),
    );
}

Widget buildContainer(Widget child) {
```

```
17      return Container(
18      decoration: BoxDecoration(
19          color: Colors.white,
20          border: Border.all(color: Colors.grey),
21          borderRadius: BorderRadius.circular(10),
22      ),
23      margin: const EdgeInsets.all(10),
24      padding: const EdgeInsets.all(10),
25      height: 150,
26      width: 300,
27      child: child,
28      );
29  }
30
31  @override
32  Widget build(BuildContext context) {
33      final mealId = ModalRoute.of(context)!.settings.arguments as String;
34      final selectedNews = dummyNews.firstWhere((meal) => meal.id == mealId);
35      return Scaffold(
36      appBar: AppBar(
37          title: Text(selectedNews.title),
38      ),
39      body: SingleChildScrollView(
40          child: Column(
41          children: <Widget>[
42              SizedBox(
43              height: 300,
44              width: double.infinity,
45              child: Image.network(
46                  selectedNews.imageURL,
47                  fit: BoxFit.cover,
48              ),
49              ),
50              buildSectionTitle(context, 'News Detail'),
51              Text(selectedNews.detail),
52          ],
53          ),
54      ),
55      );
56  }
57  }
```

10. More on Flutter UI, List, Map, and Provider Best Practices

In this chapter we'll take a close look at everything we've learned so far. However, we must remember one thing, Flutter changes with the time.

The upgraded Flutter comes with more features and at the same time, discards many faetures that we've used before.

Consequently, we're trying to evolve our knowledge with the time.

How do you use decoration in a container in Flutter?

To have a look at the screenshots we've used in this section, please visit this LINK[23]

We always want our flutter app to look not only better, but also unique. To accomplish that, we can use decorations in a container in Flutter.

Container widget is one of the main layout widgets in flutter, however, it has many properties that can change its look entirely. One of such properties is decoration.

The decoration property returns another class BoxDecoration.

What does it do?

The BoxDecoration class helps us with variety of ways to draw a box. As a result, we can use decorations to change the look of a container.

We've built our flutter News App, and that works fine. Meanwhile we can check how it looks like.

How have we built a flutter app from scratch

The above image follows a simple decoration strategy. The code is simple.

[23] https://sanjibsinha.com/decoration-in-a-container-in-flutter/

```
decoration: BoxDecoration(
        gradient: LinearGradient(
            colors: [
            color.withOpacity(0.7),
            color,
            ],
            begin: Alignment.topLeft,
            end: Alignment.bottomRight,
        ),
borderRadius: BorderRadius.circular(15),
...
// the code is incomplete for brevity
```

How do you customize a container in Flutter?

We can customize a container widget in flutter in many ways.

There are many properties that Container widget supplies us to give it an unique impression.

How about the following look?

With some slight tweak in decorations, we can completely change the look of the containers that display the news categories on the front page of our News App,

```
How do you color a container in Flutter?
Not only we will color this container, but we also add some more properties.
```

How do you outline a container in Flutter?

To change our News App front page completely, we not only outline each container, but also use some features available in Flutter.

Let us go through the code snippet that are particularly responsible for this make-over.

```
Container(
        padding: const EdgeInsets.all(15),
        child: Text(
        title,
        style: Theme.of(context).textTheme.headline6,
        ),
        /* decoration: BoxDecoration(
        gradient: LinearGradient(
            colors: [
            color.withOpacity(0.7),
            color,
            ],
            begin: Alignment.topLeft,
            end: Alignment.bottomRight,
        ),
        borderRadius: BorderRadius.circular(15),
        ), */
        alignment: Alignment.center,
//color: Colors.blue,
// we can adjust width and height
// to do that we need to commented out the constraints
        width: 350.00,
        height: 350.00,
// to skew the container
        // transform: Matrix4.rotationZ(0.1),
        decoration: BoxDecoration(
        color: Colors.blue,
        border: Border.all(
            color: Colors.red,
            width: 2.0,
            style: BorderStyle.solid,
        ),
        borderRadius: const BorderRadius.all(Radius.circular(40.0)),
        boxShadow: const [
            BoxShadow(
            color: Colors.black54,
            blurRadius: 20.0,
            spreadRadius: 20.0,
            ),
        ],
        gradient: const LinearGradient(
            begin: Alignment.centerLeft,
            end: Alignment.centerRight,
```

```
            colors: [
            Colors.red,
            Colors.white,
            ],
        ),
        ),
),
...
// the code is incomplete for brevity
```

To outline, we've used border property and supply the color outline, which is red.

In the same vein, we've also modified gradient property, changed the box shadow and used border radius.

However, we've commented out the transform property only.

Why?

Because we will use that to tweak the look later.

What is a Transform in Flutter?

Transform is a widget in flutter. It does many thing.

It changes the shape, size, position and orientation.

Only one line of code inside the Container widget has made it possible.

```
transform: Matrix4.rotationZ(0.1),
```

You can read more about the transform class and widget in Flutter documentation.

What is a RichText in flutter?

In Flutter, RichText is a widget that helps us to add more styling to a paragraph of text. Otherwise we cannot do that.

However, with Text widget we cannot achieve the same effect.

In RichText widget, we can return several TextSpan widgets and, moreover, in each TextSpan we can add different type of styling.

Not only that, the TextSpan is a subclass of InLineSpan class that is actually an immutable span of inline content, or text that forms a part of a paragraph.

In addition, the TextSpan widget has a Gesture Detector property called recognizer. Using recognizer we can easily build a navigation link in that particular part of paragraph.

In short, RichText widget gives us multiple advantages to style a paragraph of Text.

How we can do that?

We'll see in a minute.

How do you add RichText on flutter?

Before we add RichText on Flutter, let's revisit our News App, where we last decorated our categories screen.

However, the Favorites page remained the same. We didn't change the styling after that.

Now, we can change this Favorites screen completely by adding the RichText widget and several TextSpan widgets.

Not only that, we'll also display navigation link inside the paragraph, so that users can click or tap on the link and go to another page.

The code snippet of this page now looks like the following. We've already learned that a RichText widget can return a children of TextSpan widgets.

```
import 'package:flutter/gestures.dart';
import 'package:flutter/material.dart';
import 'package:news_app_extended/views/local_news.dart';

class FavoritesScreen extends StatelessWidget {
const FavoritesScreen({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return _richTextController(context);
}
}

Widget _richTextController(BuildContext context) => Container(
    color: Colors.black12,
    padding: const EdgeInsets.all(10),
    child: Center(
        child: RichText(
        text: TextSpan(
            text: 'This News App is inspired by the principle of free'
                ' Journalism. You can select ',
```

```
22              style: const TextStyle(
23              color: Colors.black,
24              fontSize: 20,
25              fontWeight: FontWeight.bold,
26              ),
27              children: [
28              TextSpan(
29                  text: 'Local',
30                  style: const TextStyle(
31                  color: Colors.deepOrange,
32                  fontSize: 20,
33                  fontWeight: FontWeight.bold,
34                  decoration: TextDecoration.underline,
35                  ),
36                  recognizer: TapGestureRecognizer()
37                  ..onTap = () {
38                      Navigator.of(context).pushNamed(LocalNews.routeName);
39                  },
40              ),
41              const TextSpan(
42                  text: ' to ',
43                  style: TextStyle(
44                  color: Colors.grey,
45                  fontSize: 20,
46                  fontWeight: FontWeight.bold,
47                  ),
48              ),
49              TextSpan(
50                  text: 'Global,',
51                  style: const TextStyle(
52                  color: Colors.deepOrange,
53                  fontSize: 20,
54                  fontWeight: FontWeight.bold,
55                  decoration: TextDecoration.underline,
56                  ),
57                  recognizer: TapGestureRecognizer()
58                  ..onTap = () {
59                      //
60                  },
61              ),
62              const TextSpan(
63                  text: ' news, and not only that, you can take part as a'
64                      ' Citizen Journalist to publish your story.',
```

```
                style: TextStyle(
                color: Colors.green,
                fontSize: 20,
                fontWeight: FontWeight.bold,
                ),
            )
            ],
        ),
        ),
    ),
);

// Although we have the full code snippet of Favorites screen, but to understand the\
 architecture of News App, please visit the respective GitHub repository
```

If we don't get a glimpse of this page, we cannot understand how RichText widget helps us to add styling to a whole paragraph of text.

How do you span text in flutter?

We not only span the text of the screen but with the help of TextSpan widget, we've added several functionality.

In addition, we've changed the background color, a part of the paragraph is in black and the other is in green.

We've also changed the navigation color, adding an underline.

```
TextSpan(
                text: 'Local',
                style: const TextStyle(
                color: Colors.deepOrange,
                fontSize: 20,
                fontWeight: FontWeight.bold,
                decoration: TextDecoration.underline,
                ),
                recognizer: TapGestureRecognizer()
                ..onTap = () {
                    Navigator.of(context).pushNamed(LocalNews.routeName);
                },
            ),
```

The above code snippet allows us to add the navigation link to another page, or screen.

As an example, we've only created a Local News Page, where we can navigate from this page.

To do that, we have added a route name in Material Design page first.

```
routes: {
        '/': (ctx) => const TabsScreen(),
        CategoryNewsScreen.routeName: (ctx) => const CategoryNewsScreen(),
        NewsDetailScreen.routeName: (ctx) => const NewsDetailScreen(),
        LocalNews.routeName: (ctx) => const LocalNews(),
},
```

After that, we've added a static route name property in the Local News screen widget.

How do you use TapGestureRecognizer flutter?

In fact, without the TapGestureRecognizer class we cannot navigate to the Local News page.

Isn't it?

```
recognizer: TapGestureRecognizer()
                ..onTap = () {
                    Navigator.of(context).pushNamed(LocalNews.routeName);
                },
```

Now we can tap the link and the TapGestureRecognizer handles the event.

And, finally we reach to a new screen.

To sum up, with the help of RichText widget and by returning several TextSpan widgets we've added more styling to our News App.

To have a look at the screenshots we've used in this section, please visit this LINK[24]

Stateful vs Stateless Flutter

To have a look at the screenshots we've used in this section, please visit this LINK[25]

Stateless widgets are immutable. Since their properties cannot change, all values are final. On the contrary, stateful widgets maintain state.

For that reason, widgets are rebuilt with the change of the state.

Does that affect flutter application?

Certainly, that rebuilds widgets and affects the performance.

Then what should we do to solve that problem?

[24] https://sanjibsinha.com/richtext-in-flutter/
[25] https://sanjibsinha.com/stateless-vs-stateful-flutter/

When should I use provider Flutter?

We should use provider package to solve that problem.

Why should we use provider package?

The main reason is, provider package will rebuild only the widget that needs the value. The widget that needs the value is known as consumer.

As a result, that consumer widget only rebuilds itself. For example, consider a simple counter. As we press the button to change the state, each pressing changes the state and rebuilds the widget.

For a stateful widget, that rebuild affects the whole tree of widgets.

In Android Studio, we can watch the flutter performance.

Firstly, we see the counter example with the stateful widget.

```
import 'package:flutter/material.dart';

void main() {
runApp(MyApp());
}

class MyApp extends StatelessWidget {
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(
        primarySwatch: Colors.blue,
    ),
    home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
}
}

class MyHomePage extends StatefulWidget {
MyHomePage({Key? key, required this.title}) : super(key: key);

final String title;

@override
_MyHomePageState createState() => _MyHomePageState();
}
```

```
29
30 class _MyHomePageState extends State<MyHomePage> {
31 int _counter = 0;
32
33 void _incrementCounter() {
34     setState(() {
35     _counter++;
36     });
37 }
38
39 @override
40 Widget build(BuildContext context) {
41
42     return Scaffold(
43     appBar: AppBar(
44         title: Text(widget.title),
45     ),
46     body: Center(
47         child: Column(
48         mainAxisAlignment: MainAxisAlignment.center,
49         children: <Widget>[
50             Text(
51             'You have pushed the button this many times:',
52             ),
53             Text(
54             '$_counter',
55             style: Theme.of(context).textTheme.headline4,
56             ),
57         ],
58         ),
59     ),
60     floatingActionButton: FloatingActionButton(
61         onPressed: _incrementCounter,
62         tooltip: 'Increment',
63         child: Icon(Icons.add),
64     ),
65     );
66 }
67 }
```

Let us press the button for five times and side by side watch the flutter performance.

Now, let us take a closer look and the screenshot looks like the following image.

It clearly shows that after pressing the button for five times, on the far right side of the screen each widget shows the number 6.

That means a stateful way always requires more memory than the stateless way.

In fact that is clearly visible in the first image, because the red colored bar represents memory usage.

This is the reason why we should use a combination of Provider package and stateless widgets.

It makes our flutter app more performant.

How do you use Flutter provider package?

We use Flutter provider package like any other packages. Firstly, we add the provider package to the dependencies section in our pubspec.yaml file.

```
dependencies:
provider: ^6.0.0
```

Secondly, we import the provider package just like any other packages.

```
import 'package:provider/provider.dart';
```

Finally, we need to notify the listeners that we're going to change the state.

We can define that method in our model folder, from where our data come.

```
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

class Counter with ChangeNotifier {
int _count = 0;

int get count => _count;

void increment() {
    _count++;
    notifyListeners();
}
}
```

Next, to run the app we must place providers above the Counter App. Not only that, we need to use other methods so that we can read and watch the change.

```dart
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'package:shop_app/models/counter.dart';
void main() {
runApp(
    MultiProvider(
    providers: [
        ChangeNotifierProvider(create: (_) => Counter()),
    ],
    child: const CounterApp(),
    ),
);
}

class CounterApp extends StatelessWidget {
const CounterApp({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: CounterHomePage(),
    );
}
}

class CounterHomePage extends StatelessWidget {
const CounterHomePage({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    final subtree = ConstantWidget(
        child: const Text("Hello World")
    );
    final anotherSubtree = _widget();
    return Scaffold(
    appBar: AppBar(
        title: const Text('A Stateless Counter App'),
    ),
    body: Center(
        child: Column(
        mainAxisSize: MainAxisSize.min,
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget> [
            Text('You have pushed the button this many times:'),
```

```
            Text(
            '${context.watch<Counter>().count}',
            key: const Key('counterState'),
            style: Theme.of(context).textTheme.headline4,
            ),
            SizedBox(width: 10.0,),
            subtree,
            SizedBox(width: 10.0,),
            anotherSubtree,
        ],
        ),
    ),
    floatingActionButton: FloatingActionButton(
        key: const Key('increment_floatingActionButton'),
        onPressed: () => context.read<Counter>().increment(),
        tooltip: 'Increment',
        child: const Icon(Icons.add),
    ),
    );
}
/// Just to test rebuild
Widget _widget() => const Text('Hi');

ConstantWidget({required Text child}) {
    /// Just to test rebuild
    return Text('Hello');
}
}
```

In the above code, a few lines are extremely important.

Firstly, we've placed the Multi Provider above our counter app.

Secondly, when the context use read and watch methods, we've supplied the type Counter that we've defined in the model folder.

```
'${context.watch<Counter>().count}',
onPressed: () => context.read<Counter>().increment(),
```

What is the difference between context.read and context.watch?

It's a key concept while we're trying to understand the role of Provider package.

To display the count number, we use context.watch.

Why?

We call context.watch to make the count property of Type Counter rebuild when the counter changes.

However, inside the floating action button on press method we return context.read.

It stops the floating action button widget from rebuilding itself.

Now, as a result, we get a different screenshot when we press the button.

With the help of Provider package, we have kept the requirement of memory usage much lower than the stateful widget.

As we can see, there are no red bars.

Not only that, now we can take a closer look at the flutter performance.

As we keep on pressing the button, only two widgets rebuild themselves without affecting the others.

Moreover, now the state change centers on only two text widgets.

The parent tree remains unaffected.

What is GridTile Flutter? How do you use grid tiles in Flutter?

To have a look at the screenshots we've used in this section, please visit this LINK[26]

The GridTile in Flutter is a subclass of Stateless Widget, which is a scrollable grid of tiles. And GridTile typically uses GridTileBar Widget either in header or footer. Not both, at the same time.

Also GridTile must have a child. However, the GridTileBar stateless widget that returns an IconButton widget.

Suppose we're going to build a book buying e-cart flutter app.

In such a scenario, we can have a scrollable grid of tiles that will place the Book items first. Next, moreover, on each book image we can have two GridTileBar widgets.

How do you make a grid Flutter?

Certainly, without the help of GridView widget, we couldn't make that. So let us start from the Material design section.

[26] https://sanjibsinha.com/gridtile-flutter/

```
Widget build(BuildContext context) {
    return ChangeNotifierProvider.value(
    value: Books(),
    child: MaterialApp(
        title: 'Book Shop',
        theme: ThemeData(
        primarySwatch: Colors.purple,
        primaryColor: Colors.deepOrange,
        ),
        home: const BooksOverviewScreen(),
        routes: {
        BookDetailScreen.routeName: (ctx) => const BookDetailScreen(),
        },
    ),
    );
}
// the code is incomplete for brevity
```

As we can see, the Books overview screen displays the book items using three main stateless widgets.

They are GridTile, GridTileBar and GridView.

We're going to the glimpse of the code snippets. For full code, please visit the respective GitHub repository.

Firstly take a look at how we have used GridTile.

```
class BookItem extends StatelessWidget {
const BookItem({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    final product = Provider.of<Book>(context, listen: false);
    return ClipRRect(
    borderRadius: BorderRadius.circular(10),
    child: GridTile(
        child: GestureDetector(
        onTap: () {
            Navigator.of(context).pushNamed(
            BookDetailScreen.routeName,
            arguments: product.id,
            );
        },
        child: Image.network(
```

```
            product.imageUrl,
            fit: BoxFit.cover,
        ),
        ),
        footer: GridTileBar(
        backgroundColor: Colors.black87,
        leading: Consumer<Book>(
            builder: (ctx, product, _) => IconButton(
            icon: Icon(
                product.isFavorite ? Icons.favorite : Icons.favorite_border,
            ),
            color: Theme.of(context).primaryColor,
            onPressed: () {
                product.toggleFavoriteStatus();
            },
            ),
        ),
        title: Text(
            product.title,
            textAlign: TextAlign.center,
        ),
        trailing: IconButton(
            icon: const Icon(
            Icons.shopping_cart,
            ),
            onPressed: () {},
            color: Theme.of(context).primaryColor,
        ),
        ),
    ),
    );
}
}
```

As we mentioned earlier, any GridTile must have a child and a GridTileBar widget either in header or footer.

In the above code, we've placed it in the footer.

Inside the GridTileBar, we've used two IconButton widgets. One represents a favorite symbol and the other displays a shopping cart.

Since we can use each IconButton to press and fire a function, therefore, we've used notify listeners to change the color of favorite sign.

As a result, we can have an image like the following where we've clicked favorites sign for three times.

We have managed the state in the stateless widgets by using Provider package.

Certainly we'll discuss that in a separate article. But before that, let us see how we've used GridView widget to display the clicked favorite book items.

```
class BooksGrid extends StatelessWidget {
final bool showFavs;

const BooksGrid({
    Key? key,
    required this.showFavs,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    final productsData = Provider.of<Books>(context);
    final products = showFavs ? productsData.favoriteItems : productsData.items;
    return GridView.builder(
    padding: const EdgeInsets.all(10.0),
    itemCount: products.length,
    itemBuilder: (ctx, i) => ChangeNotifierProvider.value(
        // builder: (c) => products[i],
        value: products[i],
        child: const BookItem(
            // products[i].id,
            // products[i].title,
            // products[i].imageUrl,
            ),
    ),
    gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
        crossAxisCount: 2,
        childAspectRatio: 3 / 2,
        crossAxisSpacing: 10,
        mainAxisSpacing: 10,
    ),
    );
}
}
```

What is change notifier provider in Flutter?

To have a look at the screenshots we've used in this section, please visit this LINK[27]

ChangeNotifierProvider is a class that comes with the Provider package.

Although I've written about change notifier before, yet this article is all together different.

We're going to build a Book Shopping Cart flutter app, and to maintain state we've used Provider package for the best result and less widget rebuilds.

To sum up, ChangeNotifierProvider listens to a ChangeNotifier. Not only listening but it also expose ChangeNotifier to its descendants and rebuilds dependents when the state changes.

When does state change?

As ChangeNotifier notifies its listeners, the state changes.

```
import 'package:flutter/foundation.dart';

class Book with ChangeNotifier {
final String id;
final String title;
final String description;
final double price;
final String imageUrl;
bool isFavorite;

Book({
    required this.id,
    required this.title,
    required this.description,
    required this.price,
    required this.imageUrl,
    this.isFavorite = false,
});

void toggleFavoriteStatus() {
    isFavorite = !isFavorite;
    notifyListeners();
}
}
```

[27] https://sanjibsinha.com/change-notifier-provider-flutter/

The above code means when we click the favorite icon on the Book item, the ChangeNotifier notify the listeners.

Now, question is what is the correct way to create a ChangeNotifier?

According to the documentation provided by the Provider package ChangeNotifierProvider, the following way is the correct way.

```
void main() {
runApp(
    MultiProvider(
    providers: [
        ChangeNotifierProvider(
        create: (_) => Books(),
        ),
        ChangeNotifierProvider(
        create: (_) => Cart(),
        ),
        ChangeNotifierProvider(
        create: (_) => Orders(),
        ),
    ],
    child: const BookApp(),
    ),
);
}
class BookApp extends StatelessWidget {
const BookApp({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Book Shop',
    theme: ThemeData(
        primarySwatch: Colors.purple,
        primaryColor: Colors.deepOrange,
    ),
    home: const BooksOverviewScreen(),
    routes: {
        BookDetailScreen.routeName: (ctx) => const BookDetailScreen(),
    },
    );
}
}
```

```
37
38 // for full code please visit Book Shopping Cart flutter app
```

We must place the Providers above the Book App. As a result, we can read and watch the change of state in Book Overview screen.

What is change notifier provider?

The ChangeNotifierProvider works as a provider-wrapper class that creates a ChangeNotifier and automatically disposes it when ChangeNotifierProvider is removed from the widget tree.

By the way, create must not be null.

Now we can click the favorite icon and change the state automatically, because ChangeNotifier will notify the listeners.

How it does that? It provides an existing ChangeNotifier.

```
1  Widget build(BuildContext context) {
2      final productsData = Provider.of<Books>(context);
3      final products = showFavs ? productsData.favoriteItems : productsData.items;
4      return GridView.builder(
5      padding: const EdgeInsets.all(10.0),
6      itemCount: products.length,
7      itemBuilder: (ctx, i) => ChangeNotifierProvider.value(
8          // builder: (c) => products[i],
9          value: products[i],
10         child: const BookItem(
11             ),
12     ),
13     gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
14         crossAxisCount: 2,
15         childAspectRatio: 3 / 2,
16         crossAxisSpacing: 10,
17         mainAxisSpacing: 10,
18     ),
19     );
20 }
21 // for full code please visit Book Shopping Cart flutter app
```

From Book overview screen we can go the book detail page quite easily now.

The book detail screen works on product ID.

```dart
class BookDetailScreen extends StatelessWidget {
static const routeName = '/product-detail';

const BookDetailScreen({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    final productId =
        ModalRoute.of(context)!.settings.arguments as String; // is the id!
    final loadedProduct = Provider.of<Books>(
    context,
    listen: false,
    ).findById(productId);
    return Scaffold(
    appBar: AppBar(
        title: Text(loadedProduct.title),
    ),
    body: SingleChildScrollView(
        child: Column(
        children: <Widget>[
            SizedBox(
            height: 300,
            width: double.infinity,
            child: Image.network(
                loadedProduct.imageUrl,
                fit: BoxFit.cover,
            ),
            ),
            const SizedBox(height: 10),
            Text(
            '\$${loadedProduct.price}',
            style: const TextStyle(
                color: Colors.grey,
                fontSize: 20,
            ),
            ),
            const SizedBox(
            height: 10,
            ),
            Container(
            padding: const EdgeInsets.symmetric(horizontal: 10),
            width: double.infinity,
            child: Text(
```

```
                loadedProduct.description,
                textAlign: TextAlign.center,
                softWrap: true,
              ),
              )
          ],
          ),
    ),
    );
}
}
// for full code please visit Book Shopping Cart flutter app
```

If we take a detail look at the Book Shopping Cart flutter app, we will find how ChangeNotifier-Provider creates the ChangeNotifier and how that notifies the listeners where it is needed.

How do you change the font on flutter?

To have a look at the screenshots we've used in this section, please visit this LINK[28]

To change the font on Flutter and give our app a unique look, we need to follow a few steps. Here we'll quickly go through them.

We'll also learn how to download unique fonts, where to place it and, in addition, how to add that font name to our text style.

This article on flutter font, will be quick and brief.

So, let's start.

Firstly, let us see our common problem and how to solve that problem.

The following image shows a detail screen of e-commerce flutter app, where we'll get the details about the product.

It's a Book App. For the primary code snippet please visit respective GitHub Repository.

The above image displays the product detail screen. However, we've not added the unique flutter font style yet.

So that was the first part. Secondly, we'll add the unique flutter fonts to change this product detail screen.

To do that, we need to download two unique Google fonts – Anton and Lato.

Keep them in a separate folder and name it "assets/fonts" in our flutter app directory.

Next, we'll add the font path to our "pubspec.yaml" file.

[28] https://sanjibsinha.com/flutter-font/

```
# To add custom fonts to your application, add a fonts section here,
# in this "flutter" section. Each entry in this list should have a
# "family" key with the font family name, and a "fonts" key with a
# list giving the asset and other descriptors for the font. For
# example:
fonts:
    - family: Lato
    fonts:
        - asset: assets/fonts/Lato-Bold.ttf
#       - asset: fonts/Schyler-Italic.ttf
#         style: italic
    - family: Anton
    fonts:
        - asset: assets/fonts/Anton-Regular.ttf
#       - asset: fonts/TrajanPro_Bold.ttf
#         weight: 700
```

Now, our unique flutter fonts has been added globally. We can add any one of these flutter font anywhere in our app.

Consequently, we can add this font to our Book or product detail screen. What font has flutter?

If we don't want to change the look, flutter manages it.

That's not a problem.

But, if we want to change the flutter font, then we have to through the above steps that we've already mentioned.

Moreover, to add some special flutter font effects to any page or screen of our flutter app, we need to mention our unique flutter font name inside TestStyle widget.

```
import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

import '../models/books.dart';

class BookDetailScreen extends StatelessWidget {
static const routeName = '/product-detail';

const BookDetailScreen({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    final productId =
```

```
15          ModalRoute.of(context)!.settings.arguments as String; // is the id!
16      final loadedProduct = Provider.of<Books>(
17      context,
18      listen: false,
19      ).findById(productId);
20      return Scaffold(
21      appBar: AppBar(
22          title: Text(loadedProduct.title),
23      ),
24      body: SingleChildScrollView(
25          child: Column(
26          children: <Widget>[
27              SizedBox(
28              height: 300,
29              width: double.infinity,
30              child: Image.network(
31                  loadedProduct.imageUrl,
32                  fit: BoxFit.cover,
33              ),
34              ),
35              const SizedBox(height: 10),
36              Text(
37              '\$${loadedProduct.price}',
38              style: const TextStyle(
39                  color: Colors.grey,
40                  fontSize: 20,
41              ),
42              ),
43              const SizedBox(
44              height: 10,
45              ),
46              Container(
47              padding: const EdgeInsets.symmetric(horizontal: 10),
48              width: double.infinity,
49              child: Text(
50                  loadedProduct.title,
51                  textAlign: TextAlign.center,
52                  softWrap: true,
53                  style: TextStyle(
54                  fontFamily: 'Lato',
55                  fontSize: 30.0,
56                  ),
57              ),
```

```
            ),
            const SizedBox(
            height: 10,
            ),
            Container(
            padding: const EdgeInsets.symmetric(horizontal: 10),
            width: double.infinity,
            child: Text(
                loadedProduct.description,
                textAlign: TextAlign.center,
                softWrap: true,
                style: TextStyle(
                fontFamily: 'Anton',
                fontSize: 20.0,
                ),
            ),
            ),
        ],
        ),
    ),
    );
}
}
// for full code please visit respective GitHub Repository.
```

As we can see that in the above code snippet, these two Container widget are responsible to display the flutter font Anton and Lato.

Watch this part carefully:

```
Container(
            padding: const EdgeInsets.symmetric(horizontal: 10),
            width: double.infinity,
            child: Text(
                loadedProduct.title,
                textAlign: TextAlign.center,
                softWrap: true,
                style: TextStyle(
                fontFamily: 'Lato',
                fontSize: 30.0,
                ),
            ),
            ),
```

```
// and this part

Container(
            padding: const EdgeInsets.symmetric(horizontal: 10),
            width: double.infinity,
            child: Text(
                loadedProduct.description,
                textAlign: TextAlign.center,
                softWrap: true,
                style: TextStyle(
                fontFamily: 'Anton',
                fontSize: 20.0,
                ),
            ),
            ),
```

For the product title we've used Lato. And, for the product description, we've used Anton.

As a result, we've got this unique look of the product detail screen.

Please compare this image with the previous image we've displayed in this page before.

The unique flutter font has changed the look of the product detail screen completely.

How do I store persistent data in Flutter?

To have a look at the screenshots we've used in this section, please visit this LINK[29]

To store persistent data in Flutter we can adopt different approaches. Either we can use local storage in key, value pair array; or, we can use SQLite database.

In addition we can use remote database also.

However, the concept revolves around state management, accessorized with Provider package.

The question is, how we can use the state management using Provider package between screens and store persistent data in flutter.

One may ask, why should we use Provider package? We not stateful widget? After all, through stateful widget, we can also store persistent data in flutter.

In addition, stateful widget comes with Flutter, an in-built feature to maintain state.

The one and only answer is, through Provider package we can avoid widget rebuilds. To get the concept, and if you want an evidence please read my previous blog post on stateful vs stateless widgets.

[29] https://sanjibsinha.com/flutter-provider-persist/

Now, let us start learning how to store persistent data in flutter. At the very beginning, let me tell you, the full code snippet is available in the respective GitHub Repository.

We cannot use full code snippets. As they are too long. Therefore, we must use part of them for brevity.

Above all, we'll follow MVC or Model-view-controller pattern.

So data comes from the models folder.

```
import 'package:flutter/foundation.dart';

class Book with ChangeNotifier {
final String id;
final String title;
final String description;
final double price;
final String imageUrl;
bool isFavorite;

Book({
    required this.id,
    required this.title,
    required this.description,
    required this.price,
    required this.imageUrl,
    this.isFavorite = false,
});

void toggleFavoriteStatus() {
    isFavorite = !isFavorite;
    notifyListeners();
}
}
```

In the above code, we have defined a Book class whose constructor would pass different properties including a boolean value isFavorite.

We've used Dart mixin to use ChangeNotifier, so that we can use ChangeNotifierProvider from provider package.

Now, we must have another data model which is actually our local storage. Based on the product ID, we can retrieve data from that storage.

```
import 'package:flutter/material.dart';

import 'book.dart';

class Books with ChangeNotifier {
final List<Book> _items = [
    Book(
    id: 'p1',
    title: 'Beginning Flutter With Dart',
    description: 'You can learn Flutter as well Dart.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2014/09/05/18/32/old-books-436498_960_720.jpg\
',
    ),
    Book(
    id: 'p2',
    title: 'Flutter State Management',
    description: 'Everything you should know about Flutter State.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960_720.jpg',
    ),
...
// code is incomplete for brevity
```

Now, the question is, how we will persist state in flutter?

How do you persist state in Flutter?

To persist state in Flutter, we must place our ChangeNotifierProvider on the top of the Book shopping cart that we've been building.

The following code snippet from the main dart file will give you an idea.

```
void main() {
runApp(
    MultiProvider(
    providers: [
        ChangeNotifierProvider(
        create: (_) => Books(),
        ),
        ChangeNotifierProvider(
        create: (_) => Cart(),
        ),
        ChangeNotifierProvider(
        create: (_) => Orders(),
        ),
    ],
    child: const BookApp(),
    ),
);
}

class BookApp extends StatelessWidget {
const BookApp({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Book Shop',
    theme: ThemeData(
        primarySwatch: Colors.purple,
        primaryColor: Colors.deepOrange,
    ),
    home: const BooksOverviewScreen(),
    routes: {
        BookDetailScreen.routeName: (ctx) => const BookDetailScreen(),
        //CartScreen.routeName: (ctx) => CartScreen(),
    },
    );
}
}
```

The home page route points to the books overview screen. However, with the help of other controllers we will be able to navigate to the book detail page,or screen.

Now, in our material design we've defined the route, so one click on the image will take us to the respective book detail screen.

In our controllers folder, we have two widgets that act as consumers who will listen to the notification sent by the ChangeNotifier.

The following code snippet will give us an idea.

```
class _BooksOverviewScreenState extends State<BooksOverviewScreen> {
var _showOnlyFavorites = false;

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Book Shop'),
        actions: <Widget>[
        PopupMenuButton(
            onSelected: (FilterOptions selectedValue) {
            setState(() {
                if (selectedValue == FilterOptions.Favorites) {
                _showOnlyFavorites = true;
                } else {
                _showOnlyFavorites = false;
                }
            });
            },
            icon: const Icon(
            Icons.more_vert,
            ),
            itemBuilder: (_) => [
            const PopupMenuItem(
                child: Text('Only Favorites'),
                value: FilterOptions.Favorites,
            ),
            const PopupMenuItem(
                child: Text('Show All'),
                value: FilterOptions.All,
            ),
            ],
        ),
        ],
    ),
    body: BooksGrid(showFavs: _showOnlyFavorites),
    );
}
}
```

As we can see, the body of the books overview screen points to books grid widget. Therefore, we can have a look at that widget in controllers folder.

```
@override
Widget build(BuildContext context) {
    final productsData = Provider.of<Books>(context);
    final products = showFavs ? productsData.favoriteItems : productsData.items;
    return GridView.builder(
    padding: const EdgeInsets.all(10.0),
    itemCount: products.length,
    itemBuilder: (ctx, i) => ChangeNotifierProvider.value(
        // builder: (c) => products[i],
        value: products[i],
        child: const BookItem(

            ),
    ),
..
// incomplete code for brevity
```

To persist data, we've used Provider.of method which passes context as parameter and uses the type Book class.

Here that plays the crucial role.

```
final productsData = Provider.of<Books>(context);
```

Not only that, the following part of the above code also helps us to persist data.

```
itemBuilder: (ctx, i) => ChangeNotifierProvider.value(
        // builder: (c) => products[i],
        value: products[i],
        child: const BookItem(

            ),
    ),
```

Although the Book App is in primary stage, still it's worth taking a look at the GitHub Repository.

Moreover, we'll keep building that e-commerce Book App in the future.

So, stay tuned and get in touch with all Flutter articles that always gives you updated information on Flutter.

What is data model in Flutter?

To have a look at the screenshots we've used in this section, please visit this LINK[30]

Model in flutter refers to data flow and it corresponds to the MVC or model view controller architecture. Although there is no hard and fast rule, but if data flow comes from the models it always works fine.

Otherwise, managing data becomes a tedious job.

In this brief introduction to data model in flutter, we'll build a model class first. After that with the help of local storage we'll display that data on the home screen.

By the way, for this small app please visit this GitHub repository.

Suppose we're going to build a shopping cart. In short, we'll sell books.

As a result we need a good display of books on our home screen, like the following image.

The data displayed on the home screen, comes from a data model class. And that class defines the id, title, description, price and the image URL.

Therefore, the model class looks like this:

```
class Book {
final String id;
final String title;
final String description;
final double price;
final String imageUrl;
bool isFavorite;

Book({
    required this.id,
    required this.title,
    required this.description,
    required this.price,
    required this.imageUrl,
    this.isFavorite = false,
});
}
```

We keep this data defining class in our models folder. Now, to add some using the constructor of this Book class, we can adopt two ways.

Either we can keep that data in our models folder, or we can create a list of books in our stateless book overview screen.

[30] https://sanjibsinha.com/data-model-flutter/

How do you get data from model in Flutter?

There are many ways by which we can get data from model in flutter. Not only that, we can also store persistent data in flutter.

Above all, retrieving data from a resource is our main target.

The resource could be a local storage or a remote source.

For the time being we prefer local storage.

Therefore, in the book overview screen, we define a list of books with all the named parameters.

```
import 'package:flutter/material.dart';
import 'package:shop_app_primary/data-model/controllers/book_item.dart';
import 'package:shop_app_primary/data-model/models/book.dart';

class BooksOverviewScreen extends StatelessWidget {
BooksOverviewScreen({Key? key}) : super(key: key);

final List<Book> books = [
    Book(
    id: 'p1',
    title: 'Beginning Flutter With Dart',
    description: 'You can learn Flutter as well Dart.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2014/09/05/18/32/old-books-436498_960_720.jpg\
',
    ),
    Book(
    id: 'p2',
    title: 'Flutter State Management',
    description: 'Everything you should know about Flutter State.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960_720.jpg',
    ),
    Book(
    id: 'p3',
    title: 'WordPress Coding',
    description:
        'WordPress coding is not difficult, in fact it is interesting.',
    price: 9.99,
    imageUrl:
```

```
        'https://cdn.pixabay.com/photo/2015/11/19/21/10/glasses-1052010_960_720.jpg',
    ),
    Book(
    id: 'p4',
    title: 'PHP 8 Standard Library',
    description: 'PHP 8 Standard Library has made developers life easier.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2015/09/05/21/51/reading-925589_960_720.jpg',
    ),
    Book(
    id: 'p5',
    title: 'Better Flutter',
    description:
        'Learn all the necessary concepts of building a Flutter App.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2015/09/05/07/28/writing-923882_960_720.jpg',
    ),
    Book(
    id: 'p6',
    title: 'Discrete Mathematical Data Structures and Algorithm',
    description:
        'Discrete mathematical concepts are necessary to learn Data Structures and A\
lgorithm.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2015/11/19/21/14/glasses-1052023_960_720.jpg',
    ),
];

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text('MyShop'),
    ),
    body: GridView.builder(
        padding: const EdgeInsets.all(10.0),
        itemCount: books.length,
        itemBuilder: (ctx, i) => BookItem(
        books[i].id,
        books[i].title,
```

```
        books[i].imageUrl,
        ),
        gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
        crossAxisCount: 2,
        childAspectRatio: 3 / 2,
        crossAxisSpacing: 10,
        mainAxisSpacing: 10,
        ),
    ),
    );
}
}
```

Although the code snippet is a little bit longer but that is due to the list of books we've added as our local storage.

However, display of data has been managed by another custom widget BookItem.

Watch this part of code:

```
body: GridView.builder(
        padding: const EdgeInsets.all(10.0),
        itemCount: books.length,
        itemBuilder: (ctx, i) => BookItem(
        books[i].id,
        books[i].title,
        books[i].imageUrl,
        ),
```

The item builder of Grid view builder passes the context and the index of the list. Next, through BookItem stateless widget constructor we can pass that data.

Now, we keep that stateless custom widget BookItem in our controllers folder.

In addition, the BookItem widget extracts the data from the data model and sends that back to the book overview screen.

How do you use lists in Flutter?

We have just seen how we can use lists of books in flutter. But extracting data from that list is the main idea.

The BookItem widget in controllers folder handles this extraction and sends the data back to the book overview screen in the views folder.

All together, the MVC structure in our small flutter app works fine. The data comes from the model class, the controller extracts that data and sends that data to the view.

The picture of data flow becomes clear.

Finally we take a look at the controller, which extracts data from the model class.

```
import 'package:flutter/material.dart';

class BookItem extends StatelessWidget {
final String id;
final String title;
final String imageUrl;

BookItem(this.id, this.title, this.imageUrl);

@override
Widget build(BuildContext context) {
    return ClipRRect(
    borderRadius: BorderRadius.circular(10),
    child: GridTile(
        child: GestureDetector(
        onTap: () {},
        child: Image.network(
            imageUrl,
            fit: BoxFit.cover,
        ),
        ),
        footer: GridTileBar(
        backgroundColor: Colors.black87,
        leading: IconButton(
            icon: const Icon(Icons.favorite),
            color: Theme.of(context).primaryColor,
            onPressed: () {},
        ),
        title: Text(
            title,
            textAlign: TextAlign.center,
        ),
        trailing: IconButton(
            icon: const Icon(
            Icons.shopping_cart,
            ),
            onPressed: () {},
```

```
                color: Theme.of(context).primaryColor,
            ),
            ),
        ),
        );
}
}
```

How do you pass data between screens in flutter?

To have a look at the screenshots we've used in this section, please visit this LINK[31]

To pass data between screens in Flutter, we need a list of data first. Most importantly that list of data must have an id.

Based on that id, we can retrieve other items of that list.

We've already learned about persistent data, however, we have not discussed how we can pass data between screens.

In this article we'll discuss on passing data between screens and for more curious readers we've kept the full code snippet at this particular GitHub Repository called Book App First.

In future, we'll try to turn that into a full shopping cart flutter app. Therefore, keep reading and, if you're a Flutter enthusiast, please get in touch with the other articles on flutter. We update regularly.

Firstly, we need a data model class Book.

```dart
import 'package:flutter/foundation.dart';

class Book with ChangeNotifier {
final String id;
final String title;
final String description;
final double price;
final String imageUrl;
bool isFavorite;

Book({
    required this.id,
    required this.title,
    required this.description,
    required this.price,
```

[31] https://sanjibsinha.com/pass-data-between-screens-flutter/

```
    required this.imageUrl,
    this.isFavorite = false,
});

void toggleFavoriteStatus() {
    isFavorite = !isFavorite;
    notifyListeners();
}
}
```

Next, we need a local storage of data where we must fill in with some dummy data.

For images, we could have used local assets. However, we prefer image network instead.

Consequently, let's get a glimpse of the dummy data.

```
import 'package:flutter/material.dart';

import 'book.dart';

class Books with ChangeNotifier {
final List<Book> _items = [
    Book(
    id: 'p1',
    title: 'Beginning Flutter With Dart',
    description: 'You can learn Flutter as well Dart.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2014/09/05/18/32/old-books-436498_960_720.jpg\
',
    ),
    Book(
    id: 'p2',
    title: 'Flutter State Management',
    description: 'Everything you should know about Flutter State.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960_720.jpg',
    ),
...
// code incomplete for brevity
```

Now, we can have a books overview screen that displays all the book items. Similarly, if we click any image that will navigate to the detail screen.

Let's take a look at the books overview screen first.

In our dummy data we've added six books. On the other hand, we have only one book-detail page or screen.

That's the advantage of using the ID of the data that passes between screens.

By the way, to maintain the state of this flutter app, we've used provider package and ChangeNotifierProvider.

How do you pass data to a child widget in flutter?

The book over view screen has been controlled by two widgets, which are kept in the controllers folder.

As a result, we see that book overview screen has the body that points to a custom Books Grid.

```
body: BooksGrid(showFavs: _showOnlyFavorites),
```

The Books Grid widget, on the other hand, have products data and a child widget Book Items where it passes the data.

```
final productsData = Provider.of<Books>(context);
    final products = showFavs ? productsData.favoriteItems : productsData.items;
    return GridView.builder(
    padding: const EdgeInsets.all(10.0),
    itemCount: products.length,
    itemBuilder: (ctx, i) => ChangeNotifierProvider.value(
        // builder: (c) => products[i],
        value: products[i],
        child: const BookItem(),
    ),
...
// the code is incomplete for brevity
```

As a result, the Books item child widget plays the crucial role to get the products id and pass it to the book detail screen.

Let' see how it has done that.

```
@override
Widget build(BuildContext context) {
    final product = Provider.of<Book>(context, listen: false);
    return ClipRRect(
    borderRadius: BorderRadius.circular(10),
    child: GridTile(
        child: GestureDetector(
        onTap: () {
            Navigator.of(context).pushNamed(
            BookDetailScreen.routeName,
            arguments: product.id,
            );
        },
        child: Image.network(
            product.imageUrl,
            fit: BoxFit.cover,
        ),
        ),
...
```

In the Books item widget on tap method, we use Navigator push named method, and as an argument we've passed the product ID.

How did we get the product ID?

Very simple.

```
final product = Provider.of<Book>(context, listen: false);
```

Since as a type Provider of method has used Book class, now we can get any book property anywhere in our flutter app.

How do you go from one class to another in flutter?

The Navigator push named method has helped us not only to navigate to another screen, but as an argument it has passed the Book class ID property.

As a result, now we can take any ID and search the dummy data to find other data that are associated with that ID.

Consequently, we can display the data on books detail page as we press any image on the overview screen.

Now, we can also take a look at the book detail screen. It must request that ID of the product that comes from the Book items widget.

```dart
import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

import '../models/books.dart';

class BookDetailScreen extends StatelessWidget {
static const routeName = '/product-detail';

const BookDetailScreen({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    final productId =
        ModalRoute.of(context)!.settings.arguments as String; // is the id!
    final loadedProduct = Provider.of<Books>(
    context,
    listen: false,
    ).findById(productId);
    return Scaffold(
    appBar: AppBar(
        title: Text(loadedProduct.title),
    ),
    body: SingleChildScrollView(
        child: Column(
        children: <Widget>[
            SizedBox(
            height: 300,
            width: double.infinity,
            child: Image.network(
                loadedProduct.imageUrl,
                fit: BoxFit.cover,
            ),
            ),
            const SizedBox(height: 10),
            Text(
            '\$${loadedProduct.price}',
            style: const TextStyle(
                color: Colors.grey,
                fontSize: 20,
            ),
            ),
            const SizedBox(
```

```
            height: 10,
            ),
            Container(
            padding: const EdgeInsets.symmetric(horizontal: 10),
            width: double.infinity,
            child: Text(
                loadedProduct.title,
                textAlign: TextAlign.center,
                softWrap: true,
                style: TextStyle(
                fontFamily: 'Lato',
                fontSize: 30.0,
                ),
            ),
            ),
            const SizedBox(
            height: 10,
            ),
            Container(
            padding: const EdgeInsets.symmetric(horizontal: 10),
            width: double.infinity,
            child: Text(
                loadedProduct.description,
                textAlign: TextAlign.center,
                softWrap: true,
                style: TextStyle(
                fontFamily: 'Anton',
                fontSize: 20.0,
                ),
            ),
            ),
        ],
        ),
    ),
    );
}
}
```

In the above code, these two lines have played crucial roles. In fact, due to these lines, we can display any book based on the product ID.

```
final productId =
        ModalRoute.of(context)!.settings.arguments as String; // is the id!
    final loadedProduct = Provider.of<Books>(
    context,
    listen: false,
    ).findById(productId);
```

How do you pass data with provider in Flutter?

To have a look at the screenshots we've used in this section, please visit this LINK[32]

When we want to pass data with Provider in Flutter we need to be careful at the very beginning. Firstly, we cannot equate it passing data between screens.

Why?

Because we can pass data between screens without the latest Provider package, ChangeNotifier-Provider, and ChangeNotifier.

Let's consider a simple data model where we have a Book class and a list of dummy data based on that Book class constructors.

We've already created a simple data model to achieve that, and have a respective GitHub Repository, where you can check the code snippet.

Let us see a glimpse of Book class first.

```
class Book {
final String id;
final String title;
final String description;
final double price;
final String imageUrl;
bool isFavorite;

Book({
    required this.id,
    required this.title,
    required this.description,
    required this.price,
    required this.imageUrl,
    this.isFavorite = false,
});
}
```

[32] https://sanjibsinha.com/pass-data-with-provider-flutter/

Now, based on that, we have some dummy data also so we can display the Book items on the books overview screen.

```
class BooksOverviewScreen extends StatelessWidget {
BooksOverviewScreen({Key? key}) : super(key: key);

final List<Book> books = [
    Book(
    id: 'p1',
    title: 'Beginning Flutter With Dart',
    description: 'You can learn Flutter as well Dart.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2014/09/05/18/32/old-books-436498_960_720.jpg\
',
    ),
    Book(
    id: 'p2',
    title: 'Flutter State Management',
    description: 'Everything you should know about Flutter State.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960_720.jpg',
    ),
    Book(
    id: 'p3',
    title: 'WordPress Coding',
    description:
        'WordPress coding is not difficult, in fact it is interesting.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2015/11/19/21/10/glasses-1052010_960_720.jpg',
    ),
    Book(
    id: 'p4',
    title: 'PHP 8 Standard Library',
    description: 'PHP 8 Standard Library has made developers life easier.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2015/09/05/21/51/reading-925589_960_720.jpg',
    ),
    Book(
    id: 'p5',
```

```
    title: 'Better Flutter',
    description:
        'Learn all the necessary concepts of building a Flutter App.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2015/09/05/07/28/writing-923882_960_720.jpg',
    ),
    Book(
    id: 'p6',
    title: 'Discrete Mathematical Data Structures and Algorithm',
    description:
        'Discrete mathematical concepts are necessary to learn Data Structures and A\
lgorithm.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2015/11/19/21/14/glasses-1052023_960_720.jpg',
    ),
];

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text('MyShop'),
    ),
    body: GridView.builder(
        padding: const EdgeInsets.all(10.0),
        itemCount: books.length,
        itemBuilder: (ctx, i) => BookItem(
        books[i].id,
        books[i].title,
        books[i].imageUrl,
        ),
        gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
        crossAxisCount: 2,
        childAspectRatio: 3 / 2,
        crossAxisSpacing: 10,
        mainAxisSpacing: 10,
        ),
    ),
    );
}
}
```

The above code is quite simple. Because we pass data between classes through class constructor. The following code snippet handles that passage of data.

```
itemBuilder: (ctx, i) => BookItem(
        books[i].id,
        books[i].title,
        books[i].imageUrl,
        ),
```

Now we can have a Book Item controller, where we can request that data through class constructor. In addition, we can display the title, image etc.

```
import 'package:flutter/material.dart';

class BookItem extends StatelessWidget {
final String id;
final String title;
final String imageUrl;

BookItem(this.id, this.title, this.imageUrl);

@override
Widget build(BuildContext context) {
    return ClipRRect(
    borderRadius: BorderRadius.circular(10),
    child: GridTile(
        child: GestureDetector(
        onTap: () {},
        child: Image.network(
            imageUrl,
            fit: BoxFit.cover,
        ),
        ),
        footer: GridTileBar(
        backgroundColor: Colors.black87,
        leading: IconButton(
            icon: const Icon(Icons.favorite),
            color: Theme.of(context).primaryColor,
            onPressed: () {},
        ),
        title: Text(
            title,
```

```
            textAlign: TextAlign.center,
        ),
        trailing: IconButton(
            icon: const Icon(
            Icons.shopping_cart,
            ),
            onPressed: () {},
            color: Theme.of(context).primaryColor,
        ),
        ),
    ),
    );
}
}
```

However, the above data model represents the whole book items. Not a single book.

How we can achieve that?

Here Provider package, and ChangeNotifierProvider come to our rescue.

If we add Provider package ChangeNotifierProvider with our book class, we can easily notify the listener widget in our widget tree.

What is provider architecture in Flutter?

The main slogan of provider architecture in Flutter is, provide the dependencies to another widget. As a result, now, we can think of a single book now.

We also need to understand another key concept. In the books overview screen, we've already instantiated the Book object.

Consequently, we want only the book, or to be very specific here, the product ID only.

Therefore, first thing we need to do is, add the dependencies of Provider package to our "pubspec.yaml" file first.

Next, we change our Book class to this:

```
import 'package:flutter/foundation.dart';

class Book with ChangeNotifier {
final String id;
final String title;
final String description;
final double price;
final String imageUrl;
bool isFavorite;

Book({
    required this.id,
    required this.title,
    required this.description,
    required this.price,
    required this.imageUrl,
    this.isFavorite = false,
});
}
```

Why should we do that?

Because we want to provide single book object to Book item widget like this:

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import '../views/book_detail_screen.dart';
import '../models/book.dart';

class BookItem extends StatelessWidget {
const BookItem({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    final product = Provider.of<Book>(context, listen: false);
    return ClipRRect(
    borderRadius: BorderRadius.circular(10),
    child: GridTile(
        child: GestureDetector(
        onTap: () {
            Navigator.of(context).pushNamed(
            BookDetailScreen.routeName,
```

```
            arguments: product.id,
            );
        },
        child: Image.network(
            product.imageUrl,
            fit: BoxFit.cover,
        ),
        ),
        footer: GridTileBar(
        backgroundColor: Colors.black87,
        leading: Consumer<Book>(
            builder: (ctx, product, _) => IconButton(
            icon: Icon(
                product.isFavorite ? Icons.favorite : Icons.favorite_border,
            ),
            color: Theme.of(context).primaryColor,
            onPressed: () {
                product.toggleFavoriteStatus();
            },
            ),
        ),
        title: Text(
            product.title,
            textAlign: TextAlign.center,
        ),
        trailing: IconButton(
            icon: const Icon(
            Icons.shopping_cart,
            ),
            onPressed: () {},
            color: Theme.of(context).primaryColor,
        ),
        ),
    ),
    );
}
}
```

In the above code, two lines are extremely important.

First, we have imported the Book data class to this widget. And, next, based on the mixin of ChangeNotifierProvider with Book class, now we can provide a single book, or product ID.

As we can see, we don't pass data through class constructor anymore.

On the contrary, we got it through this line,

```
final product = Provider.of<Book>(context, listen: false);
```

And in the Books overview screen we also have not sent all books data through Book Item constructor. Instead, we've used another widget Book Grid as the body.

```
body: BooksGrid(showFavs: _showOnlyFavorites),
```

After that, in that Book Grid widget, we return a GridView builder like the following.

```
final productsData = Provider.of<Books>(context);
final products = productsData.items;
return GridView.builder(
    padding: const EdgeInsets.all(10.0),
    itemCount: products.length,
    itemBuilder: (ctx, i) => ChangeNotifierProvider.value(
        value: products[i],
        child: const BookItem(),
    ),
```

Now as a ChangeNotifierProvider value we pass one product object from the instantiated books or products.

This part is little bit tricky, but based on that the Book Item widget now can provide one product object as an argument, while navigating to the books detail page.

```
child: GridTile(
        child: GestureDetector(
        onTap: () {
            Navigator.of(context).pushNamed(
            BookDetailScreen.routeName,
            arguments: product.id,
            );
        },
        child: Image.network(
            product.imageUrl,
            fit: BoxFit.cover,
        ),
        ),
```

Since, we've building this Book shopping cart together, let us keep in touch with the further progress. Until now, you'll get the full code snippet at this GitHub Repository.

What is provider pattern Flutter?

To have a look at the screenshots we've used in this section, please visit this LINK[33]

In Flutter when we discuss provider pattern to give examples of provider, we usually think of state management.

It is true that provider package with the help of ChangeNotifierProvider and ChangeNotifier manages state most efficiently.

But provider is actually a flutter architecture that provides the current data model to the place where we need it.

In this very short, but important article on Provider pattern flutter we'll quickly learn the concept. Let's have a look at how we can provide the current data model to anywhere in our widget tree.

We're going to design a shop app where we we'll have some products. But instead of passing data through class constructor, we'll rely on provider package with the help of ChangeNotifierProvider and ChangeNotifier.

Before we start, let me tell you where you'll find the source code. We've created a folder branch-one where we keep the main dart file for this example.

Keeping other files intact, you can run the main dart file and get the same result.

Firstly, we must have a data model at our local storage.

In our models folder we have two files Product and Products.

The Product class is the parent class based on which we have added a list of Products.

```
1  import 'package:flutter/foundation.dart';
2
3  class Product with ChangeNotifier {
4  final String id;
5  final String title;
6  final String description;
7  final double price;
8  final String imageUrl;
9
10 Product({
11     required this.id,
12     required this.title,
13     required this.description,
14     required this.price,
15     required this.imageUrl,
```

[33] https://sanjibsinha.com/provider-flutter-example/

```
});
}
```

The Products file is too long, so we cut it short for brevity. The list of products has actually instantiated the product object several times.

```
import 'package:flutter/material.dart';

import 'product.dart';

class Products with ChangeNotifier {
final List<Product> products = [
    Product(
    id: 'p1',
    title: 'Classic Watch',
    description: 'A Classic Watch accessorized with style.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2018/02/24/20/39/clock-3179167_960_720.jpg',
    ),
    Product(
    id: 'p1',
    title: 'Shoe with Gears',
    description: 'Shoes paired with excersise accessories.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2017/07/02/19/24/dumbbells-2465478_960_720.jp\
g',
    ),
...
```

The question is how we can provide this data model to our home screen. When should I use provider in Flutter?

We use provider for many purposes. However, at present we want to count how many products objects are there in our local storage.

Let's just count the length of the products.

To do that, we must place our ChangeNotifierProvider on the top of our main app.

```
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '../models/products.dart';

void main() {
runApp(
    MultiProvider(
    providers: [
        ChangeNotifierProvider(
        create: (_) => Products(),
        ),
    ],
    child: const ShopAppWithProvider(),
    ),
);
}
```

We're interested about Product objects that have already been instantiated. As a result ChangeNotifierProvider create named parameter returns Products.

Subsequently, we can retrieve all the products by two ways.

Let's see the first way.

```
class ShopAppWithProvider extends StatelessWidget {
const ShopAppWithProvider({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: ShoHomePage(),
    );
}
}

class ShoHomePage extends StatelessWidget {
const ShoHomePage({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Products App'),
    ),
    body: Center(
```

```
        child: Column(
        mainAxisSize: MainAxisSize.min,
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
            const Text('All products length:'),
            Text(
            '${context.watch<Products>().products.length}',
            key: const Key('productKeyOne'),
            style: Theme.of(context).textTheme.headline4,
            ),
        ],
        ),
    ),
    );
}
}
```

Now, directly context can watch the products length and give us a nice output like the following image.

However, we can count the products length by using Provider also.

What does provider do in Flutter?

As we've said earlier, provider does many things and one of them is to provide the current data model to any widget in the tree.

Therefore, this time we'll change the above code a little bit and will add provider.

```
class ShopAppWithProvider extends StatelessWidget {
const ShopAppWithProvider({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: ShoHomePage(),
    );
}
}

class ShoHomePage extends StatelessWidget {
const ShoHomePage({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
```

```
    final products = Provider.of<Products>(context).products;
    return Scaffold(
    appBar: AppBar(
        title: const Text('Products App'),
    ),
    body: Center(
        child: Column(
        mainAxisSize: MainAxisSize.min,
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
            const Text('All products length:'),
            Text(
            '${context.watch<Products>().products.length}',
            key: const Key('productKeyOne'),
            style: Theme.of(context).textTheme.headline4,
            ),
            const SizedBox(
            height: 10.0,
            ),
            Text(
            '${products.length}',
            key: const Key('productKeyTwo'),
            style: Theme.of(context).textTheme.headline4,
            ),
        ],
        ),
    ),
    );
}
}
```

In the above code, a few lines are worth noting. The first one is the usage of Provider.

```
final products = Provider.of<Products>(context).products;
```

Next, the provider provides the current data model products to the Text widget.

```
Text(
            '${products.length}',
            key: const Key('productKeyTwo'),
            style: Theme.of(context).textTheme.headline4,
            ),
```

Therefore, the next image shows two numbers of products length, instead of one.

And, finally, you may have noticed that each Text widget has different keys.

```
Text(
            '${context.watch<Products>().products.length}',
            key: const Key('productKeyOne'),
            style: Theme.of(context).textTheme.headline4,
            ),
            const SizedBox(
            height: 10.0,
            ),
            Text(
            '${products.length}',
            key: const Key('productKeyTwo'),
            style: Theme.of(context).textTheme.headline4,
            ),
```

Otherwise, it will throw an exception. Why, because it is final.

We'll learn more things about provider, so stay tuned as we'll build that shop app with provider step by step.

How do you use ChangeNotifierProvider in Flutter?

To have a look at the screenshots we've used in this section, please visit this LINK[34]

What is the best ChangeNotifierProvider example in Flutter? How should we use ChangeNotifier-Provider in Flutter?

What should be the correct approach to use ChangeNotifierProvider in Flutter?

Firstly, we cannot think of ChangeNotifierProvider without the provider package. And we know that provider pattern actually provides the current data model to the place where we need it.

In the previous article on provider data model, we've discussed the same topic. In the same vein, we have also seen that we can get the value of existing data quite easily using provider architecture.

[34] https://sanjibsinha.com/changenotifierprovider-flutter-example/

Moreover, it's always better than using class constructor.

In this article, we'll see how we can retrieve existing value of a shopping app using ChangeNotifier-Provider.

To do that, we must have a data model ready and instantiate the product objects first.

```
import 'package:flutter/foundation.dart';

class Product with ChangeNotifier {
final String id;
final String title;
final String description;
final double price;
final String imageUrl;

Product({
    required this.id,
    required this.title,
    required this.description,
    required this.price,
    required this.imageUrl,
});
}
```

Next, we need to instantiate the product objects in a separate class Products. So that ChangeNotifierProvider can use value method to get the existing value.

```
import 'package:flutter/material.dart';

import 'product.dart';

class Products with ChangeNotifier {
final List<Product> products = [
    Product(
    id: 'p1',
    title: 'Classic Watch',
    description: 'A Classic Watch accessorized with style.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2018/02/24/20/39/clock-3179167_960_720.jpg',
    ),
    Product(
    id: 'p1',
```

```
    title: 'Shoe with Gears',
    description: 'Shoes paired with exercise accessories.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2017/07/02/19/24/dumbbells-2465478_960_720.jp\
g',
    ),
...
// code is incomplete for brevity; please stay updated with this GitHub Repository
```

Now, as a rule, we'll keep the ChangeNotifierProvider create named parameter points to the Products class. So that later we can use that type.

```
void main() {
runApp(
    MultiProvider(
    providers: [
        ChangeNotifierProvider(
        create: (_) => Products(),
        ),
    ],
    child: const ShopAppWithProvider(),
    ),
);
}
```

We have used the default constructor because to create a value we should always use the default constructor. According to the documentation, we cannot create the instance inside build method.

It will lead to memory leaks and potentially undesired side-effects. Where is ChangeNotifierProvider used?

We use ChangeNotifierProvider here for showing existing data model. Therefore,the Shop App With Provider stateless widget will show the products overview screen.

Or, we may consider it as the home page.

Actually, we've already instantiated the product object. Therefore, we need the object individually. That means, if we can get the product ID, our job is done.

Based on that ID, we can display all the products in a scrollable Grid. Each product ID points to the respective image and title.

Now, with the help of ChangeNotifierProvider, we can use three ways to display them.

Let's see the first one's code snippet.

```
class ShopAppWithProvider extends StatelessWidget {
const ShopAppWithProvider({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: ShopHomePage(),
    );
}
}

class ShopHomePage extends StatelessWidget {
const ShopHomePage({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    final products = context.watch<Products>().products;
    return Scaffold(
    appBar: AppBar(
        title: const Text('Products App'),
    ),
    body: GridView.builder(
        padding: const EdgeInsets.all(10.0),
        itemCount: products.length,
        itemBuilder: (ctx, i) => ClipRRect(
        borderRadius: BorderRadius.circular(10),
        child: GridTile(
            child: GestureDetector(
            onTap: () {},
            child: Image.network(
                context.watch<Products>().products[i].imageUrl,
                fit: BoxFit.cover,
            ),
            ),
            footer: GridTileBar(
            backgroundColor: Colors.black87,
            title: Text(
                context.watch<Products>().products[i].title,
                textAlign: TextAlign.center,
            ),
            ),
        ),
        ),
        // ),
        gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
```

```
        crossAxisCount: 2,
        childAspectRatio: 3 / 2,
        crossAxisSpacing: 10,
        mainAxisSpacing: 10,
        ),
    ),
    );
}
}
```

In the above code, watch this line is extremely important.

```
final products = context.watch<Products>().products;
```

The next line, which plays a key role is the following one:

```
itemBuilder: (ctx, i) => ClipRRect(
...
// code is incomplete for brevity
```

In the above code we pass the context and the index.

Now depending on that logic, we can change the above home page code this way now:

```
class ShopHomePage extends StatelessWidget {
const ShopHomePage({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    final product = Provider.of<Products>(context).products;
    return Scaffold(
    appBar: AppBar(
        title: const Text('Products App'),
    ),
    body: GridView.builder(
        padding: const EdgeInsets.all(10.0),
        itemCount: product.length,
        itemBuilder: (ctx, i) => ClipRRect(
        borderRadius: BorderRadius.circular(10),
        child: GridTile(
            child: GestureDetector(
            onTap: () {},
            child: Image.network(
                product[i].imageUrl,
```

```
                fit: BoxFit.cover,
            ),
            ),
            footer: GridTileBar(
            backgroundColor: Colors.black87,
            title: Text(
                product[i].title,
                textAlign: TextAlign.center,
            ),
            ),
        ),
        ),
        // ),
        gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
        crossAxisCount: 2,
        childAspectRatio: 3 / 2,
        crossAxisSpacing: 10,
        mainAxisSpacing: 10,
        ),
    ),
    );
}
}
```

In the above code, the following code is important.

```
final product = Provider.of<Products>(context).products;
```

We have used Provider of context method where we've mentioned the type. And then get all the products.

Further, we can also use ChangeNotifierProvider value and change the above code.

```
class ShopHomePage extends StatelessWidget {
const ShopHomePage({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    final product = Provider.of<Products>(context).products;
    return Scaffold(
    appBar: AppBar(
        title: const Text('Products App'),
    ),
    body: GridView.builder(
```

```
        padding: const EdgeInsets.all(10.0),
        itemCount: product.length,
        itemBuilder: (ctx, i) => ChangeNotifierProvider.value(
        value: product[i],
        child: ClipRRect(
            borderRadius: BorderRadius.circular(10),
            child: GridTile(
            child: GestureDetector(
                onTap: () {},
                child: Image.network(
                product[i].imageUrl,
                fit: BoxFit.cover,
                ),
            ),
            footer: GridTileBar(
                backgroundColor: Colors.black87,
                title: Text(
                product[i].title,
                textAlign: TextAlign.center,
                ),
            ),
            ),
        ),
        ),
        gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
        crossAxisCount: 2,
        childAspectRatio: 3 / 2,
        crossAxisSpacing: 10,
        mainAxisSpacing: 10,
        ),
    ),
    );
}
}
```

All these three code snippet of products home page will display the products in the same way.

Furthermore, we can use the Gesture Detector on tap method to pass the product ID as an argument. As a result, we can display the product detail in a separate screen.

In the next article we will discuss that. So stay tuned to get updated on building a flutter shopping app.

What is ChangeNotifierProvider value?

To have a look at the screenshots we've used in this section, please visit this LINK[35]

The role of ChangeNotifierProvider value is not like builder or create. We need to understand this concept specifically.

Certainly, ChangeNotifierProvider value plays a crucial role. However, it has nothing to do with maintaining state.

In this article, we'll take a close look at the scope of the ChangeNotifierProvider value. How we can use it in our app.

Moreover, what is the difference between ChangeNotifierProvider value and ChangeNotifier-Provider create.

Firstly, ChangeNotifierProvider extends ChangeNotifier that flutter provides.

Secondly, ChangeNotifierProvider listens to ChangeNotifier. Not only that, after listening to ChangeNotifier, it exposes ChangeNotifier to its descendants.

The ChangeNotifierProvider also rebuilds dependents whenever ChangeNotifier notify its listeners.

Now, the million dollar question is what do we want to do?

Will we want to create a ChangeNotifier? Or, we will reuse the ChangeNotifier?

Certainly, we must create first, and after that we reuse ChangeNotifier.

Let us dig a little bit dipper into that concept. In addition, we must know that we have a GitHub repository exclusively for this article.

To understand how ChangeNotifierProvider value works, we will use part of code snippets for brevity. However, we can always check the full code at our GitHub repository.

We need to understand the app structure first. Subsequently, we'll check the code snippets so that we understand ChangeNotifierProvider value.

We have some products as our local storage. First we've created a class of Product and then we've instantiated some Product objects in a different class.

[35] https://sanjibsinha.com/changenotifierprovider-value/

```
import 'package:flutter/foundation.dart';

class Product with ChangeNotifier {
final String id;
final String title;
final String description;
final double price;
final String imageUrl;

Product({
    required this.id,
    required this.title,
    required this.description,
    required this.price,
    required this.imageUrl,
});
}
```

We don't have any method yet. Although we use dart mixin, to use ChangeNotifier.

Next, we instantiate a dew product object and keep both classes at models folder.

```
import 'package:flutter/material.dart';

import 'product.dart';

class Products with ChangeNotifier {
final List<Product> products = [
    Product(
    id: 'p1',
    title: 'Classic Watch',
    description: 'A Classic Watch accessorized with style.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2018/02/24/20/39/clock-3179167_960_720.jpg',
    ),
    Product(
    id: 'p1',
    title: 'Shoe with Gears',
    description: 'Shoes paired with excersise accessories.',
    price: 9.99,
    imageUrl:
        'https://cdn.pixabay.com/photo/2017/07/02/19/24/dumbbells-2465478_960_720.jp\
```

```
g',
    ),
...
```

As a result, our data model is now ready. Meanwhile, we can now use "ChangeNotifierProvider create" to create a ChangeNotifier and place it over the top of our shopping cart app.

```
void main() {
runApp(
    MultiProvider(
    providers: [
        ChangeNotifierProvider(
        create: (_) => Products(),
        ),
    ],
    child: const ShopAppWithProvider(),
    ),
);
}
...
```

We should not use ChangeNotifierProvider value to create a ChangeNotifier.

What is ChangeNotifierProvider Flutter?

Now the question is, why we should not use ChangeNotifierProvider value to create a ChangeNotifier in Flutter?

The first, and the foremost reason is it only references the state source, but does not manage its lifetime.

Since we have placed the ChangeNotifierProvider create on the top of our flutter app, now we can listen to ChangeNotifier and expose it to all the descendants.

Our shopping app widget tree starts with the ShopAppProvider() stateless widget.

```
class ShopAppWithProvider extends StatelessWidget {
const ShopAppWithProvider({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    return MaterialApp(
    theme: ThemeData(
        primarySwatch: Colors.brown,
        primaryColor: Colors.deepOrange,
    ),
    home: const ShopHomePage(),
    routes: {
        ShopProductDetailScreen.routename: (context) =>
            const ShopProductDetailScreen(),
    },
    );
}
}
```

Next in that widget tree comes ShopHomePage() stateless widget. This home page, or screen, whatever we wan to call it, will display the products object in a Grid View.

Therefore, we must have two separate controllers, which will not only display the products, but also, at the same time, navigate to the products detail page.

Now, in the ProductView() controller we can use ChangeNotifierProvider value to provide an existing ChangeNotifier.

```
class ProductView extends StatelessWidget {
const ProductView({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    final products = Provider.of<Products>(context).products;
    return GridView.builder(
    padding: const EdgeInsets.all(10.0),
    itemCount: products.length,
    itemBuilder: (ctx, i) => ChangeNotifierProvider.value(
        value: products[i],
        child: const ProductItem(),
    ),
    gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
        crossAxisCount: 2,
```

```
        childAspectRatio: 3 / 2,
        crossAxisSpacing: 10,
        mainAxisSpacing: 10,
    ),
    );
}
}
```

The above code snippet is self explanatory. We can use the context and index of all products and provide ChangeNotifier to the child widget ProductItem().

As a result, now we can easily access each product property through the instantiated product objects; as we've defined them in our Product class.

```
class ProductItem extends StatelessWidget {
const ProductItem({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    final product = Provider.of<Product>(context, listen: false);
    return ClipRRect(
    borderRadius: BorderRadius.circular(10),
    child: GridTile(
        child: GestureDetector(
        onTap: () {
            Navigator.of(context).pushNamed(
            ShopProductDetailScreen.routename,
            arguments: product.id,
            );
        },
        child: Image.network(
            product.imageUrl,
            fit: BoxFit.cover,
        ),
        ),
        footer: GridTileBar(
        backgroundColor: Colors.black87,
        title: Text(
            product.title,
            textAlign: TextAlign.center,
        ),
```

```
30          subtitle: Text(
31              product.description,
32              textAlign: TextAlign.center,
33          ),
34          ),
35      ),
36      );
37  }
38  }
```

Not only that, we can now pass the product ID as an argument of of the Navigator class.

Now we can click any image on the product home page, and get to the detail page.

In the next article we'll try to build the product detail page, so that it displays the product details, instead of showing a text.

What is navigator and route in Flutter?

To have a look at the screenshots we've used in this section, please visit this LINK[36]

In our previous article, we've seen how we can use ChangeNotifierProvider value. And with the help of ChangeNotifierProvider value we have used the concept of Provider and ChangeNotifier to read existing data from the local storage.

However, while doing this, we've found that we need to create another screen or page that will display the detail of a single product object.

But to do that, we need to navigate to another screen.

Right?

Firstly, we must know what route and navigator are in flutter.

What are their roles and how we can make them play their roles efficiently.

Secondly, which widgets we should use?

Finally, how we should use those widgets?

Well, let's answer one by one, so that we can solve this common problem of flutter that involves navigator and route. But before answering the questions,let me inform you that the full code snippets is available at this particular GitHub repository.

[36] https://sanjibsinha.com/navigator-and-routes-in-flutter/

What is route in flutter?

In flutter we call a page or screen as route. As we know, in flutter every page or screen is a widget. It could be either stateful or stateless widgets.

As a result we can write each route as a separate widget. However, in our case, it is not that easy.

Why?

Let's take a look at the first, or home page.

This is our home page where we display all products from a local storage, which is nothing but instantiated product objects.

Now we click any image, and it takes us to the product-detail page, like the following.

How it happens? Because, we have placed our ChangeNotifierProvider on top of the widget tree.

```
void main() {
runApp(
    MultiProvider(
    providers: [
        ChangeNotifierProvider(
        create: (_) => Products(),
        ),
    ],
    child: const ShopAppWithProvider(),
    ),
);
}
```

Next thing, we must do is, create the route.

That was our second question. Where should we use the route?

```
class ShopAppWithProvider extends StatelessWidget {
const ShopAppWithProvider({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    return MaterialApp(
    theme: ThemeData(
        primarySwatch: Colors.brown,
        primaryColor: Colors.deepOrange,
    ),
    home: const ShopHomePage(),
    routes: {
```

```
        ShopProductDetailScreen.routename: (context) =>
            const ShopProductDetailScreen(),
    },
    );
}
}
```

In the above code snippet, these two lines are extremely important. In those lines, we have defined the routes.

We should always define our routes in MaterialApp widget.

```
routes: {
        ShopProductDetailScreen.routename: (context) =>
            const ShopProductDetailScreen(),
    },
```

In our Shop Product Detail Screen we've declared a static constant property routename. Moreover, it's a string that we can find in the second image that displays the product detail screen.

```
class ShopProductDetailScreen extends StatelessWidget {
const ShopProductDetailScreen({Key? key}) : super(key: key);
static const routename = "/product-detail";

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Product Detail Page'),
    ),
    body: const Center(
        child: Text(
        'We will get product detail here',
        style: TextStyle(
            fontSize: 30.0,
        ),
        ),
    ),
    );
}
}
```

However, how could we reach at this page?

As we've defined the routes, subsequently we must have a method that should use Navigator widget.

Meanwhile, the Navigator widget uses pushnamed method so that we can pass the Shop Detail Screen routename along with the product ID.

That's the technique we have adopted in our Book Item controller, which is responsible for displaying all the products on the home page.

Therefore, we can use Gesture Detector on tap method, so that user can tap the image and reach the product detail screen, or page.

In that on tap method, we define that Navigator widget methods.

```
child: GridTile(
        child: GestureDetector(
        onTap: () {
            Navigator.of(context).pushNamed(
            ShopProductDetailScreen.routename,
            arguments: product.id,
            );
        },
        child: Image.network(
            product.imageUrl,
            fit: BoxFit.cover,
        ),
        ),
...
// code is incomplete for brevity
```

How do you pass arguments in Navigator pushNamed?

To have a look at the screenshots we've used in this section, please visit this LINK[37]

To pass arguments parameter in Navigator pushNamed method, we must have a named route.

Navigator widget is a widget that manages a set of child widgets. It also maintains a stack discipline.

In the last couple of flutter articles we have seen how to display a list of product items and, in addition how we can click any item to see the product detail. We also have a dedicated GitHub repository for that shopping cart app.

In that repository you'll get the full code snippets. However, in this article, we'll cut short our code for brevity.

To see the product detail we have navigated to a named route from the home page.

[37] https://sanjibsinha.com/navigator-pushnamed-arguments/

In our cases, we need to pass arguments to the named route, and navigated to the /product-details route.

The Products Home Page will show all Products.

We can see all product items at a glance here. Moreover, we can have title and subtitle that gives us an idea about the product.

However, to get the details of the product we need to navigate to another page. And for that we'll use Navigator widget pushNamed and pass the product ID as an argument.

To navigate to a new page we must define the routes first in our MaterialApp widget.

```
class ShopAppWithProvider extends StatelessWidget {
const ShopAppWithProvider({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) {
    return MaterialApp(
    theme: ThemeData(
        primarySwatch: Colors.brown,
        primaryColor: Colors.deepOrange,
    ),
    home: const ShopHomePage(),
    routes: {
        ShopProductDetailScreen.routename: (context) =>
            const ShopProductDetailScreen(),
    },
    );
}
}
```

Before that, we need to declare a static constant string as routename in the Shop Product Detail Screen widget.

```
class ShopProductDetailScreen extends StatelessWidget {
const ShopProductDetailScreen({Key? key}) : super(key: key);
static const routename = "/product-detail";

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Product Detail Page'),
    ),
    body: const Center(
```

```
        child: Text(
        'We will get product detail here',
        style: TextStyle(
            fontSize: 30.0,
        ),
        ),
    ),
    );
}
}
```

How do I send an argument in pushNamed Flutter?

However, the question is, how we can send the argument in Navigator pushNamed Flutter?

To understand that context in this perspective, let us first take a look at the product details page.

We've not designed this page until now. However, it will give us an idea about how we can use an argument and navigate to a named route.

Firstly, we get the products index using Provider and ChangeNotifierProvider value.

```
class ProductView extends StatelessWidget {
const ProductView({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    final products = Provider.of<Products>(context).products;
    return GridView.builder(
    padding: const EdgeInsets.all(10.0),
    itemCount: products.length,
    itemBuilder: (ctx, i) => ChangeNotifierProvider.value(
        value: products[i],
        child: const ProductItem(),
    ),
    gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
        crossAxisCount: 2,
        childAspectRatio: 3 / 2,
        crossAxisSpacing: 10,
        mainAxisSpacing: 10,
    ),
```

```
    );
}
}
```

As we can see that the code is quite straightforward. We've returned GridView builder and pass the necessary arguments to accomplish our task.

Further down the widget tree, we've used the arguments parameter of the Navigator.pushNamed method.

```
class ProductItem extends StatelessWidget {
const ProductItem({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    final product = Provider.of<Product>(context, listen: false);
    return ClipRRect(
    borderRadius: BorderRadius.circular(10),
    child: GridTile(
        child: GestureDetector(
        onTap: () {
            Navigator.of(context).pushNamed(
            ShopProductDetailScreen.routename,
            arguments: product.id,
            );
        },
        child: Image.network(
            product.imageUrl,
            fit: BoxFit.cover,
        ),
        ),
        footer: GridTileBar(
        backgroundColor: Colors.black87,
        title: Text(
            product.title,
            textAlign: TextAlign.center,
        ),
        subtitle: Text(
            product.description,
            textAlign: TextAlign.center,
        ),
```

```
        ),
    ),
    );
}
}
```

In the above code, we retrieve single instantiated product object and used Gesture Detector widget for one reason.

At the same time we can display the image on the home screen and with the help of the on tap method we can pass arguments in the Navigator.pushNamed method.

This part of the code snippet is quite self-explanatory.

```
child: GestureDetector(
        onTap: () {
            Navigator.of(context).pushNamed(
            ShopProductDetailScreen.routename,
            arguments: product.id,
            );
        },
        child: Image.network(
            product.imageUrl,
            fit: BoxFit.cover,
        ),
        ),
```

We're passing product.id as arguments parameter.

Our next task will be to get the product details with the help of that product.id. We'll discuss that topic in the next article.

You'll get all code snippets in these GitHub repositories[38]

You'll get all code snippets in these GitHub repositories[39]

You'll get all code snippets in these GitHub repositories[40]

[38] https://github.com/sanjibsinha/shop_app_with_provider
[39] https://github.com/sanjibsinha/shop_app_primary
[40] https://github.com/sanjibsinha/book_app_first

11. Google's Flutter 2.5 and Dart 2.14, What's New

To have a look at the screenshots we've used in this section, please visit this LINK[41]

Flutter 2.5 and Dart 2.14 have just arrived. What's new? What would be the coming changes that will really matter in the future?

Let's take a brief look at this topic today. Google has released Flutter 2.5 and Dart 2.14.

Due to these changes, the latest updates take place at the UI framework for multiple form factors. At the same time, it affects the app development programming language.

On Android devices, Flutter 2.5 makes some improvements for flutter apps. It happens when it runs in full screen mode. The customization options also changes.

We'll discuss all the changes later in a separate article. However, today, we'll learn how Material Banner works in new Flutter 2.5.

What is MaterialBanner?

The MaterialBanner is a Material Design banner. Just like any other banner it displays an urgent message.

In addition, it also provides actions for users to address the issue.

If the issue seems to be not-bothering at all, then the user can ignore and dismiss that message.

Now as usual, banners should be displayed at the top of the screen. Just below the app bar.

To make this happen, we can use the following code:

[41] https://sanjibsinha.com/flutter-2-5-changes/

```
import 'package:flutter/material.dart';
import 'package:flutter/widgets.dart';

void main() {
runApp(const MyApp());
}

class MyApp extends StatelessWidget {
const MyApp({Key? key}) : super(key: key);

// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(
        primarySwatch: Colors.blue,
    ),
    home: const MyHomePage(),
    );
}
}

class MyHomePage extends StatelessWidget {
const MyHomePage({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) => Scaffold(
        appBar: AppBar(
        title: const Text('MaterialBanner'),
        ),
        body: Center(
            child: Column(
        children: [
            Container(
            margin: const EdgeInsets.all(
                30.0,
            ),
            padding: const EdgeInsets.all(
                25.0,
            ),
            child: const Text(
                'Click the Button below to show MaterialBanner',
                style: TextStyle(
```

```
                fontSize: 40.0,
                fontWeight: FontWeight.bold,
                ),
            ),
            ),
            Container(
            margin: const EdgeInsets.all(
                30.0,
            ),
            padding: const EdgeInsets.all(
                25.0,
            ),
            child: ElevatedButton(
                child: const Text(
                'Click Me!',
                style: TextStyle(
                    fontSize: 25.0,
                    fontWeight: FontWeight.bold,
                ),
                ),
                onPressed: () =>
                    ScaffoldMessenger.of(context).showMaterialBanner(
                MaterialBanner(
                    content: const Text(
                    'Hello from Material Banner',
                    style: TextStyle(
                        fontSize: 20.0,
                        fontWeight: FontWeight.bold,
                    ),
                    ),
                    leading: const Icon(Icons.balcony_outlined),
                    backgroundColor: Colors.amberAccent,
                    actions: [
                    TextButton(
                        child: const Text('Dismiss'),
                        onPressed: () => ScaffoldMessenger.of(context)
                            .hideCurrentMaterialBanner(),
                    ),
                    ],
                ),
                ),
            ),
            ),
```

```
87          ],
88          )),
89      );
90  }
```

For flutter SDK release you can see the previous releases also.

How do you Map a dart list in Flutter 2.5?

To have a look at the screenshots we've used in this section, please visit this LINK[42]

To map a Dart list in Flutter, we need to know one thing first.

Flutter layout widgets like Column, Row, or Wrap always has children that returns a List. Not a Map.

As a result, in some circumstances, we need to map a dart list in flutter as per our requirements.

We have a lot of articles on Flutter List and Map, however, we need to understand the relationship between Dart List and Map in flutter.

Let us first see, how we can map a list in pure dart console programming.

Let us have a Student class and

```
1  class Student {
2  String name;
3  int age;
4
5  Student({
6      required this.name,
7      required this.age,
8  });
9  }
```

And, we can create a List of some Student objects in Dart.

[42] https://sanjibsinha.com/list-and-map-in-flutter/

```
import 'package:dart_common_problems/student.dart';

List<Student> students = [
Student(
    name: 'John',
    age: 10,
),
Student(
    name: 'Json',
    age: 11,
),
Student(
    name: 'Allen',
    age: 9,
),
Student(
    name: 'Maria',
    age: 8,
),
Student(
    name: 'Becky',
    age: 10,
),
];
```

Now, in our bin folder we can map this Dart list, quite easily.

```
import 'package:dart_common_problems/students.dart';

void main(List<String> arguments) {
var map1 = {for (var e in students) e.name: e.age};
print(map1);
var map2 = students.asMap().entries.map((e) => e.value);
print(map2);
var map3 = students.asMap().entries.map((e) => e.key);
print(map3);
var map4 = students.asMap().entries.map((e) => e.key);
print(map4.toList().length);
}
```

Let us first check the outputs of the lines first. Then we will discuss the code.

```
{John: 10, Json: 11, Allen: 9, Maria: 8, Becky: 10}

(Instance of 'Student', Instance of 'Student', Instance of 'Student',

Instance of 'Student', Instance of 'Student')

(0, 1, 2, 3, 4)

5
```

In the first line, we get each student's name and age as we loop through the List of instantiated students.

But in the second line, firstly, we map the Dart list with asMap() method. Secondly we go through each entries and this time we use map() method to get the key and value respectively.

Quite naturally, as value we've got five instances of Student class. And as key, we get the index of each list item.

However, we can convert that map into List again and count the length.

As expected, we've got the number 5.

Now, question is can we do the same thing in Flutter?

How do I convert a map to a List in flutter?

Yes, we can do the same thing in Flutter and get the length of the Students objects. Even we can pass that length as Navigator arguments and get an output just like before.

Subsequently, we can pass each index of the Student object also and navigate to another page and see the details.

We can do that either using Provider data model and ChangeNotifierProvider or we can use Widget class constructors.

However, this time we want to understand how we can map a dart list in flutter.

Therefore, we'll take the same example that we've just seen in the Dart console application. Consequently we'll try to apply that same example in our Flutter app.

For brevity we'll see the code in parts. Although you'll get the full code snippet in the respective GitHub repository.

Firstly, we'll create a home page where we define the route and navigation.

```
import 'package:flutter/material.dart';
import 'package:shop_app_with_provider/list-map-navigation/all_pages.dart';

class HomePage extends StatelessWidget {
const HomePage({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'List Map and Navigation',
    home: FirstPage(),
    routes: {
        AllPage.routename: (context) => const AllPage(),
    },
    );
}
}
```

Secondly, we define a Student class as we've defined in our Dart console application.

```
class Student {
String id;
String name;
int age;

Student({
    required this.id,
    required this.name,
    required this.age,
});
}
```

In the above code we've made a little change. We've added one more property ID to Student class. It'll help us to navigate and find all details of a particular student.

Now, we can instantiate ten Student objects inside the our first page stateless widget.

```
class FirstPage extends StatelessWidget {
FirstPage({Key? key}) : super(key: key);

/* final List _icons = [
    '1700',
    '1800',
    '1900',
]; */

final List<Student> students = [
    Student(
    id: 's1',
    name: 'John',
    age: 10,
    ),
    Student(
    id: 's2',
    name: 'Json',
    age: 11,
    ),
    Student(
    id: 's3',
    name: 'Allen',
    age: 9,
    ),
    Student(
    id: 's4',
    name: 'Maria',
    age: 8,
    ),
    Student(
    id: 's5',
    name: 'Becky',
    age: 10,
    ),
    Student(
    id: 's6',
    name: 'John',
    age: 10,
    ),
    Student(
    id: 's7',
    name: 'Json',
```

```
    age: 11,
    ),
    Student(
    id: 's8',
    name: 'Allen',
    age: 9,
    ),
    Student(
    id: 's9',
    name: 'Maria',
    age: 8,
    ),
    Student(
    id: 's10',
    name: 'Becky',
    age: 10,
    ),
];

...
```

Thirdly, we need a custom widget to get the length of the all Student list.

```
Widget _buildIcons(BuildContext context, int index) {
    final List x = students.asMap().entries.map((e) => e.key).toList();
    final int y = x.length;
    return GestureDetector(
    onTap: () {
        Navigator.of(context).pushNamed(
        AllPage.routename,
        arguments: y,
        );
    },
    child: Container(
        margin: const EdgeInsets.all(10.0),
        width: 80.0,
        height: 40.0,
        alignment: Alignment.center,
        decoration: const BoxDecoration(
        borderRadius: BorderRadius.all(Radius.circular(30.00)),
        boxShadow: [
            BoxShadow(
            color: Colors.red,
```

```
21              blurRadius: 4.00,
22              spreadRadius: 2.00,
23              ),
24          ],
25          gradient: LinearGradient(
26              begin: Alignment.centerLeft,
27              end: Alignment.centerRight,
28              colors: [
29              Colors.yellow,
30              Colors.white,
31              ],
32          ),
33          ),
34          child: Text(
35          '$index',
36          textAlign: TextAlign.center,
37          style: const TextStyle(
38              fontSize: 25.00,
39              fontFamily: 'Trajan Pro',
40              fontWeight: FontWeight.bold,
41          ),
42          ),
43      ),
44      );
45  }
46
47  ...
```

Our next challenge is to map a Dart list in Flutter.

Just like the Dart console application code, we've used the map() method to map the List of Students objects that have already been instantiated.

As a result, this part of the above code is quite self explanatory.

```
final List x = students.asMap().entries.map((e) => e.key).toList();
    final int y = x.length;
    return GestureDetector(
    onTap: () {
        Navigator.of(context).pushNamed(
        AllPage.routename,
        arguments: y,
        );
    },
```

Now, we're in a position to navigate to another page, here All Pages stateless widget, and pass the length of the List of instantiated Students objects there.

How do you Map a dart list?

Now we must use iterable map() method as we have seen in our Dart console application above.

```
var map4 = students.asMap().entries.map((e) => e.key);
print(map4.toList().length);
```

However, replicate the same code in Flutter and retrieve all the Student object individually in Flutter requires further operations.

To sum up, we know, that Flutter returns a list of widgets as children through some layout widgets like Row, Wrap, ListView or Column.

For example, in our custom first page widget, it has done the same.

```
@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('List, Map and Navigation'),
    ),
    body: Wrap(
        children: students
            .asMap()
            .entries
            .map((MapEntry map) => _buildIcons(context, map.key))
            .toList(),
    ),
    );
}
```

Now, if we compare the above code with the Dart console code, then we'll understand how Flutter uses map() method and passes MapEntry object and passes the map key to the _buildIcons() method.

However, we need to convert it to a List.

Why?

Because the Wrap children returns a List of widgets.

Now, we can run our app, and see the output where it displays the index or key of each instance of Student object.

As we have already used the GestureDetector onTap() method to navigate to another screen, we can see how many students are there in the class.

Our next challenge will be to get student details in this screen. So that we can display student's name individually on the home page, and if we tap each student name, we'll arrive in this screen to get the details.

How to start with an app template in Flutter 2.5

To have a look at the screenshots we've used in this section, please visit this LINK[43]

We've already seen that Flutter 2.5 has brought many changes for the mobile app developers and peers. However, in that article we've not discussed how we can start with an app template in Flutter 2.5.

Certainly, Flutter 2.5 includes performance enhancements, DevTool updates, new Material You support, and with those changes also comes another great feature.

In addition, we can start with an app template in Flutter 2.5.

In normal default way, we start with this command to create a flutter app:

flutter create new_app

But, to start with an app template we change this command in the following way:

```
1  flutter create -t skeleton new_flutter_template
```

To see the full code please visit the respective GitHub repository. [44]

As we have expected, this new app template has added many features that we can tweak quite easily. However, this article will give a gentle introduction. Not more than that.

The pubspec.yaml has a skeleton code. And it comes with many more features. If we take a look at the file, we'll understand how it differs from the default Flutter application pubspec.yaml file.

[43] https://sanjibsinha.com/flutter-2-5-template/
[44] https://github.com/sanjibsinha/new_flutter_template

```
name: new_flutter_template
description: A new Flutter project.

# Prevent accidental publishing to pub.dev.
publish_to: 'none'

version: 1.0.0+1

environment:
sdk: ">=2.12.0 <3.0.0"

dependencies:
flutter:
    sdk: flutter
flutter_localizations:
    sdk: flutter

dev_dependencies:
flutter_test:
    sdk: flutter

flutter_lints: ^1.0.0

flutter:
uses-material-design: true

# Enable generation of localized Strings from arb files.
generate: true

assets:
    # Add assets from the images directory to the application.
    - assets/images/
```

Now we can take a look when we run the app.

If we click any Sample item it uses the static routename we've seen before. And takes us to the detail page.

As a beginner you may find the folder structure difficult to follow.

Why?

Because, the item class and view pages come from the "/lib/src/sample_feature/" folder. Subsequently, three items have been passed through the class constructor.

First we have a simple sample item class:

```
class SampleItem {
const SampleItem(this.id);

final int id;
}
```

And after that we have passed three ids through class constructor as list items.

```
class SampleItemListView extends StatelessWidget {
const SampleItemListView({
    Key? key,
    this.items = const [SampleItem(1), SampleItem(2), SampleItem(3)],
}) : super(key: key);

static const routeName = '/';

final List<SampleItem> items;
...
// code is incomplete for brevity
```

Then we use the ListView.builder to get three items displayed on the home page.

What is the latest version of Flutter?

As we can see the latest version of Flutter as on 8th September, 2021, is Flutter 2.5 stable release.

And, we're discussing the features of the latest version. One of the great features is, certainly, this app template.

If we go through the detail page, there is nothing new.

```
import 'package:flutter/material.dart';

/// Displays detailed information about a SampleItem.
class SampleItemDetailsView extends StatelessWidget {
const SampleItemDetailsView({Key? key}) : super(key: key);

static const routeName = '/sample_item';

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
```

```
        title: const Text('Item Details'),
    ),
    body: const Center(
        child: Text('More Information Here'),
    ),
    );
}
}
```

There is a static constant routename, and that has been defined in the Material App page.

And we've set the route or navigation in the "app.dart" file.

```
onGenerateRoute: (RouteSettings routeSettings) {
            return MaterialPageRoute<void>(
            settings: routeSettings,
            builder: (BuildContext context) {
                switch (routeSettings.name) {
                case SettingsView.routeName:
                    return SettingsView(controller: settingsController);
                case SampleItemDetailsView.routeName:
                    return const SampleItemDetailsView();
                case SampleItemListView.routeName:
                default:
                    return const SampleItemListView();
                }
            },
            );
        },
...
// code is incomplete for brevity
```

There are a lot of other things that we should know about this app template in Flutter 2.5.

In the next article, we'll discuss settings controller and settings service. We'll also see how we can use localization feature with this app template in Flutter 2.5.

So stay tuned and keep reading about Flutter new features.

The latest version of Flutter comes with many new features

To have a look at the screenshots we've used in this section, please visit this LINK[45]

[45] https://sanjibsinha.com/latest-version-flutter/

The latest version of Flutter is 2.5 release. We have already seen how they work. Goggle's Flutter 2.5 and Dart 2.14, what's new and how to start with an app template in Flutter.

In this series of articles we examine what is new in the latest version of Flutter 2.5. The app template generates a skeleton of flutter app, where we have found a completely new feature – settings.

We'll see how it works in a minute. For full code snippet please visit the respective GitHub repository.

In the new folder structure, we find a new file:

```
    analysis_options.yaml
    # The following line activates a set of recommended lints for Flutter apps,
    # packages, and plugins designed to encourage good coding practices.
    include: package:flutter_lints/flutter.yaml
```

Besides, the pubspec.yaml file has added the assets from the images directory to the application.

```
assets:
    # Add assets from the images directory to the application.
    - assets/images/
```

There are many more other features. However, let's concentrate on settings only for this article.

In the main dart file the skeleton sets the settings controller that automatically adds an image on the top right section of the app bar.

```
import 'package:flutter/material.dart';

import 'src/app.dart';
import 'src/settings/settings_controller.dart';
import 'src/settings/settings_service.dart';

void main() async {

final settingsController = SettingsController(SettingsService());


await settingsController.loadSettings();


runApp(MyApp(settingsController: settingsController));
}
```

As we can see, from the very beginning settings controller plays a big role in the app building process. Which was absent in the previous version.

As a result we find the settings version on the app bar top right section.

If we click the settings icon what happens?

We can change this to the Dark theme.

Is flutter only for UI?

The answer is NO! Flutter is never intended to only design and build an awesome app that can simultaneously run on Android and iOS from a single codebase.

Flutter can manage state management most efficiently and with the help of Dart programming language, we can build any kind of app that takes help from the backend.

The latest version skeleton main dart file imports app dart file where we define the Material page route.

```
/// The Widget that configures your application.
class MyApp extends StatelessWidget {
const MyApp({
    Key? key,
    required this.settingsController,
}) : super(key: key);

final SettingsController settingsController;

@override
Widget build(BuildContext context) {

    return AnimatedBuilder(
    animation: settingsController,
    builder: (BuildContext context, Widget? child) {
        return MaterialApp(
        debugShowCheckedModeBanner: false,

        restorationScopeId: 'app',

        localizationsDelegates: const [
            AppLocalizations.delegate,
            GlobalMaterialLocalizations.delegate,
            GlobalWidgetsLocalizations.delegate,
            GlobalCupertinoLocalizations.delegate,
        ],
        supportedLocales: const [
            Locale('en', ''), // English, no country code
```

```
        ],


        onGenerateTitle: (BuildContext context) =>
            AppLocalizations.of(context)!.appTitle,

        theme: ThemeData(),
        darkTheme: ThemeData.dark(),
        themeMode: settingsController.themeMode,


        onGenerateRoute: (RouteSettings routeSettings) {
            return MaterialPageRoute<void>(
            settings: routeSettings,
            builder: (BuildContext context) {
                switch (routeSettings.name) {
                case SettingsView.routeName:
                    return SettingsView(controller: settingsController);
                case SampleItemDetailsView.routeName:
                    return const SampleItemDetailsView();
                case SampleItemListView.routeName:
                default:
                    return const SampleItemListView();
                }
            },
            );
        },
        );
    },
    );
}
}
```

The question is why we wrap the MaterialApp with the AnimatedBuilder widget?

The reason is, the AnimatedBuilder listens to the SettingsController for changes. Because of that, we can change the theme mode and provides a temporary state to that particular theme.

If we take a look at the SettingsController code snippet it will be clear in a minute.

```
class SettingsController with ChangeNotifier {
SettingsController(this._settingsService);

final SettingsService _settingsService;

late ThemeMode _themeMode;


ThemeMode get themeMode => _themeMode;


Future<void> loadSettings() async {
    _themeMode = await _settingsService.themeMode();

    notifyListeners();
}

Future<void> updateThemeMode(ThemeMode? newThemeMode) async {
    if (newThemeMode == null) return;

    if (newThemeMode == _themeMode) return;

    _themeMode = newThemeMode;

    notifyListeners();

    await _settingsService.updateThemeMode(newThemeMode);
}
}
```

Firstly, SettingsController is a class that many widgets can interact with. Moreover, widgets can listen to any changes. The SettingsController uses mixins to use ChangeNotifier functionalities to notify listeners.

Secondly, we find two methods load settings and update theme mode. The first one helps to load user's settings from settings service. And finally inform listeners that a change has occurred.

On the other hand, the update theme mode method updates and persists the theme mode based on the user's selection.

Finally, after informing listeners this method persists the change to a local database or internet. While doing so, it uses the SettingsService class.

Not only settings, but flutter latest version 2.5 that has been released on 8th September, 2021 brings in more features.

Localization is one of them. In the next article, we'll discuss that. So stay tuned.

How do you do localization in flutter 2.5?

To have a look at the screenshots we've used in this section, please visit this LINK[46]

The latest version Flutter 2.5 has added many features. But how do we do localization in the latest version of Flutter? All we know that Flutter 2.5 has made that process rather easy.

In this tutorial we will discuss that topic.

We have already discussed a few of them. In addition, in the previous post on the latest version of Flutter, we have seen that localization has been made easy with the Flutter 2.5 app template.

Firstly, we can get the full code snippet at our respective GitHub repository. We'll use the part of code here for brevity.

Secondly, Flutter 2.5 app template skeleton has added a localization folder, where we get a file "app_-en.arb" , like the following code snippet:

```
{
"appTitle": "new_flutter_template",
"@appTitle": {
    "description": "The title of the application"
}
}
```

It has defined the app title: "new_flutter_template". Similarly we can create another file "app_fr.arb". And change the app title so that it displays French instead of English.

```
{
    "appTitle": "new_flutter_template_en_Francais",
    "@appTitle": {
    "description": "Le titre de la application"
    }
}
```

However, our task is not finished with this only. We have to change this part in our "app.dart" file.

[46] https://sanjibsinha.com/localization-in-flutter/

```
supportedLocales: const [
            Locale('en', ''), // English, no country code
            Locale('fr', ''), // French, no country code
        ],

        // Use AppLocalizations to configure the correct application title
        // depending on the user's locale.
        //
        // The appTitle is defined in .arb files found in the localization
        // directory.
        onGenerateTitle: (BuildContext context) =>
            AppLocalizations.of(context)!.appTitle,
...
// code is incomplete for brevity
```

In the comments section of on generate title named parameter, we see that it is clearly mentioned that app title is defined in .arb files found in the localization.

How do you change the localization in flutter?

Before we want to change the localization in Flutter, we need to run the file and see the default home page first.

The above image has been generated from the sample_item_list_view.dart file in lib/src/sample_-feature folder.

As a result we need to change the App Bar title there like the following:

```
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text(AppLocalizations.of(context)!.appTitle),
...
```

However, it will throw error if we don't import this file on the top of the file.

```
import 'package:flutter_gen/gen_l10n/app_localizations.dart';
```

Now, we need to change the language settings from English to French. And, as a result, the home page app title will change as the following screenshot.

Please visit the website link, to get an idea.

The Flutter 2.5 latest version app template skeleton also shows us how to pass data through the class constructors.

In our next tutorial on Flutter 2.5 changes, we'll discuss that.

How do you pass data to a widget in flutter 2.5?

To have a look at the screenshots we've used in this section, please visit this LINK[47]

To pass data to a Flutter widget we can use various methods. However, today, we'll discuss the simplest method, so that a beginner can understand.

We'll pass data through class constructors. We've been already working on Flutter latest version 2.5. After having seen a couple of articles about the changes and features of Flutter 2.5, we'll try to pass data from one widget to the other using Flutter app template skeleton.

As we've seen if we run the app without tweaking the code, it shows a ListView items.

When we click any one of the item, it takes us to the detail page where actually we don't pass any data.

To pass data to the detail page we need to use Navigator arguments.

As a result, when we launch the fresh app, we see the detail page like the following image:

Now, we don't want to see such a blank page, where no data has been passed from the home page ListView items.

We want to pass data from the home page so that that data gets displayed on this detail page.

To do that, let's create a simple Student class. Where each student has an ID,and a name.

Now, in our app, when we click on the ID of any student, the name should be displayed on the Student details page.

How do we pass data between classes in flutter?

As we've said earlier, the simplest method to pass data between classes in flutter is class constructor.

In our "/lib/src/models/" folder let's have a Student class first.

```
1 class Student {
2 final String id;
3 final String name;
4 const Student(this.id, this.name);
5 }
```

Firstly, for brevity, we need to be succinct in displaying the code snippets here. Therefore, please visit the respective GitHub Repository for the full code, if you're interested.

Secondly, we need to instantiate a few Student objects before we pass the student details.

Without two or three students how we can show their details in the detail page?

[47] https://sanjibsinha.com/pass-data-to-a-widget-in-flutter/

```
class SampleItemListView extends StatelessWidget {
const SampleItemListView({
    Key? key,
    this.students = const [
    Student('s1', 'John'),
    Student('s2', 'Json'),
    Student('s3', 'Maria'),
    ],
}) : super(key: key);

static const routeName = '/';

final List<Student> students;
...
```

In this Sample Item ListView page, we'll show the Student ID only. But, we'll pass the name as Navigator argument.

We've already created three Students with their respective ID and name. As a result, it'll not be difficult to show all the IDs in a ListView. At the same time, we can pass the name as argument also.

```
body: ListView.builder(
        restorationId: 'sampleItemListView',
        itemCount: students.length,
        itemBuilder: (BuildContext context, int index) {
        final student = students[index];

        return ListTile(
            title: Text('Student ID: ${student.id}'),
            leading: const CircleAvatar(
                // Display the Flutter Logo image asset.
                foregroundImage: AssetImage('assets/images/flutter_logo.png'),
            ),
            onTap: () {
                Navigator.restorablePushNamed(
                context,
                SampleItemDetailsView.routeName,
                arguments: student.name,
                );
            });
        },

),
```

The ListView builder counts the students number, and show only the clickable IDs on the page.

The first important part is the following one:

```
itemCount: students.length,
        itemBuilder: (BuildContext context, int index) {
        final student = students[index];
```

Next, we show the student ID with an image that we get from the assets/images folder.

```
title: Text('Student ID: ${student.id}'),
            leading: const CircleAvatar(
                // Display the Flutter Logo image asset.
                foregroundImage: AssetImage('assets/images/flutter_logo.png'),
            ),
```

And, finally, we have the on Tap method, where we pass the name of the student as an argument.

```
Navigator.restorablePushNamed(
                context,
                SampleItemDetailsView.routeName,
                arguments: student.name,
                );
```

Consequently, when we run the app, we get the students IDs displayed on the home page.

How do we get data through class constructor?

To get data through class constructor, we need to have some other mechanism.

We must get the data by this line of code:

```
final String name = ModalRoute.of(context)!.settings.arguments as String;
```

Although we have got the argument as a simple String here, the functionality of ModalRoute arguments could be quite complex. We'll see that in our next article.

However, to make this process simple, we can just have the data as a string and show that on a Text widget.

```
class SampleItemDetailsView extends StatelessWidget {
const SampleItemDetailsView({Key? key}) : super(key: key);

static const routeName = '/student_details';

@override
Widget build(BuildContext context) {
    final String name = ModalRoute.of(context)!.settings.arguments as String;

    return Scaffold(
    appBar: AppBar(
        title: const Text('Student Details'),
    ),
    body: Center(
        child: Container(
        margin: const EdgeInsets.all(
            20.0,
        ),
        padding: const EdgeInsets.all(
            20.0,
        ),
        child: ListView(
            children: [
            Text(
                'Student Name: $name',
                style: const TextStyle(
                fontSize: 40.0,
                fontWeight: FontWeight.w600,
                ),
            ),
            ],
        ),
        ),
    ),
    );
}
}
```

As a result, instead of a blank page, we can have at least the respective student's name displayed on the details page.

In the next article, we'll see more complex examples examples, where we can use List and Map concepts so that we can pass more data.

However, to do that we need to tweak the above code.

Until then, stay tuned.

How to Pass and Receive data in Flutter 2.5

To have a look at the screenshots we've used in this section, please visit this LINK[48]

What is the simplest method by which we can pass and receive data in Flutter? Certainly, through the widget class constructors we can easily pass and receive data.

However, we also need to understand another concept named route, which involves another Flutter widget, such as Navigator.

In our previous article, we've seen how we can pass data to a widget. We've done that using Flutter 2.5 app template skeleton.

In addition to the previous post, we'd like to move forward, and pass more than one data as a string.

To do that, we need to understand two concepts. One is class constructors and another is named route.

Before we start, let me tell you where to get the full code snippets. We've made a separate GitHub repository to study the full code snippets.

Therefore, please visit that repository to get the full code, or you can download that repository in particular to study and run locally.

For brevity, here we'll see a few portion of code snippet, not all.

We'll move step by step, so that even a beginner in Flutter can understand what is happening.

Firstly, we need to take a look at the main dart file, which is the entry point of our small flutter app.

```
1  import 'package:flutter/material.dart';
2  import 'named_routes/src/settings/settings_controller.dart';
3  import 'named_routes/src/settings/settings_service.dart';
4  import 'named_routes/src/app.dart';
5
6
7  void main() async {
8  final settingsController = SettingsController(SettingsService());
9
10 await settingsController.loadSettings();
11
12 runApp(MyApp(settingsController: settingsController));
13 }
```

[48] https://sanjibsinha.com/pass-receive-data-in-flutter/

From the above code, it's clear that the name of our app is MyApp() and it has passed an argument.

Let's forget about that argument. That MyApp() widget is in app.dart file, which is in "named_-routes/src/" folder.

Let's take a look at the app.dart file. Especially where we've generated and mentioned the named route.

```
onGenerateRoute: (RouteSettings routeSettings) {
            return MaterialPageRoute<void>(
            settings: routeSettings,
            builder: (BuildContext context) {
                switch (routeSettings.name) {
                case SettingsView.routeName:
                    return SettingsView(controller: settingsController);
                case StudentDetailPage.routeName:
                    return const StudentDetailPage(
                    id: '',
                    name: '',
                    studentClass: '',
                    );
                case StudentHome.routeName:
                default:
                    return const StudentHome();
                }
            },
            );
...
// the code is incomplete for brevity, please consult GitHub repository
```

A very simple switch case logic allows us to understand that we have two widgets or pages in particular that will handle the process of passing data.

Now, the question is, from where that data will come? We keep our local data in models folder and they are respectively a Student class and some instantiated Student object.

How do you get data in Flutter 2.5?

In this case, we get data in Flutter from our models folder where we have kept our local storage.

Let us see the Student class first.

```
class Student {
final String id;
final String name;
final String studentClass;
const Student(
    this.id,
    this.name,
    this.studentClass,
);
}
```

Next, we'll instantiate that class and created a few Student objects that will act as the dummy data.

```
import 'student.dart';

const dummyStudents = [
Student(
    's1',
    'John',
    'IX',
),
Student(
    's2',
    'Json',
    'X',
),
Student(
    's3',
    'Mary',
    'XI',
),
];
```

There are three students who have their respective IDs, name and class.

Now, we must Map that List of Students in our Flutter widget, so that we can display all the students' name on the screen.

At the same time, we must take care so that we can click each Student name and reach another page, where we can display their name and class.

Here is the home page of Student.

```
import 'package:flutter/material.dart';
import '../models/dummy_students.dart';

class StudentHome extends StatelessWidget {
const StudentHome({Key? key}) : super(key: key);

static const routeName = '/';

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('All Students'),
    ),
    body: GridView(
        gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(
        maxCrossAxisExtent: 200,
        childAspectRatio: 3 / 2,
        crossAxisSpacing: 20,
        mainAxisSpacing: 20,
        ),
        children: dummyStudents.map((e) {
        return StudentLists(
            id: e.id,
            name: e.name,
            studentClass: e.studentClass,
        );
        }).toList(),
    ),
    );
}
}
```

As we can see that the list of Students use a method map() and return StudentLists widget where we have passed the data by class constructors.

Consequently, we can take a look at the StudentLists widget.

```
class StudentLists extends StatelessWidget {
final String id;
final String name;
final String studentClass;

const StudentLists({
    Key? key,
    required this.id,
    required this.name,
    required this.studentClass,
}) : super(key: key);

void selectStudent(BuildContext context) {
    Navigator.push(context, MaterialPageRoute(builder: (_) {
    return StudentDetailPage(id: id, name: name, studentClass: studentClass);
    }));
}

@override
Widget build(BuildContext context) {
    return InkWell(
    onTap: () => selectStudent(context),
    child: Container(
        alignment: Alignment.center,
        padding: const EdgeInsets.all(
        10.0,
        ),
        margin: const EdgeInsets.all(
        10.0,
        ),
        color: Colors.amberAccent,
        child: Text(
        name,
        style: const TextStyle(
            fontSize: 20.0,
            fontWeight: FontWeight.bold,
        ),
        ),
    ),
    );
}
}
```

We've received data through class constructors and again using the selectStudent() method, we pass the same data to the StudentDetailPage widget.

Subsequently, selectStudent() method plays the most important role in sending data. It also uses Navigator push method to pass a Material Page Route and again pass the data through class constructors.

How to receive data in Flutter 2.5?

Now we can click any student's name and see his or her detail in StudentDetailPage widget.

In the above code snippet this part is extremely important. Therefore, we must take a close look at the code snippet.

```
void selectStudent(BuildContext context) {
    Navigator.push(context, MaterialPageRoute(builder: (_) {
    return StudentDetailPage(id: id, name: name, studentClass: studentClass);
    }));
}
```

As we notice that Navigator pushes two arguments basically. One is context.

Why?

Because we're going to use this method inside build() method.

Secondly, the method passes another argument or parameter that basically builds a Material Page Route and returns another widget StudentDetailPage.

It's a stateless widget, moreover, it also passes the same id, name and student's class as class constructor parameters.

As a result, in that custom widget, we can receive data quite easily and display exactly what we need to display.

As we've clicked the first student name, John, we get his other details. If we click another student Mary, we'll also get her details the same way.

Certainly, we could have added many other properties in Student class, and displayed those details in the same way.

However, to understand the concepts, this is enough.

Finally, we'll take a look at the StudentDetailPage widget. We'll see how we have received data and displayed them.

```
class StudentDetailPage extends StatelessWidget {
final String id;
final String name;
final String studentClass;

const StudentDetailPage({
    Key? key,
    required this.id,
    required this.name,
    required this.studentClass,
}) : super(key: key);

static const routeName = '/student_details';

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text(name),
        backgroundColor: Colors.redAccent,
    ),
    body: Center(
        child: ListView(
        children: [
            Container(
            height: 100,
            width: 100,
            margin: const EdgeInsets.all(
                10.0,
            ),
            alignment: Alignment.center,
            color: Colors.amberAccent,
            child: Text(
                'Student\'s name: $name',
                style: const TextStyle(
                fontSize: 30,
                fontWeight: FontWeight.bold,
                ),
            ),
            ),
            const SizedBox(
            height: 5.0,
            ),
```

```
44              Container(
45              height: 100,
46              width: 100,
47              margin: const EdgeInsets.all(
48                  10.0,
49              ),
50              alignment: Alignment.center,
51              color: Colors.redAccent,
52              child: Text(
53                  '$name studies in class $studentClass',
54                  style: const TextStyle(
55                  fontSize: 30,
56                  fontWeight: FontWeight.bold,
57                  ),
58              ),
59              ),
60          ],
61          ),
62      ),
63      );
64  }
65  }
```

Since the Navigator pushes the same data from the same resources, now we can receive them through class constructors.

After that, we can display them not in the AppBar title, but also in the body.

We hope, the whole concept of passing data through class constructors and named route is now clear.

But, we can handle the same data in a more efficient way, with the help of Provider data model.

In the next Chapter we'll discuss that. So, stay tuned.

12. Understanding Material Design in Flutter

Most of the times we need to wrap hundreds of widgets with MaterialApp. But Why? Moreover, we need to use Scaffold widget to use its properties so that we can create a visually good-looking Flutter App.

In this chapter, we'll try to understand the material design principles that are associated with every MaterialApp widget.

Every widgets that are child to MaterialApp create the Flutter's visual layout structure. We try to explain these concepts so a beginner can understand.

Moreover, each section of this chapter gives a link to an article, where you can view the screenshots, whcih are associated with that section.

What is Material Design in flutter?

To view all the related images please visit the following link

- All related images with this section - https://sanjibsinha.com/material-design-in-flutter/[49]

Material design in Flutter refers to an adaptable design system. However, if we don't explain this statement a little further, it really doesn't make any sense if you're a beginner.

To sum up, with the help of Material design system we create Widgets in Flutter.

What are Widgets?

It says that in Flutter, everything is Widgets.

Firstly, widgets are classes written in Dart programing language. As a result,widgets follow Dart coding guidelines. However, we don't have to worry about this now.

With the help of Material design system we build myriad of widgets. At some point we may feel the widgets are countless.

As there are lots of different types of Widgets that Flutter come with. Besides, we can make our custom widgets.

Some of the default widgets are visually stunning. We use them to give users visually rich experience.

[49] https://sanjibsinha.com/material-design-in-flutter/

Secondly, widgets may have behavioral patterns, such as a widget may maintain its state. It can inherits from its parent, or maintains it itself, or passes state to its child. When state changes, certainly the behavior changes.

Finally, we need widgets that are motion rich. That means it may use animation.

However, all widgets don't have Material components widgets, like MaterialApp widget as its ancestor. We'll discuss that later and see which widgets don't care about Material components widgets.

But, at present, we'll show you how material design widgets affect a flutter app.

If we don't understand that concept, as a beginner we'll stumble at the very beginning.

Let's run the following code. All we want is to display a TextButton widget on the screen,

```
void main() {
runApp(
    Center(
    child: TextButton(
        style: TextButton.styleFrom(
        textStyle: const TextStyle(fontSize: 20),
        ),
        onPressed: () {},
        child: const Text('Enabled'),
    ),
    ),
);
}
```

When we ran the app, it should have displayed the text button, which is a widget, on the screen.

However, that didn't happen. It gave us a long message that basically was an error message.

We can read the same message on our console.

What does that say?

```
An Observatory debugger and profiler on Chrome is available at: http://127.0.0.1:400\
89/ituDJNtr1rE=
══╡ EXCEPTION CAUGHT BY WIDGETS LIBRARY ╞═══════════════════════════════════════\
═══════════
The following assertion was thrown building _InkResponseStateWidget(gestures: , mous\
eCursor:
ButtonStyleButton_MouseCursor, clipped to BoxShape.rectangle, dirty, state:
_InkResponseState#a0527):
No Directionality widget found.
```

```
_InkResponseStateWidget widgets require a Directionality widget ancestor.
The specific widget that could not find a Directionality ancestor was:
_InkResponseStateWidget
The ownership chain for the affected widget is: "_InkResponseStateWidget ◻ InkWell ◻
DefaultTextStyle ◻ AnimatedDefaultTextStyle ◻ _InkFeatures-[GlobalKey#679ad ink rend\
erer] ◻
NotificationListener ◻ CustomPaint ◻ _ShapeBorderPaint ◻ PhysicalShape
◻ _MaterialInterior ◻ ◻"
Typically, the Directionality widget is introduced by the MaterialApp or WidgetsApp \
widget at the
top of your application widget tree. It determines the ambient reading direction and\
 is used, for
example, to determine how to lay out text, how to interpret "start" and "end" values\
, and to resolve
EdgeInsetsDirectional, AlignmentDirectional, and other *Directional objects.
The relevant error-causing widget was:
TextButton TextButton:file:///home/ss/Documents/development/flutter_artisan/lib/main\
.dart:5:11
```

Quite a big error message, a beginner may stumble to decipher the meaning.

However, a seasoned flutter developer can take a look and know the summary. To sum up, the long error message says, while running the app you didn't use Material components widgets as an ancestor of the TextButton widget.

It means we should have wrapped the TextButton with a MaterialApp widget. What is MaterialApp in Flutter?

For any material design related widgets, this convenience widget, MaterialApp plays a key role. MaterialApp in Flutter wraps a number of widgets that we need to design any flutter app.

There are lots of material-design specific functionalities that we need while building a flutter app.

If you compare convenience, MaterialApp is more convenient than WidgetsApp. Besides adding design-specific functionalities it adds styling, color, controlling sizes and many more.

As per as convenience is concerned, after MaterialApp, we need Scaffold widget that implements material design specific layout.

Therefore, we need to wrap the TextButton widget with MaterialApp and Scaffold widgets.

```
import 'package:flutter/material.dart';
import 'package:flutter/widgets.dart';

void main() {
runApp(
    MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: const Text('Material Design'),
        ),
        body: Center(
        child: Column(
            children: [
            TextButton(
                style: TextButton.styleFrom(
                textStyle: const TextStyle(
                    fontSize: 20,
                ),
                ),
                onPressed: null,
                child: const Text(
                'Text Button Disabled',
                ),
            ),
            const SizedBox(height: 30),
            TextButton(
                style: TextButton.styleFrom(
                textStyle: const TextStyle(
                    fontSize: 20,
                ),
                ),
                onPressed: () {},
                child: const Text(
                'Text Button Enabled',
                ),
            ),
            ],
        ),
        ),
    ),
    ),
);
}
```

Now, as a result, we can run the flutter app safely.

Why the first TextButton looks different than the second one?

That's a completely different issue. We'll discuss it later.

AppBar Flutter: How Do I use AppBar?

To view all the related images please visit the following link

- All related images with this section - https://sanjibsinha.com/appbar-flutter/[50]

On Android toolbars represent AppBar. Certainly, it's another Material design widget that we've discussed earlier.

To use AppBar we firstly need another widget – Scaffold.

As we've just learned, AppBar consists of many toolbars.

Therefore, we can use our AppBar in many ways. We can navigate to another page, we can use search icon to search our app, and many more.

We'll see that in a minute.

Before that, to see more Material design widgets in one place, let's organize our code. Let's create a folder "controllers" and "views".

In "controllers" we create our AppBar widget. In addition, we can visit the full code snippet in this GitHub repository.

```
import 'package:flutter/material.dart';
import 'package:flutter_artisan/views/app_bar_home.dart';

class AppBarWidget extends StatelessWidget {
const AppBarWidget({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const AppBarHome();
}
}
```

And inside we have an AppBar home page, which will display all the AppBar tools.

[50] https://sanjibsinha.com/appbar-flutter/

```dart
import 'package:flutter/material.dart';
import 'package:flutter_artisan/views/app_bar_next.dart';

class AppBarHome extends StatelessWidget {
const AppBarHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('AppBar Example'),
        actions: <Widget>[
        IconButton(
            icon: const Icon(Icons.add_alert),
            tooltip: 'Show Snackbar',
            onPressed: () {
            ScaffoldMessenger.of(context).showSnackBar(
                const SnackBar(
                content: Text('A SnackBar'),
                ),
            );
            },
        ),
        IconButton(
            icon: const Icon(Icons.search_outlined),
            tooltip: 'Search',
            onPressed: () {
            // our code
            },
        ),
        IconButton(
            icon: const Icon(Icons.navigate_next),
            tooltip: 'Next page',
            onPressed: () {
            Navigator.push(
                context,
                MaterialPageRoute<void>(
                builder: (BuildContext context) {
                    return const AppBarNext();
                },
                ),
            );
            },
```

```
        ),
        ],
    ),
    body: const Center(
        child: Text(
        'This is the AppBar Example home page',
        style: TextStyle(fontSize: 30),
        textAlign: TextAlign.center,
        ),
    ),
    );
}
}
```

Before explaining the code, let's take a look at how our AppBar home page looks like.

As we've learned earlier, an app bar, resting on the top of our flutter app, consists of a toolbar and potentially other widgets.

That may include TabBar and a FlexibleSpaceBar, which we'll discuss later.

The most important role of AppBar is it expose one or more common actions with IconButtons.

In our case, we can see one bell icon, search icon and navigate icon.

They all have one common named parameter – onTap method. With the help of that method, we can even change the state of the flutter app.

That's why, the nearest ancestor of AppBar widget is StatefulWidget.

Flutter places AppBar as a fixed height widget at the top of the screen. It happens because App bars are typically used in the Scaffold.appBar property.

We can even add scrollable AppBar. However, that's a different topic. For a scrollable app bar, we use SliverAppBar, which embeds an AppBar in a sliver for use in a CustomScrollView.

Usually, any AppBar displays the toolbar widgets, such as leading, title, and actions.

In the above code, we've used actions to display our icons.

```
appBar: AppBar(
        title: const Text('AppBar Example'),
        actions: <Widget>[
        IconButton(
            icon: const Icon(Icons.add_alert),
            tooltip: 'Show Snackbar',
            onPressed: () {
            ScaffoldMessenger.of(context).showSnackBar(
                const SnackBar(
                content: Text('A SnackBar'),
                ),
            );
            },
        ),
...
```

Where do I put AppBar in flutter?

The advantage we enjoy in Flutter is that it allows us to create menu, search,or leading button in app bar.

With a single AppBar property, we can create several functionalities in our Flutter app.

If we click the bell icon button, it opens up the snack bar.

```
IconButton(
            icon: const Icon(Icons.add_alert),
            tooltip: 'Show Snackbar',
            onPressed: () {
            ScaffoldMessenger.of(context).showSnackBar(
                const SnackBar(
                content: Text('A SnackBar'),
                ),
            );
            },
        ),
...
```

In a separate article we'll discuss the role of SnackBar widget.

As a result we can see that snack bar pops up at the bottom of our app.

Inside our SnackBar, we can place many more widgets.

Subsequently, we can navigate to another page from AppBar.

```
IconButton(
            icon: const Icon(Icons.navigate_next),
            tooltip: 'Next page',
            onPressed: () {
            Navigator.push(
                context,
                MaterialPageRoute<void>(
                builder: (BuildContext context) {
                    return const AppBarNext();
                },
                ),
            );
            },
        ),
...
```

We have placed the custom stateless AppBarNext widget inside our views folder. Let's take a look at the code first.

```
import 'package:flutter/material.dart';
import 'package:flutter/widgets.dart';

class AppBarNext extends StatelessWidget {
const AppBarNext({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('App Bar Next Page'),
    ),
    body: const AppBarNextPage(),
    );
}
}

class AppBarNextPage extends StatelessWidget {
const AppBarNextPage({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
```

```
25          margin: const EdgeInsets.all(
26          10,
27          ),
28          padding: const EdgeInsets.all(
29          10,
30          ),
31          child: const Text(
32          'AppBar Next Page',
33          style: TextStyle(
34              fontWeight: FontWeight.bold,
35              fontSize: 30,
36          ),
37          ),
38      ),
39      );
40  }
41  }
```

Most importantly, we can use another AppBar, because it's another page. And, of course, we can add other functionalities in this AppBar.

As a result, for this page we've used another AppBar.

How do I style AppBar flutter?

Remember, every page may have different AppBar in Flutter. Therefore, to style AppBar we can use different approaches.

In the above code snippets we've restricted ourselves to certain features only.

But any AppBar comes with many functionalities that include background color, text font, leading, and many more.

We can change the alignment or position of the title in our AppBar.

In the above code snippet, we've kept it on the left side. However, if we feel, we change its position and move it either to the center or right.

In actions widget, we can add more icons with different functionalities.

That includes a pop up menu.

It depends on what type of Flutter app we're going to build.

In this Material design series we'll discuss more useful widgets, so please stay tuned.

How do I use BottomNavigationBar in flutter?

To view all the related images please visit the following link

- All related images with this section - https://sanjibsinha.com/bottomnavigationbar-flutter/[51]

We can use BottomNavigationBar within a Scaffold in Flutter. We can use bottom navigation bar at the bottom of an app.

Why do we use this particular material design widget?

When we want to select a small number of views, typically between three and five, we use BottomNavigationBar.

Now, the name suggests that BottomNavigationBar is a widget that helps us to display multiple views. BottomNavigationBar helps us to navigate to another page or screen.

As we've just said, the number of screens or pages may vary.

For this article, we'll keep it within three.

As a result, it looks like this:

At the bottom of the screen, the BottomNavigationBar shows up and it opens up with the first page.

However, there are lots of things we need to learn about this particular widget that follows Material design.

In this article, we'll understand those key concepts and besides we'll learn how to design this BottomNavigationBar. How do you fix a bottom navigation bar in flutter?

Firstly, we must remember that BottomNavigationBar is a Material design widget.

Flutter provides a number of widgets that help us build apps that follow Material Design.

What is Material Design?

A Material app starts with the MaterialApp widget. Secondly, it builds a number of useful widgets at the root of our app.

Certainly, that includes a Navigator widget.

Due to the Navigator widget, we can manage a stack of widgets.

Although this is not today's topic, still as a footnote we must add that the Navigator lets us transition smoothly between screens. Using the MaterialApp widget is a good practice.

Why?

[51] https://sanjibsinha.com/bottomnavigationbar-flutter/

Because when we build a flutter app, we may have forgotten which widget follows the Material design widgets as its ancestor.

As always, BottomNavigationBar is a Material design widget, so we must use MaterialApp widget at the top and keep BottomNavigationBar at the bottom.

Let us see the full code first. After that we'll take a close look at the different features, step by step.

```
import 'package:flutter/material.dart';

class BottomNavigationBarTest extends StatelessWidget {
const BottomNavigationBarTest({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: BottomNavigationHome(),
    );
}
}

class BottomNavigationHome extends StatefulWidget {
const BottomNavigationHome({Key? key}) : super(key: key);

@override
_BottomNavigationHomeState createState() => _BottomNavigationHomeState();
}

class _BottomNavigationHomeState extends State<BottomNavigationHome> {
int _selectedIndex = 0;
static const List<Widget> _widgetOptions = <Widget>[
    Text(
    'First Page',
    style: TextStyle(
        fontSize: 35,
        fontWeight: FontWeight.bold,
    ),
    ),
    Text(
    'Second Page',
    style: TextStyle(
        fontSize: 35,
        fontWeight: FontWeight.bold,
    ),
```

```
    ),
    Text(
    'Third Page',
    style: TextStyle(
        fontSize: 35,
        fontWeight: FontWeight.bold,
    ),
    ),
];

void _onItemTapped(int index) {
    setState(() {
    _selectedIndex = index;
    });
}

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('BottomNavigationBar Example'),
        backgroundColor: Colors.green,
    ),
    body: Center(
        child: _widgetOptions.elementAt(_selectedIndex),
    ),
    bottomNavigationBar: BottomNavigationBar(
        /// customizing background color
        ///
        backgroundColor: Colors.amber,
        mouseCursor: SystemMouseCursors.grab,
        unselectedItemColor: Colors.deepOrangeAccent,
        showSelectedLabels: false,
        showUnselectedLabels: false,
        type: BottomNavigationBarType.shifting,
        selectedFontSize: 20,
        selectedIconTheme: const IconThemeData(
        color: Colors.amberAccent,
        ),
        selectedItemColor: Colors.amberAccent,
        selectedLabelStyle: const TextStyle(
        fontWeight: FontWeight.bold,
        ),
```

```
        items: const <BottomNavigationBarItem>[
        BottomNavigationBarItem(
            icon: Icon(Icons.home),
            label: 'First Page',
            backgroundColor: Colors.black38,
        ),
        BottomNavigationBarItem(
            icon: Icon(Icons.search),
            label: 'Second Page',
            backgroundColor: Colors.black26,
        ),
        BottomNavigationBarItem(
            icon: Icon(Icons.person),
            label: 'Third Page',
            backgroundColor: Colors.black45,
        ),
        ],
        currentIndex: _selectedIndex,
        iconSize: 50,
        onTap: _onItemTapped,
        elevation: 5,
    ),
    );
}
}
```

To start with we've started with MaterialApp, keeping it at the top of our app.

```
@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: BottomNavigationHome(),
    );
}
...
```

Next, we have started designing the bottom navigation bar.

As we've said earlier, the bottom navigation bar consists of multiple items.

It could be in the form of text labels, icons, or both. In addition, it is laid out on top of a piece of material.

As a rule of thumb, a bottom navigation bar is usually used in conjunction with a Scaffold.

And when we define Scaffold, BottomNavigationBar is provided as the Scaffold.bottomNavigationBar argument.

Just take a look at the code:

```
return Scaffold(
    appBar: AppBar(
        title: const Text('BottomNavigationBar Example'),
        backgroundColor: Colors.green,
    ),
    body: Center(
        child: _widgetOptions.elementAt(_selectedIndex),
    ),
    bottomNavigationBar: BottomNavigationBar(
        /// customizing background color
        ///
        backgroundColor: Colors.amber,
...
```

In the typical Scaffold.bottomNavigationBar argument we define all other properties of BottomNavigationBar.

```
/// customizing background color
        ///
        backgroundColor: Colors.amber,
        mouseCursor: SystemMouseCursors.grab,
        unselectedItemColor: Colors.deepOrangeAccent,
        showSelectedLabels: false,
        showUnselectedLabels: false,
        type: BottomNavigationBarType.shifting,
        selectedFontSize: 20,
        selectedIconTheme: const IconThemeData(
        color: Colors.amberAccent,
        ),
        selectedItemColor: Colors.amberAccent,
        selectedLabelStyle: const TextStyle(
        fontWeight: FontWeight.bold,
        ),
...
        currentIndex: _selectedIndex,
        iconSize: 50,
        onTap: _onItemTapped,
        elevation: 5,
...
```

Now, we can see that each BottomNavigationBar arguments, such as background color, mouse cursor, elevation, and many more define how it will look like.

We must remember that the length of items must be at least two and each item's icon and title/label must not be null.

How do I customize the bottom navigation bar flutter?

And that's how we can customize the bottom navigation bar in flutter.

As a result, we define the number of items.

```
static const List<Widget> _widgetOptions = <Widget>[
    Text(
    'First Page',
    style: TextStyle(
        fontSize: 35,
        fontWeight: FontWeight.bold,
    ),
    ),
    Text(
    'Second Page',
    style: TextStyle(
        fontSize: 35,
        fontWeight: FontWeight.bold,
    ),
    ),
    Text(
    'Third Page',
    style: TextStyle(
        fontSize: 35,
        fontWeight: FontWeight.bold,
    ),
    ),
];
```

In the above code, we find that there are four items. When there are less than four items, the BottomNavigationBarType is set to BottomNavigationBarType.fixed.

Otherwise, bottom navigation bar's type changes with the help of BottomNavigationBarType.shifting.

Take a look at the bottom navigation bar's type argument.

```
type: BottomNavigationBarType.shifting,
```

Now, as a result, we can click bottom navigation bar's icons and see different pages.

As a result, when we click any item, that is highlighted with a different color. Moreover, it has got an elevation.

Subsequently, we can move to the third page.

What is a drawer in flutter?

To view all the related images please visit the following link

- All related images with this section - https://sanjibsinha.com/what-is-drawer-flutter/[52]

Drawer in Flutter is a widget that helps us to navigate to other links. We can imagine it as a sub-router that consists of various links to other routes.

So with the help of Drawer widget we can navigate to other useful screens or pages in Flutter.

Drawer has a horizontal movement from the edge of the Scaffold. It actually helps to navigate the link to different routes in the flutter app.

How does the Drawer widget work?

Drawer plays a key role in Flutter User Interface or UI Material Design implementation. When our flutter use Material design, we use to navigate with two primary widgets.

One is tab and the other is Drawer.

When we don't have sufficient space, we avoid tabs. We use Drawer.

In Flutter, we use the Drawer widget in combination with a Scaffold.

We need to use Scaffold to create a layout with a Material Design drawer and then populate the Drawer with items.

After that we close the Drawer pro grammatically.

The First step is, we wrap the Drawer with Scaffold widget. Let us consider a home page that starts with MaterialApp widget and we define two extra route.

We define those so that we can navigate to these pages with the help of our Drawer widget.

[52] https://sanjibsinha.com/what-is-drawer-flutter/

```
import 'package:flutter/material.dart';
import '/drawer_example/drawer_about.dart';
import '/drawer_example/drawer_contact.dart';

class DrawerExample extends StatelessWidget {
const DrawerExample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Drawer Example',
    home: DrawerHome(),
    routes: {
        DrawerAbout.routename: (context) => const DrawerAbout(),
        DrawerContact.routename: (context) => const DrawerContact(),
    },
    );
}
}

class DrawerHome extends StatelessWidget {
const DrawerHome({Key? key}) : super(key: key);
static const routename = "/";

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(title: const Text('')),
    body: Center(
        child: Container(
        margin: const EdgeInsets.all(20),
        padding: const EdgeInsets.all(20),
        child: Text(
            'A Drawer Example.',
            style: TextStyle(fontSize: 50.0),
        ),
        ),
    ),
    drawer: Drawer(
        child: ListView(
        // Important: Remove any padding from the ListView.
        padding: EdgeInsets.zero,
        children: <Widget>[
```

```
44              const UserAccountsDrawerHeader(
45              accountName: Text("Sanjib Sinha"),
46              accountEmail: Text("sanjib@sanjibsinha.com"),
47              currentAccountPicture: CircleAvatar(
48                  backgroundColor: Colors.orange,
49                  child: Text(
50                  "S",
51                  style: TextStyle(fontSize: 40.0),
52                  ),
53              ),
54              ),
55              ListTile(
56              leading: const Icon(Icons.home),
57              title: const Text("Home"),
58              onTap: () {
59                  Navigator.pop(context);
60              },
61              ),
62              ListTile(
63              leading: const Icon(Icons.settings),
64              title: const Text("About"),
65              onTap: () {
66                  Navigator.of(context).pushNamed(DrawerAbout.routename);
67              },
68              ),
69              ListTile(
70              leading: const Icon(Icons.contacts),
71              title: const Text("Contact Us"),
72              onTap: () {
73                  Navigator.of(context).pushNamed(DrawerContact.routename);
74              },
75              ),
76          ],
77          ),
78      ),
79      );
80  }
81  }
```

In the above code, we have defined the MaterialApp with two extra routes.

```
return MaterialApp(
    title: 'Drawer Example',
    home: DrawerHome(),
    routes: {
        DrawerAbout.routename: (context) => const DrawerAbout(),
        DrawerContact.routename: (context) => const DrawerContact(),
    },
    );
...
```

To add a Drawer widget to the flutter app, we've wrapped it with Scaffold widget.

We've done that for one single purpose.

The Scaffold widget maintains that our flutter app should follow the Material design guidelines and builds a consistent visual structure.

Two extra routes that we've defined, also follow the same guidelines. In addition, two extra pages also use Scaffold widgets for the same purpose.

So that they can support other Material design components, such as AppBar or SnackBar.

Watch this part of the code, where we've populated the Drawer with items.

And to do that we've used ListView widget.

```
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(title: const Text('')),
    body: Center(
        child: Container(
        margin: const EdgeInsets.all(20),
        padding: const EdgeInsets.all(20),
        child: Text(
            'A Drawer Example.',
            style: TextStyle(fontSize: 50.0),
        ),
        ),
    ),
    drawer: Drawer(
        child: ListView(
        // Important: Remove any padding from the ListView.
        padding: EdgeInsets.zero,
        children: <Widget>[
            const UserAccountsDrawerHeader(
            accountName: Text("Sanjib Sinha"),
```

```
21              accountEmail: Text("sanjib@sanjibsinha.com"),
22              currentAccountPicture: CircleAvatar(
23                  backgroundColor: Colors.orange,
24                  child: Text(
25                  "S",
26                  style: TextStyle(fontSize: 40.0),
27                  ),
28              ),
29              ),
30              ListTile(
31              leading: const Icon(Icons.home),
32              title: const Text("Home"),
33              onTap: () {
34                  Navigator.pop(context);
35              },
36              ),
37              ListTile(
38              leading: const Icon(Icons.settings),
39              title: const Text("About"),
40              onTap: () {
41                  Navigator.of(context).pushNamed(DrawerAbout.routename);
42              },
43              ),
44              ListTile(
45              leading: const Icon(Icons.contacts),
46              title: const Text("Contact Us"),
47              onTap: () {
48                  Navigator.of(context).pushNamed(DrawerContact.routename);
49              },
50              ),
51          ],
52          ),
53      ),
54      );
55  }
```

As a result now we have a home page like the following image.

On the top left side, when we tap the Drawer icon, the drawer opens up to navigate to two pages, About and Contact us.

As we can see that it takes no time to populate the Drawer with DrawerHeader arguments and, moreover, we can add other routes with the help of ListTile widgets.

How do you customize the drawer in flutter and navigate?

We've already seen that to customize the Drawer in Flutter there are several options.

Moreover, we can follow the same Material Design guidelines and structure with help of Scaffold and Drawer widgets in other pages also.

Consider the About page Drawer instance. As we've clicked in the home page Drawer link, the About page opens us.

Moreover, the About or the Contact page can follow the same Material Design guidelines to add Drawer widgets.

Let's consider the About page code.

```
import 'package:flutter/material.dart';
import '/drawer_example/drawer_contact.dart';
import '/drawer_example/drawer_example.dart';

class DrawerAbout extends StatelessWidget {
const DrawerAbout({Key? key}) : super(key: key);
static const routename = "/about";

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(title: const Text('')),
    body: const Center(
        child: Text(
        'About Us Page.',
        style: TextStyle(fontSize: 50.0),
    )),
    drawer: Drawer(
        child: ListView(
        // Important: Remove any padding from the ListView.
        padding: EdgeInsets.zero,
        children: <Widget>[
            const UserAccountsDrawerHeader(
            accountName: Text("Sanjib Sinha"),
            accountEmail: Text("sanjib@sanjibsinha.com"),
            currentAccountPicture: CircleAvatar(
                backgroundColor: Colors.orange,
                child: Text(
                "S",
                style: TextStyle(fontSize: 40.0),
```

```
                ),
            ),
            ),
            ListTile(
            leading: const Icon(Icons.home),
            title: const Text("Home"),
            onTap: () {
                Navigator.of(context).pushNamed(DrawerHome.routename);
            },
            ),
            ListTile(
            leading: const Icon(Icons.contacts),
            title: const Text("Contact Us"),
            onTap: () {
                Navigator.of(context).pushNamed(DrawerContact.routename);
            },
            ),
        ],
        ),
    ),
    );
}
}
```

For full code please visit the respective GitHub repository.

As a result when we click the About Drawer, it opens up and displays different navigation links than home page.

To sum up, we wrap Drawer widget with Scaffold so that we can implement the structure that follows the Material Design guidelines.

Moreover, we can add Drawer items according to our needs so that user can navigate to other links.

Therefore, Drawer and Navigator widgets has relations between them. And the relation is a close one.

What is Material App in Flutter?

To view all the related images please visit the following link

- All related images with this section - https://sanjibsinha.com/material-app-flutter/[53]

[53] https://sanjibsinha.com/material-app-flutter/

The material app in flutter builds an application that uses material design. At the same time we treat it as a convenience widget that wraps a number of widgets.

In a series of articles we're delving into the material design widgets so that we can understand the basic concepts of Flutter user interface.

As we've just said, Material App widget wraps a number of widgets that we use for material design applications.

Moreover the Material App builds upon a WidgetsApp by adding material-design specific functionalities.

Consequently, these functionalities could be numerous and complex; such as AnimatedTheme or GridPaper. We'll come to that point later and discuss how we can use such complex flutter user interface.

Since the Material App builds upon a WidgetsApp, let's first try to understand what is WidgetsApp. How it wraps the material app.

If we understand WidgetsApp, we'll dig into the Flutter architecture and that will help us later. What is the difference between WidgetsApp and MaterialApp?

The WidgetsApp class is the base class for both MaterialApp and CupertinoApp to implement base functionality for an app.

As a parent to both MaterialApp and CupertinoApp, the WidgetsApp class provides all material design functionalities that we have been talking about in this material design category.

So we can define the WidgetsApp class as a convenience widget that wraps a number of material design specific widgets that we commonly use for an application.

However, what is one of the most important roles of WidgetsApp?

WidgetsApp binds the system back button to popping the Navigator or quitting the application.

To clarify, any flutter application must know how to navigate from one screen to another. We must provide any screen that facility.

WidgetsApp helps its child class to implement that feature. As a result, we can define all kind of routes in our MaterialApp class. We'll see that in a minute.

Is MaterialApp a widget?

Certainly, the MaterialApp is a widget. In addition, the MaterialApp configures the top-level Navigator to search for routes in a particular order. So that we can move to other widgets.

Let us build a MaterialApp widget and define the routes so that we can understand how it works.

Although we're trying to give full code snippet, still to get an more detail idea, you may visit the respective GitHub repository.

```dart
import 'package:flutter/material.dart';
import 'default_page.dart';
import 'first_page.dart';
import 'home_page.dart';
import 'second_page.dart';

class MaterialAppExample extends StatelessWidget {
const MaterialAppExample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Daily News',
    theme: ThemeData(
        primarySwatch: Colors.pink,
        primaryColor: Colors.blue,
        canvasColor: const Color.fromRGBO(255, 254, 229, 1),
        textTheme: ThemeData.light().textTheme.copyWith(
            bodyText2: const TextStyle(
                color: Color.fromRGBO(20, 51, 51, 1),
            ),
            bodyText1: const TextStyle(
                color: Color.fromRGBO(20, 51, 51, 1),
            ),
            headline6: const TextStyle(
                fontSize: 20,
                fontWeight: FontWeight.bold,
            ),
            ),
    ),

    initialRoute: '/', // default is '/'
    routes: {
        '/': (ctx) => const HomePage(),
        FirstPage.routename: (ctx) => const FirstPage(),
        SecondPage.routename: (ctx) => const SecondPage(),
    },
    onUnknownRoute: (settings) {
        return MaterialPageRoute(
        builder: (ctx) => const DefaultPage(),
        );
    },
    );
```

```
}
}
```

We've defined the theme of our flutter app in MaterialApp widget. However, we'll discuss that in a later article in great detail.

At present we concentrate on route and navigation part.

Besides this MaterialApp widget we've created three more screens. The first page, second page and a fall back page Default page.

Watch this part of the above code:

```
initialRoute: '/', // default is '/'
    routes: {
        '/': (ctx) => const HomePage(),
        FirstPage.routename: (ctx) => const FirstPage(),
        SecondPage.routename: (ctx) => const SecondPage(),
    },
    onUnknownRoute: (settings) {
        return MaterialPageRoute(
        builder: (ctx) => const DefaultPage(),
        );
    },
...
```

We've defined both the initial route and the '/' route that points to actually the home property.

As a matter of fact, we could have written the above code as the following:

```
home: const HomePage(),
    routes: {
        FirstPage.routename: (ctx) => const FirstPage(),
        SecondPage.routename: (ctx) => const SecondPage(),
    },
    onUnknownRoute: (settings) {
        return MaterialPageRoute(
        builder: (ctx) => const DefaultPage(),
        );
    },
...
```

The above code will also work.

Subsequently we've also found that any MaterialApp widget starts with Scaffold widget.

What is the difference between material app and scaffold?

The Scaffold widget acts as a base class or child of material app. Once flutter framework knows that we're going to use material app, it organizes all material components and follow material design.

However, it directs the next child Scaffold to provide many basic functionalities that only Scaffold can provide.

Let's see the code of home page, so we can understand how it works.

```
import 'package:flutter/material.dart';
import 'first_page.dart';

class HomePage extends StatelessWidget {
const HomePage({Key? key}) : super(key: key);
static const routename = '/first';

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Home Page'),
    ),
    body: GestureDetector(
        onTap: () {
        Navigator.push(
            context,
            MaterialPageRoute(builder: (context) => const FirstPage()),
        );
        },
        child: Container(
        margin: const EdgeInsets.all(10.0),
        padding: const EdgeInsets.all(10.0),
        child: ClipRRect(
            borderRadius: const BorderRadius.only(
            topLeft: Radius.circular(15.0),
            topRight: Radius.circular(15.0),
            bottomLeft: Radius.circular(15.0),
            bottomRight: Radius.circular(15.0),
            ),
            child: Image.network(
                'https://cdn.pixabay.com/photo/2020/05/15/14/03/lake-5173683_960_720\
.jpg'),
        ),
```

```
        ),
    ),
    );
}
}
```

The Scaffold starts with AppBar, that displays the back button when we navigate to another page. In addition, it allows us to use Drawer, Title, etc.

At the same time, at the bottom, beyond the body property it allows us to use BottomNavigationBar, FloatingActionButton, etc.

The body property of Scaffold uses here GestureDetector widget so that we can use onTap method to navigate to the first page.

If we tap the image, we can navigate to the first page, which again navigates to the second page.

This is one of the greatest advantage of MaterialApp that we can navigate from one screen to the other quite easily.

Let us take a look at the first page widget, whose AppBar automatically creates the back button to navigate back to the home page again.

Furthermore, we can tap the image and as we've reached here from the home page, we can navigate to the second page from here.

The following code snippet gives us an idea of how we can follow the material design principles.

```dart
import 'package:flutter/material.dart';
import 'second_page.dart';

class FirstPage extends StatelessWidget {
const FirstPage({Key? key}) : super(key: key);
static const routename = '/first';

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('First Page'),
    ),
    body: GestureDetector(
        onTap: () {
        Navigator.push(
            context,
            MaterialPageRoute(builder: (context) => const SecondPage()),
        );
```

```
        },
        child: Container(
        margin: const EdgeInsets.all(10.0),
        padding: const EdgeInsets.all(10.0),
        child: ClipRRect(
            borderRadius: const BorderRadius.only(
            topLeft: Radius.circular(15.0),
            topRight: Radius.circular(15.0),
            bottomLeft: Radius.circular(15.0),
            bottomRight: Radius.circular(15.0),
            ),
            child: Image.network(
                'https://cdn.pixabay.com/photo/2021/07/12/19/43/swans-6421355_960_72\
0.jpg'),
        ),
        ),
    ),
    );
}
}
```

To sum up, we've learned many basic concepts of material design that starts from WidgetsApp. After that MaterialApp widget along with with the Scaffold widget implement material design principles.

Subsequently, we've learned how MaterialApp implements one of the primary roles of material design principles, which is nothing but Navigation.

What is a theme in Flutter?

To view all the related images please visit the following link

- All related images with this section - https://sanjibsinha.com/flutter-theme-example/[54]

A theme in Flutter is an integral part of the user interface. Moreover, it is an extended part of Material design.

In this article, we'll try to understand how we can use theme to design color,and font to make our flutter app more visually good-looking.

Firstly, we can define our global theme in the MaterialApp widget.

[54] https://sanjibsinha.com/flutter-theme-example/

In our previous articles, we've seen how we can influence the child widgets through MaterialApp. We could've done that because MaterialApp is the parent widget and maintains the material design principles.

Therefore, we can apply the same logic in our theme also.

Subsequently, we can define our global theme in the MaterialApp widget.

Before we delve into detail, let me tell you that we keep the full code snippet in this GitHub Repository.

Firstly, let us take a look at the MaterialApp theme definition.

```
import 'package:flutter/material.dart';
import 'default_page.dart';
import 'first_page.dart';
import 'home_page.dart';
import 'second_page.dart';

class MaterialAppExample extends StatelessWidget {
const MaterialAppExample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Daily News',
    theme: ThemeData(
        primarySwatch: Colors.brown,
        primaryColor: Colors.blue,
        canvasColor: const Color.fromRGBO(255, 254, 229, 1),
        textTheme: ThemeData.light().textTheme.copyWith(
            bodyText2: const TextStyle(
                color: Color.fromRGBO(0, 155, 0, 1.0),
                fontSize: 30,
                fontWeight: FontWeight.bold,
                //fontFamily: 'Allison',
            ),
            bodyText1: const TextStyle(
                color: Color.fromRGBO(20, 51, 51, 1),
                fontSize: 30,
                fontWeight: FontWeight.bold,
            ),
            headline6: const TextStyle(
                fontSize: 20,
                fontWeight: FontWeight.bold,
```

```
            ),
            ),
    ),
    // home: CategoriesScreen(),
    initialRoute: '/', // default is '/'
    routes: {
        '/': (ctx) => const HomePage(),
        FirstPage.routename: (ctx) => const FirstPage(),
        SecondPage.routename: (ctx) => const SecondPage(),
    },
    onUnknownRoute: (settings) {
        return MaterialPageRoute(
        builder: (ctx) => const DefaultPage(),
        );
    },
    );
}
}
```

Since we've defined our theme in MaterialApp widget, we can apply that theme globally with the help of Theme.of method.

Through this method we can pass the context and moreover, we can select what type of text style we need to adopt.

Let us take a close look at the theme property.

```
theme: ThemeData(
        primarySwatch: Colors.brown,
        primaryColor: Colors.blue,
        canvasColor: const Color.fromRGBO(255, 254, 229, 1),
        textTheme: ThemeData.light().textTheme.copyWith(
            bodyText2: const TextStyle(
                color: Color.fromRGBO(0, 155, 0, 1.0),
                fontSize: 30,
                fontWeight: FontWeight.bold,
                fontFamily: 'Allison',
            ),
            bodyText1: const TextStyle(
                color: Color.fromRGBO(20, 51, 51, 1),
                fontSize: 30,
                fontWeight: FontWeight.bold,
            ),
            headline6: const TextStyle(
```

```
                fontSize: 20,
                fontWeight: FontWeight.bold,
            ),
            ),
    ),
```

The theme property of MaterialApp widget returns ThemeData constructor. Consequently, we can define many things. The primary color, font, different text style, etc.

When we apply this theme, the primary variant makes the same text color darker. We'll see that in a minute. Before that, we need to know more about Flutter theme.

How do you give a theme on Flutter?

As we've said, we share colors and font styles throughout a flutter app. That's why we define ThemeData to the MaterialApp constructor.

When we provide ThemeData to the MaterialApp constructor theme property, we can use that theme app wide.

Moreover, when we defina a Theme, we can use it within our own widgets as well as we can set the background colors and font styles for AppBars, Buttons,Checkboxes, and moreTo share a Theme across an entire app.

As a result, we can provide that theme to the immediate child, home page.

```
import 'package:flutter/material.dart';
import 'first_page.dart';

class HomePage extends StatelessWidget {
const HomePage({Key? key}) : super(key: key);
static const routename = '/first';

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Home Page'),
    ),
    body: Column(
        children: [
        Container(
            margin: const EdgeInsets.all(10.0),
            padding: const EdgeInsets.all(10.0),
            child: GestureDetector(
```

```
            onTap: () {
                Navigator.push(
                context,
                MaterialPageRoute(builder: (context) => const FirstPage()),
                );
            },
            child: ClipRRect(
                borderRadius: const BorderRadius.only(
                topLeft: Radius.circular(15.0),
                topRight: Radius.circular(15.0),
                bottomLeft: Radius.circular(15.0),
                bottomRight: Radius.circular(15.0),
                ),
                child: Image.network(
                    'https://cdn.pixabay.com/photo/2020/05/15/14/03/lake-5173683_960\
_720.jpg'),
            ),
            ),
        ),
        const SizedBox(
            height: 10,
        ),
        Text(
            'Material App Home First',
            style: TextStyle(
            color: Theme.of(context).colorScheme.primary,
            fontSize: 30,
            ),
        ),
        const SizedBox(
            height: 10,
        ),
        Text(
            'Material App Home Second',
            style: TextStyle(
            color: Theme.of(context).colorScheme.primaryVariant,
            fontSize: 20,
            fontWeight: FontWeight.bold,
            ),
        ),
        const SizedBox(
            height: 10,
        ),
```

```
        Text(
            'Material App Home Third',
            style: Theme.of(context).textTheme.bodyText1,
        ),
        const SizedBox(Text(
            'Material App Home Third',
            style: Theme.of(context).textTheme.bodyText1,
        ),
            height: 10,
        ),
        Text(
            'Material App Home Fourth',
            style: Theme.of(context).textTheme.bodyText2,
        ),
        ],
    ),
    );
}
}
```

Inside home page, now, after the image there are four text widgets, which use different text styles defined in MaterialApp.

Now, if we add a special font “Allison” to a certain text style defined in MaterialApp theme property, it changes its look.

However, we need to create a “font” folder inside the “lib” folder of our app,and then we need to add the dependencies in our “pubspec.yaml” file.

```
# To add custom fonts to your application, add a fonts section here,
# in this "flutter" section. Each entry in this list should have a
# "family" key with the font family name, and a "fonts" key with a
# list giving the asset and other descriptors for the font. For
# example:
fonts:

    - family: Allison
    fonts:
        - asset: lib/fonts/Allison-Regular.ttf
```

As a result, that special font reflects in our flutter app.

We’ve declared the font in this way in MaterialApp theme property. The font family property of text style widget points to the “Allison” font.

Subsequently, each text that follows this text style will display text in that font.

```
bodyText2: const TextStyle(
                color: Color.fromRGBO(0, 155, 0, 1.0),
                fontSize: 30,
                fontWeight: FontWeight.bold,
                fontFamily: 'Allison',
            ),
```

After that, we've declared that in our home page code:

```
Text(
            'Material App Home Third',
            style: Theme.of(context).textTheme.bodyText2,
        ),
```

As a result, each text style that uses "bodyText2" style displays the text in that font only.

How do I change the text theme in Flutter?

To change the text theme in Flutter, we can follow another method. We'll see that in another page, that is first page.

```
    import 'package:flutter/material.dart';
    import 'second_page.dart';

    class FirstPage extends StatelessWidget {
    const FirstPage({Key? key}) : super(key: key);
    static const routename = '/first';

    @override
    Widget build(BuildContext context) {
        return Scaffold(
        appBar: AppBar(
            title: const Text('First Page'),
        ),
        body: GestureDetector(
            onTap: () {
            Navigator.push(
                context,
                MaterialPageRoute(builder: (context) => const SecondPage()),
            );
            },
            child: Column(
```

```
            children: [
                Container(
                margin: const EdgeInsets.all(10.0),
                padding: const EdgeInsets.all(10.0),
                child: ClipRRect(
                    borderRadius: const BorderRadius.only(
                    topLeft: Radius.circular(15.0),
                    topRight: Radius.circular(15.0),
                    bottomLeft: Radius.circular(15.0),
                    bottomRight: Radius.circular(15.0),
                    ),
                    child: Image.network(
                        'https://cdn.pixabay.com/photo/2021/07/12/19/43/swans-642135\
5_960_720.jpg'),
                ),
                ),
                const SizedBox(
                height: 10,
                ),
                Text(
                'Using Color Constants',
                style: TextStyle(
                    fontSize: 28,
                    fontFamily: 'Allison',
                    color: Colors.blue,
                ),
                ),
                Text(
                'Using Hexadecimal Color',
                style: TextStyle(
                    fontSize: 28,
                    color: Color(0xFF89B5F7),
                ),
                ),
                Text(
                'Using ARGB Color',
                style: TextStyle(
                    fontSize: 28,
                    color: Color.fromARGB(255, 255, 128, 200),
                ),
                ),
                Text(
                'Using RGBO Color',
```

```
                style: TextStyle(
                    fontSize: 28,
                    color: Color.fromRGBO(0, 255, 0, 100),
                ),
                )
            ],
            ),
        ),
        );
    }
    }
```

What is the difference between the above code and the home page code?

In this case, we've not defined the configuration of the overall visual Theme for a MaterialApp or a widget sub-tree within the app.

We can use the MaterialApp theme to configure the appearance of the entire app. However, in the above case we've not followed that principle.

On the contrary, we've defined each text style explicitly.

Take one instance and see how it works.

```
Text(
            'Using Hexadecimal Color',
            style: TextStyle(
                fontSize: 28,
                color: Color(0xFF89B5F7),
            ),
            ),
```

The above code configures the hexadecimal value. Although as a beginner, we think, using the constant value is better.

```
Text(
            'Using Color Constants',
            style: TextStyle(
                fontSize: 28,
                fontFamily: 'Allison',
                color: Colors.blue,
            ),
            ),
```

As a result, we can view the image of the first page, where we have defined text style explicitly.

If we compare this image with the previous home page image, we'll understand the difference in color and font style.

Actually when we want to align the appearance of any widget with the overall theme, we must obtain the current theme's configuration with Theme.of.

It happens because, the Material components typically depend exclusively on the colorScheme and textTheme. These properties are guaranteed to have non-null values.

In addition, our null safety principle also keeps intact.

The question is, how does this process work?

When we define theme property in MaterialApp, the static Theme.of method finds the ThemeData value specified for the nearest BuildContext ancestor.

After defining a theme, we can use it into the widget build() methods with the Theme.of(context) method.

Since we've defined it earlier, the method looks into the widget tree and returns the very first theme in the tree.

If we have a not defined above our widget, the app's material theme is returned.

To sum up, we can say that, certainly we can define theme globally in our MaterialApp widget and all the sub-trees can follow the same material design principles.

However, if we want to change the look for a certain section of our flutter app, we can change the color and font explicitly.

What is scaffold in Flutter?

To view all the related images please visit the following link

- All related images with this section - https://sanjibsinha.com/scaffold-flutter/[55]

Scaffold is a widget in Flutter that implements the basic material design visual layout structure. Scaffold provides APIs for showing drawers and bottom sheets.

In this material design related tutorial we'll learn everything about Scaffold.

What is Scaffold? How and why we should use it to make a Flutter app complete.

Considering the visual aspect of a Mobile device, or Tab we should remember that the screen size varies from device to device.

[55] https://sanjibsinha.com/scaffold-flutter/

The first job of Scaffold is to expand and fill the available space. The Scaffold widget fills entire window or device screen.

Subsequently, when the keyboard appears the Scaffold's ancestor MediaQuery widget's MediaQueryData.viewInsets changes and the Scaffold will be rebuilt.

The Scaffold's body is resized to make room for the keyboard. In addition, there are areas that might not be completely visible. The MediaQueryData.padding value defines that area.

Therefore the Scaffold widget can control from top to bottom of the screen including the invisible areas. At the top, there is an an appBar and at the bottom there is bottomNavigationBar.

We'll see how they work in a minute.

In this introductory note, to sum up, we can say that Scaffold is mainly responsible for creating a base. And on that base the app screen appears.

Next, on that app screen various child widgets hold on and render on the screen.

At the same time Scaffold provides many widgets or APIs for showing Drawer, SnackBar, BottomNavigationBar, AppBar, FloatingActionButton, and many more.

To start with, we'll describe Scaffold properties part by part. As a result, the code snippet may look incomplete. Although finally we'll give the full code snippet at the end, still you may get the full code at this GitHub Repository.

Why scaffold is used in Flutter?

As we have said earlier, scaffold is used to implement the basic material design visual layout structure. Moreover, Scaffold contains almost everything we need to create a functional and responsive flutter app.

Let's start with AppBar.

How does it look like?

The AppBar is a horizontal bar at the top of the Scaffold widget.

The Scaffold widget starts with AppBar and displays other properties at the top of the screen such as Drawer. We can see the Drawer at the left of the AppBar.

Without AppBar, the Scaffold widget appears incomplete.

```
return Scaffold(
    appBar: AppBar(
        title: const Text(
        'Scaffold Example',
        style: TextStyle(
            fontFamily: 'Allison',
        ),
        ),
    ),
...
```

Certainly, we can add more functionalities to the AppBar. And in a separate discussion we'll look into that.

After AppBar, the body property plays very extremely important role for Scaffold.

The code snippet of body property is quite simple here, as we have only give a Text output.

```
body: const Center(
        child: Text(
        'A Scaffold Example',
        style: TextStyle(
            fontFamily: 'Allison',
            fontSize: 50,
            fontWeight: FontWeight.bold,
        ),
        ),
    ),
```

As a result, it looks like the following image.

During this discussion we must admit that the body property of Scaffold might not look such simple in any Flutter app.

In this body part we can build a chain of other widgets and a humongous tree of sub-classes may form a complex visual layout.

Just to get an idea, we may take a look at the following image where we have built a book app and the body property of Scaffold displays many categories of books.

How did we use the body property in the above image?

You can read the post on GridTile.

Let us move to the next feature of Scaffold. Now we will see how the Drawer widget helps us to navigate to other screens. Drawer is an integral part of Scaffold.

```
drawer: Drawer(
        child: ListView(
        // Important: Remove any padding from the ListView.
        padding: EdgeInsets.zero,
        children: <Widget>[
            const UserAccountsDrawerHeader(
            accountName: Text("Sanjib Sinha"),
            accountEmail: Text("sanjib@sanjibsinha.com"),
            currentAccountPicture: CircleAvatar(
                backgroundColor: Colors.orange,
                child: Text(
                "S",
                style: TextStyle(fontSize: 40.0),
                ),
            ),
            ),
            ListTile(
            leading: const Icon(Icons.home),
            title: const Text("Home"),
            onTap: () {
                Navigator.pop(context);
            },
            ),
            ListTile(
            leading: const Icon(Icons.settings),
            title: const Text("About"),
            onTap: () {
                Navigator.pop(context);
            },
            ),
            ListTile(
            leading: const Icon(Icons.contacts),
            title: const Text("Contact Us"),
            onTap: () {
                Navigator.pop(context);
            },
            ),
        ],
        ),
    ),
...
```

Drawer is a slider panel mainly used for navigation. On the AppBar, either at the top left or top right

section of the AppBar we can place it.

We can choose an appropriate icon for the Drawer. If we click on the Drawer the drawer menu opens up and user can navigate to other useful screens.

Take a look at the code, and compare that with the image. Now user can click any link shown on drawer and navigate to the destination.

Should I use scaffold Flutter?

Yes, we should use scaffold widget in flutter.

Why?

There are many reasons.

Let us discuss the main reasons and try to understand why we should use Scaffold.

The first and foremost of all reasons is Flutter Scaffold widget provides many default properties like AppBar, Body, Bottom Navigation, Floating Action and Persistent Footer.

Not only that, the scaffold widget controls the Material look and feel of the flutter app.

As a result, as long as we deal with material design and use MaterialApp, every page or Screen should have the scaffold widget as its immediate parent.

Consider the lower part of the screen. The Scaffold can add many functionalities there too.

Let us start with floating action button.

```
floatingActionButton: FloatingActionButton(
        elevation: 10.0,
        child: const Icon(Icons.access_alarms_outlined),
        onPressed: () {},
    ),
    floatingActionButtonLocation: FloatingActionButtonLocation.centerFloat,
...
```

Every floating action button has an onPressed method. Therefore we can use it for many purposes.

At present we've used it just for display and give an idea how it looks like.

As we can see in the image, the floating action button is primarily a button displayed at the bottom.

We can change its position with the help of another Scaffold property floating action button location.

At present we place it at the middle of the lower screen.

The floating action button is a circular icon button that floats over the body of a screen.

In addition, this button may promote an action in the application, which plays an important role.

These are the reasons why we should use Scaffold widget.

In the lower section of any flutter app, we can add more functionalities with the help of Scaffold widget. Not only the floating action button, but there is a Scaffold property like persistent footer buttons.

As the name suggests, we can add not one but many buttons.

```
persistentFooterButtons: <Widget>[
        TextButton(
        onPressed: () {},
        child: const Icon(
            Icons.backpack_rounded,
            color: Colors.green,
        ),
        ),
        TextButton(
        onPressed: () {},
        child: const Icon(
            Icons.handyman_rounded,
            color: Colors.red,
        ),
        ),
    ],
...
```

The persistent footer buttons property returns a list of buttons displayed at the bottom of the Scaffold widget.

The advantage of persistent footer button is that buttons are always visible.

Suppose we've used a ListView of items, and scroll the body of the Scaffold, these buttons will remain at their place. It is always wrapped in a ButtonBar widget.

The Scaffold widget renders these buttons at the lower section of the body but just above the bottom Navigation Bar.

In the above image we see the persistent footer buttons on the right side of the lower section. The two buttons in green and red color are sandwiched between floating action button and the bottom navigation bar.

Finally we'll take a look at one of the most important properties of bottom navigation bar.

The bottom navigation bar acts as a list of menus that we use for multiple purposes. However the name suggests that it is meant for navigating to other pages.

We can add at least five icons or text items in the bar as items.

```
bottomNavigationBar: BottomNavigationBar(
        currentIndex: 0,
        fixedColor: Colors.teal,
        items: const [
        BottomNavigationBarItem(
            label: 'Home',
            icon: Icon(Icons.home),
        ),
        BottomNavigationBarItem(
            label: 'Search',
            icon: Icon(Icons.search),
        ),
        BottomNavigationBarItem(
            label: 'Add',
            icon: Icon(Icons.add_box),
        ),
        ],
        onTap: (int index) {},
    ),
...
```

We can take a look at the above image of persistent footer buttons where the bottom navigation bar is visible at the very bottom.

Finally, with the help of all Scaffold properties the flutter app looks like the following image.

And if you're interested to take a look at the full code, here it is:

```
import 'package:flutter/material.dart';
import 'package:flutter/widgets.dart';

class ScaffoldExample extends StatelessWidget {
const ScaffoldExample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Scaffold Example',
    home: ScaffoldHome(),
    );
}
}

class ScaffoldHome extends StatelessWidget {
```

```
17  const ScaffoldHome({Key? key}) : super(key: key);
18
19  @override
20  Widget build(BuildContext context) {
21      return Scaffold(
22      appBar: AppBar(
23          title: const Text(
24          'Scaffold Example',
25          style: TextStyle(
26              fontFamily: 'Allison',
27          ),
28          ),
29      ),
30      body: const Center(
31          child: Text(
32          'A Scaffold Example',
33          style: TextStyle(
34              fontFamily: 'Allison',
35              fontSize: 50,
36              fontWeight: FontWeight.bold,
37          ),
38          ),
39      ),
40      drawer: Drawer(
41          child: ListView(
42          // Important: Remove any padding from the ListView.
43          padding: EdgeInsets.zero,
44          children: <Widget>[
45              const UserAccountsDrawerHeader(
46              accountName: Text("Sanjib Sinha"),
47              accountEmail: Text("sanjib@sanjibsinha.com"),
48              currentAccountPicture: CircleAvatar(
49                  backgroundColor: Colors.orange,
50                  child: Text(
51                  "S",
52                  style: TextStyle(fontSize: 40.0),
53                  ),
54              ),
55              ),
56              ListTile(
57              leading: const Icon(Icons.home),
58              title: const Text("Home"),
59              onTap: () {
```

```
                Navigator.pop(context);
            },
            ),
            ListTile(
            leading: const Icon(Icons.settings),
            title: const Text("About"),
            onTap: () {
                Navigator.pop(context);
            },
            ),
            ListTile(
            leading: const Icon(Icons.contacts),
            title: const Text("Contact Us"),
            onTap: () {
                Navigator.pop(context);
            },
            ),
        ],
        ),
    ),
    floatingActionButton: FloatingActionButton(
        elevation: 10.0,
        child: const Icon(Icons.access_alarms_outlined),
        onPressed: () {},
    ),
    floatingActionButtonLocation: FloatingActionButtonLocation.centerFloat,
    persistentFooterButtons: <Widget>[
        TextButton(
        onPressed: () {},
        child: const Icon(
            Icons.backpack_rounded,
            color: Colors.green,
        ),
        ),
        TextButton(
        onPressed: () {},
        child: const Icon(
            Icons.handyman_rounded,
            color: Colors.red,
        ),
        ),
    ],
    bottomNavigationBar: BottomNavigationBar(
```

```
        currentIndex: 0,
        fixedColor: Colors.teal,
        items: const [
        BottomNavigationBarItem(
            label: 'Home',
            icon: Icon(Icons.home),
        ),
        BottomNavigationBarItem(
            label: 'Search',
            icon: Icon(Icons.search),
        ),
        BottomNavigationBarItem(
            label: 'Add',
            icon: Icon(Icons.add_box),
        ),
        ],
        onTap: (int index) {},
    ),
    backgroundColor: Colors.white,
    );
}
}
```

To sum up, Scaffold widget has many properties that we should use with discretion. Since Scaffold has many properties, so we must be choosy to select what property would fit our criteria.

How do you make a TabBar in flutter?

To add tabs to the app, we need to create a TabBar and TabBarView and attach them with the TabController.

To view all the related images please visit the following link

- All related images with this section[56]

TabBar is a material design widget that displays a horizontal row of tabs, and we can control the number of tabs.

We create the horizontal row of tabs as the AppBar.bottom part of an AppBar. Moreover, we need two extra widgets to fulfill our mission.

[56] https://sanjibsinha.com/tabbar-flutter/

The first one is TabController, and if that is not provided, then we should use a DefaultTabController as the ancestor.

In Addition we need a TabBarView that returns a list of widgets where we can place the associated tab views.

While building the TabBar widget, we always remember a few key concepts. The tab controller's TabController.length must equal the length of the tabs list and the length of the TabBarView.children list.

Let us imagine a two tabs "Home" and "Contact". When we run the app, the Home tab initially selected and opens up. However, just below the AppBar we can have two tabs displayed side by side.

The AppBar bottom property holds the key to two tabs displayed in the image above.

```
return MaterialApp(
    home: DefaultTabController(
        length: 2,
        child: Scaffold(
        appBar: AppBar(
            title: const Text(
            'Scaffold Example',
            style: TextStyle(
                fontFamily: 'Allison',
            ),
            ),
            bottom: TabBar(
            tabs: [
                Tab(icon: Icon(Icons.home_max_outlined), text: "Home"),
                Tab(icon: Icon(Icons.contact_page_outlined), text: "Contact")
            ],
            ),
        ),
...
// code is incomplete for brevity
```

In the above image what we see is this part of the code:

```
bottom: TabBar(
            tabs: [
                Tab(icon: Icon(Icons.home_max_outlined), text: "Home"),
                Tab(icon: Icon(Icons.contact_page_outlined), text: "Contact")
            ],
            ),
```

Although this code snippet is incomplete, we can take a look at the full code at this GitHub repository.

Therefore, a TabBar has two distinct parts. Firstly we need DefaultTabController as the ancestor. Secondly we need TabBarView widget to return the body part of the Scaffold.

As a result the full code looks similar to this.

```
import 'package:flutter/material.dart';
import 'tabbar_contact.dart';
import 'tabbar_home.dart';

class TabBarExample extends StatelessWidget {
const TabBarExample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: DefaultTabController(
        length: 2,
        child: Scaffold(
        appBar: AppBar(
            title: const Text(
            'Scaffold Example',
            style: TextStyle(
                fontFamily: 'Allison',
            ),
            ),
            bottom: TabBar(
            tabs: [
                Tab(icon: Icon(Icons.home_max_outlined), text: "Home"),
                Tab(icon: Icon(Icons.contact_page_outlined), text: "Contact")
            ],
            ),
        ),
        body: TabBarView(
            children: [
            TabBarHome(),
```

```
31              TabBarContact(),
32              ],
33          ),
34          drawer: Drawer(
35              child: ListView(
36              // Important: Remove any padding from the ListView.
37              padding: EdgeInsets.zero,
38              children: <Widget>[
39                  const UserAccountsDrawerHeader(
40                  accountName: Text("Sanjib Sinha"),
41                  accountEmail: Text("sanjib@sanjibsinha.com"),
42                  currentAccountPicture: CircleAvatar(
43                      backgroundColor: Colors.orange,
44                      child: Text(
45                      "S",
46                      style: TextStyle(fontSize: 40.0),
47                      ),
48                  ),
49                  ),
50                  ListTile(
51                  leading: const Icon(Icons.home),
52                  title: const Text("Home"),
53                  onTap: () {
54                      Navigator.pop(context);
55                  },
56                  ),
57                  ListTile(
58                  leading: const Icon(Icons.settings),
59                  title: const Text("About"),
60                  onTap: () {
61                      Navigator.pop(context);
62                  },
63                  ),
64                  ListTile(
65                  leading: const Icon(Icons.contacts),
66                  title: const Text("Contact Us"),
67                  onTap: () {
68                      Navigator.pop(context);
69                  },
70                  ),
71              ],
72              ),
73          ),
```

```
74          floatingActionButton: FloatingActionButton(
75              elevation: 10.0,
76              child: const Icon(Icons.access_alarms_outlined),
77              onPressed: () {},
78          ),
79          floatingActionButtonLocation:
80              FloatingActionButtonLocation.centerFloat,
81          persistentFooterButtons: <Widget>[
82              TextButton(
83              onPressed: () {},
84              child: const Icon(
85                  Icons.backpack_rounded,
86                  color: Colors.green,
87              ),
88              ),
89              TextButton(
90              onPressed: () {},
91              child: const Icon(
92                  Icons.handyman_rounded,
93                  color: Colors.red,
94              ),
95              ),
96          ],
97          bottomNavigationBar: BottomNavigationBar(
98              currentIndex: 0,
99              fixedColor: Colors.teal,
100             items: const [
101             BottomNavigationBarItem(
102                 label: 'Home',
103                 icon: Icon(Icons.home),
104             ),
105             BottomNavigationBarItem(
106                 label: 'Search',
107                 icon: Icon(Icons.search),
108             ),
109             BottomNavigationBarItem(
110                 label: 'Add',
111                 icon: Icon(Icons.add_box),
112             ),
113             ],
114             onTap: (int index) {},
115         ),
116         backgroundColor: Colors.amberAccent,
```

```
        ),
    ),
    );
}
}
```

In the above code snippet, the body part plays an important role also.

```
body: TabBarView(
            children: [
            TabBarHome(),
            TabBarContact(),
            ],
        ),
```

We need to have two separate widgets to show two pages once the tabs are clicked.

What is TabBarView flutter?

The TabBarView is nothing but a page that displays the particular widget that corresponds to the selected tab. If we select the Home tab, we get this view.

A page view that displays the widget which corresponds to the currently selected tab. As we've learned earlier, we use the TabBarView in conjunction with a TabBar.

We can take a look at the home page code. That is pretty simple.

```
import 'package:flutter/material.dart';

class TabBarHome extends StatelessWidget {
const TabBarHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Container(
    margin: const EdgeInsets.all(20),
    padding: const EdgeInsets.all(20),
    child: Text(
        'Home Page',
        style: TextStyle(
        fontFamily: "Allison",
        fontSize: 50,
        fontWeight: FontWeight.bold,
```

```
        ),
    ),
    );
}
}
```

If we select the contact tab, we can see the contact page.

In the similar vein, we can write the same code changing slightly.

```
import 'package:flutter/material.dart';

class TabBarContact extends StatelessWidget {
const TabBarContact({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Container(
    margin: const EdgeInsets.all(20),
    padding: const EdgeInsets.all(20),
    child: Text(
        'Conact Page',
        style: TextStyle(
        fontFamily: "Allison",
        fontSize: 50,
        fontWeight: FontWeight.bold,
        ),
    ),
    );
}
}
```

As a result, we can view the contact page like the following image.

While working with TabBar and a TabBarView, we need TabController, and if that is not provided, then we should use a DefaultTabController as the ancestor.

The TabController class coordinates tab selection between a TabBar and a TabBarView.

The index property is the index of the selected tab.

In our case, how did we use that?

We've not mentioned that. However, in case we use a List of widgets beforehand, then we can select a current index.

Consequently we need to remember that the TabBar inherits from a stateful widget and therefore can create a TabController and share it directly.

In our case, since the TabBar and TabBarView don't have a convenient stateful ancestor, we've provided a DefaultTabController inherited widget.

How do I create a DropdownButton in flutter?

We need a matrial design button like DropdownButton to select from a list of items.

To view all the related images please visit the following link

- All related images with this section[57]

To create a material design button like DropdownButton we need to use DropdownMenuItem class.

In this short but insightful tutorial on material design Flutter UI structure we'll discuss how we can add a drop down button for selecting from a list of items.

When we use the DropdownMenuItem<T> class, DropdownButton actually instantiate a DropdownMenuItem object or creates an item in the menu.

Usually the button shows the currently selected item as well as an arrow that opens a menu so that we can select another item.

The question is what type of item the value represents. The type T is the type of the value the entry represents. Since it represents type of the value,the menu must represent values of consistent types.

Although the parent of DropdownMenuItem is a stateless widget, we need a stateful widget to use the onChanged callback. So that the function can update a state variable that defines the drop down's value.

Subsequently, it also calls State.setState to rebuild the DropdownButton with the new value.

In the following code, we'll see how we can add multiple drop downs in Flutter.

How do you add two drop downs in flutter?

We need a drop down button for selecting from a list of items. Therefore, adding more than on drop downs is not a big issue as long as we can create on drop down.

Let us see how it looks like first.

Let's take a close look, so we can understand how it looks like on the mobile device.

The DropdownButton class has many properties and tow most important properties are items and onChanged method.

[57] https://sanjibsinha.com/dropdown-button-flutter/

```
DropdownButton example() => DropdownButton<String>(
        items: [
        DropdownMenuItem(
            value: "1",
            child: Row(
            mainAxisAlignment: MainAxisAlignment.spaceBetween,
            children: const <Widget>[
                Icon(Icons.build),
                SizedBox(width: 10),
                Text(
                'Example',
                style: TextStyle(
                    fontSize: 20,
                    fontWeight: FontWeight.bold,
                ),
                ),
            ],
            ),
        ),
        DropdownMenuItem(
            value: "2",
            child: Row(
            mainAxisAlignment: MainAxisAlignment.spaceBetween,
            children: const <Widget>[
                Icon(Icons.settings),
                SizedBox(width: 10),
                Text(
                "Setting",
                style: TextStyle(
                    fontSize: 20,
                    fontWeight: FontWeight.bold,
                ),
                ),
            ],
            ),
        ),
        ],
        onChanged: (value) {
        setState(() {
            _value = value!;
        });
        },
        value: _value,
```

```
        isExpanded: true,
    );
```

The DropdownButton items property consists of a list. Therefore, whenever we want to display we must remember that.

In many cases, we can directly map the list items and use toList() method.

However, in this case, we directly call inside the build method of a stateful widget.

```
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'DropDownButton Example',
    home: Scaffold(
        appBar: AppBar(
        title: const Text('DropDownButton Example'),
        ),
        body: SingleChildScrollView(
        child: Column(
            children: <Widget>[
            const SizedBox(height: 10),
            Container(
                padding: const EdgeInsets.symmetric(horizontal: 20),
                color: Colors.green,
                child: example(),
            ),
            const SizedBox(height: 10),
            ],
        ),
        ),
    ),
    );
}
// For full code please visit GitHub repository
```

As a result, we can now click the drop down icon to select any item.

We must use a stateful widget because if the list of items is null or the onChanged callback is null, the drop down button will not be enabled anymore.

What does that mean?

It means the arrow will be displayed in grey and it will not respond to input.

And, finally, since we've created a material design button like DropdownButton, one of its ancestors should be a Material widget.

What is Material State in Flutter?

MaterialState enum and MaterialStateProperty class are siblings. They help to change state of material widgets.

To view all the related images please visit the following link

- All related images with this section[58]

Material State is an interactive state that some of the Material widgets can use. In addition, with the help of MaterialState enum, those widgets receive inputs from users and change its state.

How they do that, we'll see in a minute. However, before that we'll try to understand how it works.

Let us consider two material widgets like checkbox and text button. In our subsequent part, we'll test how we can implement material state on these two widgets.

As we know, these material widgets usually work on user inputs. A user can tick the checkbox or press the button. They can also hover over those widgets.

As a result the interactive state of these material widgets change. If a text button looks red, when a user hover over the button, it may change to green.

The same is true for the check box.

Now, the question is, how these widgets track their state?

They track their current state using Set<MaterialState>.

Suppose we want to change color of two stateful material widgets. We need to write some code like this:

```
Color changeColor(Set<MaterialState> states) {
    const Set<MaterialState> interactiveStates = <MaterialState>{
        MaterialState.pressed,
        MaterialState.hovered,
        MaterialState.focused,
    };
    if (states.any(interactiveStates.contains)) {
        return Colors.green;
    }
    return Colors.deepOrange;
    }
```

We can call these function inside text button or checkbox, like the following way:

[58] https://sanjibsinha.com/material-state-flutter/

```
TextButton(
            style: ButtonStyle(
                foregroundColor: MaterialStateProperty.resolveWith(changeColor),
            ),
            onPressed: () {},
            child: const Text('TextButton'),
            ),
        ...
            Checkbox(
            checkColor: Colors.white,
            fillColor: MaterialStateProperty.resolveWith(changeColor),
            value: isChecked,
            onChanged: (bool? value) {
                setState(
                () {
                    isChecked = value!;
                },
                );
            },
            ),
```

We can see that each stateful material widgets has different property to call that changeColor method. For full code please visit the respective GitHub repository.

And to do that they take help of another interface MaterialStateProperty. This interface helps these widget objects to change their state. Although it sounds simple, yet, it goes through a complex process.

MaterialStateProperty acts as an interface for material widgets, here checkbox and text button. Consequently, these widgets resolve to a value of type T that links to MaterialState, and moreover, it controls the widget's interactive state.

Now MaterialState has many values, such as, MaterialState.focused, MaterialState.hovered, MaterialState.pressed.

We've seen that in the above code.

For an example, if we take a look at the initial state of the flutter app, it looks like the following image:

Initially the text button and the check box are deep orange. When we had taken the screenshot that had been their initial state.

Due to quality of the image the deep orange looks like red. ☒ Anyway, let us take a look at this part of code of

```
Set<MaterialState>:

return Colors.deepOrange;
```

What is material state property in Flutter?

To sum up in one sentence, material state properties represent values that depend on text button or checkbox widget's material state.

We have already encoded that as a Set<MaterialState> values and check whether the values are getting changed:

```
if (states.any(interactiveStates.contains)) {
        return Colors.green;
    }
```

As a result, if we hover, the color of text button changes.

The same is true for the checkbox. The state changes as we hover over this material widget.

To sum up, we can say that the relationship between MaterialState enum and MaterialStateProperty class plays a key role here.

How do I add a checkbox in flutter?

Usually, when we see a checkbox on the screen, it displays a square box with white space. But the real story starts from here…

To view all the related images please visit the following link

- All related images with this section[59]

We can use several input components in flutter. Checkbox is one of them. Moreover, we can add a checkbox quite easily.

However, we need to understand the main concept behind any checkbox. Firstly, one of the ancestor of any checkbox should be a Material widget.

Secondly, since checkbox is a type of input component, it always holds a Boolean value.

While we check a checkbox or mark it with a tick sign, that means yes. If we don't, it means no. However, there is another value. Null.

[59] https://sanjibsinha.com/checkbox-flutter/

A checkbox can display null also.

A material design checkbox doesn't maintain any state.

Then what happens when a user marks a checkbox with a tick sign?

Only when a user checks or marks a checkbox, the state of checkbox changes, and the widget calls the onChanged callback.

Each time, when a widget uses a checkbox, will listen to the onChanged callback and rebuilds the checkbox with a new value. Following that an update takes place and that changes the visual appearance of the checkbox.

Enough talking, let's see a snippet of code that displays one checkbox. Let's suppose, a checkbox widget is the child of a Material widget and it will use the checkbox.

```
class CheckBoxExample extends StatelessWidget {
const CheckBoxExample({Key? key}) : super(key: key);

static const String _title = 'CheckBox Example';

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: _title,
    home: Scaffold(
        appBar: AppBar(title: const Text(_title)),
        body: const Center(
        child: CheckBoxWidget(),
        ),
    ),
    );
}
}
```

The first condition is CheckBoxWidget must be a stateful widget. So that when the value of checkbox changes the onChanged callback will be applied to the value, which can be yes.

Suppose, the callback is null. In that case, the checkbox will not respond to any input gestures. However, if we tap the checkbox, the callback cycle changes from no to yes, or null.

Let us see the next part of the code.

```
class CheckBoxWidget extends StatefulWidget {
const CheckBoxWidget({Key? key}) : super(key: key);

@override
State<CheckBoxWidget> createState() => _CheckBoxWidgetState();
}

class _CheckBoxWidgetState extends State<CheckBoxWidget> {
bool isChecked = false;

@override
Widget build(BuildContext context) {
    Color changeColor(Set<MaterialState> states) {
    const Set<MaterialState> interactiveStates = <MaterialState>{
        MaterialState.pressed,
        MaterialState.hovered,
        MaterialState.focused,
    };
    if (states.any(interactiveStates.contains)) {
        return Colors.blue;
    }
    return Colors.red;
    }

    return Checkbox(
    checkColor: Colors.white,
    fillColor: MaterialStateProperty.resolveWith(changeColor),
    value: isChecked,
    onChanged: (bool? value) {
        setState(() {
        isChecked = value!;
        });
    },
    );
}
}
```

The Checkbox onChanged property now updates the state of the parent StatefulWidget using the State.setState method.

As a result, the parent StatefulWidget gets rebuilt, and the color of the checkbox changes.

In the next example we'll see another example of checkbox along with the Switch widget.

How do you change the check box color in flutter?

There are several options to change the color of a checkbox.

Usually, when we see a checkbox on the screen, it displays a square box with white space. When a user hover over the checkbox, it may change its color. To clarify, we can revisit the previous code where we've used Set<MaterialState>.

With the help of MaterialState and MaterilaStateProperty we can manage that change in color as state changes.

But there is another easier method to change the color of checkbox.

In the above image when we tap the checkbox, the checkbox color changes to blue, and the color of Switch changes to red.

How did that happen?

Very simple.

Both checkbox and switch widgets have a property called activeColor. We can mention the color constant value there so that each time user tap any one of them, the concerned widgets change to that activeColor.

Let us see the respective code snippet.

```
body: SingleChildScrollView(
        child: Column(
            children: <Widget>[
            Checkbox(
                value: _value,
                onChanged: (bool? value) {
                setState(
                    () {
                    _value = value!;
                    },
                );
                },
                activeColor: Colors.blue,
                tristate: true,
            ),
            Switch.adaptive(
                value: _value,
                activeColor: Colors.redAccent,
                activeTrackColor: Colors.redAccent,
                onChanged: (bool? value) {
                setState(
```

```
                    () {
                    _value = value!;
                    },
                );
                },
            ),
            ],
        ),
        ),
// code is incomplete for brevity. Please visit the GitHub repository to get the ful\
l code snippet.
```

To sum up, in this article we've learned how to use checkbox in flutter. However, besides this compact version of checkbox, we have CheckboxListTile widget, in which we can describe the header and subtitle using the respective properties.

In the next articles, we'll take a look at how CheckboxListTile widget works.

How do I use a checkbox widget in flutter?

CheckboxListTile is a mixture of Checkbox and ListTile widgets, sharing many common properties.

To view all the related images please visit the following link

- All related images with this section[60]

In our previous post we've seen how we can use checkbox in Flutter. However, we've not discussed another handy widget.

Checkbox list tile widget, in many ways, plays the similar role as a checkbox. In addition, it acquires many properties from ListTile widget in flutter.

Therefore, we can say, CheckboxListTile is a mixture of Checkbox and ListTile widgets.

In this article, we'll see how we can use checkbox and check box list tile side by side. So that we can compare these two widgets.

In every way, the value, onChanged, activeColor and checkColor properties of CheckboxListTile widget play the same role as Checkbox does.

On the other hand, properties like title, subtitle, isThreeLine, dense, and contentPadding of ListTIle are also there.

Let us first take a look at these two widgets.

[60] https://sanjibsinha.com/checkbox-widget-flutter/

The following image shows one checkbox and two check box list tiles widgets side by side.

Let's see the code of checkbox, so that we can have idea about the similar properties these two widgets share.

```
Row(
                children: [
                const SizedBox(
                    width: 10,
                ),
                Text(
                    'A Checkbox Example: ',
                    style: TextStyle(fontSize: 17.0),
                ),
                Checkbox(
                    checkColor: Colors.greenAccent,
                    activeColor: Colors.red,
                    value: this.valuefirst,
                    onChanged: (bool? value) {
                    setState(() {
                        this.valuefirst = value!;
                    });
                    },
                ),
                Checkbox(
                    value: this.valuesecond,
                    onChanged: (bool? value) {
                    setState(
                        () {
                        this.valuesecond = value!;
                        },
                    );
                    },
                ),
                ],
            ),
// code is incomplete for brevity, for full code please visit the respective GitHub \
repository
```

The first checkbox example shares the same properties that a checkbox list tile does have. In the first checkbox sample, we have seen the above mentioned value, onChanged, activeColor and checkColor properties.

```
Checkbox(
                    checkColor: Colors.greenAccent,
                    activeColor: Colors.red,
                    value: this.valuefirst,
                    onChanged: (bool? value) {
                    setState(() {
                        this.valuefirst = value!;
                    });
                    },
                ),
```

Now, we can have a look at the checkbox list tile code example, so that we can compare them side by side.

```
Column(
                children: [
                Text(
                    'CheckboxListTile Example',
                    style: TextStyle(fontSize: 20.0),
                ),
                CheckboxListTile(
                    secondary: const Icon(Icons.home_outlined),
                    title: const Text('This is header'),
                    subtitle: const Text('This is subtitle'),
                    value: this.valuefirst,
                    onChanged: (bool? value) {
                    setState(
                        () {
                        this.valuefirst = value!;
                        },
                    );
                    },
                ),
                CheckboxListTile(
                    controlAffinity: ListTileControlAffinity.trailing,
                    secondary: const Icon(Icons.home_filled),
                    title: const Text('This is header'),
                    subtitle: Text('This is subtitle'),
                    value: this.valuesecond,
                    onChanged: (bool? value) {
                    setState(
                        () {
                        this.valuesecond = value!;
```

```
                        },
                    );
                    },
                ),
                ],
            ),
// code is incomplete for brevity, for full code please visit the respective GitHub \
repository
```

By default the value is set to false as this is a Boolean value and takes three values. True, false and null.

```
class _CheckboxListtileExampleState extends State<CheckboxListtileExample> {
bool valuefirst = false;
bool valuesecond = false;
...
```

Since in our example the checkbox and checkbox list tile have the similar values, when we select one, another gets selected automatically.

As the value changes from false to true, the active color property changes. In checkbox widget we've mentioned it explicitly. However, in checkbox list tile widget we've not mentioned it.

That means active color is null in the checkbox list tile widget.

Due to that reason, it enables the ThemeData.toggleableActiveColor and makes it blue.

If we want to show the CheckboxListTile as disabled, we can pass null as the onChanged callback.

What is elevated Button in flutter?

ElevatedButton is not like TextButton or OutlinedButton. What is its specialty?

To view all the related images please visit the following link

- All related images with this section[61]

There is something fancy in elevated button in flutter. Firstly, there are other buttons.

Text button is one of them. TextButton is a simple flat button without a shadow. Secondly, we have Outlined Button. It's nothing but a TextButton with a border outline. In another article we'll differentiate and compare them.

[61] https://sanjibsinha.com/elevated-button-flutter/

However, what is the specialty of ElevatedButton?

We'll learn that in this flutter tutorial.

To start with, let us know that any button is a part of Material Design in flutter. We've been discussing this topic for a time being. For all material design components source code you may visit this GitHub repository. You'll get plenty of examples.

Therefore, we can say that ElevatedButton is a material design button, however, it has some specialty.

We use elevated buttons to add more dimensions to our material design.

As a result, ElevatedButton has many properties that we can control in a various way and give users rich experience in the flutter app.

We know that on a mobile screen mostly the layouts are flat. Hardly there is elevation or elevated content.

Subsequently, we must remember that we shouldn't use elevated buttons in already elevated content widgets like Card or Dialog.

Since the name suggests that the button must have an elevation, therefore, while we press, on a Material widget, the Material.elevation increases automatically.

The ElevatedButton is a direct child of ButtonStyleButton class, as a result, it inherits a few properties from the parent class.

The ElevatedButton displays its Text and Icon widgets in style's ButtonStyle.foregroundColor and the button's filled background is the ButtonStyle.backgroundColor.

The ButtonStyleButton class has provided those features.

Before taking a look at the code, to get an idea, let us have a quick look at how an ElevatedButton looks like.

We have two elevated buttons. The first one is in blue and the second one is in red.

When we press the first elevated button, it'll turn to deep purple and the long press will add a white tone in the background.

However, the second one will not change its color. Although it will get the elevation by adding a whitish tone in the background.

In addition, we'll take a look at the code in a minute.

How do you get the elevated button on flutter?

Before taking a look at the code, let's know how we can get the elevated button on flutter?

We can place elevated button inside, any layout widgets, like Center, Column, etc.

In our code, we've kept two elevated buttons inside Column.

```
child: Column(
            children: [
            Text(
                'First ElevatedButton Example',
                style: TextStyle(fontSize: 30.0),
            ),
            ElevatedButton(
                style: ButtonStyle(
                backgroundColor: MaterialStateProperty.resolveWith<Color>(
                    (Set<MaterialState> states) {
                    if (states.contains(MaterialState.pressed))
                        return Colors.deepPurple;
                    return Colors.blue;
                    },
                ),
                ),
                child: Text(
                'First ElevatedButton',
                style: TextStyle(fontSize: 30),
                ),
                onPressed: () {},
            ),
            const SizedBox(
                height: 10,
            ),
            Text(
                'Second ElevatedButton Example: ',
                style: TextStyle(fontSize: 30),
            ),
            ElevatedButton(
                child: Text('Elevated Button'),
                onPressed: () {},
                style: ButtonStyle(
                backgroundColor: MaterialStateProperty.all(Colors.red),
                padding: MaterialStateProperty.all(
                    EdgeInsets.all(50),
                ),
                textStyle: MaterialStateProperty.all(
                    TextStyle(fontSize: 30),
                ),
                ),
            ),
            ],
```

```
        ),
// code is incomplete for brevity, please visit the respective GitHub repository for\
 full code snippet
```

The first elevated button looks more complicated than the first one. Let's take a closer look.

```
ElevatedButton(
                style: ButtonStyle(
                backgroundColor: MaterialStateProperty.resolveWith<Color>(
                    (Set<MaterialState> states) {
                    if (states.contains(MaterialState.pressed))
                        return Colors.deepPurple;
                    return Colors.blue;
                    },
                ),
                ),
                child: Text(
                'First ElevatedButton',
                style: TextStyle(fontSize: 30),
                ),
                onPressed: () {},
            ),
```

We've overridden the style parameter of the elevated button with ButtonStyle and as the background property we've used MaterialState and MaterialStateProperty. We've an elaborate discussion on that topic before, please read that article on MaterialState and MaterialStateProperty.

Because of these Material State properties, the elevated button changes its color while we press it.

The first elevated button changes its color to deep purple from blue while we press it.

However, the second elevated button doesn't change the color while it gets pressed.

Actually, defaultStyleOf defines elevated button's default style, and one can override the style of this elevated button, as we've done with button style class.

If we take a closer look at the second elevated button, we'll find that we can define background color, text style with the help of MaterialStateProperty.

```
ElevatedButton(
                child: Text('Elevated Button'),
                onPressed: () {},
                style: ButtonStyle(
                backgroundColor: MaterialStateProperty.all(Colors.red),
                padding: MaterialStateProperty.all(
                    EdgeInsets.all(50),
                ),
                textStyle: MaterialStateProperty.all(
                    TextStyle(fontSize: 30),
                ),
                ),
```

The static styleFrom method is another convenient way to create a elevated button ButtonStyle from simple values.

```
ElevatedButton(
onPressed: () => {},
child: Text(' ADD '),
style: ElevatedButton.styleFrom(
primary: Colors.purple,
padding: EdgeInsets.symmetric(
horizontal: 20, vertical: 20),
textStyle: TextStyle(
fontSize: 16, fontWeight: FontWeight.bold),
),
),
),
```

What is text button in Flutter?

- All related images with this section - https://sanjibsinha.com/text-button-flutter/[62]

By default, a text button in flutter has no distinct feature. However, we can make its presence felt.

A text button is a Material widget label without elevation. We've discussed about elevated button in our previous post.

This short tutorial about TextButton will teach us about the nature of this button. In addition, we'll know what are the primary differences between elevated button, outlined button and a text button.

[62] https://sanjibsinha.com/text-button-flutter/

Firstly, any button is basically a user input component and a child of material widget.

Secondly, in a flat layout we should think how we can decorate our text button. Otherwise the text button will not catch eyes.

We can use text buttons with other inline contents, however, we should differentiate that button with the help of padding so user can notice its presence.

Since a text button doesn't have any visual border, we need to decorate it.

To get an idea, we must have a look at two buttons.

We've not decorated the first one, but decorated the second one, so that we can differentiate between them. Moreover, we'll take a look at each code separately. In addition, one can get the full code snippet at this GitHub repository.

Let's consider the first text button's code. It's a simple one. As we've told,we've not decorated it.

```
TextButton(
            style: TextButton.styleFrom(
            textStyle: const TextStyle(fontSize: 30),
            ),
            onPressed: () {},
            child: const Text('TextButton Simple'),
        ),
```

Next, we'll take a look at the second Text button which we've decorated with gradient.

```
ClipRRect(
            borderRadius: BorderRadius.circular(4),
            child: Stack(
            children: <Widget>[
                Positioned.fill(
                child: Container(
                    padding: const EdgeInsets.all(15),
                    alignment: Alignment.center,
                    width: 350.00,
                    height: 350.00,
                    decoration: BoxDecoration(
                    color: Colors.blue,
                    border: Border.all(
                        color: Colors.red,
                        width: 2.0,
                        style: BorderStyle.solid,
                    ),
                    borderRadius:
```

```
                        const BorderRadius.all(Radius.circular(40.0)),
                    gradient: const LinearGradient(
                        begin: Alignment.centerLeft,
                        end: Alignment.centerRight,
                        colors: [
                        Colors.blue,
                        Colors.yellow,
                        ],
                    ),
                    ),
                ),
                ),
                TextButton(
                style: TextButton.styleFrom(
                    padding: const EdgeInsets.all(16.0),
                    primary: Colors.black,
                    textStyle: const TextStyle(fontSize: 30),
                ),
                onPressed: () {},
                child: const Text('TextButton Gradient'),
                ),
            ],
            ),
        ),
```

It's quite obvious that we've used different widgets to decorate our second text button. Otherwise we couldn't have made its presence obvious in our bare eyes.

Subsequently, we'll try to understand the difference between two text buttons.

The first text button takes the style including color from the defaultStyleOf. However, in the second one we have overridden that defaultStyleOf with our custom gradient style with its style parameter.

As a result, the second button reacts to touches by filling with the style's ButtonStyle.backgroundColor.

In the first case, the defaultStyleOf manages it.

Since a text button doesn't come with great style and decoration, it's good not to use it where it'll blend with other content, such as in the middle of a list of items.

However, we can add great style from with the help of static styleFrom method. It's a convenient way to create a text button ButtonStyle from simple values.

```
TextButton(
            style: TextButton.styleFrom(
            elevation: 40.0,
            backgroundColor: Colors.yellow,
            ),
            onPressed: () {},
            child: Text(
            'Press TextButton',
            style: TextStyle(
                fontFamily: 'Allison',
                fontSize: 50,
                color: Colors.black,
            ),
            ),
        ),
```

As a result, we can greatly modify the style of our Text button's appearance. Moreover, we can use our custom Google font Allison to give it extra edge over the other text buttons.

Consequently, there is no fear of blending with the inline text anymore. Our text buttons now visually attractive, and in addition, it makes its presence felt.

Let's take a look at the appearance of the third text button, now.

To sum up, we've covered material design text buttons from different angles. And hopefully at the end, we have an idea of how we can use text buttons in different ways.

What is outlined button in flutter?

- All related images with this section - https://sanjibsinha.com/outlined-button-flutter/[63]

The outlined button in flutter is basically a text button. But there are some differences.

The outlined button in flutter is basically a material design widget. Actually it's a TextButton with an outlined border.

We've an exhaustive tutorial on text button with reference to its style, border, color, and elevation. Moreover, we've discussed elevated button in our previous tutorial also.

Therefore, you if you had read those articles already, you'd find this tutorial on outlined button rather familiar.

For an example, we've displayed three outlined buttons in a column widget. So that we can distinguish each with the other.

[63] https://sanjibsinha.com/outlined-button-flutter/

Let's first take a look at the first two outlined buttons first.

Subsequently, let's take a look at the first outlined button's code.

```
OutlinedButton(
            onPressed: () {},
            child: const Text(
            'Click Me',
            style: TextStyle(
                fontSize: 30,
                color: Colors.red,
            ),
            ),
        ),
```

In fact, nothing fancy is there. If you're familiar with the text button, then you'll find this code quite similar.

And the second outlined button uses color gradient to make its presence felt. And, indeed, it's successful in its attempt.

As a result, the code is quite lengthy.

```
ClipRRect(
            borderRadius: BorderRadius.circular(4),
            child: Stack(
            children: <Widget>[
                Positioned.fill(
                child: Container(
                    padding: const EdgeInsets.all(15),
                    alignment: Alignment.center,
                    width: 350.00,
                    height: 350.00,
                    decoration: BoxDecoration(
                    color: Colors.blue,
                    border: Border.all(
                        color: Colors.red,
                        width: 2.0,
                        style: BorderStyle.solid,
                    ),
                    borderRadius:
                        const BorderRadius.all(Radius.circular(40.0)),
                    gradient: const LinearGradient(
                        begin: Alignment.centerLeft,
```

```
                        end: Alignment.centerRight,
                        colors: [
                        Colors.blue,
                        Colors.red,
                        ],
                    ),
                    ),
                ),
                ),
                OutlinedButton(
                onPressed: () {},
                child: const Text(
                    'Click Me',
                    style: TextStyle(
                    fontSize: 30,
                    color: Colors.white,
                    ),
                ),
                ),
            ],
            ),
        ),
```

We've used two other layout widgets like ClipRRect and Card so that we can decorate our outlined button.

However, in less code, we can decorate our outlined button quite easily.

To do that, we need to use the static styleFrom method.

```
OutlinedButton(
            style: OutlinedButton.styleFrom(
            primary: Colors.white,
            backgroundColor: Colors.red,
            side: BorderSide(
                color: const Color(4278190000),
            ),
            elevation: 40.0,
            ),
            onPressed: () {},
            child: Text(
            'Press OutlineButton',
            style: TextStyle(
                fontFamily: 'Allison',
```

```
                fontSize: 50,
            ),
            ),
        ),
```

As a result, we can distinguish an outlined button quite easily. For full code, please visit the respective GitHub repository.

The third outlined button is now quite distinguishable.

Now we need to remember a few things while we use an outlined button.

Firstly, the OutlinedButton widget passes two required parameters that we must know. One is child, and the other is onPressed.

Secondly, defaultStyleOf defines the default style of any outlined button.

And finally, the TextButton or ElevatedButton doesn't have a default ButtonStyle.side static method. It actually borders the outline.

Keeping these simple rules in mind will help us to deal with the outlined button in a better way.

How do you use the icon button flutter?

- All related images with this section - https://sanjibsinha.com/icon-button-flutter/[64]

IconButton is a handy tool in flutter. Using icon button in flutter, we can accomplish many tasks.

Icons represent many motifs and serve many purposes. Therefore, in flutter too, the icon button helps us to accomplish many tasks. Now it depends on what we need to accomplish with the icon button.

For an example, we can think of any IconButton that represents directions or locations. It can navigate to another page, or opens up a drawer.

Simply by touching the icon button, we can make the state change from off to on. Therefore, we can use that onPressed callback in our advantage.

As usual, the icon button is a material design widget. As a consequence we need to have MaterialApp widget as one of its ancestors. And, moreover, as a picture printed on a Material widget, an icon button reacts to touches by changing its color.

We use Icon buttons in the AppBar.actions field, but we can use it in many other places as well.

Any Icon button has an onPressed callback and to make it respond we should not make it null.

If we make the callback null, then the button will be disabled and will not react to touch.

Another important thing to remember, any icon button needs to be of minimum pixels in size, regardless of the actual iconSize. We need that to satisfy the touch target size requirements in the Material Design specification.

[64] https://sanjibsinha.com/icon-button-flutter/

How do you change the color of the IconButton in flutter?

Before we want to dig deep into this question, let's see two icon buttons in one ListView widget.

The first one is wrapped by an Ink widget to give it a background color.

```
Ink(
            decoration: const ShapeDecoration(
            color: Colors.redAccent,
            shape: CircleBorder(),
            ),
            child: IconButton(
            icon: const Icon(
                Icons.directions_transit,
            ),
            tooltip: 'Good, bad and ugly',
            iconSize: 50,
            onPressed: () {},
            ),
        ),
```

The second IconButton, although is not wrapped by any widget to add color, still can use its own various color parameters to display color, or even change its color.

Let's see the code snippet of the second icon button.

```
IconButton(
            icon: const Icon(Icons.h_mobiledata_outlined),
            tooltip: 'Hover and change color to green!',
            color: Colors.grey,
            highlightColor: Colors.red,
            hoverColor: Colors.green,
            focusColor: Colors.purple,
            splashColor: Colors.yellow,
            disabledColor: Colors.amber,
            iconSize: 48,
            onPressed: () {
            setState(() {
                _isPressed = !_isPressed;
            });
            },
        ),
```

Now, let's see the first image.

According to the code that the second icon button follows, it has a hoverColor parameter that changes to green.

Let's test and see whether it works or not.

As we can see, the color changes to green, and also it has a tooltip parameter, that displays on the screen.

Flutter allows us to give user's a rich experience so that a mobile application looks good,

How do you add text to icon button on flutter?

To add text to icon button, we can take various steps. One of the most common steps that we use is tooltip parameter.

However, if we go through the full code snippet of the above two icon buttons,we can find one interesting parameter that each icon button possesses.

The onPressed parameter.

How this parameter comes to our help?

In many ways.

Let's take a simple test.

Suppose we have a simple Boolean message that comes False by default.

Now through the onPressed parameter we can change its state and make it True. Even we can grab that change in one Text widget.

However, without going through the full code we cannot understand how we can achieve that feat.

```
import 'package:flutter/material.dart';

class IconExample extends StatelessWidget {
const IconExample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Iconbutton Example',
    home: IconButtonExample(),
    );
}
}

Color colorWhite = Colors.white;

```

```
class IconButtonExample extends StatefulWidget {
const IconButtonExample({Key? key}) : super(key: key);

@override
_IconButtonExampleState createState() => _IconButtonExampleState();
}

class _IconButtonExampleState extends State<IconButtonExample> {
bool _isPressed = false;
@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('IconButton Example'),
    ),
    body: ListView(
        children: <Widget>[
        Ink(
            decoration: const ShapeDecoration(
            color: Colors.redAccent,
            shape: CircleBorder(),
            ),
            child: IconButton(
            icon: const Icon(
                Icons.directions_transit,
            ),
            tooltip: 'Good, bad and ugly',
            iconSize: 50,
            onPressed: () {},
            ),
        ),
        const SizedBox(
            height: 20,
        ),
        IconButton(
            icon: const Icon(Icons.h_mobiledata_outlined),
            tooltip: 'Hover and change color to green!',
            color: Colors.grey,
            highlightColor: Colors.red,
            hoverColor: Colors.green,
            focusColor: Colors.purple,
            splashColor: Colors.yellow,
            disabledColor: Colors.amber,
```

```
            iconSize: 48,
            onPressed: () {
            setState(() {
                _isPressed = !_isPressed;
            });
            },
        ),
        const SizedBox(
            height: 20,
        ),
        Text(
            'It is pressed : $_isPressed',
            style: const TextStyle(
            fontSize: 30,
            ),
        ),
        ],
    ),
    );
}
}
// the full code snippet
```

Now, we can take a look and feel that each icon buttons are parts of Material designs, moreover, both have MaterialApp as one of its ancestors.

Firstly, we can tap the second icon and it changes its color to red. It happens because of its highlightColor property.

Now, we've added the Boolean value and to change its state, we can use the Text widget.

As a result, before pressing the second button, the message is like the following image.

However it changes its state once we press the second icon button.

Accordingly, the message below also changes.

To sum up, we can conclude that, icon buttons in flutter are very handy tools. They serve various purposes.

In addition we can use them to change the state also.

How do you use a TextField in Flutter?

- All related images with this section - https://sanjibsinha.com/textfield-flutter/[65]

[65] https://sanjibsinha.com/textfield-flutter/

TextField is the most commonly used material design text input widget. However, it has many interesting features.

Although there are different text input widget in flutter, yet the TextField is the most commonly used.

The TextField is a material design widget. Therefore, one of its ancestors should be MaterialApp.

By default, a TextField is decorated with an underline. In addition we can add a label, icon, inline hint text, and error text with the help of decoration property.

What does decoration do?

Let's see an image to get an idea.

By default, decoration property draws a divider below the text field as we can see in the above image.

Certainly, the role of the decoration property doesn't end here. We can control the other decoration parts.

For instance, we can add a label or an icon.

To get an idea, let's see the first code example of the above text field.

```
TextField(
                    decoration: InputDecoration(
                    border: OutlineInputBorder(),
                    labelText: 'ITEM',
                    suffixStyle: TextStyle(
                        fontSize: 50,
                        fontWeight: FontWeight.bold,
                    ),
                    ),
                    controller: titleController,
                ),
```

After the decoration the most important property or parameter of a TextField is controller.

Why controller is so important?

Let's try to understand the concept and role of text field.

Firstly, the TextField is a material design text input widget. As a result, user should be able to type text and add that inputs to a local or remote storage.

Naturally, the TextField in Flutter is the most commonly used text input widget that allows users to collect inputs from the keyboard into an app. Without the controller we cannot do that.

Secondly, controller is the most important property because it controls the selection, composing region, and finally, it observes the changes.

For example, take a look at the second image.

To submit data to our local storage, we need to declare controllers beforehand.

```
final titleController = TextEditingController();
final amountController = TextEditingController();
```

Meanwhile we have defined a data model also in our model sub-directory.

```
import 'package:flutter/foundation.dart';

class ExpenseList {
String id;
String title;
double amount;
DateTime date;

ExpenseList({
    @required this.id,
    @required this.title,
    @required this.amount,
    @required this.date,
});
}
```

Next, inside our flutter app, we've used a callback and pass the two text controllers.

```
TextButton(
                    onPressed: () {
                    addTaskAndAmount(
                        titleController.text,
                        double.parse(amountController.text),
                    );
                    },
                    child: Text(
                    'SUBMIT',
                    style: TextStyle(
                        fontSize: 25,
                        fontWeight: FontWeight.bold,
                    ),
                    ),
                ),
```

Now, we can define the required callback quite easily inside our stateful widget.

```
class _ExpenseFirstPageState extends State<ExpenseFirstPage> {
final List<ExpenseList> expenseList = [];

void addTaskAndAmount(String title, double amount) {
    final expense = ExpenseList(
    id: DateTime.now().toString(),
    title: title,
    amount: amount,
    date: DateTime.now(),
    );
    setState(() {
    expenseList.add(expense);
    });
}

void deleteExpenseList(String id) {
    setState(() {
    expenseList.removeWhere((element) => element.id == id);
    });
}
....
//code is incomplete for brevity, please visit the GitHub repository for full code
```

As a result, we can add the input data to our local storage and it will display automatically.

To sum up, we'll take a look at another important property of the TextField.

How can I get data from multiple TextField in flutter?

We know that the text field calls the onChanged callback whenever the user changes the text in the field.

However, there is another important property onSubmitted callback.

When a use has finished typing in the text field, by pressing the submit button as shown on the above code snippet, or pressing the soft keyboard, the text field immediately calls the onSubmitted callback.

To get an idea, we can take a look at the second text field in our example app.

```
TextField(
                    decoration: InputDecoration(
                    border: OutlineInputBorder(),
                    labelText: 'AMOUNT',
                    suffixStyle: TextStyle(
                        fontSize: 50,
                        fontWeight: FontWeight.bold,
                    ),
                    ),
                    controller: amountController,
                    onSubmitted: (String value) async {
                    await showDialog<void>(
                        context: context,
                        builder: (BuildContext context) {
                        return AlertDialog(
                            title: const Text('Thanks!'),
                            actions: <Widget>[
                            TextButton(
                                onPressed: () {
                                Navigator.pop(context);
                                },
                                child: const Text('OK'),
                            ),
                            ],
                        );
                        },
                    );
                    },
                ),
```

As a result, when the user is done typing and press the submit button, the onSubmitted callback fires a AlertDialog widget.

Consequently, we can see an alert on the screen.

Besides these properties of a text field, we can integrate the TextField into a Form with other FormField widgets. And to do that we can use TextFormField.

For most applications the onSubmitted callback will be sufficient for reacting to user input.

To sum up, we can have an idea of how the TextField widget works takes inputs from the users. Moreover, we can store that value with the help of text controller in our local storage.

How do you make a flutter card?

- All related images with this section - https://sanjibsinha.com/card-in-flutter/[66]

With the help of material design widget Card we can display related information and give users a rich experience.

According to the Flutter documentation a Card widget is a sheet of Material used to represent some related information.

Since a card widget represents some related information, let us represent information about our expense list that we've been building in our previous article.

Above all as a beginner the first question will pop up.

How does a card widget looks like?

How does this sheet of Material design layout widget keeps related information closer, so that it looks good and gives user a rich experience.

For example, firstly, let's see an image of flutter app that uses card widget.

As you can see in the above image, we've built an expense list flutter app, where in two text fields we submit data and it gets added to the local storage. And, after that, the data displays in Card widget below.

If you feel interested get the full code in respective GitHub repository.

Let's see, how we've displayed the related information using Card widget.

```
Card(
                    elevation: 10,
                    child: Row(
                        children: [
                        displayAmount(e),
                        displayTaskAndDate(e),
                        ],
                    ),
                    ),
```

As we see, inside Card widget we've used elevation parameter and gives a value 10.

Next, as child parameter we use a Row widget that places two custom widgets.

One is for displaying amount and the other is for displaying task and date.

[66] https://sanjibsinha.com/card-in-flutter/

Moreover, when we'll examine the code of each custom widget we'll find that they again have used Card widget for storing content and action of a single object.

Using Card widget is also very simple. For instance, here we've called Card constructor and then pass a Row widget as child parameter.

We should always remember that Card, is a wrapper class. In addition a Card is a special type, a Material of type MaterialType.card.

In another article we'll discuss this concept in detail.

As we see, there might be a confusion over what is the exact difference between a Container widget and a Card widget.

The main difference is the Card widget must have a material design as its ancestor.

The Container widget doesn't have that compulsion. Moreover, a Container widget can have a child Card inside it.

What is difference between card and container in flutter?

To understand this difference properly, let's see a code snippet that houses both Container and Card.

Subsequently, let's compare this code with the above image, so that we can understand this difference in a clear way.

```
// this is displayAmount(e),
Container displayAmount(ExpenseList e) {
return Container(
    margin: EdgeInsets.all(8),
    padding: EdgeInsets.all(8),
    decoration: BoxDecoration(
    color: Colors.yellow[100],
    border: Border.all(
        color: Colors.red,
        width: 5,
    ),
    ),
    child: Card(
    child: Text(
        '\$${e.amount}',
        style: TextStyle(
        fontSize: 20,
        fontWeight: FontWeight.bold,
        ),
    ),
```

```
    ),
);
}
```

In the above code a Container uses Card as its sub-class in the tree of widgets.

However, if we go up above the tree, we'll find that this Container is a sub-class or child of a Card widget.

```
Card(
                    elevation: 10,
                    child: Row(
                        children: [
                        displayAmount(e),
                        displayTaskAndDate(e),
                        ],
                    ),
                    ),
```

In the similar vein, we can see the display task and date widget code.

```
Column displayTaskAndDate(ExpenseList e) {
return Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: [
    Container(
        margin: EdgeInsets.all(5),
        padding: EdgeInsets.all(8),
        decoration: BoxDecoration(
        color: Colors.blue[100],
        border: Border.all(
            color: Colors.red,
            width: 5,
        ),
        ),
        child: Card(
        child: Text(
            '${e.title}',
            style: TextStyle(
                fontSize: 25,
                fontWeight: FontWeight.bold,
                backgroundColor: Colors.blue[100]),
        ),
```

```
        ),
    ),
    Container(
        margin: EdgeInsets.all(2),
        padding: EdgeInsets.all(5),
        child: Card(
        child: Text(
            DateFormat('yyyy/MM/dd').format(e.date),
            style: TextStyle(
            fontSize: 20,
            fontWeight: FontWeight.normal,
            fontStyle: FontStyle.italic,
            ),
        ),
        ),
    ),
    ],
);
}
```

In conclusion, we can say that a Container plays the same role just like in HTML “div”.

In web development, we wrap other content with the help of HTML “div”.

In Flutter Container widget allows us to wrap other widgets. On the contrary, Card is an implementation of material widgets.

We can not only control the display of related content, we can control the shape. We can make it rounded corners and add a shadow to it.

What is the grid view in Flutter?

- All related images with this section - https://sanjibsinha.com/grid-view-flutter/[67]

GridView is a widget that consists of a grid list consists of a repeated pattern of cells. However, it must have a MaterialApp as an ancestor.

A grid view in flutter is a material component widget that must have a MaterialApp as one of its ancestors.

Since this tutorial is intended for beginners, we’ll try to describe GridView as simple as possible.

As a graphical control element a grid view shows items in a tabular form. Consequently, we can say that GridView is a widget that consists of a grid list consists of a repeated pattern of cells, which are arrayed in a vertical and horizontal layout.

[67] https://sanjibsinha.com/grid-view-flutter/

To clarify, a GridView displays items in a two dimensional rows and columns. And GridView widget implements this material component.

How do we show a GridView?

As the name suggests, while GridView displays items in a grid we can also tap any item to see the detail of that item.

As a result, to show items in a grid, GridView uses many other widgets, such as Image, Text, Icon, etc.

To make GridView work we can use various constructors, such as, count, builder, custom and extent.

We'll learn the usage of GridView widget with each constructor, however, each method has some advantages and disadvantages.

For example, in most common cases, grid layouts use GridView.count constructor. But it creates a layout with a fixed number of tiles in the cross axis.

On the other hand, GridView.extent constructor creates a layout with tiles that have a maximum cross-axis extent.

Therefore, if we want to create a grid with a large number of children, we use GridView.builder constructor with either SliverGridDelegateWithFixedCrossAxisCount, or a SliverGridDelegateWithMaxCrossAxisExtent for the gridDelegate.

The main axis direction of a grid is the direction in which it scrolls, later in a separate article we'll learn the scrollDirection that helps GridView to scroll.

In this tutorial we'll concentrate on GridView.builder constructor only.

Firstly, let's see how we can display a list of items with the help of GridView.

Secondly, we'll see the code.

So the image first.

The above screen shows us a grid view that displays book items. After that, we can tap each item and see the detail page like the following image. GridView flutter detail page GridView flutter detail page

To make it happen, we have used many advanced concepts, such as, provider package, Navigator, List and Map and other layout widgets.

At present, we should concentrate on GridView only, so let's see how GridView.builder constructor works with SliverGridDelegateWithFixedCrossAxisCount.

Widget build(BuildContext context) { final productsData = Provider.of<Books>(context); final products = showFavs ? productsData.favoriteItems : productsData.items; return GridView.builder(padding: const EdgeInsets.all(10.0), itemCount: products.length, itemBuilder: (ctx, i) ⇒ ChangeNotifierProvider.value(// builder: (c) ⇒ products[i], value: products[i], child: const BookItem(),), gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 2, childAspectRatio:

3 / 2, crossAxisSpacing: 10, mainAxisSpacing: 10,),); } // the code is incomplete for brevity, for full code snippet please visit the GitHub repository

The GridView.builder constructor uses “itemBuilder” parameter that plays a major role in building the grid that displays the items.

Therefore, let’s take a look at the custom BookItem widget.

Widget build(BuildContext context) { final product = Provider.of<Book>(context, listen: false); return ClipRRect(borderRadius: BorderRadius.circular(10), child: GridTile(child: GestureDetector(onTap: () { Navigator.of(context).pushNamed(BookDetailScreen.routeName, arguments: product.id,); }, child: Image.network(product.imageUrl, fit: BoxFit.cover,),), footer: GridTileBar(backgroundColor: Colors.black87, leading: Consumer<Book>(builder: (ctx, product, _) ⇒ IconButton(icon: Icon(product.isFavorite ? Icons.favorite : Icons.favorite_border,), color: Theme.of(context).primaryColor, onPressed: () { product.toggleFavoriteStatus(); },),), title: Text(product.title, textAlign: TextAlign.center,), trailing: IconButton(icon: const Icon(Icons.shopping_cart,), onPressed: () {}, color: Theme.of(context).primaryColor,),),); } // the code is incomplete for brevity, for full code snippet please visit the GitHub repository

The GridView display is incomplete without the GridTile widget and GestureDetector widget. We’ll discuss them separately in another article.

What is GridView count in flutter?

- All related images with this section - https://sanjibsinha.com/gridview-count-flutter/[68]

GridView count is a GridView constructor that we often use in flutter to display a fixed list of items.

In our previous tutorial we’ve discussed how GridView widget in #flutter uses different constructors. However, each constructor serves the same purpose.

What is that?

Each GridView shows items in a tabular form.

To clarify, a GridView displays items in a two dimensional rows and columns. And GridView widget implements this material component.

Now, GridView.count constructor is no different. It serves the same purpose, although in a different way.

Let’s see the image first. Then we’ll try to understand how the code works.

What are the different types of GridView available in flutter?

Firstly, in most common cases, grid layouts use GridView.count constructor. But it creates a layout with a fixed number of tiles in the cross axis, as we see in the above image.

[68] https://sanjibsinha.com/gridview-count-flutter/

Secondly, to overcome this disadvantage, we use another GridView constructor. if we want to create a grid with a large number of children, we use GridView.builder constructor with either SliverGridDelegateWithFixedCrossAxisCount, or a SliverGridDelegateWithMaxCrossAxisExtent for the gridDelegate.

Finally, we'll see how we've used GridView.count constructor to display items as we see in the above image.

Let's see the code first.

```
import 'package:flutter/material.dart';

class GridviewCountExample extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: Text("Flutter GridView Count Constructor"),
        ),
        body: GridView.count(
        crossAxisCount: 3,
        crossAxisSpacing: 6.0,
        mainAxisSpacing: 10.0,
        children: List.generate(
            Books.length,
            (index) {
            return Center(
                child: SelectBook(
                book: Books[index],
                ),
            );
            },
        ),
        ),
    ),
    );
}
}

class Book {
const Book({
    required this.title,
    required this.icon,
```

```
});
final String title;
final IconData icon;
}

const List<Book> Books = const <Book>[
const Book(
    title: 'Home Decor Guide',
    icon: Icons.home,
),
const Book(
    title: 'City Guide Map',
    icon: Icons.map,
),
const Book(
    title: 'Phone Directory',
    icon: Icons.phone,
),
const Book(
    title: 'Camera Accessories',
    icon: Icons.camera_alt,
),
const Book(
    title: 'Car Setting Manual',
    icon: Icons.car_rental_outlined,
),
];

class SelectBook extends StatelessWidget {
const SelectBook({
    Key? key,
    required this.book,
}) : super(key: key);
final Book book;

@override
Widget build(BuildContext context) {
    final TextStyle? textStyle = TextStyle(
    fontFamily: 'Lato Bold',
    fontSize: 20,
    color: Colors.white,
    );
    return Card(
```

```
    color: Colors.red,
    child: Center(
        child: Column(
        crossAxisAlignment: CrossAxisAlignment.center,
        children: <Widget>[
            Expanded(
            child: Icon(
                book.icon,
                size: 50.0,
                color: textStyle!.color,
            ),
            ),
            Text(
            book.title,
            style: textStyle,
            ),
        ],
        ),
    ),
    );
}
}
```

As the code shows, we have created a fixed List of book items. The GridView.count constructor through its children parameter generates a List that returns the List index.

With the help of that index we can get the other Book properties and display them on the screen.

How do you create a grid list in flutter?

With the help of GridView.count constructor creating a grid list is quite easy.

Watch this part of code.

```
body: GridView.count(
        crossAxisCount: 3,
        crossAxisSpacing: 6.0,
        mainAxisSpacing: 10.0,
        children: List.generate(
            Books.length,
            (index) {
            return Center(
                child: SelectBook(
```

```
                book: Books[index],
                ),
            );
            },
        ),
        ),
    ),
```

We can see that GridView.count constructor has different parameters.

Moreover, we can create a grid list inside it. As we have already created a fixed list already, we can use that to display the items.

Consequently, we can see how different GridView.count parameters work.

In the above code, we've used GridView.count constructor parameter crossAxisCount.

The crossAxisCount specifies the number of columns in a grid view.

```
crossAxisCount: 3,
```

In our code, we've mentioned it as 3.

The next parameter we've used is crossAxisSpacing.

```
crossAxisSpacing: 6.0,
```

Each child widget, here we've used Icon and Text widgets to display the item properties, has pixels between them. The crossAxisSpacing specifies the pixels.

The same way we use the mainAxisSpacing to specify the number of pixels between each child widget listed in the main axis.

```
mainAxisSpacing: 10.0,
```

We've also seen the children parameter that returns a List. So we need to be careful about this when we return a List of items.

Sometimes, we need to map the list.

Besides, we have other useful parameters, such as, padding that specifies the space around the whole list of widgets.

On the other hand, the scrollDirection specifies the direction in which the items on GridView scrolls. By default, it scrolls in a vertical direction. We can also use the reverse parameter, which is a Boolean. If it is true, it will reverse the list in the opposite direction along the main axis.

What is GridView.extent in Flutter?

- All related images with this section - https://sanjibsinha.com/gridview-extent-flutter/[69]

GridView.extent constructor creates a scrollable, two dimensional array of widgets with tiles that each have a maximum cross-axis extent.

We have discussed GridView and GridView.count before. However, the GridView.extent constructor is also very interesting and we can use it for the same purpose.

What is the purpose?

To create a scrollable, two dimensional array of widgets with tiles that each have a maximum cross-axis extent.

We'll talk about the cross axis and main axis, in a minute. Before that let's learn what is the biggest advantage of GridView?

The greatest advantage of GridView widget is it lets us scroll its content.

How does it happen?

Like CustomScrollView and ListView, the GridView's parent class is the class, ScrollView. As a result, GridView can scroll its content.

Moreover, many parameters of GridView gets their stimulation from the parent class ScrollView.

Since GridView is a two dimensional array of widgets, so we can define cross axis and main axis quite easily.

The cross axis represents the column, we can also say the horizontal part of the tiles. On the contrary, the main axis represents the rows down below the vertical side.

Similarly, we can also say that GridView.extent constructor has one property, which is required. If we look at the implementation of the parameters, we'll have an idea.

GridView.extent(

```
{Key? key,
Axis scrollDirection = Axis.vertical,
bool reverse = false,
ScrollController? controller,
bool? primary,
ScrollPhysics? physics,
bool shrinkWrap = false,
EdgeInsetsGeometry? padding,
required double maxCrossAxisExtent,
double mainAxisSpacing = 0.0,
```

[69] https://sanjibsinha.com/gridview-extent-flutter/

```
double crossAxisSpacing = 0.0,
double childAspectRatio = 1.0,
bool addAutomaticKeepAlives = true,
bool addRepaintBoundaries = true,
bool addSemanticIndexes = true,
double? cacheExtent,
List<Widget> children = const <Widget>[],
int? semanticChildCount,
DragStartBehavior dragStartBehavior = DragStartBehavior.start,
ScrollViewKeyboardDismissBehavior keyboardDismissBehavior = ScrollViewKeyboardDismis\
sBehavior.manual,
String? restorationId,
Clip clipBehavior = Clip.hardEdge}
```

)

The named parameter maxCrossAxisExtent plays a key role in displaying the items.

required double maxCrossAxisExtent,

If maxCrossAxisExtent is 400, we cannot keep two columns side by side. Because that will take almost all of the horizontal screen space.

Therefore we need to be careful about make it 200, so that we can at least place two columns side by side.

To get an idea, let's see an image of the upper part of GridView.extent constructor.

Let's look at the full code snippet, so that we can have a better idea.

After that, we'll discuss the code.

```
import 'package:flutter/material.dart';
import 'package:flutter/widgets.dart';

class GridViewExtent extends StatelessWidget {
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: Text("Flutter GridView Demo"),
        backgroundColor: Colors.green,
        ),
        body: Center(
        child: GridView.extent(
```

```
            primary: false,
            padding: const EdgeInsets.all(16),
            crossAxisSpacing: 10,
            mainAxisSpacing: 10,
            maxCrossAxisExtent: 200.0,
            children: <Widget>[
            Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(8),
                width: 250,
                height: 250,
                child: Text(
                'The Beginning',
                style: TextStyle(
                    backgroundColor: Colors.amberAccent,
                    fontSize: 30,
                    fontFamily: 'Anton',
                    fontWeight: FontWeight.bold,
                ),
                ),
            ),
            Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(8),
                child: Image.network(
                    'https://cdn.pixabay.com/photo/2015/09/05/07/28/writing-923882_9\
60_720.jpg'),
            ),
            Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(8),
                child: Image.network(
                    'https://cdn.pixabay.com/photo/2015/11/19/21/14/glasses-1052023_\
960_720.jpg'),
            ),
            Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(8),
                child: Image.network(
                    'https://cdn.pixabay.com/photo/2015/09/05/21/51/reading-925589_9\
60_720.jpg'),
            ),
            Container(
```

```
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(8),
                child: Image.network(
                    'https://cdn.pixabay.com/photo/2015/11/19/21/10/glasses-1052010_\
960_720.jpg'),
            ),
            Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(8),
                child: Image.network(
                    'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960\
_720.jpg'),
            ),
            Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(8),
                child: Image.network(
                    'https://cdn.pixabay.com/photo/2014/09/05/18/32/old-books-436498\
_960_720.jpg'),
            ),
            Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(8),
                child: Image.network(
                    'https://cdn.pixabay.com/photo/2015/09/05/21/51/reading-925589_9\
60_720.jpg'),
            ),
            Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(8),
                child: Image.network(
                    'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960\
_720.jpg'),
            ),
            Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(8),
                width: 250,
                height: 250,
                child: Text(
                'The End',
                style: TextStyle(
                    backgroundColor: Colors.lightBlueAccent,
```

```
                    fontSize: 30,
                    fontFamily: 'Anton',
                    fontWeight: FontWeight.bold,
                ),
                ),
            ),
            ],
        ),
        ),
    ),
    );
}
}
```

Especially this part of the code needs our attention.

```
child: GridView.extent(
            primary: false,
            padding: const EdgeInsets.all(16),
            crossAxisSpacing: 10,
            mainAxisSpacing: 10,
            maxCrossAxisExtent: 200.0,
....
```

There are many other named parameters, which at present we have not used.

Certainly, we can scroll down and see the end row, like the following screenshot.

To sum up, we can say that a graphical control element a grid view shows items in a tabular form. Consequently, we can say that GridView is a widget that consists of a grid list consists of a repeated pattern of cells, which are arrayed in a vertical and horizontal layout.

Moreover, a GridView displays items in a two dimensional rows and columns. And GridView widget implements this material component.

How do you use chip in flutter?

- All related images with this section - https://sanjibsinha.com/chip-flutter/[70]

Want to give users to choose any entity from a list of blocks? It could be locations, rules, etc. Chip is the answer.

[70] https://sanjibsinha.com/chip-flutter/

In this quick and short tutorial we'll see how we can use chip in flutter. Chip is a material design component widget.

Flutter Chips represent complex entities in small blocks, and that starts from locations, rules to contacts, or something else.

There are different types of Chip widgets available in flutter material design component library.

Before digging further, let's check what are the different types of Chips that are available in Flutter. In addition, we'll also learn what are their roles.

Certainly, in separate articles we'll discuss those Chip widgets in detail.

There is InputChip that represents various piece of information, such as person, place, or conversational text.

Subsequently, we can name FilterChip. As the name suggests, we can use FilterChip as tags or descriptive words as a way to filter contents.

On the contrary, ChoiceChip allows us to select from a set of options. As a result, those options contain related text, or categories.

Finally, we can think of ActionChip that represents an action related to the content displayed on the Chip.

Enough talking, let us first see a simple list of Chips on our flutter app.

As we can see that Chips are compact elements that represent an attribute, text, etc. However, it has another interesting parameter that we can use to delete any particular chip from the list.

Before discussing that parameter, let's see the full code snippet first.

```
import 'package:flutter/material.dart';

class WrapExample extends StatelessWidget {
const WrapExample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    debugShowCheckedModeBanner: false,
    title: '',
    home: WrapHome(),
    );
}
}

class WrapHome extends StatelessWidget {
const WrapHome({Key? key}) : super(key: key);
```

```
18
19  @override
20  Widget build(BuildContext context) {
21      return Scaffold(
22      appBar: AppBar(
23          title: Text('Wrap Example'),
24      ),
25      body: Center(
26          child: Wrap(
27          spacing: 18.0, // gap between adjacent chips
28          runSpacing: 14.0, // gap between lines
29          children: <Widget>[
30              Container(
31              margin: const EdgeInsets.all(20),
32              padding: const EdgeInsets.all(20),
33              child: Text(
34                  'Choose Locations',
35                  style: TextStyle(
36                  fontFamily: 'Allison',
37                  fontSize: 60,
38                  fontWeight: FontWeight.bold,
39                  ),
40              ),
41              ),
42              const SizedBox(
43              height: 10,
44              ),
45              Chip(
46              avatar: CircleAvatar(
47                  backgroundColor: Colors.blue.shade900,
48                  child: const Text('WB'),
49              ),
50              label: const Text(
51                  'West Bengal',
52                  style: TextStyle(
53                  fontSize: 20,
54                  fontWeight: FontWeight.bold,
55                  ),
56              ),
57              shadowColor: Colors.black,
58              elevation: 20,
59              onDeleted: () {},
60              ),
```

```
61              Chip(
62              avatar: CircleAvatar(
63                  backgroundColor: Colors.blue.shade900,
64                  child: const Text('MH'),
65              ),
66              label: const Text(
67                  'Maharastra',
68                  style: TextStyle(
69                  fontSize: 20,
70                  fontWeight: FontWeight.bold,
71                  ),
72              ),
73              shadowColor: Colors.black,
74              elevation: 20,
75              ),
76              Chip(
77              avatar: CircleAvatar(
78                  backgroundColor: Colors.blue.shade900,
79                  child: const Text('TM'),
80              ),
81              label: const Text(
82                  'Tamilnadu',
83                  style: TextStyle(
84                  fontSize: 20,
85                  fontWeight: FontWeight.bold,
86                  ),
87              ),
88              shadowColor: Colors.black,
89              elevation: 20,
90              ),
91              Chip(
92              avatar: CircleAvatar(
93                  backgroundColor: Colors.blue.shade900,
94                  child: const Text('AP'),
95              ),
96              label: const Text(
97                  'Andhra Pradesh',
98                  style: TextStyle(
99                  fontSize: 20,
100                 fontWeight: FontWeight.bold,
101                 ),
102             ),
103             shadowColor: Colors.black,
```

```
            elevation: 20,
            ),
        ],
        ),
    ),
    );
}
}
```

The above image shows us we are asking user to choose any locations from a given set of options. However, user cannot delete any one of them.

Right?

To solve that, let's first check one Chip code snippet first.

```
Chip(
            avatar: CircleAvatar(
                backgroundColor: Colors.blue.shade900,
                child: const Text('WB'),
            ),
            label: const Text(
                'West Bengal',
                style: TextStyle(
                fontSize: 20,
                fontWeight: FontWeight.bold,
                ),
            ),
            shadowColor: Colors.black,
            elevation: 20,
            ),
```

This Chip represents the very first location. But it has no option that allows users to delete it.

To make it happen, let's add a non-null onDeleted callback that will cause the chip to include a button for deleting the chip.

Watch the code below, we've added a non-null onDeleted callback with the first code.

```
Chip(
            avatar: CircleAvatar(
                backgroundColor: Colors.blue.shade900,
                child: const Text('WB'),
            ),
            label: const Text(
                'West Bengal',
                style: TextStyle(
                fontSize: 20,
                fontWeight: FontWeight.bold,
                ),
            ),
            shadowColor: Colors.black,
            elevation: 20,
            onDeleted: () {},
            ),
```

As a result, the first Chip has a delete button added next to the text.

To sum up, the ancestors of any Chip widget must have many material component widgets that include Material, MediaQuery, Directionality, and MaterialLocalizations.

You have guessed it right. All of these widgets are provided by MaterialApp and Scaffold. And remember that two parameters of any Chip Widget could never be null, they are label and clipBehavior arguments.

13. Slivers and Scrolling Widgets

When we delve deep into Flutter user interface, we find some interesting widgets that help users to scroll either vertically or horizontally.

Moreover, we can add more functionalities to those scrolling effects with the help of Transfom widget.

With reference to those magical effects in our Android and iOS App, let's take a deep look at those special widgets.

- All related code snippets with this section - https://github.com/sanjibsinha/flutter_artisan/tree/-main/lib[71]

What is SliverAppBar in flutter?

- All related images with this section - https://sanjibsinha.com/silver-app-bar-flutter/[72]

SliverAppBar is a special type of AppBar in flutter that can change its appearance and collapses as we scroll up or down.

SliverAppBar is a material design component widget, which must have a MaterialApp as its ancestors.

However, the name suggests that it is a kind of AppBar.

Then what it is exactly? And how do we use a SliverAppBar?

Here, the word sliver is important, because it defines a scrollable area in the AppBar, which has the ability to collapse.

As a result, we can conclude that SliverAppBar is a kind of AppBar that can change with body of our flutter app.

To make it more clear, we can add that this particular AppBar can change its appearance, if we scroll up it can shrink and gets smaller in size, even it can disappear. And, on the contrary, it can also gets to its normal size by blending with the body, as we scroll down again.

As a result, we can use an image as its background. Since it occupies the upper part, when we scroll up or down, it plays the role of a navigation bar in iOS and acts as a toolbar in Android app.

[71] https://github.com/sanjibsinha/flutter_artisan/tree/main/lib
[72] https://sanjibsinha.com/silver-app-bar-flutter/

Let's see two images one after the other, so that we can have a more clear picture of what is SliverAppBar and how it works.

Now, if we scroll up to see the bottom part, the image disappears and the text only remains. The SliverAppBar collapses.

To make it simple, we will take a look at the full code where we'll show how SliverAppBar integrates with a CustomScrollView.

```
import 'package:flutter/material.dart';

class SliverAppBarExample extends StatelessWidget {
const SliverAppBarExample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Sliver AppBar Example',
    home: SliverAppBarHome(),
    );
}
}

class SliverAppBarHome extends StatelessWidget {
const SliverAppBarHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    body: CustomScrollView(
        slivers: [
        SliverAppBar(
            expandedHeight: 200.0,
            floating: false,
            pinned: true,
            flexibleSpace: FlexibleSpaceBar(
            centerTitle: true,
            title: Text(
                'It will collapse',
                style: TextStyle(
                color: Colors.white,
                fontSize: 16.0,
                ),
            ),
```

```
            background: Image.network(
                'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960_720\
.jpg',
                fit: BoxFit.cover,
            ),
            ),
        ),
        SliverFixedExtentList(
            itemExtent: 50,
            delegate: SliverChildListDelegate([
            Container(color: Colors.red),
            Container(color: Colors.green),
            Container(color: Colors.blue),
            Container(color: Colors.red),
            Container(color: Colors.green),
            Container(color: Colors.blue),
            Container(color: Colors.red),
            Container(color: Colors.green),
            Container(color: Colors.blue),
            Container(color: Colors.red),
            Container(color: Colors.green),
            Container(color: Colors.blue),
            ]),
        ),
        ],
    ),
    );
}
}
```

The Custom scroll view has the slivers parameter that returns a list of widgets.

```
CustomScrollView(
        slivers: [
        SliverAppBar(
            expandedHeight: 200.0,
            floating: false,
            pinned: true,
            flexibleSpace: FlexibleSpaceBar(
...
```

As a result we have added a sliver fixed extent list widget and used a combination of colors.

In our next tutorial we'll show how we this special collapsible AppBar can act as a toolbar or any other widgets, such as a TabBar and a FlexibleSpaceBar.

How do I make my collapsing toolbar flutter?

- All related images with this section - https://sanjibsinha.com/sliverappbar-with-nestedscrollview/[73]

To make SliverAppBar behave like a collapsing toolbar in flutter we need to configure NestedScrollView.

We've already learned in our previous article on SliverAppBar that SliverAppBar is a material design component widget, which must have a MaterialApp as its ancestors.

However, to make a collapsing toolbar in Flutter, we need to use SliverAppBar along with a NestedScrollView.

Why do we need to do that?

Because we want our outer and inner sliver should work in one single scroll view, such as a CustomScrollView.

With reference to the above statement we must also know what does sliver mean in Flutter.

Let's try to understand that concept first. Suppose we have a scrollable area where a portion must behave in a special way. In that case, sliver is that portion.

The name SliverAppBar comes from that concept. To make an AppBar behave in a special way, flutter has invented SliverAppBar that acts as a collapsing toolbar.

As a result, our special AppBar becomes scrollable also. As we scroll up it collapses and blends with the inner scrollable body.

However, to make that happen, the headerSliverBuilder of a NestedScrollView may require special configuration. So that the outer and the inner act as one single scroll view.

To get an idea, let us see the first screenshot of our flutter app.

We've used SliverAppBar in the outer scroll view.

Moreover, we have made the pinned parameter of SliverAppBar true.

Why did we do that?

Because a pinned SliverAppBar works in a NestedScrollView exactly as it would in another scroll view, like CustomScrollView.

Although the SliverAppBar remains at the outer, but it never disappears when we scroll up the inner portion.

[73] https://sanjibsinha.com/sliverappbar-with-nestedscrollview/

The SliverAppBar.pinned, the Boolean parameter of SliverAppBar can still expand and contract as the user scrolls. Subsequently it works as a collapsing toolbar in flutter.

In addition, it will remain visible and behaves in a special manner when user scrolls up and down. It has never scrolled out of view.

The SliverAppBar always remains in the visible portion of the viewport.

Certainly, it would not be possible, if we had not configured headerSliverBuilder of a NestedScrollView.

Consequently, we must discuss NestedScrollView in the next article, and that we'll certainly do. So stay tuned.

And, finally, let's see how we've configured headerSliverBuilder of a NestedScrollView in our code.

```
import 'package:flutter/material.dart';

class SliverNestedExample extends StatelessWidget {
const SliverNestedExample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Sliver and Nested Scroll',
    home: NestedSliverHome(),
    );
}
}

class NestedSliverHome extends StatelessWidget {
const NestedSliverHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    body: NestedScrollView(
        headerSliverBuilder: (BuildContext context, bool innerBoxIsScrolled) {
        return <Widget>[
            SliverAppBar(
            expandedHeight: 200.0,
            floating: false,
            pinned: true,
            flexibleSpace: FlexibleSpaceBar(
                centerTitle: true,
                title: Container(
```

```
                color: Colors.blue,
                child: Text(
                    "The Toolbar Collapses",
                    style: TextStyle(
                    color: Colors.white,
                    fontSize: 25.0,
                    ),
                ),
                ),
                background: Image.network(
                'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960_720\
.jpg',
                fit: BoxFit.cover,
                ),
            ),
            ),
        ];
        },
        body: ListView(
        children: [
            Container(
            margin: const EdgeInsets.all(5),
            padding: const EdgeInsets.all(5),
            width: 250,
            height: 250,
            color: Colors.red,
            ),
            Container(
            margin: const EdgeInsets.all(5),
            padding: const EdgeInsets.all(5),
            width: 250,
            height: 250,
            color: Colors.yellow,
            ),
            Container(
            margin: const EdgeInsets.all(5),
            padding: const EdgeInsets.all(5),
            width: 250,
            height: 250,
            color: Colors.green,
            ),
            Container(
            margin: const EdgeInsets.all(5),
```

```
74              padding: const EdgeInsets.all(5),
75              width: 250,
76              height: 250,
77              color: Colors.pink,
78              ),
79              Container(
80              margin: const EdgeInsets.all(5),
81              padding: const EdgeInsets.all(5),
82              width: 250,
83              height: 250,
84              color: Colors.amber,
85              ),
86              Container(
87              margin: const EdgeInsets.all(5),
88              padding: const EdgeInsets.all(5),
89              width: 250,
90              height: 250,
91              color: Colors.teal,
92              ),
93          ],
94          ),
95      ),
96      );
97  }
98  }
```

This special configuration works naturally in a NestedScrollView, as the pinned SliverAppBar always remains in the visible portion of the viewport.

It perfectly acts as a collapsing toolbar in Flutter.

What is SliverGrid in flutter?

- All related images with this section - https://sanjibsinha.com/slivergrid-flutter/[74]

SliverGrid a sliver places multiple box children horizontally in a column and vertically in a row.

We're currently writing a few articles on flutter sliver series. As a result, we've already seen how flutter SliverAppBar works in a CustomScrollView. In addition we've also learned how to create a collapsing toolbar with the help of a NestedScrollView widget.

Considering the beginner's point of view let's us start with the concept of Sliver in flutter first.

[74] https://sanjibsinha.com/slivergrid-flutter/

Why?

Because the word sliver is always important, because it defines a scrollable area, either in the outer or in the inner space of a flutter app.

In case of SliverGrid a sliver places multiple box children in a two dimensional arrangement, which means horizontally in a column and vertically in a row.

The SliverGrid has been controlled by one parameter gridDelegate.

How does this property of SliverGrid work?

It places its children in arbitrary positions. However, each child has a specific size, which is determined by the gridDelegate.

How do you use Slivergrid Flutter?

Enough talking. Let's see the code first. After that, we'll discuss a few specific part of the code.

```
import 'package:flutter/material.dart';

class SliverGridExample extends StatelessWidget {
const SliverGridExample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Sliver and Nested Scroll',
    home: SliverGridHome(),
    );
}
}

class SliverGridHome extends StatelessWidget {
const SliverGridHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    body: NestedScrollView(
        headerSliverBuilder: (BuildContext context, bool innerBoxIsScrolled) {
        return <Widget>[
            SliverAppBar(
            expandedHeight: 200.0,
```

```
26              floating: false,
27              pinned: true,
28              flexibleSpace: FlexibleSpaceBar(
29                  centerTitle: true,
30                  title: Container(
31                  color: Colors.blue,
32                  child: const Text(
33                      "The Toolbar Collapses",
34                      style: TextStyle(
35                      color: Colors.white,
36                      fontSize: 25.0,
37                      ),
38                  ),
39                  ),
40                  background: Image.network(
41                  'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960_720\
42  .jpg',
43                  fit: BoxFit.cover,
44                  ),
45              ),
46              ),
47          ];
48          },
49          body: CustomScrollView(
50          slivers: <Widget>[
51              SliverGrid(
52              delegate: SliverChildBuilderDelegate(
53                  (context, index) {
54                  return Container(
55                      alignment: Alignment.center,
56                      color: Colors.orange[100 * (index % 9)],
57                      child: Text('grid item $index'),
58                  );
59                  },
60                  childCount: 30,
61              ),
62              gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
63                  crossAxisCount: 3,
64                  mainAxisSpacing: 15,
65                  crossAxisSpacing: 15,
66                  childAspectRatio: 2.0,
67              ),
68              ),
```

```
69          ],
70          ),
71      ),
72      );
73  }
74  }
```

In the above code, there are two distinct parts.

The ancestor is obviously the MaterialApp. Then, we've used NestedScrollView widget as the body parameter of the Scaffold widget.

But, we didn't mention AppBar, which naturally comes to mind when we define any Scaffold. Because we wanted that our AppBar should work as a collapsing toolbar, we've used SliverAppBar.

```
1   SliverAppBar(
2               expandedHeight: 200.0,
3               floating: false,
4               pinned: true,
5               flexibleSpace: FlexibleSpaceBar(
6                   centerTitle: true,
7                   title: Container(
8                   color: Colors.blue,
9                   child: const Text(
10                      "The Toolbar Collapses",
11                      style: TextStyle(
12                      color: Colors.white,
13                      fontSize: 25.0,
14                      ),
15                  ),
16                  ),
17                  background: Image.network(
18                  'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960_720\
19  .jpg',
20                  fit: BoxFit.cover,
21                  ),
22              ),
23              ),
```

That defines the look of the flutter app like the following screnshot.

In the outer space resides the SliverAppBar, however, in the inner space resides the SliverGrid with two parameters. One is delegate and the other is gridDelegate.

```
SliverGrid(
            delegate: SliverChildBuilderDelegate(
                (context, index) {
                return Container(
                    alignment: Alignment.center,
                    color: Colors.orange[100 * (index % 9)],
                    child: Text('grid item $index'),
                );
                },
                childCount: 30,
            ),
            gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
                crossAxisCount: 3,
                mainAxisSpacing: 15,
                crossAxisSpacing: 15,
                childAspectRatio: 2.0,
            ),
            ),
```

The delegate parameter defines the number of children by the childCount parameter. In the above code, we've mentioned it as 30.

Therefore, as we scroll down, two events take place.

Firstly, the SliverAppBar starts acting as a collapsing toolbar.

Secondly, the child count increases.

The toolbar image shrinks and gets shorter. And down below, the grid item shows the number 20.

If we move further down, the toolbar will collapse fully, however, it'll not disappear. And the grid child count shows the last number 29.

What is gridDelegate Flutter?

As we've told before, the gridDelegate parameter of SliverGrid plays a crucial role. We can always always create our own delegate class, however, using the gridDelegate directly we can control the layout of the children within SliverGrid.

```
gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
                crossAxisCount: 3,
                mainAxisSpacing: 15,
                crossAxisSpacing: 15,
                childAspectRatio: 2.0,
            ),
```

The gridDelegate has other functions in the GridView. However, in SliverGrid it acts as a delegate that controls the size and position of the children.

SliverPersistentHeader Flutter, a sliver whose size varies

- All related images with this section - https://sanjibsinha.com/sliverpersistentheader-flutter/[75]

We need this slivers widget when the SliverAppBar completely disappears. .

SliverPersistentHeader is a sliver whose size varies when we scroll down.

Keeping the above statement in mind, when we scroll to the edge of the viewport, which is opposite to the sliver's GrowthDirection, the SliverPersistentHeader doesn't disappear. On the contrary, it adjusts its size when we scroll.

Granted, we can have the same effect with SliverAppBar, and we can make it act like a collapsible toolbar. But, when the CustomScrollView has no centred sliver that changes its size with each scroll, SliverPersistentHeader rescues us.

Even the SliverAppBar may disappear, but SliverPersistentHeader remains in the inner space and adjusts its size.

That's the reason, why we use SliverPersistentHeader in our Flutter app.

Let us take a look at the first screenshot.

In the above image, the outer space is covered by SliverAppBar. However, under that special AppBar, we have our two SliverPersistentHeader defined.

[75] https://sanjibsinha.com/sliverpersistentheader-flutter/

```
return CustomScrollView(
    slivers: [
        SliverAppBar(
        title: const Text(
            'SliverPersistentHeader Sample',
            style: TextStyle(
            fontFamily: 'Allison',
            fontSize: 60,
            fontWeight: FontWeight.bold,
            ),
        ),
        backgroundColor: Colors.deepPurple,
        expandedHeight: 200,
        flexibleSpace: FlexibleSpaceBar(
            background: Image.network(
            'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960_720.jpg\
',
            fit: BoxFit.cover,
            ),
        ),
        ),
        const ACustomSliverHeader(
        backgroundColor: Colors.amber,
        headerTitle: 'First Sliver Persistent Header Sample',
        ),
        const ACustomSliverHeader(
        backgroundColor: Colors.green,
        headerTitle: 'Second Sliver Persistent Header Sample',
        ),
...
```

Next, we have created two custom Sliver persistent header widgets in this way.

```
class ACustomSliverHeader extends StatelessWidget {
final Color backgroundColor;
final String headerTitle;
// {Key? key}) : super(key: key);
const ACustomSliverHeader({
    Key? key,
    required this.backgroundColor,
    required this.headerTitle,
}) : super(key: key);

```

```
@override
Widget build(BuildContext context) {
    return SliverPersistentHeader(
    pinned: true,
    floating: false,
    delegate: Delegate(backgroundColor, headerTitle),
    );
}
}
```

The SliverPersistentHeader has a parameter called delegate.

It appears that we have created our own Delegate class that extends the abstract class SliverPersistentHeaderDelegate.

It is something that we can do in this way:

```
class Delegate extends SliverPersistentHeaderDelegate {
final Color backgroundColor;
final String headerTitle;

Delegate(this.backgroundColor, this.headerTitle);

@override
Widget build(
    BuildContext context, double shrinkOffset, bool overlapsContent) {
    return Container(
    color: backgroundColor,
    child: Center(
        child: Text(
        headerTitle,
        style: const TextStyle(
            color: Colors.black,
            fontSize: 20,
        ),
        ),
    ),
    );
}

@override
double get maxExtent => 150;

@override
```

```
28  double get minExtent => 60;
29
30  @override
31  bool shouldRebuild(SliverPersistentHeaderDelegate oldDelegate) {
32      return true;
33  }
```

In a separate article we'll take a detailed look at how we can create our own delegate class that we can use in different slivers.

Since, we have displayed part of code for brevity, you can get the full code at the respective GitHub repository.

Now, as a result, we can scroll down to see the sliver eggect.

As the SliverAppBar starts disappearing, the Sliver persistent header also starts adjusting its size.

Finally, when we have scrolled to the edge of the viewport inverse the sliver's GrowthDirection, the SliverAppBar completely disappears and the SliverPersistentHeader shrinks and fist to the screen.

To sum up, not only we've put the sliver inside a CustomScrollView, we've also put the the other sliver to make our custom scrollable region.

That's why, we've put SliverGrid below two custom SliverPersistentHeader.

```
1   SliverGrid(
2           delegate: SliverChildBuilderDelegate(
3               (context, index) {
4               return Container(
5                   alignment: Alignment.center,
6                   color: Colors.orange[100 * (index % 9)],
7                   child: Text(
8                   'grid item $index',
9                   style: const TextStyle(
10                      fontSize: 15,
11                  ),
12                  ),
13              );
14              },
15              childCount: 30,
16          ),
17          gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
18              crossAxisCount: 3,
19              mainAxisSpacing: 15,
20              crossAxisSpacing: 15,
21              childAspectRatio: 2.0,
```

```
22          ),
23          ),
```

So when we scroll down entirely, the sliver grid item count reaches the number and sliver persistent header shrinks and fits the screen.

SliverPersistentHeader varies its size in flutter

How do you use slivers flutter?

- All related images with this section - https://sanjibsinha.com/slivers-flutter/[76]

In this article we've tried to sum up all sliver widgets in flutter trying to know how they work.

We've already learned different aspects of sliver flutter. However, since we want to tap every stone and see if there is something down below, let's dig further. Let's know how to use other slivers in flutter.

We've already learned what is SliverAppBar. In addition, we've learned how to make collapsing toolbar, what is sliver grid, and sliver persistent head whose size varies.

Now, as regards those sliver articles let's finish learning other sliver widgets.

What is sliver in flutter?

In Flutter, a sliver is a slice of a scrollable area that we can use to achieve custom scrolling behaviours. However, it makes more sense if we say, that scrollable area adjusts its size as we scroll up or down.

Let's see two separate screenshots, that will make sense.

After that, let's see another image where the SliverAppBar acts as a collapsing toolbar. To see the effect, let's scroll down completely.

As a result, the above SliverAppBar or toolbar in Android, has collapsed completely. That's the magic of slivers in flutter.

In this respect, let's try to understand one key concept. Flutter creates its layouts organising widgets in trees. That means when Flutter creates a new widget, the parent widget passes constraint information to its children and this process goes down. Reversely, it's also true. Sometimes we also lift our state up. However, what we really want to say is that, on the top of these all sliver widgets lie RenderObject that paints everything. The RenderObject implements RenderSliver class.

[76] https://sanjibsinha.com/slivers-flutter/

What is SliverFixedExtentList?

As the name suggests, the SliverFixedExtentList assumes a fixed extent for its children and places multiple box children with the same main axis extent in a linear array.

If we compare SliverFixedExtentList with SliverList, then we'll find that the former one is more efficient than SliverList.

Why does that happen?

It's because SliverFixedExtentList does not need to perform layout on its children to obtain their extent in the main axis.

Let's see a code snippet where these two Sliver widgets place their children. Moreover that also explains what we want to say.

```
SliverFixedExtentList(
        delegate: SliverChildListDelegate([
            Container(color: Colors.red),
            Container(color: Colors.green),
            Container(color: Colors.blue),
        ]),
        itemExtent: 50,
        ),
        SliverList(
        delegate: SliverChildBuilderDelegate(
            (context, index) {
            return Container(
                height: 50,
                alignment: Alignment.center,
                color: Colors.orange[100 * (index % 9)],
                child: Text('orange $index'),
            );
            },
            childCount: 9,
        ),
        ),
```

Incidentally, we can take a look at the screenshot of our app that paints these two sliver widgets.

By the way, just down below the SliverAppBar, we can see three children container widgets whose parent is SliverFixedExtentList. And after that right below, there are SliverList children.

Next, we can see how SliverToBoxAdapter works.

What is SliverToBoxAdapter in Flutter?

Contrary to those above RenderSliver children, the SliverToBoxAdapter is much simpler and it only contains a single box widget.

As we have been saying from the very beginning, slivers are specially designed and these widgets serve special purposes.

While talking off slivers we know that inside a CustomScrollView, we want to create custom scroll effects. However, sometimes we need a basic sliver that always stays in its place.

For instance, we can take a look at a simple code snippet that defines SliverToBoxAdapter.

```
SliverToBoxAdapter(
        child: Container(
            color: Colors.yellow,
            padding: const EdgeInsets.all(8.0),
            child: const Text('Grid Header', style: TextStyle(fontSize: 24)),
        ),
        ),
```

Next, we can take a look at the screenshot that helps us to visualise how it looks like.

Most importantly, we shouldn't use multiple SliverToBoxAdapter widgets to display multiple box widgets in a CustomScrollView.

On the contrary, we can use other sliver widgets, which are more efficient in instantiating their children that are actually visible while we scroll. And these sliver widgets are SliverList, SliverFixedExtentList, SliverPrototypeExtentList, or SliverGrid.

Finally, we'll use two SliverGrid constructors count and extent.

```
SliverGrid.count(
        crossAxisCount: 3,
        mainAxisSpacing: 10.0,
        crossAxisSpacing: 10.0,
        childAspectRatio: 4.0,
        children: <Widget>[
            Container(color: Colors.red),
            Container(color: Colors.green),
            Container(color: Colors.blue),
            Container(color: Colors.red),
            Container(color: Colors.green),
            Container(color: Colors.blue),
        ],
        ),
```

```
15          SliverGrid.extent(
16          maxCrossAxisExtent: 200,
17          mainAxisSpacing: 10.0,
18          crossAxisSpacing: 10.0,
19          childAspectRatio: 4.0,
20          children: <Widget>[
21              Container(color: Colors.pink),
22              Container(color: Colors.indigo),
23              Container(color: Colors.orange),
24              Container(color: Colors.pink),
25              Container(color: Colors.indigo),
26              Container(color: Colors.orange),
27          ],
28          ),
```

The SliverGrid.count constructor creates a sliver that places multiple box children in a two dimensional arrangement with a fixed number of tiles horizontally.

The SliverGrid.extent constructor, on the contrary, does the same thing, only with tiles that each have a maximum cross-axis extent.

The lower part of the above image shows two different fixed number of tiles. The first one displays red, green and blue; and the second one arranges tiles in pink, indigo and orange.

With reference to the above code snippets, we must admit that they are cut short for brevity. However, for all the sliver related code snippets, you may visit the relevant GitHub code repository.

To sum up, we've learned many things about different sliver widgets, although our understanding will not be complete unless we learn CustomScrollView and NestedScrollView.

Therefore, our next tutorials will discuss those useful flutter widgets separately.

How to use CustomScrollView in Flutter?

- All related images with this section - https://sanjibsinha.com/customscrollview-flutter/[77]

CustomScrollView in Flutter uses slivers using which we can create various scrolling effects.

Before we use CustomScrollView, let's know what it is. The CustomScrollView is a ScrollView that uses slivers. And the slivers create various scrolling effects.

How the slivers create various scrolling effects, we'll see in a minute.

Let's see the first screenshot.

[77] https://sanjibsinha.com/customscrollview-flutter/

If we check the code of this flutter app, we'll find that we've used AppBar and besides, we've also used SliverAppBar inside the CustomScrollView slivers. As a result, when we press the plus button in the AppBar, the items are being added in the inner space and as we scroll down, the SliverAppBar shrinks.

The AppBar remains at its place and, SliverAppBar starts working as a collapsing toolbar.

Let's go through the code from the very beginning, so that we can understand how the CustomScrollView works.

Firstly, we need to run the app.

```
import 'package:flutter/material.dart';
import 'custom_and_nested_scroll/custom_scroll_sample.dart';

import 'package:provider/provider.dart';
import 'models/counter.dart';
void main() {
runApp(

    MultiProvider(
    providers: [
        ChangeNotifierProvider(
        create: (_) => Counter(),
        ),
    ],

    child: CustomScrollSample(),
    ),
);
}
```

We've used the latest Provider package to maintain our state and increment the item's index.

```
import 'package:flutter/foundation.dart';

class Counter with ChangeNotifier {
int _count = 0;

int get count => _count;

void increment() {
    _count++;
    notifyListeners();
```

```
}
}
```

Next, we've used Scaffold and through the body parameter we use a CustomScrollView.

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '/models/counter.dart';

class CustomScrollSample extends StatelessWidget {
const CustomScrollSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'CustomScroll Sample',
    home: CustomScrollHome(),
    );
}
}

class CustomScrollHome extends StatelessWidget {
CustomScrollHome({Key? key}) : super(key: key);
List<int> top = <int>[];
List<int> bottom = <int>[0];

@override
Widget build(BuildContext context) {
    int index = context.watch<Counter>().count;
    const Key centerKey = ValueKey<String>('bottom-sliver-list');
    return Scaffold(
    appBar: AppBar(
        title: const Text('Press to add items above and below'),
        leading: IconButton(
        icon: const Icon(Icons.add),
        onPressed: () {
            top.add(-top.length - 1);
            bottom.add(bottom.length);
            context.read<Counter>().increment();
        },
        ),
    ),
    body: CustomScrollView(
```

```
39          slivers: [
40          SliverAppBar(
41              expandedHeight: 200.0,
42              floating: false,
43              pinned: true,
44              flexibleSpace: FlexibleSpaceBar(
45              centerTitle: true,
46              title: const Text(
47                  'It will collapse',
48                  style: TextStyle(
49                  color: Colors.white,
50                  fontSize: 16.0,
51                  ),
52              ),
53              background: Image.network(
54                  'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960_720\
55  .jpg',
56                  fit: BoxFit.cover,
57              ),
58              ),
59          ),
60          SliverList(
61              delegate: SliverChildBuilderDelegate(
62              (BuildContext context, int index) {
63                  return Container(
64                  alignment: Alignment.center,
65                  color: Colors.blue[200 + top[index] % 4 * 100],
66                  height: 100 + top[index] % 4 * 20.0,
67                  child: Text('Item: ${top[index]}'),
68                  );
69              },
70              childCount: top.length,
71              ),
72          ),
73          SliverList(
74              key: centerKey,
75              delegate: SliverChildBuilderDelegate(
76              (BuildContext context, int index) {
77                  return Container(
78                  alignment: Alignment.center,
79                  color: Colors.blue[200 + bottom[index] % 4 * 100],
80                  height: 100 + bottom[index] % 4 * 20.0,
81                  child: Text('Item: ${bottom[index]}'),
```

```
                );
            },
            childCount: bottom.length,
            ),
        ),
        ],
    ),
    );
}
}
```

As we see inside CustomScrollView slivers we've used two slivers, SliverAppBar, and SliverList. Both they are Widgets producing RenderSliver objects.

As a result, when we've pressed the plus button on the AppBar for several times and scroll down, it looks like the following image.

We've already learned what is SliverAppBar. In addition, we've learned how to make collapsing toolbar, what is sliver grid, and sliver persistent head whose size varies.

However, the CustomScrollView has a close relation with those slivers widgets.

How to use NestedScrollView in flutter?

- All related images with this section - https://sanjibsinha.com/nestedscrollview-flutter/[78]

NestedScrollView in flutter has many advantages when we plan to scroll up and down.

As the name suggests the NestedScrollView in flutter is nothing but a scrolling view at first sight. However, it has many advantages. And the main advantage is we can nest other scrolling views inside it.

Talking off scrolling views reminds us that in a common use case in any flutter app, we often use SliverAppBar containing a TabBar in the header and TabBarView in the body, inside NestedScrollView.

But we're not going to use that example here.

Because we have a plan to discuss TabBar and TabBarView in a separate flutter article. They deserve it.

Therefore let us concentrate on how we can use NestedScrollView widget in a different way. While doing so, we'll keep AppBar in our Scaffold widget, and SliverAppBar in NestedScrollView.

Besides we will also use CustomScrollView in the body of the NestedScrollView.

[78] https://sanjibsinha.com/nestedscrollview-flutter/

By the way, we must keep in mind that there are other scroll view widgets that help us to scroll. For instance, we can name ScrollView, which we'll definitely discuss in a separate post.

To start with, firstly let's get the full code snippet and start watching the screenshots, so that we can discuss the parts of the code along with the image of our flutter app.

```
import 'package:flutter/material.dart';

class NestedScrollViewSampleOne extends StatelessWidget {
const NestedScrollViewSampleOne({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'NestedScrollView first sample',
    home: NestedScrollViewFirstHome(),
    );
}
}

class NestedScrollViewFirstHome extends StatelessWidget {
const NestedScrollViewFirstHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text('NestedScrollView Sample'),
    ),
    body: NestedScrollView(
        headerSliverBuilder: (BuildContext context, bool innerBoxIsScrolled) {
        return <Widget>[
            SliverAppBar(
            expandedHeight: 200.0,
            //forceElevated: innerBoxIsScrolled,
            //floating: true,

            pinned: true,
            flexibleSpace: FlexibleSpaceBar(
                centerTitle: true,
                title: Container(
                color: Colors.blue,
                child: const Text(
```

```
                    "The Collapsing Toolbar",
                    style: TextStyle(
                    color: Colors.white,
                    fontSize: 25.0,
                    ),
                ),
                ),
                background: Image.network(
                'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960_720\
.jpg',
                fit: BoxFit.cover,
                ),
            ),
            ),
        ];
        },
        body: CustomScrollView(
        slivers: <Widget>[
            SliverGrid(
            delegate: SliverChildBuilderDelegate(
                (context, index) {
                return Container(
                    alignment: Alignment.center,
                    color: Colors.orange[100 * (index % 9)],
                    child: Text('grid item $index'),
                );
                },
                childCount: 30,
            ),
            gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
                crossAxisCount: 3,
                mainAxisSpacing: 15,
                crossAxisSpacing: 15,
                childAspectRatio: 2.0,
            ),
            ),
        ],
        ),
    ),
    );
}
}
```

In the above code we've set SliverAppBar pinned parameter true. As a result, we get these two screenshots when we run the app and after that when we scroll down to the down below.

Next, we scroll down to the end of the bottom screen and our SliverAppBar has collapsed totally, adjusts its size and the image disappears keeping only the text.

The main advantage of NestedScrollView is that the pinned SliverAppBar works in a NestedScrollView exactly as it would in another scroll view, like CustomScrollView.

When we use SliverAppBar.pinned true, the SliverAppBar remains visible at the top of the scroll view.

As the user scrolls, it contracts and adjusts its size and finally it remains at the top. Since in our code, we've also used AppBar, it remains just down below the AppBar.

To recognize that property, let's take a brief look at that part of the above code.

```
body: NestedScrollView(
        headerSliverBuilder: (BuildContext context, bool innerBoxIsScrolled) {
        return <Widget>[
            SliverAppBar(
            expandedHeight: 200.0,
            pinned: true,
            flexibleSpace: FlexibleSpaceBar(
                centerTitle: true,
                title: Container(
                color: Colors.blue,
                child: const Text(
                    "The Collapsing Toolbar",
                    style: TextStyle(
                    color: Colors.white,
                    fontSize: 25.0,
                    ),
                ),
                ),
                background: Image.network(
                'https://cdn.pixabay.com/photo/2016/09/10/17/18/book-1659717_960_720\
.jpg',
                fit: BoxFit.cover,
                ),
            ),
            ),
        ];
        },
```

As the user scrolls, the pinned SliverAppBar doesn't move out of the scroll view. However, to make

it scroll out of the viewport, we should make the floating parameter true.

```
body: NestedScrollView(
        headerSliverBuilder: (BuildContext context, bool innerBoxIsScrolled) {
        return <Widget>[
            SliverAppBar(
            expandedHeight: 200.0,
            forceElevated: innerBoxIsScrolled,
            floating: true,

            // pinned: true,
```

Since we've made the floating parameter true and pinned parameter has been commented out, the SliverAppBar shrinks as the user scrolls down further.

As a result, the SliverAppBar that acts as a collapsing toolbar is collapsing indeed and scrolling out of the view.

Finally, it will disappear completely as the user scrolls and reaches the bottom part of the screen.

Now, at the top, only the AppBar remains. The SliverAppBar completely disappears.

As we've noticed that when we use NestedScrollView headerSliverBuilder, we can use not only a SliverAppBar, but other slivers as well. Consequently, we can take advantage of this along with the body part where we can use CustomScrollView.

How to use PageView in Flutter

- All related images with this section - https://sanjibsinha.com/pageview-flutter/[79]

PageView is not just another Scrolling widget. It has many features that are simply amazing.

PageView belongs to the Scrolling widgets. Good news is , we've started writing on Scrolling widgets and we've already written about CustomScrollView and NestedScrollView.

As we've just said, PageView is just like any other Scrolling widgets, although this widget has its own pros and cons that include many amazing features. Moreover, it depends on how we'll use them.

Certainly, in some scenario, CustomScrollView or NestedScrollView works better than PageView. Therefore, we must take that into account and remember that the opposite is also true.

Besides, we need to understand Slivers. And, of course, some scrolling widgets use slivers in more efficient way.

[79] https://sanjibsinha.com/pageview-flutter/

Anyway, in this article, we'll cover the very basic concepts of PageView. And after that, in our next article, we'll cover more advanced features of PageView widget.

Firstly, page view widget is a scrollable list of pages, as the name suggests. Either vertically, or horizontally, we can view either infinite number of pages, or a fixed list of pages.

However, in most cases, we don't want infinite pages. And in this case, we have a simple code snippet that a beginner can understand. So let's see that first. Secondly, we'll discuss the code and see the screenshots also. And, finally, we'll take a look at how we can learn further.

```
import 'package:flutter/material.dart';

class PageViewSampleSimple extends StatelessWidget {
const PageViewSampleSimple({Key? key}) : super(key: key);

static const String _title = 'PageView Simple Sample';

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    debugShowCheckedModeBanner: false,
    title: _title,
    home: PageViewSampleSimpleHome(),
    );
}
}

class PageViewSampleSimpleHome extends StatelessWidget {
const PageViewSampleSimpleHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('PageView Simple Sample'),
    ),
    body: PageView(
        scrollDirection: Axis.vertical,
        children: <Widget>[
        Container(
            color: Colors.green,
            child: const Text(
            'Page One',
            style: TextStyle(
```

```
                fontFamily: 'Allison',
                fontSize: 60,
                fontWeight: FontWeight.bold,
            ),
            ),
        ),
        Container(
            color: Colors.yellow,
            child: const Text(
            'Page Two',
            style: TextStyle(
                fontFamily: 'Allison',
                fontSize: 60,
                fontWeight: FontWeight.bold,
            ),
            ),
        ),
        Container(
            color: Colors.deepPurple,
            child: const Text(
            'Page Three',
            style: TextStyle(
                fontFamily: 'Allison',
                fontSize: 60,
                fontWeight: FontWeight.bold,
            ),
            ),
        ),
        ],
    ),
    );
}
}
```

The code snippet shows, we have three pages as children. By default, each child is forced to be the same size as the viewport.

Our code shows that each child is a container filled with different colour.

The first page is Green.

Now, we can control the scroll direction and in our case, we've instructed that parameter to scroll vertically.

```
scrollDirection: Axis.vertical,
```

As a result, we can swipe our flutter app upwards and the second page slowly comes to take the centre stage. This page is yellow.

Let's scroll up more to find out the third page, which is deep purple.

Most importantly, this combination of code and images is pretty basic. But we've certainly understood the basic principle of PageView widget.

Now we can use a PageController and control the visibility of pages. It means, which page is visible in the view is controlled by PageController. It has other powers as well, and we'll discuss that in detail in the next article that covers PageView.builder constructor.

To sum up, a PageView widget generates scrollable pages on the screen. What we've seen is a fixed list of pages. If we had used PageView.builder, we would have generated repeating pages.

As a matter of fact, PageView acts in similar vein as ListView does. Since this article is a gentle introduction to PageView widget, let's know a few more details.

There are three types of PageView widgets. We've already seen one and heard the name of PageView.builder constructor. In addition, there are another PageView constructor PageView.custom.

We'll meet those PageView constructors in our next article and definitely say "Hello".

By the way, before concluding this article, we must know that every PageView widget either scroll vertically, or horizontally. However, there exists other parameters that may add special effects to that scroll. So the scroll effect looks like the page is scrolling diagonally. We'll learn those tricks in the coming article.

What is PageView builder in flutter?

- All related images with this section - https://sanjibsinha.com/pageview-builder-flutter/[80]

We can use PageView.builder constructor in flutter to add many custom effects to pages.

We've been doing a series of articles on Scrolling Widgets. As a result, we've already written an article on PageView. However, PageView has two other constructors that are also important and demand a place in this Scrolling series.

As we've just said, PageView is just like any other Scrolling widgets, although this widget has its own pros and cons that include many amazing features. Moreover, it depends on how we'll use them.

However, PageView.builder constructor plays also a crucial role and acts differently than PageView. PageView.builder creates a scrollable list that works page by page using widgets.

[80] https://sanjibsinha.com/pageview-builder-flutter/

When we need to show a large number of children, this PageView.builder constructor is incomparable. Moreover, the builder is called only for those children that are actually visible.

How much can we scroll?

With reference to that question, we can say PageView.builder controls that item count with the itemCount parameter. We cannot provide a null value.

As long as the indices are less than item count parameter, and greater than or equal to zero, PageView.builder constructor calls the parameter itemBuilder. We'll see that in a minute.

Before that, let's know a few other key information about PageView.builder constructor.

Firstly, by default, PageView.builder doesn't support child reordering. We can use either use PageView or PageView.custom constructor to do that.

Secondly, PageView.builder doesn't allow null value for the allowImplicitScrolling parameter.

Since PageView.builder is a constructor of PageView, which has StatefulWidget as its immediate ancestor, therefore, PageView.builder effects are not going to work with Stateless widget. Although we can always experiment with the Provider package.

Certainly, we can start with a stateless widget first and our code snippet is like the following one.

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '/models/counter.dart';

class PageViewBuilderSimple extends StatelessWidget {
const PageViewBuilderSimple({Key? key}) : super(key: key);

static const String _title = 'PageView Simple Sample';

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    debugShowCheckedModeBanner: false,
    title: _title,
    home: PageViewBuilderHome(),
    );
}
}

class PageViewBuilderHome extends StatelessWidget {
const PageViewBuilderHome({Key? key}) : super(key: key);

@override
```

```
Widget build(BuildContext context) {
    double thisPage = context.watch<Counter>().x;
    return Scaffold(
    appBar: AppBar(
        title: const Text('PageView Simple Sample'),
    ),
    body: PageView.builder(
        itemCount: 4,
        itemBuilder: (context, position) {
        Color color;
        if (position == thisPage.floor()) {
            color = Colors.pinkAccent;
            return Container(
            color: color,
            child: const Text(
                "First Page",
                style: TextStyle(
                color: Colors.white,
                fontSize: 120.0,
                fontFamily: 'Allison',
                ),
            ),
            );
        } else if (position == thisPage.floor() + 1) {
            color = Colors.blueAccent;
            return Container(
            color: color,
            child: const Text(
                "Second Page",
                style: TextStyle(
                color: Colors.white,
                fontSize: 120.0,
                fontFamily: 'Allison',
                ),
            ),
            );
        } else if (position == thisPage.floor() + 2) {
            color = Colors.deepOrangeAccent;
            return Container(
            color: color,
            child: const Text(
                "Third Page",
                style: TextStyle(
```

```
                color: Colors.white,
                fontSize: 120.0,
                fontFamily: 'Allison',
                ),
            ),
            );
        } else {
            color = Colors.greenAccent;
            return Container(
            color: color,
            child: const Text(
                "Fourth Page",
                style: TextStyle(
                color: Colors.white,
                fontSize: 120.0,
                fontFamily: 'Allison',
                ),
            ),
            );
        }
        },
    ),
    );
}
}
```

Before we run the code, let's understand that we have used provider package and using a data model where we've declared our current page double value equal to 0.0.

```
import 'package:flutter/foundation.dart';

class Counter with ChangeNotifier {

double x = 0.0;

void increase() {
    x = x + 1.0;
    notifyListeners();
}
}
```

After that, we've used this line of code to watch the value from our data model.

```
double thisPage = context.watch<Counter>().x;
```

Now, if we run the app, what we do we see?

Next, we can swipe the page to see other pages as expected. However, we've declared the item count parameter to 4. And we have 4 pages to display.

In addition, we could have added a special effect while swiping the pages, with the help of Transform widget.

In that case, when we'll swipe, it won't swipe horizontally; instead it has a special scroll effect with the help of Transform widget.

Certainly we'll learn more about this Transform widget in flutter and we'll also see how we can use Transform widget with PageView.builder constructor.

What is PageView custom in flutter?

- All related images with this section - https://sanjibsinha.com/pageview-custom-flutter/[81]

PageView custom constructor supplies children for slivers. That makes it interesting.

Since we've started writing on Scrolling widgets series, we've decided that we'll cover PageView custom constructor in a separate article. As a result, in this article we're going to learn what is PageView.custom constructor and, moreover, how we can use this PageView custom constructor in our flutter app.

More or less, as we progress, we'll be going to find that PageView.custom constructor has many similarities with PageView.builder constructor. However, if they had shared the same features just like a clone, the flutter creators wouldn't have created another constructor.

Therefore, they have differences and as well as they have some similarities.

Firstly, to be more precise, a PageView custom constructor creates a scrollable list of custom pages that a user can swipe and view. Secondly, with some tweak in our code, we can add some transforming effects to these pages. And finally, we can do so because PageView widgets has some parameters that are dedicated to help Transform widgets.

Now, a PageView custom constructor always allows us to create a child model that we can return as custom pages.

Therefore we can take a look at the full code snippet first.

[81] https://sanjibsinha.com/pageview-custom-flutter/

```dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '/models/counter.dart';

class PageViewCustomSimple extends StatelessWidget {
const PageViewCustomSimple({Key? key}) : super(key: key);

static const String _title = 'PageView Custom Sample';

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    debugShowCheckedModeBanner: false,
    title: _title,
    home: PageViewBuilderHome(),
    );
}
}

class PageViewBuilderHome extends StatelessWidget {
const PageViewBuilderHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    double thisPage = context.watch<Counter>().x;
    return Scaffold(
    appBar: AppBar(
        title: const Text('PageView Custom Sample'),
    ),
    body: PageView.custom(
        childrenDelegate: SliverChildBuilderDelegate(
        (BuildContext context, int index) {
            Color color;
            if (index == thisPage.floor()) {
            color = Colors.yellow;
            return Container(
                color: color,
                child: const Text(
                "First Page",
                style: TextStyle(
                    color: Colors.red,
                    fontSize: 120.0,
                    fontFamily: 'Allison',
```

```
                ),
                ),
            );
            } else if (index == thisPage.floor() + 1) {
            color = Colors.purple;
            return Container(
                color: color,
                child: const Text(
                "Second Page",
                style: TextStyle(
                    color: Colors.yellow,
                    fontSize: 120.0,
                    fontFamily: 'Allison',
                ),
                ),
            );
            } else if (index == thisPage.floor() + 2) {
            color = Colors.white;
            return Container(
                color: color,
                child: const Text(
                "Third Page",
                style: TextStyle(
                    color: Colors.black,
                    fontSize: 120.0,
                    fontFamily: 'Allison',
                ),
                ),
            );
            } else {
            color = Colors.black54;
            return Container(
                color: color,
                child: const Text(
                "Fourth Page",
                style: TextStyle(
                    color: Colors.white,
                    fontSize: 120.0,
                    fontFamily: 'Allison',
                ),
                ),
            );
            }
```

```
87          },
88          childCount: 4,
89          ),
90      ),
91      );
92  }
93  }
```

At the very end of our code, we find that childCount parameter is 4. PageView.custom constructor has a parameter called childrenDelegate. As the name suggests, we can pass a SliverChildBuilderDelegate class and supply children for slivers using a builder callback.

These children represent custom pages. And as a result, if a user swipes pages, she can view four custom pages in tow, following one another.

With reference to that, we can say that the child count could have been 10, instead of 4, and we could have made 10 pages.

Since in PageView custom constructor SliverChildBuilderDelegate class plays an important role and supplies children pages, so we can try to know this class closely.

Firstly, as the name points out quite precisely, this class belongs to flutter Slivers.

Secondly, it's fun to note that the SliverChildBuilderDelegate delegate provides children in a succinct way. When the user swipes and decides to go from the first page to view the second page, in actuality, until the second page is not displayed fully, it is not built.

How does it happen? The SliverChildBuilderDelegate delegate uses NullableIndexedWidgetBuilder callback.

We don't want to make the discussion more complex and scare the beginners. However, before closing down, let us know a few more important things about the PageView widget.

It has a parameter called allowImplicitScrolling. This parameter must not be null.

How to use DraggableScrollableSheet

This widget can be dragged along the vertical axis and we can scroll to view the contents.

We can use the DraggableScrollableSheet widget for many purposes. However, in this tutorial, we'll see how with the help of this widget, we can add a list of words, which will be displayed below on a sheet that we can scroll to view.

Before jumping in to the code and related screenshots, let's try to understand a few basic concepts about draggable scrollable sheet widget.

Certainly, we can compare this widget with CustomScrollView, NestedScrollView, PageView and ListView widgets. The functionality of this widget, in many ways, is similar to other Scrolling

Widgets. However, since this widget is draggable and scrollable at the same time, it is at an advantage.

Quite naturally, this advantage puts this widget in a superior position.

Why so?

Let's see the screenshots firstly. So that we can visualise the advantages. As regards, the draggable and scrollable effect, we can have a total grip on the size of the sheet that has been generated.

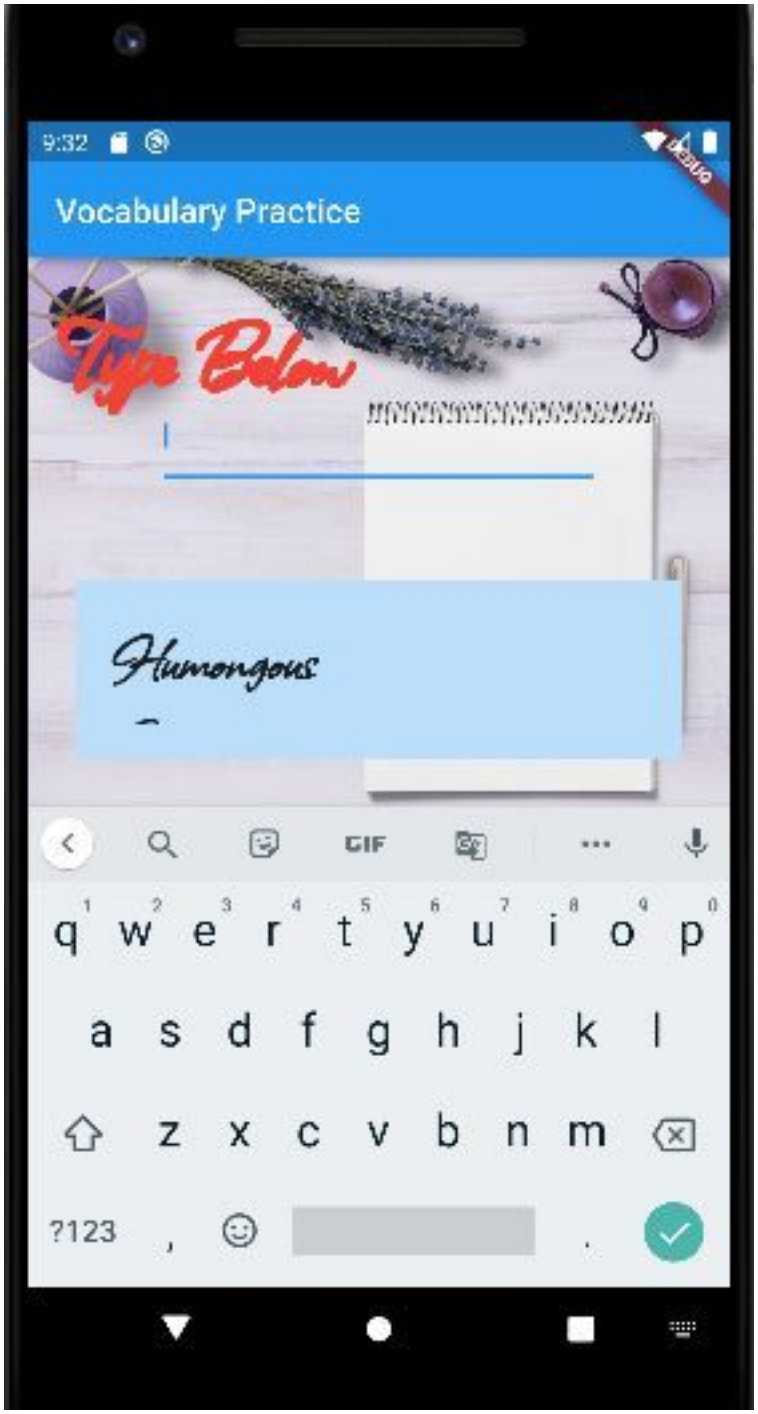

Figure 13.1 – DraggableScrollableSheet first look

We've built a small but functional app using which we can practise vocabulary. Once we type the words, we can add them to the bottom sheet by clicking the keypad.

After that we can view the list from top to bottom by scrolling just like the following second screenshot that displays the upper side.

Figure 13.2 – DraggableScrollableSheet flutter keeping words

Since the lower bottom sheet is scrollable, we can scroll vertically and see the last word that we've typed.

Figure 13.3 – DraggableScrollableSheet flutter allows to scroll and view the last word

Of course, we could have made the image much smaller or made the draggable scrollable sheet much bigger.

How does that happen?

The DraggableScrollableSheet can extend up to a certain fraction of the total screen of our app. Meanwhile, the scrolling effect can occur inside its expanded height. As it expands its size, it also contains the list of words that we're typing above.

The parameters of draggable scrollable sheet

Let us see the full code, so that we can discuss the parts of the code separately.

```
import 'package:flutter/material.dart';

class DraggableScrollableSheetSample extends StatelessWidget {
const DraggableScrollableSheetSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Draggable Scrollable Sheet Sample',
    home: DraggableScrollableSheetHome(),
    );
}
}

class DraggableScrollableSheetHome extends StatefulWidget {
const DraggableScrollableSheetHome({Key? key}) : super(key: key);

@override
State<DraggableScrollableSheetHome> createState() =>
    _DraggableScrollableSheetHomeState();
}

List<String> listOfItems = [];
final TextEditingController textController = TextEditingController();

class _DraggableScrollableSheetHomeState
    extends State<DraggableScrollableSheetHome> {
@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Vocabulary Practice'),
    ),
    body: Stack(
        children: <Widget>[
        Image.network(
```

```
            'https://cdn.pixabay.com/photo/2018/04/07/08/28/notepad-3297994_960_720.\
jpg',
            fit: BoxFit.cover,
            width: double.infinity,
        ),
        const SizedBox(
            height: 30,
        ),
        Container(
            margin: const EdgeInsets.all(10),
            padding: const EdgeInsets.all(10),
            child: Text(
            'Type Below',
            style: TextStyle(
                fontFamily: 'Allison',
                fontSize: 60,
                fontWeight: FontWeight.bold,
                foreground: Paint()
                ..color = Colors.red
                ..strokeWidth = 2.0
                ..style = PaintingStyle.stroke,
            ),
            ),
        ),
        const SizedBox(
            height: 60,
        ),
        Container(
            margin: const EdgeInsets.all(40),
            padding: const EdgeInsets.all(40),
            child: TextField(
            controller: textController,
            onSubmitted: (text) {
                listOfItems.add(text);
                textController.clear();
                setState(() {});
            },
            ),
        ),
        DraggableScrollableSheet(
            builder: (BuildContext context, ScrollController scrollController) {
            return Padding(
                padding: const EdgeInsets.all(8.0),
```

```
                child: Container(
                margin: const EdgeInsets.all(20),
                padding: const EdgeInsets.all(20),
                color: Colors.blue[100],
                child: ListView.builder(
                    itemCount: listOfItems.length,
                    itemBuilder: (BuildContext ctxt, int index) {
                    return Text(
                        listOfItems[index],
                        style: const TextStyle(
                        fontFamily: 'Allison',
                        fontSize: 40,
                        fontWeight: FontWeight.bold,
                        ),
                    );
                    },
                ),
                ),
            );
            },
        ),
        ],
    ),
    );
}
}
```

For more flutter user interface widgets please visit the relevant GitHub repository.

Since we've planned to add a list of words, so we need to have the following part of the code where we have a text field, text editing controller and a mechanism that will add words to the list.

```
List<String> listOfItems = [];
final TextEditingController textController = TextEditingController();
...
TextField(
            controller: textController,
            onSubmitted: (text) {
                listOfItems.add(text);
                textController.clear();
                setState(() {});
            },
            ),
...
```

The draggable scrollable sheet can be dragged along the vertical axis. What does determine its size?

The size varies between its minChildSize, which defaults to 0.25 and maxChildSize, which defaults to 1.0. These sizes are percentages of the height of the parent container.

As we have used a ListView.builder, and started adding words, the draggable scrollable widget coordinates resizing and scrolling of the widget returned by the builder.

```
ListView.builder(
                    itemCount: listOfItems.length,
                    itemBuilder: (BuildContext ctxt, int index) {
                    return Text(
                        listOfItems[index],
                        style: const TextStyle(
                        fontFamily: 'Allison',
                        fontSize: 40,
                        fontWeight: FontWeight.bold,
                        ),
                    );
                    },
                ),
...
```

As soon as we open the app, we can see the initial size, which is controlled by the parameter initialChildSize which defaults to 0.5.

Dragging will work between the range of minChildSize and maxChildSize parameters. They are percentages of the parent container's height. If we create the widget by the ScrollableWidgetBuilder, and it does not use the provided ScrollController, the sheet will remain at the initialChildSize.

The expand parameter takes the boolean value true or false. With reference to the size, this parameter decides whether the widget should expand available space in its parent or not, having default value true.

Flutter Scrollbar Interactive

Providing a unique ScrollController to each Scrollable widget is important. Let's see why?

In our previous section we've seen how we can create a Scrollbar. We have also seen how we can read about other Scrolling Widgets. A Scrollbar always belongs to the Scrolling widgets category. And there are more you may have interest in.

At the end of this post we'll provide you a list of other Scrolling Widgets.

When we say that a Scrollbar is interactive, actually what does that mean?

Let's try to understand that concept first. Why so? Because we've seen in our previous section that a Scrollbar thumb can be dragged along the main axis of the ScrollView to change the ScrollPosition.

If we tap along the track where the Scrollbar thumb operates, it will trigger a ScrollIncrement-Type.page based on the relative position to the thumb. In addition, a Scrollbar, due to its interactive nature, can use the PrimaryScrollController if a controller is not set.

Moreover, if we can provide a unique ScrollController to each Scrollable, we can prevent multiple ScrollPositions being attached to the PrimaryScrollController.

In our previous Scrollbar example we've seen how a Scrollbar's ScrollView.scrollDirection of Axis.vertical automatically attaches its ScrollPosition to the PrimaryScrollController. It takes place because we've not provided a ScrollController.

What is the role of ScrollController in a Scrollbar?

When we say a Scrollbar is interactive, it directly has a link to the State of the object. In fact, to be more precise, we can say that each scroll controller is stored as member variable in State objects.

Why?

Because they can be reused in each State.build. Therefore, a ScrollController actually controls a scrollable widget.

As a result, we can subsequently confirm that a ScrollController creates a ScrollPosition to manage the state specific to an individual Scrollable widget. Consequently, a ScrollController always notifies its listeners whenever a user scrolls and the ScrollPosition changes.

To understand this concept let's see the screenshots firstly.

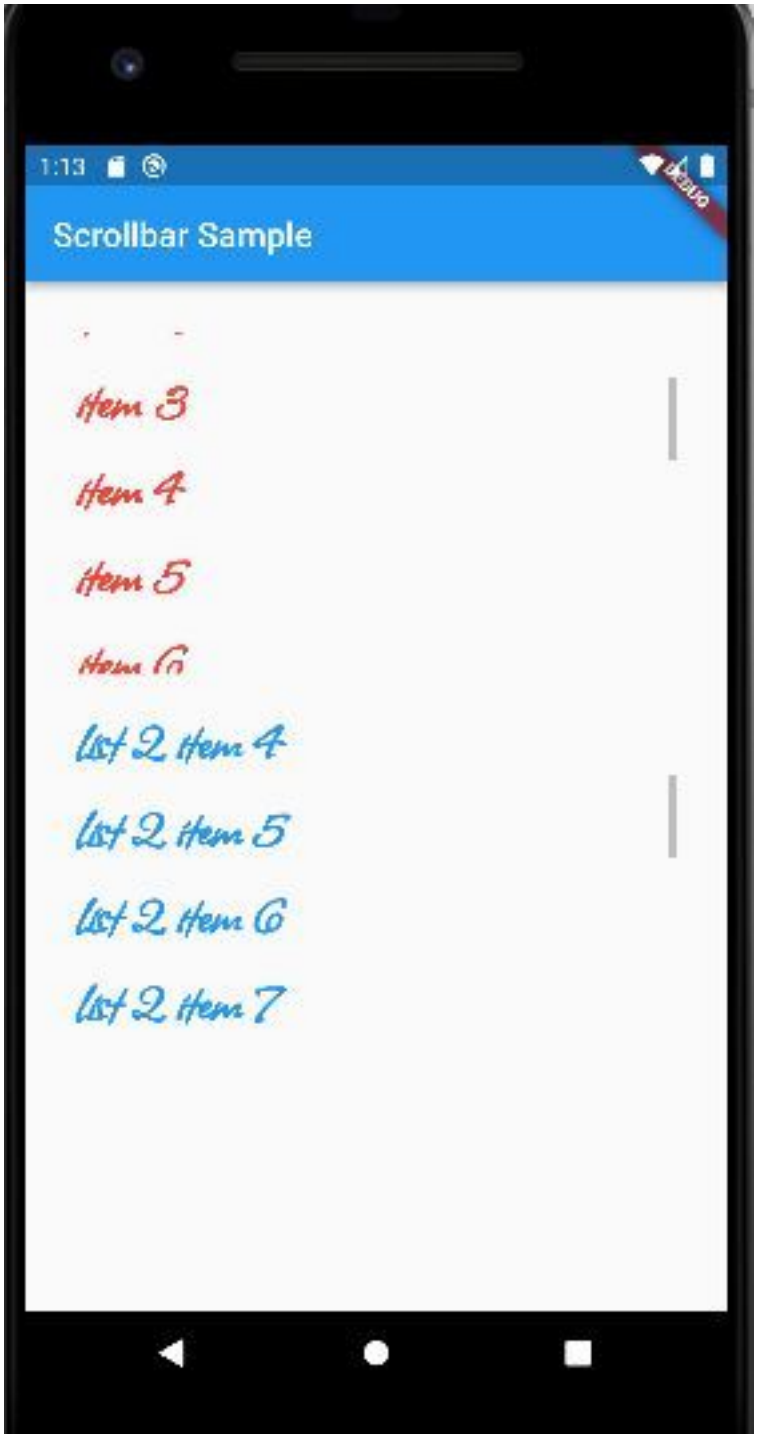

Figure 13.4 – Scrollbar controller sample one

The above image shows us a column and inside that column we have two scrollable widgets. Watch the first Scrollbar code.

```
Scrollbar(
            isAlwaysShown: true,
            controller: _controllerOne,
            child: ListView.builder(
                controller: _controllerOne,
                itemCount: 20,
                itemBuilder: (BuildContext context, int index) => Text(
                'item $index',
                style: const TextStyle(
                    color: Colors.red,
                    fontSize: 40.0,
                    fontFamily: 'Allison',
                    fontWeight: FontWeight.bold,
                ),
                ),
            ),
            ),
```

The parameter isAlwaysShown is true. That means the Scrollbar thumb will always show. However, in the above screenshot the second Scrollbar thumb also is being displayed. It's because we were dragging that so it showed itself.

```
isAlwaysShown: true,
```

Although in the next screenshot, the second Scrollbar thumb is not being displayed as it has faded out.

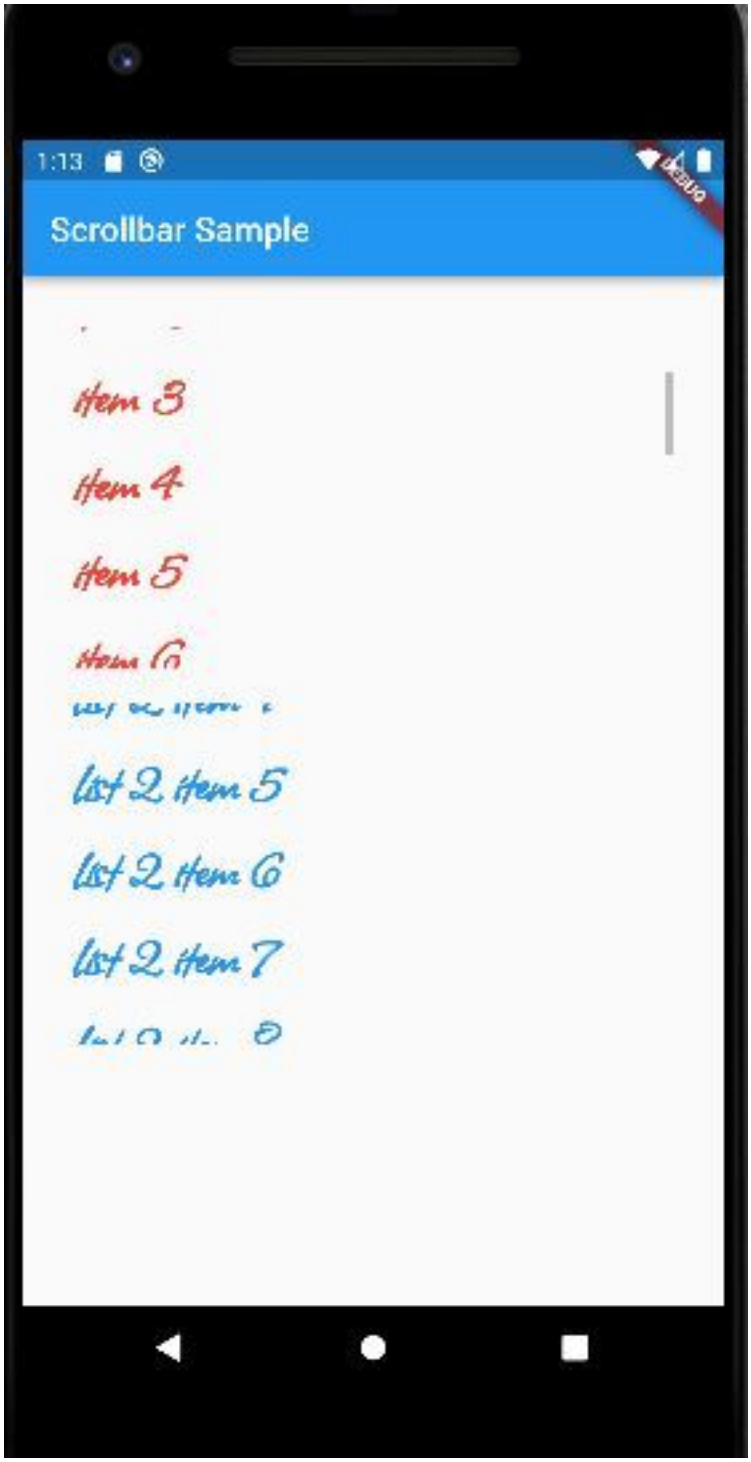

Figure 13.5 – Scrollbar controller sample two

It happened because the parameter isAlwaysShown is not true anymore in the second Scrollbar.

Let's see the full code so that we can get an idea how a ScrollController acts as a Listenable. And, how it updates the state of the scroll position.

In addition to that it also manages state by maintaining the ScrollPosition.

```
import 'package:flutter/material.dart';

/// This is the main application widget.
class ScrollbarControllerSample extends StatelessWidget {
const ScrollbarControllerSample({Key? key}) : super(key: key);

static const String _title = 'Scrollbar Sample';

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: _title,
    home: Scaffold(
        appBar: AppBar(title: const Text(_title)),
        body: const Center(
        child: ScrollbarControllerSampleHome(),
        ),
    ),
    );
}
}

/// This is the stateful widget that the main application instantiates.
class ScrollbarControllerSampleHome extends StatefulWidget {
const ScrollbarControllerSampleHome({Key? key}) : super(key: key);

@override
State<ScrollbarControllerSampleHome> createState() =>
    _ScrollbarControllerSampleHomeState();
}

/// This is the private State class that goes with ScrollbarControllerSampleHome.
class _ScrollbarControllerSampleHomeState
    extends State<ScrollbarControllerSampleHome> {
// Generate a dummy list
//final myProducts = List<String>.generate(10, (i) => 'Product $i');
final ScrollController _controllerOne = ScrollController();
final ScrollController _controllerTwo = ScrollController();
@override
Widget build(BuildContext context) {
    return Padding(
```

```
42      padding: const EdgeInsets.all(30),
43      child: Column(
44          children: <Widget>[
45          SizedBox(
46              height: 200,
47              child: Scrollbar(
48              isAlwaysShown: true,
49              controller: _controllerOne,
50              child: ListView.builder(
51                  controller: _controllerOne,
52                  itemCount: 20,
53                  itemBuilder: (BuildContext context, int index) => Text(
54                  'item $index',
55                  style: const TextStyle(
56                      color: Colors.red,
57                      fontSize: 40.0,
58                      fontFamily: 'Allison',
59                      fontWeight: FontWeight.bold,
60                  ),
61                  ),
62              ),
63              ),
64          ),
65          const SizedBox(
66              height: 20,
67          ),
68          SizedBox(
69              height: 200,
70              child: Scrollbar(
71              controller: _controllerTwo,
72              child: ListView.builder(
73                  controller: _controllerTwo,
74                  itemCount: 20,
75                  itemBuilder: (BuildContext context, int index) => Text(
76                  'list 2 item $index',
77                  style: const TextStyle(
78                      color: Colors.blue,
79                      fontSize: 40.0,
80                      fontFamily: 'Allison',
81                      fontWeight: FontWeight.bold,
82                  ),
83                  ),
84              ),
```

```
85                 ),
86             ),
87             ],
88       ),
89       );
90   }
91   }
```

Although we have used ListView.builder constructor as scrollable widgets inside the Scrollbar, we could have used ListView, GridView, or CustomScrollView.

To sum up, when we say that flutter Scrollbar is interactive, it actually means change of ScrollPosition attached to the controller. When we pass a ScrollController, it implements Scrollbar dragging. When we drag, the Scrollbar thumb updates its position.

How to use Scrollbar in flutter

Scrollbar in flutter is a material design widget that needs a ScrollView widget to show the thumb.

Not only Scrollbar widget belongs to only Scrolling widgets, but also it's a Material Design widget.

To add a Scrollbar, we need to have a ScrollView. Why? Because we can only add a Scrollbar to a ScrollView; otherwise, it doesn't make any sense.

To make our point strong, let's see the screenshot of a flutter app that we've built for this purpose.

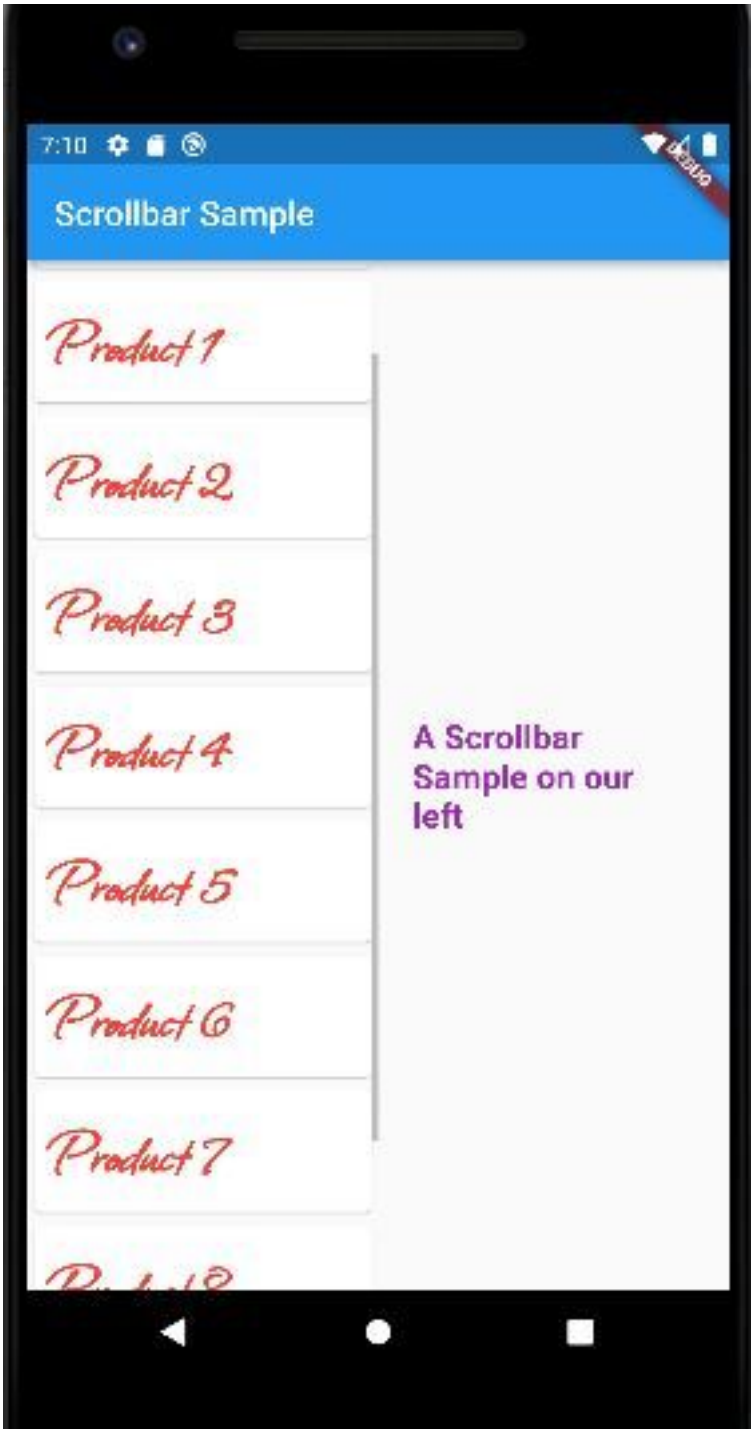

Figure 13.6 – Scrollbar in flutter

As the above screenshot displays, our items are not many. We have only 10 items to show on the screen. As a result, our Scrollbar thumb looks bigger.

What if we've added more items on the screen?

As a result the Scrollbar thumb gets smaller. As we scroll vertically, the Scrollbar thumb moves.

The Scrollbar thumb shows the exact spot of the ScrollView, which is actually visible

By the way, in our code, as a ScrollView widget we've used ListView.builder constructor of ListView widget.

Therefore, now it makes sense if we take a look at the full code. After that, we'll discuss a few parts of the code to understand Scrollbar widget in great detail.

```
import 'package:flutter/material.dart';

/// This is the main application widget.
class ScrollbarSample extends StatelessWidget {
const ScrollbarSample({Key? key}) : super(key: key);

static const String _title = 'Scrollbar Sample';

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: _title,
    home: Scaffold(
        appBar: AppBar(title: const Text(_title)),
        body: const Center(
        child: ScrollbarSampleHome(),
        ),
    ),
    );
}
}

/// This is the stateful widget that the main application instantiates.
class ScrollbarSampleHome extends StatefulWidget {
const ScrollbarSampleHome({Key? key}) : super(key: key);

@override
State<ScrollbarSampleHome> createState() => _ScrollbarSampleHomeState();
}

/// This is the private State class that goes with ScrollbarSampleHome.
class _ScrollbarSampleHomeState extends State<ScrollbarSampleHome> {
// Generate a dummy list
final myProducts = List<String>.generate(10, (i) => 'Product $i');
@override
```

```
Widget build(BuildContext context) {
    return LayoutBuilder(
    builder: (BuildContext context, BoxConstraints constraints) {
        return Row(
        children: [
            SizedBox(
            width: constraints.maxWidth / 2,
            child: Scrollbar(
                isAlwaysShown: true,
                child: ListView.builder(
                itemCount: myProducts.length,
                itemBuilder: (context, index) {
                    return Card(
                    //key: UniqueKey(),
                    child: Padding(
                        padding: const EdgeInsets.all(10),
                        child: Text(
                        myProducts[index],
                        style: const TextStyle(
                            color: Colors.red,
                            fontSize: 40.0,
                            fontFamily: 'Allison',
                            fontWeight: FontWeight.bold,
                        ),
                        ),
                    ),
                    );
                },
                ),
            ),
            ),
            SizedBox(
            width: constraints.maxWidth / 2,
            child: Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(10),
                child: const Text(
                'A Scrollbar Sample on our left',
                style: TextStyle(
                    color: Colors.purple,
                    fontSize: 20.0,
                    fontWeight: FontWeight.bold,
                ),
```

```
                ),
            ),
            )
        ],
        );
    },
    );
}
}
```

If you're interested to know more about such Flutter widgets, please visit the relevant GitHub repository.

As we can see, we have distinctly divided the screen in two parts by using a Row widget.

On the left side we have our Scrollbar widget with a ListView.builder constructor.

```
return Row(
        children: [
            SizedBox(
            width: constraints.maxWidth / 2,
            child: Scrollbar(
                isAlwaysShown: true,
                child: ListView.builder(
                itemCount: myProducts.length,
                itemBuilder: (context, index) {
                    return Card(
                    //key: UniqueKey(),
                    child: Padding(
                        padding: const EdgeInsets.all(10),
                        child: Text(
                        myProducts[index],
                        style: const TextStyle(
                            color: Colors.red,
                            fontSize: 40.0,
                            fontFamily: 'Allison',
                            fontWeight: FontWeight.bold,
                        ),
                        ),
                    ),
                    );
                },
                ),
            ),
```

```
            ),
...
```

Certainly, above that, we have declared our dummy products.

```
final myProducts = List<String>.generate(10, (i) => 'Product $i');
```

Please have a look at how we have declared our products in a list and generated 10 items. This is a nice sample of declarative programming style that Flutter uses and lifts the heavy burden off developers

Since we've not provided the vertical scroll view a ScrollController, it's using the PrimaryScrollController.

As the scroll view scrolls, the Scrollbar thumb will fade in and out. It happens by default.

If we've made the Scrollbar parameter isAlwaysShown true, the Scrollbar thumb will remain visible without the fade animation.

With reference to what we've learned so far, we can say that, if the child ScrollView (here we've used ListView.builder constructor) is infinitely long, the RawScrollbar will not be painted. In that case, the Scrollbar cannot precisely show the location of the visible area.

By the way, every Scrollbar is interactive and can use the PrimaryScrollController if a controller is not set. However, we'll discuss that property in our next section.

How to use ReorderableListView

ReorderableListView looks magical as we can drag and drop any item and reorder a list.

The ReorderableListView widget is another good example of Scrolling widgets, which we've covering in a great detail.

As the name suggests, it's a kind of ListView of items where the user can reorder the items by dragging one item over another.

However, this flutter widget is ideal for small items.

Why?

Because initially the ReorderableListView widget has a children parameter that returns a list of items where each item gets involved when the dragging takes place.

In other words, when the list is constructed it requires doing work for every child; not only those children, which are visible.

In addition, all list items must have a key that is not null.

Let us create a list of numerical items and order them from 1 to 10.

After all, the ReorderableListView's children parameter will return a list that can be scrolled to the end.

In our code, we have a list of 10 numerical items that are in perfect order.

As a result, the first screenshot displays the number in order, starting from 1 and the incremental expansion of numbers move downwards.

Now, let's try to reorder the number.

We can see that the number 10 is being dragged and we're trying to lace it over the number 8.

We've successfully dragged and placed 10 over 8.

Let's come back to the topic we were talking about. How this reordering takes place?

Let's see the full code first.

```
import 'package:flutter/material.dart';

/// This is the main application widget.
class ReorderableListViewSample extends StatelessWidget {
const ReorderableListViewSample({Key? key}) : super(key: key);

static const String _title = 'ReorderableListView Sample';

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: _title,
    home: Scaffold(
        appBar: AppBar(title: const Text(_title)),
        body: const ReorderableListViewHome(),
    ),
    );
}
}

/// This is the stateful widget that the main application instantiates.
class ReorderableListViewHome extends StatefulWidget {
const ReorderableListViewHome({Key? key}) : super(key: key);

@override
State<ReorderableListViewHome> createState() =>
    _ReorderableListViewHomeState();
}

```

```dart
class _ReorderableListViewHomeState extends State<ReorderableListViewHome> {
/// generating 10 items
final List<int> _items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

@override
Widget build(BuildContext context) {
    return ReorderableListView(
    padding: const EdgeInsets.symmetric(horizontal: 40),
    children: _items
        .map(
            (e) => Container(
            key: ValueKey(e),
            decoration: BoxDecoration(
                color: Colors.blue,
                border: Border.all(
                color: Colors.red,
                width: 2.0,
                style: BorderStyle.solid,
                ),
                borderRadius: const BorderRadius.all(Radius.circular(40.0)),
                boxShadow: const [
                BoxShadow(
                    color: Colors.black54,
                    blurRadius: 20.0,
                    spreadRadius: 20.0,
                ),
                ],
                gradient: const LinearGradient(
                begin: Alignment.centerLeft,
                end: Alignment.centerRight,
                colors: [
                    Colors.red,
                    Colors.white,
                ],
                ),
            ),
            child: ListTile(
                contentPadding: const EdgeInsets.all(25),
                leading: const Icon(Icons.input_outlined),
                title: Text(
                '$e',
                key: ValueKey(e),
                style: const TextStyle(
```

```
                    color: Colors.black,
                    fontSize: 80.0,
                    fontFamily: 'Allison',
                    fontWeight: FontWeight.bold,
                ),
                ),
                trailing: const Icon(Icons.drag_indicator_outlined),
            ),
            ),
        )
        .toList(),
    onReorder: (int oldIndex, int newIndex) {
        setState(() {
        if (oldIndex < newIndex) {
            newIndex -= 1;
        }
        final int item = _items.removeAt(oldIndex);
        _items.insert(newIndex, item);
        });
    },
    );
}
}
```

The first thing first, so we have a list of items.

```
/// generating 10 items
final List<int> _items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

Nothing special in this. As we have wanted to arrange the numbers in order, we've declared a list of numerical items where numbers are arranged in order.

Next we've taken necessary steps so that the ReorderableListView's children parameter will return this list. As flutter follows the declarative style of programming, we don't have to run for loop manually. On the contrary, flutter maps that list and gives us an output that can be displayed using layout widgets.

Therefore we've decorated the list using the decoration parameter of the Container widget. And as a child parameter we pass a ListTile.

```
_items
        .map(
            (e) => Container(
            key: ValueKey(e),
            decoration: BoxDecoration(
                color: Colors.blue,
                border: Border.all(
                color: Colors.red,
                width: 2.0,
                style: BorderStyle.solid,
                ),
                borderRadius: const BorderRadius.all(Radius.circular(40.0)),
                boxShadow: const [
                BoxShadow(
                    color: Colors.black54,
                    blurRadius: 20.0,
                    spreadRadius: 20.0,
                ),
                ],
                gradient: const LinearGradient(
                begin: Alignment.centerLeft,
                end: Alignment.centerRight,
                colors: [
                    Colors.red,
                    Colors.white,
                ],
                ),
            ),
            child: ListTile(
                contentPadding: const EdgeInsets.all(25),
                leading: const Icon(Icons.input_outlined),
                title: Text(
                '$e',
                key: ValueKey(e),
                style: const TextStyle(
                    color: Colors.black,
                    fontSize: 80.0,
                    fontFamily: 'Allison',
                    fontWeight: FontWeight.bold,
                ),
                ),
                trailing: const Icon(Icons.drag_indicator_outlined),
            ),
```

```
            ),
        )
        .toList(),
```

However, most importantly, we need to handle two things to make things happen.

Granted we cannot think of ReorderableListView widget as a magical widget, but we need to understand the logic behind it.

Firstly, we need to provide key to each item of the list.

```
Container(
            key: ValueKey(e),
...
title: Text(
                '$e',
                key: ValueKey(e),
....
```

Secondly, we need a callback by the list to report that a list item has been dragged to a new position in the list. The ancestor StatefulWidget then updates the order of the item.

```
onReorder: (int oldIndex, int newIndex) {
        setState(() {
        if (oldIndex < newIndex) {
            newIndex -= 1;
        }
        final int item = _items.removeAt(oldIndex);
        _items.insert(newIndex, item);
        });
    },
```

On the whole, now it makes sense. And lastly we achieve almost a kind of magical aura where we can drag any item on the screen and reorder them as we wish.

How to rearrange list in flutter

ReorderableListView.builder constructor is an efficient way to rearrange list in flutter.

In our previous section we've seen how we can rearrange list in Flutter. Yes, we are talking off ReorderableListView widget. By the way, we can also rearrange list in flutter using ReorderableListView.builder constructor.

The ReorderableListView.builder constructor creates a reorderable list from widget items that are created on demand. In a different way ReorderableListView also allows us to build a reorderable list with all the items that we pass into the constructor.

The ReorderableListView widget is good for small number of items. On the contrary, we can use ReorderableListView.builder constructor for list views with a large number of children.

Because ReorderableListView.builder constructor performs in a different way, we can handle a large number of list view.

To understand that, let us see all the parameters of ReorderableListView.builder constructor.

```
const ReorderableListView.builder(
{Key? key,
required IndexedWidgetBuilder itemBuilder,
required int itemCount,
required ReorderCallback onReorder,
double? itemExtent,
Widget? prototypeItem,
ReorderItemProxyDecorator? proxyDecorator,
bool buildDefaultDragHandles,
EdgeInsets? padding,
Widget? header,
Axis scrollDirection,
bool reverse,
ScrollController? scrollController,
bool? primary,
ScrollPhysics? physics,
bool shrinkWrap,
double anchor,
double? cacheExtent,
DragStartBehavior dragStartBehavior,
ScrollViewKeyboardDismissBehavior keyboardDismissBehavior,
String? restorationId,
Clip clipBehavior}
)
```

The above code snippet shows us the three parameters that are required.

One of them is itemBuilder callback that will be called only with indices greater than or equal to zero and less than another required parameter itemCount.

In addition, another required parameter is onReorder, which is a callback by the list, which reports that a list item has been dragged to a new position in the list. Certainly, the name fully justifies its significance.

Because we're dragging an item and placing it at a new location, we're reordering the list.

To make more sense to our statement, let's take a look at the full code snippet first. After that, we'll discuss how it works.

```
import 'package:flutter/material.dart';

/// This is the main application widget.
class ReorderableListViewBuilderSample extends StatelessWidget {
const ReorderableListViewBuilderSample({Key? key}) : super(key: key);

static const String _title = 'ReorderableListView Sample';

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: _title,
    home: Scaffold(
        appBar: AppBar(title: const Text(_title)),
        body: const ReorderableListViewBuilderHome(),
    ),
    );
}
}

/// This is the stateful widget that the main application instantiates.
class ReorderableListViewBuilderHome extends StatefulWidget {
const ReorderableListViewBuilderHome({Key? key}) : super(key: key);

@override
State<ReorderableListViewBuilderHome> createState() =>
    _ReorderableListViewBuilderHomeState();
}

class _ReorderableListViewBuilderHomeState
    extends State<ReorderableListViewBuilderHome> {
/// generating 10 items
final List<int> _items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

@override
Widget build(BuildContext context) {
    return ReorderableListView.builder(
        itemCount: _items.length,
```

```
39          itemBuilder: (context, index) {
40          final String itemName = '${_items[index]}';
41          return Container(
42              key: ValueKey(itemName),
43              decoration: BoxDecoration(
44              color: Colors.blue,
45              border: Border.all(
46                  color: Colors.red,
47                  width: 2.0,
48                  style: BorderStyle.solid,
49              ),
50              borderRadius: const BorderRadius.all(Radius.circular(40.0)),
51              boxShadow: const [
52                  BoxShadow(
53                  color: Colors.black54,
54                  blurRadius: 20.0,
55                  spreadRadius: 20.0,
56                  ),
57              ],
58              gradient: const LinearGradient(
59                  begin: Alignment.centerLeft,
60                  end: Alignment.centerRight,
61                  colors: [
62                  Colors.red,
63                  Colors.white,
64                  ],
65              ),
66              ),
67              child: ListTile(
68              contentPadding: const EdgeInsets.all(25),
69              leading: const Icon(Icons.input_outlined),
70              title: Text(
71                  itemName,
72                  key: ValueKey(itemName),
73                  style: const TextStyle(
74                  color: Colors.black,
75                  fontSize: 80.0,
76                  fontFamily: 'Allison',
77                  fontWeight: FontWeight.bold,
78                  ),
79              ),
80              trailing: const Icon(Icons.drag_indicator_outlined),
81              ),
```

```
        );
        },
        // The reorder function
        onReorder: (oldIndex, newIndex) {
        setState(() {
            if (newIndex > oldIndex) {
            newIndex = newIndex - 1;
            }
            final element = _items.removeAt(oldIndex);
            _items.insert(newIndex, element);
        });
        });
}
}
```

If we compare the above code with the previous code snippet of ReorderableListView, we'll realise that this time we don't have to map the list anymore. Moreover, the item Builder parameter creates the widget instances when called, returning a non-null widget.

It's more efficient as long as performance is concerned. Why? Because it creates the instances on demand using constructor's itemBuilder callback.

However, the reordering is arranged by our good old friend and one of the required parameters, onReorder.

```
onReorder: (oldIndex, newIndex) {
        setState(() {
            if (newIndex > oldIndex) {
            newIndex = newIndex - 1;
            }
            final element = _items.removeAt(oldIndex);
            _items.insert(newIndex, element);
        });
        }
```

Finally the ReorderableListView.builder constructor creates the list on demand by using the IndexedWidgetBuilder, so that the list items are built lazily on demand.

What is Scrollable in flutter

The Scrollable widget not only implements scrolling model, it handles many other things.

When a widget scrolls, we call it Scrollable. The StatefulWidget is the direct ancestor of a Scrollable class. Therefore when we scroll down or up, as location changes, state of position also changes.

Although on the whole we use the Scrollable class to implement the interaction model for a scrollable widget, but it's rare to construct a Scrollable directly.

Actually, there are reasons for that.

A Scrollable class doesn't have an opinion about how the children will be displayed using the viewport.

Therefore, as a result, we use different scrollable widgets, such as ListView,or GridView that are built for different purposes.

To tell the truth, a Scrollable class does have an interaction model including gesture recognition that includes different type of semantic actions, such as tap, drag, and scale. Consequently they also involve state.

But, a Scrollable class doesn't have any idea about the viewport that actually displays the children.

What is Viewport?

As the name suggests, we cannot view the materials without viewport. Most importantly, we don't have an idea on what is going to be displayed.

The scrolling machinery would stop if viewport didn't do the hard work. Since viewport widget toils hard, a subset of its children is being displayed.

The viewport widget manages the dimension and the given offset. The offset is nothing but a property that decides which part of the content inside the viewport should be visible.

How does it happen?

Here the ViewportOffset.pixels plays the important role, determining the scroll offset. Using this scroll offset viewport selects which part of its content to display.

Because we know that as the user scrolls the viewport, the value changes, and the changing value again changes the content, which is displayed.

It also happens in case of other widgets that combines Scrollable and Viewport into widgets that are easier to use. Consider the case of PageView.

At the same time, Viewport widget also hosts a list of Slivers.

You may have interest in reading a few articles on Slivers as well; because viewport cannot contain box children directly it uses different type of slivers.

Why don't we construct Scrollable directly?

The reason is simple. We treat a Scrollable as an implementation model.

Subsequently, we choose ListView or GridView that combines many facets at the same time. It scrolls, uses viewport, and a layout model.

On the other hand, the CustomScrollView combines layout models.

Summing it up, without the Scrollable, in most cases, we cannot think of building a flutter app that perform well. We always expect that a user will choose a product out of many other products. And in a static page that doesn't scroll or interact with gestures, we cannot provide such rich experiences. Besides scrolling, gesture also plays a key role.

What is ListView in flutter

ListView is not like other ListView constructors and good for a small list of children.

We've already discussed how a custom scroll view works. A ListView is basically a CustomScrollView.

However, a custom scroll view has many features that a ListView doesn't have. As a result, a ListView is not always sufficient.

We may ask, what is the limitation of ListView?

Firstly, although a ListView is a custom scroll view, it's got only useful feature from custom scroll view; and, that is a single SliverList in its CustomScrollView.slivers property.

Let's try to understand this concept and dig deep.

Sometimes, and in many case, most of the time, we need a SliverAppBar. With reference to this, you may read our previous post on collapsing toolbar.

In such cases, ListView is not sufficient. Therefore, in those cases, it's wise to use CustomScrollView directly.

However, a ListView and a CustomScrollView has some similarities too. At least, if you consider a few properties, such as, key, controller, etc.

How do I use ListView in flutter?

Before we use ListView let us try to understand a few other key concepts.

Apart from these similarities and differences, we need to use ListView frequently. In fact, although there are many scrolling widgets, still ListView is the most commonly used. It displays children one after another while we scroll.

Let's see the one example of ListView where it displays a list of student instances one after another.

Now, as the scroll direction takes us towards the lower section, we can see the third and the last student instance.

How many types of ListView are there in flutter?

We can construct ListView using four methods. In the coming sections, we're going to see how we can use ListView by constructors like ListView.builder, ListView.custom, and ListView.separated.

Likewise, the ListView constructor takes an List<Widget> of children. We'll see in a minute, how this constructor fits for the list of instances to be displayed as children.

But, there is a caveat that we should take care of. ListView is good for small number of children. Since, it doesn't build child items on demand, it requires heavy work in constructing the List of children.

The ListView constructs every child; whether they are in the viewport being displayed in the list view, or not. As a result it's wise to work with small number of items. And, that saves system resources, making our flutter app highly performant.

Let's take a look at the full code below.

```
import 'package:flutter/material.dart';

import 'package:flutter/foundation.dart';

class Student {
final String id;
final String name;
final String imageUrl;

Student({
    required this.id,
    required this.name,
    required this.imageUrl,
});
}

/// adding id and images
List<Student> students = [
Student(
    id: 's1',
    name: 'Json',
    imageUrl:
        'https://cdn.pixabay.com/photo/2018/09/11/19/49/education-3670453_960_720.jp\
g',
),
Student(
    id: 's2',
    name: 'Limpi',
    imageUrl:
        'https://cdn.pixabay.com/photo/2016/11/29/13/56/asian-1870022_960_720.jpg',
),
Student(
```

```
    id: 's3',
    name: 'Maria',
    imageUrl:
        'https://cdn.pixabay.com/photo/2017/09/21/13/32/girl-2771936_960_720.jpg',
),
];

class ListViewSample extends StatelessWidget {
const ListViewSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Students Name',
    home: StudentsHomePage(),
    );
}
}

class StudentsHomePage extends StatelessWidget {
const StudentsHomePage({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Students Home Page'),
    ),
    body: ListView(
        children: students.map((e) {
        return AllStudents(title: e.name, image: e.imageUrl);
        }).toList(),
    ),
    );
}
}

class AllStudents extends StatelessWidget {
final String title;
final String image;
const AllStudents({
    Key? key,
    required this.title,
```

```
    required this.image,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Center(
    child: Column(
        children: [
        Container(
            padding: const EdgeInsets.all(
            5.0,
            ),
            margin: const EdgeInsets.all(
            5.0,
            ),
            child: Text(
            title,
            style: const TextStyle(
                fontFamily: 'Allison',
                fontSize: 80.0,
                fontWeight: FontWeight.bold,
            ),
            ),
        ),
        Image.network(
            image,
            fit: BoxFit.cover,
            width: 150,
            height: 150,
        ),
        ],
    ),
    );
}
}
```

The above code snippet is quite simple. As the ListView constructor takes an List<Widget> of children, we map our Student list and return name and image properties through constructor of another widget.

Because it's a simple list, we don't have to work very hard to display the small student instances using ListView.

What is ListView builder in flutter

ListView builder constructor is a ListView that we can use to build widgets on demand.

A ListView is a widget that arranges a scrollable list of widgets linearly. Moreover, we can construct ListView using four ways. We'll see each construction one by one. In today's section, we'll discuss ListView.builder constructor.

The ListView.builder constructor builds the children widget on demand based on the index of a list. And, to do that, the ListView.builder constructor takes an IndexedWidgetBuilder. In fact, the IndexedWidgetBuilder builds the children on demand.

When there are a large number of children widgets that we're going to display on the screen, the ListView.builder constructor is ideal for that purpose.

You may wonder, how the builder is called when there are an infinite number of children widgets. In reality, builder is called only for those children that are actually visible.

When we use ListView.builder constructor, we'll always provide a non-null item count.

To understand this mechanism, let's consider a model class of students that will have id, name and an image property.

```
import 'package:flutter/foundation.dart';

class Student with ChangeNotifier {
final String id;
final String name;
final String imageUrl;

Student({
    required this.id,
    required this.name,
    required this.imageUrl,
});
}

class Students with ChangeNotifier {
/// adding id and images
List<Student> students = [
    Student(
    id: 's1',
    name: 'Json',
    imageUrl:
        'https://cdn.pixabay.com/photo/2018/09/11/19/49/education-3670453_960_720.jp\
g',
```

```
    ),
    Student(
    id: 's2',
    name: 'Limpi',
    imageUrl:
        'https://cdn.pixabay.com/photo/2016/11/29/13/56/asian-1870022_960_720.jpg',
    ),
    Student(
    id: 's3',
    name: 'Maria',
    imageUrl:
        'https://cdn.pixabay.com/photo/2017/09/21/13/32/girl-2771936_960_720.jpg',
    ),
];
}
```

By the way, we use provider package to provide the data model objects.

As we can see in the above code, there are three student objects that we've already created.

Next, we'll see the rest part of the code, where we will use a ListView builder constructor and display each student's name and image.

```
ListView.builder(
        padding: const EdgeInsets.all(10.0),
        itemCount: students.length,
        itemBuilder: (ctx, i) => ChangeNotifierProvider.value(
...
/// the code is incomplete for brevity
/// you'll find the full code below
```

The ListView builder constructor has many properties, yet we've used only three of them. Since we have several children widgets, so we can use padding property to make them separated.

However, here most important properties are itemCount and itemBulder.

We've counted the length of the students objects by this line:

```
itemCount: students.length,
```

Actually, we've already created three student objects, therefore, there are three students, so to say. To clarify, we've provided a non-null value to the itemCount parameter of the ListView builder constructor. It helps ListView to measure how much it has to scroll.

Another very important parameter is itemBuilder callback. Until the indices become less than the itemCount, the itemBuilder is called.

```
itemBuilder: (ctx, i) => ChangeNotifierProvider.value(...)
```

As the name suggests, the item builder builds the items. Subsequently, it cannot return null widgets.

We're going to see the full code soon. But before that let's run the app and see the first screenshot.

The ListView builder constructor displays the first student object belonging to our list. However, if we want to child reordering, this constructor doesn't support that.

To do that either we have to use ListView or ListView.custom.

What is the difference between ListView and ListView builder in flutter?

The biggest difference between ListView and ListView builder is in the process of building children widgets.

The ListView builder constructor builds children widgets on demand. The ListView widget cannot do that.

With the help of ListView builder constructor we should not return a previously constructed widget. That sounds quite natural. Because it will build children on demand.

If we have already constructed widgets, then use ListView. That is also another big difference between them.

Let's show the full code now.

```
import 'package:flutter/material.dart';
import 'package:flutter_artisan/models/student.dart';
import 'package:provider/provider.dart';

class ListViewBuilderSample extends StatelessWidget {
const ListViewBuilderSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Students Name',
    home: StudentsHomePage(),
    );
}
}

class StudentsHomePage extends StatelessWidget {
const StudentsHomePage({Key? key}) : super(key: key);
```

```
19
20  @override
21  Widget build(BuildContext context) {
22      final students = Provider.of<Students>(context).students;
23      return Scaffold(
24      appBar: AppBar(
25          title: const Text('Students Home Page'),
26      ),
27      body: ListView.builder(
28          padding: const EdgeInsets.all(10.0),
29          itemCount: students.length,
30          itemBuilder: (ctx, i) => ChangeNotifierProvider.value(
31          value: students[i],
32          child: Center(
33              child: Column(
34              children: [
35                  Container(
36                  padding: const EdgeInsets.all(5),
37                  child: Text(
38                      students[i].name,
39                      style: const TextStyle(
40                      fontFamily: 'Allison',
41                      fontSize: 30,
42                      fontWeight: FontWeight.bold,
43                      ),
44                  ),
45                  ),
46                  Image.network(
47                  students[i].imageUrl,
48                  fit: BoxFit.cover,
49                  ),
50              ],
51              ),
52          ),
53          ),
54      ),
55      );
56  }
57  }
```

Is ListView builder scrollable?

Certainly, ListView builder constructor is scrollable. Now, we can scroll and see the other student instances.

Run the above code, and see the second student instance.

And after that, we can also scroll again and see the third student instance.

What is ListView separated

ListView separated constructor builds separator children in between child items.

There are similarities in ListView.builder and ListView.separated constructors, as they both take at least one IndexedWidgetBuilder. That is itemBulder parameter, which builds the children on demand.

However, there is one difference between these two ListView constructors.

The ListView.separated constructor has another parameter separatorBuilder, which similarly builds separator children which appear in between the child items.

As a whole, the ListView.separated constructor is a very useful API. We can use it to add separators between child items inside a flutter ListView.

Firstly, let us take a look at the screenshots. Secondly, after that, we'll discuss the code snippets. You may compare ListView.separated with the previous article on ListView.builder constructor. That'll clarify if any doubt arises.

The separators have a special character. They only appear between list items. That means, the last separator appears before the last item.

The first screenshot clearly displays the separator after the first student instance. The separator, a horizontal grey line, is attached with the image, as we've not kept the divider inside any container with a well-defined padding parameter.

Now, we'll take a look at the second screenshot that shows no separator after the last student instance.

Therefore, we have come to know one key point about the ListView.separated constructor. There will be no separator above the first item, and below the last item.

As we scroll, we find this characteristic quite logical. After the last instance we don't really need any separator.

It's also true for the first one. Why? Because it doesn't look good if one unnecessary separator shows up above the first instance.

Although we've only used a Divider widget as the separatorBuilder here, but we could have used a list of other widgets as its separator children in between the student instances.

For a fixed number of children, the ListView separated constructor is very useful.

How do you add a separator to a ListView in flutter?

Now it's time to add a separator to a ListView in flutter. And, it's a high time to show the code snippets.

We've already seen the screenshots that show the separators in between the student items.

To understand this mechanism, let's consider a model class of students that will have id, name and an image property.

```
import 'package:flutter/foundation.dart';

class Student with ChangeNotifier {
final String id;
final String name;
final String imageUrl;

Student({
    required this.id,
    required this.name,
    required this.imageUrl,
});
}

class Students with ChangeNotifier {
/// adding id and images
List<Student> students = [
    Student(
    id: 's1',
    name: 'Json',
    imageUrl:
        'https://cdn.pixabay.com/photo/2018/09/11/19/49/education-3670453_960_720.jp\
g',
    ),
    Student(
    id: 's2',
    name: 'Limpi',
    imageUrl:
        'https://cdn.pixabay.com/photo/2016/11/29/13/56/asian-1870022_960_720.jpg',
    ),
    Student(
    id: 's3',
    name: 'Maria',
    imageUrl:
```

```
        'https://cdn.pixabay.com/photo/2017/09/21/13/32/girl-2771936_960_720.jpg',
    ),
];
}
```

By the way, we use provider package to provide the data model objects.

As we can see in the above code, there are three student objects that we've already created.

Next, we'll see the rest part of the code, where we will use a ListView.separated constructor and display each student's name and image, along with the separators.

```
import 'package:flutter/material.dart';
import 'package:flutter_artisan/models/student.dart';
import 'package:provider/provider.dart';

class ListViewSeperatedSample extends StatelessWidget {
const ListViewSeperatedSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Students Name',
    home: StudentsHomePage(),
    );
}
}

class StudentsHomePage extends StatelessWidget {
const StudentsHomePage({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    final students = Provider.of<Students>(context).students;
    return Scaffold(
    appBar: AppBar(
        title: const Text('Students Home Page'),
    ),
    body: ListView.separated(
        padding: const EdgeInsets.all(10.0),

        /// required
        itemCount: students.length,

```

```
        /// required
        itemBuilder: (context, index) => ChangeNotifierProvider.value(
        value: students[index],
        child: Center(
            child: Column(
            children: [
                Container(
                padding: const EdgeInsets.all(5),
                child: Text(
                    students[index].name,
                    style: const TextStyle(
                    fontFamily: 'Allison',
                    fontSize: 30,
                    fontWeight: FontWeight.bold,
                    ),
                ),
                ),
                Container(
                padding: const EdgeInsets.all(5),
                child: Text(
                    'Student ID: ${students[index].id}',
                    style: const TextStyle(
                    fontSize: 10,
                    fontWeight: FontWeight.bold,
                    ),
                ),
                ),
                Image.network(
                students[index].imageUrl,
                fit: BoxFit.cover,
                ),
            ],
            ),
        ),
        ),

        /// required
        separatorBuilder: (BuildContext context, int index) =>
            ChangeNotifierProvider.value(
        value: students[index],
        child: const Center(
            child: Divider(
            height: 10,
```

```
76              thickness: 10,
77              color: Colors.black38,
78              ),
79          ),
80          ),
81      ),
82      );
83  }
84  }
```

The above code snippet clearly shows us how we can create fixed-length scrollable linear array of list – student instances, which again are separated by list item separators.

Three parameters are required when we use ListView separated constructor. They are itemCount, itemBuilder, and separatorBuilder.

Between these three required parameters, itemBuilder, and separatorBuilder are callbacks; however, they serve different purposes.

The itemBuilder callback works at tandem with itemCount parameter.

Consequently, the item builder callback will only be called with the indices greater than, or equal to zero. Whatsoever, it always should be less than the item count parameter.

The same way, separator builder callback will only be called with the indices greater than, or equal to zero. Whatsoever, it always should be less than itemCount – 1.

That's why, we don't see any separator above the first item, and below the last item.

In addition to these properties, the item builder and separator builder callbacks should always return a non-null widget and always create widgets when called.

What is ListView custom

The ListView custom constructor is another type of ListView. It's different than others.

We've already learned that a ListView is a widget that arranges a scrollable list of widgets linearly. Moreover, we can construct ListView using four ways. We've seen two ListView constructors in our previous sections. ListView.builder and ListView.separated constructors play different roles in building a scrollable, linear array of widgets.

However, with a custom child model a ListView can control the algorithm used to estimate the size of children that are not actually visible.

The ListView.custom constructor is not like other two constructors. It takes a SliverChildDelegate, which provides the ability to customise additional aspects of the child model.

In this section we'll see how we can use ListView.custom constructor to display a list of Student instances, using text and image. Moreover, we can also use a method to reverse the list, by changing the state of our UI.

Whenever, we discuss the scrolling widgets or slivers, we must remember that we need to preserve the state of child elements when they are scrolled in and out of view,

There are several options available, whatsoever. But, at present we will use provider package to manage the state.

How do I create a ListView with image and text in flutter?

Certainly, we can create a ListView with image and text in flutter with either ListView.builder and ListView.separated constructors. But, we can use ListView.custom constructor with a required parameter SliverChildDelegate childrenDelegate that helps us to customise the child elements.

To understand this mechanism, let's consider a model class of students that will have id, name and an image property. Along with these properties, we've also added a method that will reverse the list on demand.

```
import 'package:flutter/foundation.dart';

class Student with ChangeNotifier {
final String id;
final String name;
final String imageUrl;

Student({
    required this.id,
    required this.name,
    required this.imageUrl,
});
}

class Students with ChangeNotifier {
/// adding id and images
List<Student> students = [
    Student(
    id: 's1',
    name: 'Json',
    imageUrl:
        'https://cdn.pixabay.com/photo/2018/09/11/19/49/education-3670453_960_720.jp\
g',
    ),
    Student(
    id: 's2',
```

```
    name: 'Limpi',
    imageUrl:
        'https://cdn.pixabay.com/photo/2016/11/29/13/56/asian-1870022_960_720.jpg',
    ),
    Student(
    id: 's3',
    name: 'Maria',
    imageUrl:
        'https://cdn.pixabay.com/photo/2017/09/21/13/32/girl-2771936_960_720.jpg',
    ),
];

void reverse() {
    students = students.reversed.toList();
    notifyListeners();
}
}
```

Now, we're going to take a look at the required parameter of ListView.custom constructor, which plays a very important role in building children.

```
child: ListView.custom(
/// required
        childrenDelegate: SliverChildBuilderDelegate(
            (BuildContext context, int index) {
            return ChangeNotifierProvider.value(
                value: students[index],
...
// the code is incomplete for brevity
// you'll get the full code below
```

The childrenDelegate is the required parameter that is nothing but a delegate,which supplies children for slivers. Now, as a result, rather than receiving their children as an explicit List, they receive their children using a SliverChildDelegate.

Consequently, we can lay out the list now, using SliverChildListDelegate. Based on the existing widgets, visible children elements are created. Or, we can use SliverChildBuilderDelegate that provides widgets.

We've used the SliverChildBuilderDelegate, that passes context and index of the list items. While doing that it builds child elements on demand.

Let's see the full code snippet now.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_artisan/models/student.dart';
import 'package:provider/provider.dart';

class ListViewCustomSample extends StatelessWidget {
const ListViewCustomSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Students Name',
    home: StudentsHomePage(),
    );
}
}

class StudentsHomePage extends StatelessWidget {
const StudentsHomePage({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    final students = Provider.of<Students>(context).students;
    return Scaffold(
    appBar: AppBar(
        title: const Text('Students Home Page'),
    ),
    body: SafeArea(
        child: ListView.custom(
        childrenDelegate: SliverChildBuilderDelegate(
            (BuildContext context, int index) {
            return ChangeNotifierProvider.value(
                value: students[index],
                child: Center(
                child: Column(
                    children: [
                    Container(
                        padding: const EdgeInsets.all(5),
                        child: Text(
                        students[index].name,
                        style: const TextStyle(
                            fontFamily: 'Allison',
                            fontSize: 30,
                            fontWeight: FontWeight.bold,
```

```
                        ),
                        ),
                    ),
                    Image.network(
                        students[index].imageUrl,
                        fit: BoxFit.cover,
                    ),
                    ],
                ),
                ),
            );
            },
            childCount: students.length,
        ),
        ),
    ),
    bottomNavigationBar: BottomAppBar(
        child: Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
            Container(
            padding: const EdgeInsets.all(10),
            child: TextButton(
                onPressed: () => context.read<Students>().reverse(),
                child: const Text(
                'Reverse students',
                style: TextStyle(
                    fontFamily: 'Allison',
                    fontSize: 30,
                    fontWeight: FontWeight.bold,
                ),
                ),
            ),
            ),
        ],
        ),
    ),
    );
}
}
```

When we run the app, as usual the first student instance is created and being displayed on the screen. At the bottom, the bottom navigation bar allows us to use a Text Button to reverse the list.

If we click the button, the list of student instances is reversed.

We've used latest provider package that helps us to notify listeners and that reverse the list for us.

```
TextButton(
                onPressed: () => context.read<Students>().reverse(),
                child: const Text(
                'Reverse students',
                style: TextStyle(
                    fontFamily: 'Allison',
                    fontSize: 30,
                    fontWeight: FontWeight.bold,
                ),
                ),
            ),
...
```

As a result, the last student instance comes up at the top.

By the way, you may want to take a look at the other Scrolling Widgets, before we close down further reading on this topic.

What is single child ScrollView in flutter

The SingleChildScrollView widget in flutter belongs to scrolling widgets. But,it's different.

In a series of articles we've been discussing scrolling widgets. The SingleChildScrollView is another important addition to that series. It's true that SingleChildScrollView is not like other scrolling widgets; nevertheless, it's useful in some special cases.

Here, we'll look into that speciality that a SingleChildScrollView possesses.

Let's start with a screenshot first. So that, we can understand the topic. Moreover, we'll understand why we need single child scroll view widget.

The above image shows an overflown error message.

Why is that so?

It happened because the Column widget cannot be scrolled. As a result, it gives us error. However, we can make this single Column widget scrollable with the help of SingleChildScrollView.

And we'll also learn how the Single child scroll view widget comes to our help. It not only makes the single Column widget scrollable, but enhances the performance also.

How we can do it, we'll see in a minute.

Let us take a look at the code snippet first. This is our data model.

```
class Student {
final String id;
final String name;
final String imageUrl;

Student({
    required this.id,
    required this.name,
    required this.imageUrl,
});
}

/// adding id and images
List<Student> students = [
Student(
    id: 's1',
    name: 'Json',
    imageUrl:
        'https://cdn.pixabay.com/photo/2018/09/11/19/49/education-3670453_960_720.jp\
g',
),
Student(
    id: 's2',
    name: 'Limpi',
    imageUrl:
        'https://cdn.pixabay.com/photo/2016/11/29/13/56/asian-1870022_960_720.jpg',
),
Student(
    id: 's3',
    name: 'Maria',
    imageUrl:
        'https://cdn.pixabay.com/photo/2017/09/21/13/32/girl-2771936_960_720.jpg',
),
];
```

Next, we'll map the list and return the list items as child items inside Column widget.

```
Column(
                    children: students.map((e) {
                    return AllStudents(title: e.name, image: e.imageUrl);
                    }).toList(),
```

Since the number of child items are not large, we can fit SingleChildScrollView for wrapping the

single Column widget.

However, if the number of child items are too large, consider using ListView constructors like ListView.builder, ListView.custom, and ListView.separated. These widgets create items visible on demand. That definitely saves system resources.

The problem with the above code is different. We've mapped a list of Student instances, and tried to return using single Column widget. But, the Column widget has not enough room to display the entire contents.

Hence, we've got the above error message. Moreover, there are several reasons for that. It might happen, because some devices have unusually small screens. Otherwise, one can also use the flutter app in landscape mode. That may throw an error.

The Column widget always tries to expand as big as it can. Subsequently, it results in a conflict.

To mitigate that conflict, in some special cases, we can wrap the Column widget with the SingleChildScrollView.

How do I use SingleChildScrollView in flutter?

The SingleChildScrollView gives its children infinite amount of space. However, if our child items are expected to fit on the screen, we cannot make our flutter app more performant using ListView or CustomScrollView. We can easily resolve the conflict using SingleChildScrollView.

On the contrary, if our child items are not supposed to fit the screen, then using the SingleChild-ScrollView could be expensive as far as the performance is concerned.

Let us see the second screenshot and after that we can discuss the code that resolves the conflict and helps us to avoid getting the overflown error message.

Now, the error message has gone, and the scroll direction takes us to the bottom viewport where we can view the last child element.

Let's see the full code now.

```
import 'package:flutter/material.dart';

import 'package:flutter/foundation.dart';

class Student {
final String id;
final String name;
final String imageUrl;

Student({
    required this.id,
    required this.name,
```

```
    required this.imageUrl,
});
}

/// adding id and images
List<Student> students = [
Student(
    id: 's1',
    name: 'Json',
    imageUrl:
        'https://cdn.pixabay.com/photo/2018/09/11/19/49/education-3670453_960_720.jp\
g',
),
Student(
    id: 's2',
    name: 'Limpi',
    imageUrl:
        'https://cdn.pixabay.com/photo/2016/11/29/13/56/asian-1870022_960_720.jpg',
),
Student(
    id: 's3',
    name: 'Maria',
    imageUrl:
        'https://cdn.pixabay.com/photo/2017/09/21/13/32/girl-2771936_960_720.jpg',
),
];

class SingleChildScrollViewSample extends StatelessWidget {
const SingleChildScrollViewSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Students Name',
    home: StudentsHomePage(),
    );
}
}

class StudentsHomePage extends StatelessWidget {
const StudentsHomePage({Key? key}) : super(key: key);

@override
```

```
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Students Home Page'),
    ),
    body: DefaultTextStyle(
        style: Theme.of(context).textTheme.bodyText2!,
        child: LayoutBuilder(
        builder: (BuildContext context, BoxConstraints viewportConstraints) {
            return SingleChildScrollView(
            child: ConstrainedBox(
                constraints: BoxConstraints(
                minHeight: viewportConstraints.maxHeight,
                ),
                child: IntrinsicHeight(
                child: Column(
                    children: students.map((e) {
                    return AllStudents(title: e.name, image: e.imageUrl);
                    }).toList(),
                ),
                ),
            ),
            );
        },
        ),
    ),
    );
}
}

class AllStudents extends StatelessWidget {
final String title;
final String image;
const AllStudents({
    Key? key,
    required this.title,
    required this.image,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Center(
    child: Column(
```

```
        children: [
        Container(
            padding: const EdgeInsets.all(
            5.0,
            ),
            margin: const EdgeInsets.all(
            5.0,
            ),
            child: Text(
            title,
            style: const TextStyle(
                fontFamily: 'Allison',
                fontSize: 80.0,
                fontWeight: FontWeight.bold,
            ),
            ),
        ),
        Image.network(
            image,
            fit: BoxFit.cover,
            width: 250,
            height: 250,
        ),
        ],
    ),
    );
}
}
```

In the above code, we've used a LayoutBuilder. The layout builder widget obtains the size of the viewport.

However, to obtain the size of the viewport, we need the ConstrainedBox widget that sets the minimum height of the Column widget. Now, the rule of thumb is, the Column widget cannot be smaller than its BoxConstraints.minHeight.

As a result, the height of the Column widget gets bigger than the minimum height provided by the ConstrainedBox and the sum of the heights of the children.

To sum up, the SingleChildScrollView is not always preferable. But, to wrap a single widget for making it scrollable, it's ideal.

14. A Close Encounter with Provider package and State Management

We've already discussed and given a gentle introduction to State Management in Flutter. However, a closer look and comparison between different types of classes in Provider package is necessary.

That's the reason we recapitulate and visit Provider package and State management again.

What is provider in Flutter?

The provider is a wrapper around InheritedWidget, making it simple and reusable.

Since flutter uses declarative programming style and we have a dedicated category to state management, we can try to learn concepts and fundamentals such as simple app state management. And we'll do that with the help of provider package; however, before doing that let's know what is provider package in flutter.

In this section we're going to use provider package in a simple way, so firstly, let's know how it works.

Provider package in Flutter serves us in many ways. Flutter team recommends it as the best state management architecture. But that is not the end of the story. We can use Provider for many purposes, that also includes passing a global style across the flutter app. However, understanding Provider will remain incomplete if we don't understand the concept of generics value or the type of Provider.

This section on Provider is intended for beginners, so we assume we need to start from the beginning.

Firstly, we can provide any type of value through Provider.

Take a look at the code below.

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'model/global_theme.dart';
import 'view/home.dart';

void main() {
runApp(
    /// Providers are above [Root App] instead of inside it, so that tests
    /// can use [Root App] while mocking the providers
```

```
    MultiProvider(
    providers: [
        ChangeNotifierProvider(create: (_) => GlobalTheme()),
        Provider<int>(create: (context) => 1),
        Provider<String>(create: (context) => 'Hello Flutter'),
    ],
    child: const Home(),
    ),
);
}
```

Forget about the first line. Take a look at the middle line and let's try to understand it first.

```
Provider<int>(create: (context) => 1),
```

In this case, Provider passes an integer value inside <> brackets. That generic value tells Flutter what type of provider to look for. And we've returned a value 1.

The second line tells us another stroy.

```
Provider<String>(create: (context) => 'Hello Flutter'),
```

This time Flutter will look for a String or text; so we've returned a text: "Hello Flutter".

Once we've passed this generic value we can try to access that value anywhere,in the widget tree. However, that value must be provided there inside the build method.

Why so?

Because context works like a road. Each different Provider takes different road, or context. Inside the build method, all roads, contexts merge and separate the value. As a result we can access different value quite easily.

Flutter works seamlessly with Provider and goes up through the widget tree to find the exact value.

If it doesn't find that generic value, exception is thrown.

Let's try to understand more about the generic value or type that Provider understands.

Not that it should be only integer or String. It could be any type, even an Object. That means, we can pass a whole class properties and methods through Provider.

The first line of code tells that story. However, in a different way, although.

```
ChangeNotifierProvider(create: (_) => GlobalTheme()),
```

Here, in the above line we've used ChangeNotifierProvider, instead of Provider. There are reasons to do that. We've discussed that already.

However, the most important reason is, if we want to change the state, or call a class method through Provider, the ChangeNotifierProvider is the best choice.

Now, we can try to grab those values. First of all, consider the above line.

We have passed a custom class GlobalTheme. That defines the global theme of the app. Certainly, we can define different ThemeData object to make every page of our app look different.

But at present, it's enough to get the idea.

```
import 'package:flutter/material.dart';

class GlobalTheme with ChangeNotifier {
final globalTheme = ThemeData(
    primarySwatch: Colors.deepOrange,
    textTheme: const TextTheme(
    bodyText1: TextStyle(
        fontSize: 22,
    ),
    bodyText2: TextStyle(
        color: Colors.blue,
        fontSize: 18,
        fontWeight: FontWeight.bold,
    ),
    caption: TextStyle(
        fontSize: 16,
        fontWeight: FontWeight.bold,
        fontStyle: FontStyle.italic,
    ),
    headline1: TextStyle(
        color: Colors.deepPurple,
        fontSize: 50,
        fontFamily: 'Allison',
    ),
    headline2: TextStyle(
        color: Colors.deepOrange,
        fontSize: 30,
        fontWeight: FontWeight.bold,
    ),
    ),
    appBarTheme: const AppBarTheme(
    backgroundColor: Colors.amber,
```

```
    // This will control the "back" icon
    iconTheme: IconThemeData(color: Colors.red),
    // This will control action icon buttons that locates on the right
    actionsIconTheme: IconThemeData(color: Colors.blue),
    centerTitle: false,
    elevation: 15,
    titleTextStyle: TextStyle(
        color: Colors.deepPurple,
        fontWeight: FontWeight.bold,
        fontFamily: 'Allison',
        fontSize: 40,
    ),
    ),
);
}
```

Next, we can have these values in our home page. To understand the whole widget tree, you can read the full code snippets in this GitHub repository.

```
import 'package:build_blog_with_flutter/model/global_theme.dart';
import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

class HomePage extends StatelessWidget {
const HomePage({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text(
        'Testing Global Theme with Provider',
        style: Theme.of(context).appBarTheme.titleTextStyle,
        ),
    ),
    body: const HomeBody(),
    );
}
}

/// pushing to main now
class HomeBody extends StatelessWidget {
```

```
25  const HomeBody({Key? key}) : super(key: key);
26
27  @override
28  Widget build(BuildContext context) {
29      final int globalInt = Provider.of<int>(context);
30      final String globalString = Provider.of<String>(context);
31      final ThemeData globalTheme = Provider.of<GlobalTheme>(context).globalTheme;
32      return Center(
33      child: Column(
34          children: [
35          Container(
36              margin: const EdgeInsets.all(5),
37              padding: const EdgeInsets.all(5),
38              child: Text(
39              '$globalInt',
40              //style: Theme.of(context).textTheme.bodyText1,
41              style: globalTheme.textTheme.bodyText1,
42              ),
43          ),
44          Container(
45              margin: const EdgeInsets.all(5),
46              padding: const EdgeInsets.all(5),
47              child: Text(
48              globalString,
49              //style: Theme.of(context).textTheme.bodyText2,
50              style: globalTheme.textTheme.bodyText2,
51              ),
52          ),
53          Container(
54              margin: const EdgeInsets.all(5),
55              padding: const EdgeInsets.all(5),
56              child: Text(
57              'Headline 1 theme style again provided by provider',
58              //style: Theme.of(context).textTheme.headline2,
59              style: globalTheme.textTheme.headline1,
60              ),
61          ),
62          ],
63      ),
64      );
65  }
66  }
```

As a result, we see the following screenshot where we get the provided integer, String value; and, along with that we can also apply our custom theme to our app.

In the above code, we can avoid using the custom theme. Actually we had provided the global custom theme through the following lines:

```
ChangeNotifierProvider(create: (_) => GlobalTheme()),
...
final ThemeData globalTheme = Provider.of<GlobalTheme>(context).globalTheme;
```

If we don't do that, no longer the provided custom theme works. And in that case, the rest provided value appears on the screen with the default ThemeData that take another context or road.

As a result, the look of the app gets changed.

We hope, it now makes sense, how Provider works and we get provided value anywhere in the widget tree.

Provider is a wrapper around InheritedWidget

The provider is a wrapper around InheritedWidget to make them easier to use and more reusable. Certainly, we could have used InheritedWidget manually rather than using provider package.

If we had done that, we could have maintained state management. But, at the same time, we would have missed some advantages that provider package provides.

As a result, it's better to see what advantages we can enjoy by using provider package.

We can simplify allocation and disposal of our resources. We can change the value of a variable, call a method and do some more complicated staff that involves inputs and outputs.

The lazy loading helps us to use system resources more efficiently. If we had used inherited widget manually, we would need to make a new class every time. The provider package reduces that boilerplate. That helps us to write less code.

We can consume inherited widgets by three ways. However, in this section we'll limit ourselves to practical considerations of Provider.of method. What is provider used for?

Provider is used for one single reason. It's easy to use and understand. Moreover, we write less code and we need to understand a few practical steps necessary to achieve our goals.

For an instance, as we aim to introduce beginners to the skill of state management, let us consider a simple counter app.

However, pressing any button and changing the number mean that UI also changes and rebuilds itself. If it changes the whole widget tree, then that would unnecessarily eat up the system resources, affecting the performance.

Right?

The good news is we can avoid that bloody widget rebuilding.

The provider makes it possible.

Let's see how.

Firstly, add the dependency on provider package to pubspec.yaml file.

```
dependencies:
flutter:
    sdk: flutter

provider: ^6.0.0
```

Now we can import 'package:provider/provider.dart' to start using provider and building our app.

How does provider work in Flutter?

We need a counter class to have a variable that will hold the state. And, besides, we need two methods, one for incremental purpose and the other to decrement.

```
import 'package:flutter/foundation.dart';

class Counter with ChangeNotifier {
int _count = 0;
int get count => _count;

void increment() {
    _count++;
    notifyListeners();
}

void decrement() {
    _count--;
    notifyListeners();
}
}
```

But, the business is not finished yet, though. In fact it starts from here.

ChangeNotifier comes from Flutter SDK that helps us to notify listeners. Since Counter class is with ChangeNotifier or it may have extended ChangeNotifier, any widget can subscribe to its changes.

As we can see that, when we call method increment, or decrement, the number changes.

But that should be reflected in our widget. As a result, when the changes occur, that simultaneously builds the widget in a declarative way.

But we don't want to rebuild the whole widget tree.

Therefore, we have declared that we're going to use provider at the top of our app.

```
void main() {
runApp(
    ChangeNotifierProvider(
    create: (context) =>
        Counter(),
    child: const ProviderSampleOne(),
    ),
}
```

Now, the ChangeNotifierProvider from provider package uses create parameter to return the Counter class and passes the context, so that in our child widget we can use that context to call the variable and methods of the class.

As a result, we can now declare a Counter variable inside our Build method and use Provider.of method to remain with that context.

```
final Counter counter = Provider.of<Counter>(context);
```

Now, the rest is quite easy. If we go the lowest part from the uppermost part of our widget tree, we can use this counter variable and access the variable and methods of Counter class.

However, that doesn't mean, pressing the button and changing the state will rebuild the whole widget tree. On the contrary, it only affects the widget where it has been called.

```
import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

import '/models/counter.dart';

class ProviderSampleOne extends StatelessWidget {
const ProviderSampleOne({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: ProviderSampleOneHome(),
```

```
    );
}
}

class ProviderSampleOneHome extends StatelessWidget {
const ProviderSampleOneHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    final Counter counter = Provider.of<Counter>(context);
    return Scaffold(
    body: Center(
        child: Column(
        children: [
            Container(
            margin: const EdgeInsets.all(20),
            padding: const EdgeInsets.all(20),
            child: Text(
                'You have pressed this ${counter.count} times!',
                style: const TextStyle(
                fontSize: 100,
                fontFamily: 'Allison',
                fontWeight: FontWeight.bold,
                color: Colors.red,
                ),
            ),
            ),
            const SizedBox(
            height: 20,
            ),
            ElevatedButton(
            onPressed: () => counter.increment(),
            child: const Text(
                'Press to Increment',
                style: TextStyle(
                fontSize: 30,
                color: Colors.white,
                ),
            ),
            ),
            const SizedBox(
            height: 20,
            ),
```

```
57              ElevatedButton(
58              onPressed: () => counter.decrement(),
59              child: const Text(
60                  'Press to Decrement',
61                  style: TextStyle(
62                  fontSize: 30,
63                  color: Colors.white,
64                  ),
65              ),
66              ),
67          ],
68          ),
69      ),
70      );
71  }
72  }
```

We can write the same counter app in a different way too. In the coming sections we'll explore those options.

By the way, in this way, although the widget rebuilds the tree from bottom up,the uppermost parts, which are not associated with this counter variable do not get affected.

What is Consumer Flutter?

We don't have to call Provider.of. The Consumer does it and takes care of it.

Consumer in flutter is an object in the provider library package that obtains Provider<T> from its ancestors and passes its value to builder. Actually, Consumer calls Provider.of in a new widget, and delegates its build implementation to builder.

In our previous section we've seen how we can manage state in our flutter app by using Provider.of from Provider package. However, we've not tested how, at the same time, we can avoid unnecessary widget rebuilds.

Therefore, we'll supplement the proof during this write-up with the help of screenshots.

Before that, for beginners, let's clarify the role of Consumer.

We have a Counter model class that has one number variable and and two methods. Pressing one method will increment the number and the other method will decrement.

What Consumer does is nothing but exposes instances of provided models. As a result, we can display data (here number) and call any method (here increment)on that provided Counter model.

What is the difference between provider and Consumer?

The creator of provider package Rémi Rousselet has distinguished it in a Stack overflow answer.

Let us quote him because he has made the distinction in a lucid way so that a beginner can understand.

Provider.of is the only way to obtain and listen to an object. Consumer, Selector, and all the *ProxyProvider calls Provider.of to work.

Provider.of vs Consumer is a matter of personal preference. But there's a few arguments for both

```
Provider.of
1. can be called in all the widgets lifecycle, including click handlers and didChang\
eDependencies
2. doesn't increase the indentation

Consumer
1. allows more granular widgets rebuilds
2. solves most BuildContext misuse
```

To use Provider.of, we need to call it.

```
final Counter counter = Provider.of<Counter>(context);
```

However, Consumer can call Provider.of in a new widget. Therefore, we don't have to call it explicitly. On the contrary, the Consumer widget, will call Provider.of with its own BuildContext.

The good news is, when the Consumer widget will call Provider.of with its own BuildContext, inside that BuildContext, we can use any number of expensive widgets without any worry. Because as the state changes, they will not rebuild.

Here Consumer child may take a vital role here.

What is Consumer child?

The Consumer child is any widget that doesn't need the data inside of the provider, so when the data gets updated, they don't get re-created.

Subsequently, we can say, Consumer helps us to lessen the widget rebuilds. How it does, we'll see in a moment.

In this section, we've used the same Counter model.

```
import 'package:flutter/foundation.dart';

class Counter with ChangeNotifier {
int _count = 0;
int get count => _count;

void increment() {
    _count++;
    notifyListeners();
}

void decrement() {
    _count--;
    notifyListeners();
}
}
```

After that, we've created two separate expensive widgets that we're going to use inside Consumer BuildContext.

Tow expensive widgets return a few Text widgets.

```
class HumongousWidget extends StatelessWidget {
const HumongousWidget({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Center(
    /// building another humongous widget tree
    child: Wrap(
        direction: Axis.vertical,
        children: [
        Column(
            children: const [
            Text(
                'First Row of Consumer Child',
                style: TextStyle(
                fontSize: 25.0,
                color: Colors.blue,
                fontFamily: 'Allison',
                fontWeight: FontWeight.bold,
                ),
            ),
```

```
22              SizedBox(
23                  height: 10.0,
24              ),
25              Text(
26                  'Second Row of Consumer Child',
27                  style: TextStyle(
28                  fontSize: 25.0,
29                  color: Colors.blue,
30                  fontFamily: 'Allison',
31                  fontWeight: FontWeight.bold,
32                  ),
33              ),
34              ],
35          ),
36          const Divider(
37              color: Colors.black,
38              height: 20,
39              thickness: 5,
40              indent: 20,
41              endIndent: 0,
42          ),
43          Column(
44              children: const [
45              Text(
46                  'Third Row of Consumer Child',
47                  style: TextStyle(
48                  fontSize: 25.0,
49                  color: Colors.blue,
50                  fontFamily: 'Allison',
51                  fontWeight: FontWeight.bold,
52                  ),
53              ),
54              SizedBox(
55                  height: 10.0,
56              ),
57              Text(
58                  'Fourth Row of Consumer Child',
59                  style: TextStyle(
60                  fontSize: 25.0,
61                  color: Colors.blue,
62                  fontFamily: 'Allison',
63                  fontWeight: FontWeight.bold,
64                  ),
```

```
            ),
            ],
        ),
        ],
    ),
    );
}
}

class ExpensiveWidget extends StatelessWidget {
const ExpensiveWidget({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Center(
    /// building another humongous widget tree
    child: Wrap(
        direction: Axis.vertical,
        children: [
        Column(
            children: const [
            Text(
                'First Row inside Consumer',
                style: TextStyle(
                fontSize: 25.0,
                color: Colors.blue,
                fontFamily: 'Allison',
                fontWeight: FontWeight.bold,
                ),
            ),
            SizedBox(
                height: 10.0,
            ),
            Text(
                'Second Row inside Consumer',
                style: TextStyle(
                fontSize: 25.0,
                color: Colors.blue,
                fontFamily: 'Allison',
                fontWeight: FontWeight.bold,
                ),
            ),
            ],
```

```
        ),
        ],
    ),
    );
}
}
```

Next, we'll use those expensive widgets inside Consumer widget with its own BuildContext. So the final code looks like this:

```dart
import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

import '/models/counter.dart';

class ProviderSampleTwo extends StatelessWidget {
const ProviderSampleTwo({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: ProviderSampleTwoHome(),
    );
}
}

class ProviderSampleTwoHome extends StatelessWidget {
const ProviderSampleTwoHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    /// Only this widget will be rebuilt
    //final CountingTheNumber message = Provider.of<CountingTheNumber>(context);
    /// we're using Consumer widget instead of Provider.of().
    /// we've put our Consumer widget as deep as possible in the tree
    return Scaffold(
    appBar: AppBar(
        title: const Text('Consumer Example'),
    ),
    body: ListView(
        /// building a humongous widget tree
```

```
        children: [
        Center(
            child: Container(
            margin: const EdgeInsets.all(
                25.0,
            ),
            padding: const EdgeInsets.all(
                25.0,
            ),
            child: Consumer<Counter>(
                builder: (context, cart, child) => Stack(
                children: [
                    Container(
                    padding: const EdgeInsets.only(top: 60),
                    child: Text(
                        'You pressed: ${cart.count} times!',
                        style: const TextStyle(
                        fontFamily: 'Allison',
                        fontSize: 60,
                        fontWeight: FontWeight.bold,
                        ),
                    ),
                    ),
                    const SizedBox(
                    height: 160,
                    ),
                    // Use SomeExpensiveWidget here, without rebuilding every time.
                    const ExpensiveWidget(),
                    const SizedBox(
                    height: 30,
                    ),
                    if (child != null) child,
                ],
                ),
                // Build the expensive widget here.
                child: Container(
                padding: const EdgeInsets.only(top: 160),
                child: const HumongousWidget(),
                ),
            ),
            ),
        ),
        Center(
```

```
76              child: Container(
77              margin: const EdgeInsets.all(10),
78              padding: const EdgeInsets.all(10),
79              child: Consumer<Counter>(
80                  builder: (context, message, _) {
81                  return Column(
82                      children: [
83                      FloatingActionButton(
84                          onPressed: () {
85                          message.increment();
86                          },
87                          tooltip: 'Increment',
88                          child: const Icon(Icons.add),
89                      ),
90                      ],
91                  );
92                  },
93              ),
94              ),
95          ),
96          ],
97      ),
98      );
99  }
100 }
```

With reference to the widget rebuilds, let's run the app with Android studio flutter performance.

As a result, this is our first look.

On the right side of the Android Studio screen, we can watch how rebuilds occur.

Next, we click the button 3 times and see how changes take place on the right side, Flutter performance screen.

The flutter performance in Android Studio shows that only Consumer related widgets have been rebuilt. Other widgets, including those expensive widgets have not been affected.

To be more precise, we can take a close look.

Although we have changed the state three times, one of the expensive widgets have not been rebuilt.

Let's see another proof.

To sum up, there are many advantages we can enjoy by using provider package.

We can simplify allocation and disposal of our resources. We can change the value of a variable, call a method and do some more complicated staff that involves inputs and outputs.

The lazy loading helps us to use system resources more efficiently. If we had used inherited widget manually, we would need to make a new class every time. The provider package reduces that boilerplate. That helps us to write less code.

In this section we've seen how we can avoid widget rebuilds and save system resources more efficiently.

What is Flutter Selector?

There is one similarity between Selector and Consumer belonging to the provider package.

Selector is just another class like Consumer from provider package. However, there is a distinct difference between Selector and Consumer. Before we proceed, a gentle reminder; we're discussing flutter state management in a specific category, so don't forget to check that for the latest update.

In this section we'll look into the role of Selector closely. We'll also learn how we can avoid unnecessary widget rebuilds with the help of Selector class.

Although Selector and Consumer tackles state management using different methods, yet they have one similarity. Just like Consumer, the Selector class also obtains a value using Provider.of, and after that, the Selector passes that value to the selector callback.

The selector callback then returns an object that contains the information of change, which is needed to build that widget again.

One may ask, why we need to do that?

The answer is, sometimes, we want to filter the updates by selecting an information on change of behaviour related to the model class. Not only that, this process of filtering triggers the change only on the selected value. And it also prevents widget rebuilds.

It's true that Selector class from provider package is less known compared to Consumer. However, so far we've learned, at least by theory, Selector is useful.

As we progress, we'll see how selector allows us to select a specific value to listen to. And, subsequently, we'll see only when the selected value changes, how the associated widget only changes.

How do you make a selector in Flutter?

We can make a selector in flutter using two methods. In this section we'll learn one that is quite simple and useful with the latest provider package version 6.

Here we use selector as an extension of context keyword in our build method.

To know more, let's take a look at the code snippet. Then we'll discuss it in great detail.

Firstly, the Counter data model class.

```
import 'package:flutter/foundation.dart';

class Counter with ChangeNotifier {
int _count = 0;
int _num = 0;

int get count => _count;
int get num => _num;

void increment() {
    _count++;
    notifyListeners();
}

void addOne() {
    _num++;
    notifyListeners();
}

}
```

We've two integer variables and two methods that will add 1 to them.

Secondly, we need different widgets to see how Selector works.

The following widgets will display the change when we press the associated buttons. The first value changes with the first button; and, the second value changes with the second button.

```
class SelectorValue extends StatelessWidget {
const SelectorValue({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    print('Counter first value building');
    return Text(
    'You pressed this ${context.select((Counter c) => c.count)} times.',
    style: const TextStyle(
        fontSize: 45,
        fontFamily: 'Allison',
        fontWeight: FontWeight.bold,
        color: Colors.red,
    ),
```

```
    );
}
}
...
class SelectorAnotherValue extends StatelessWidget {
const SelectorAnotherValue({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    print('Counter second value building');
    return Text(
    'You pressed this ${context.select((Counter c) => c.num)} times.',
    style: const TextStyle(
        fontSize: 45,
        fontFamily: 'Allison',
        fontWeight: FontWeight.bold,
        color: Colors.red,
    ),
    );
}
}
```

In the above code snippet, for each instance, the the extension of context keyword in the build method inside the respective widget, exposes a value from the Counter provider model.

It takes the exact data type, which is integer, in this case. Moreover, it listens to the related provider method.

Therefore, let us see the button widgets that define associated listening mechanism.

```
class SelectorFirstMethod extends StatelessWidget {
const SelectorFirstMethod({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    print('Counter first method building');
    return ElevatedButton(
    onPressed: context.select((Counter c) => c.increment),
    child: const Text(
        'Increment first value',
```

```
        style: TextStyle(
        fontSize: 20,
        color: Colors.white,
        ),
    ),
    );
}
}
...
class SelectorSecondMethod extends StatelessWidget {
const SelectorSecondMethod({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    print('Counter second method building');
    return ElevatedButton(
    onPressed: context.select((Counter c) => c.addOne),
    child: const Text(
        'Increment second value',
        style: TextStyle(
        fontSize: 20,
        color: Colors.white,
        ),
    ),
    );
}
}
```

As a result, if we press the second button, the second selected object instance will only change; however, only that widget gets rebuilt. It doesn't affect the other.

Since we've added a print statement with each build method, it prints out when the button presses and state changes. Not only that, that also confirms how many widgets get rebuilt or not.

When we run the app, we get the following print statement.

```
Home page building
Counter first value building
Counter second value building
Counter first method building
Counter second method building
```

Next, we press the second button, so the related widget only rebuilds itself and it prints out the message on the terminal.

```
Counter second value building
```

Comparing the above statement with the above image, we can now easily understand how Selector class filters the value.

Consequently, the first value changes. And we get the following print statement.

```
Counter first value building
```

You may wonder, how we've used those widgets! Therefore, the rest of code goes as the following.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_artisan/models/counter.dart';
import 'package:provider/provider.dart';

class ProviderSampleThree extends StatelessWidget {
const ProviderSampleThree({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: ProviderSampleThreeHome(),
    );
}
}

class ProviderSampleThreeHome extends StatelessWidget {
const ProviderSampleThreeHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    print('Home page building');
    return Scaffold(
    body: Center(
```

```
24          child: Column(
25          children: [
26              Container(
27              margin: const EdgeInsets.all(10),
28              padding: const EdgeInsets.all(10),
29              child: const SelectorValue(),
30              ),
31              Container(
32              margin: const EdgeInsets.all(10),
33              padding: const EdgeInsets.all(10),
34              child: const SelectorAnotherValue(),
35              ),
36              const SizedBox(
37              height: 5,
38              ),
39              const SelectorFirstMethod(),
40              const SizedBox(
41              height: 5,
42              ),
43              const SelectorSecondMethod(),
44          ],
45          ),
46      ),
47      );
48  }
49  }
```

Finally, we've also checked how Selector class helps us to avoid unnecessary widget rebuilds by using Android Studio Flutter performance.

The above image clearly indicates how with the help of flutter Selector from provider package, we can avoid unnecessary widget rebuilds.

To sum up, only when the selected value is changed, the associated builder method is triggered and the widget gets rebuilt. Other widgets don't get affected.

That is the greatest advantage of the Selector class.

How to use Selector Flutter

To use Selector in Flutter, we need to know what type of Selector we should use.

Certainly the most popular state management in flutter is provider. While discussing provider package we usually talk about using Consumer and Provider.of. Compared to other two provider mechanism, Selector is less popular or talked about class.

Although Selector is just another class like Consumer from provider package, yet we don't use it often. On the contrary, we use popular Consumer class. Or,when we build a small flutter app, we use Provider.of directly.

Of course, there is a distinct difference between Selector and Consumer, but quite often we forget about that difference.

As a consequence, we pick up Consumer, forgetting that in some specific cases,Selector helps us to filter a particular data and always is the better choice than Consumer.

However, before we proceed further, a gentle reminder; we're discussing flutter state management in a specific category, so don't forget to check that for the latest update.

We've already started looking into the role of Selector closely. As a result, we've discussed one type of Selector in our previous post.

We've also learned how we can avoid unnecessary widget rebuilds with the help of Selector class.

Although Selector and Consumer tackles state management using different methods, yet they have one similarity. Just like Consumer, the Selector class also obtains a value using Provider.of, and after that, the Selector passes that value to the selector callback.

The selector callback then returns an object that contains the information of change, which is needed to build that widget again.

Selector Filters data

When we want to filter the updates by selecting an information on change of behaviour related to the model class we use Selector. Not only that, this process of filtering triggers the change only on the selected value. And it also prevents widget rebuilds. We'll see that in a minute.

As we can see we've pressed the first button twice and the second button thrice.

Since we've attached print statement with each build method, with the change of state we can get messages like the following.

```
Building first value
Building first value

Building second value
Building second value
Building second value
```

It clearly shows that when we change state of a particular value that does not rebuild the whole widget tree. Even it does not affect the other provided object.

Thus Selector helps us to avoid unnecessary widget rebuilds.

We can track the widget rebuilds with the help of Android Studio Flutter Performance also. That shows us the same result in a more specific way.

Firstly, the Counter data model class that has two integer variables that get changed with associated methods. As an extension each method notify listeners.

```
import 'package:flutter/foundation.dart';

class Counter with ChangeNotifier {
int _count = 0;
int _num = 0;

int get count => _count;
int get num => _num;

void increment() {
    _count++;
    notifyListeners();
}

void addOne() {
    _num++;
    notifyListeners();
}

}
```

How do we listen to this change?

Let us see the full code snippet, that obviously gives us opportunity to compare with another type of Selector that uses context as an extension.

```
import 'package:flutter/material.dart';
import 'package:flutter_artisan/models/counter.dart';
import 'package:provider/provider.dart';

class ProviderSampleFour extends StatelessWidget {
const ProviderSampleFour({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
```

```
    home: ProviderSampleFourHome(),
    );
}
}

class ProviderSampleFourHome extends StatelessWidget {
const ProviderSampleFourHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    print('Home page building');
    return Scaffold(
    body: Center(
        child: Column(
        children: [
            Selector<Counter, int>(
            selector: (_, provider) => provider.count,
            builder: (context, firstValue, child) {
                print('Building first value');
                return Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(10),
                child: Text(
                    'You pressed $firstValue times',
                    style: const TextStyle(
                    fontSize: 60,
                    fontFamily: 'Allison',
                    fontWeight: FontWeight.bold,
                    color: Colors.red,
                    ),
                ),
                );
            },
            ),
            Selector<Counter, int>(
            selector: (_, provider) => provider.num,
            builder: (context, secondValue, child) {
                print('Building second value');
                return Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(10),
                child: Text(
                    'You pressed $secondValue times',
```

```
                    style: const TextStyle(
                    fontSize: 60,
                    fontFamily: 'Allison',
                    fontWeight: FontWeight.bold,
                    color: Colors.red,
                    ),
                ),
                );
            },
            ),
            const SizedBox(
            height: 5,
            ),
            const SelectorFirstMethod(),
            const SizedBox(
            height: 5,
            ),
            const SelectorSecondMethod(),
        ],
        ),
    ),
    );
}
}

class SelectorSecondMethod extends StatelessWidget {
const SelectorSecondMethod({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    print('Counter second method building');
    return ElevatedButton(
    onPressed: context.select((Counter c) => c.addOne),
    child: const Text(
        'Increment second value',
        style: TextStyle(
        fontSize: 20,
        color: Colors.white,
        ),
    ),
    );
```

```
}
}

class SelectorFirstMethod extends StatelessWidget {
const SelectorFirstMethod({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    print('Counter first method building');
    return ElevatedButton(
    onPressed: context.select((Counter c) => c.increment),
    child: const Text(
        'Increment first value',
        style: TextStyle(
        fontSize: 20,
        color: Colors.white,
        ),
    ),
    );
```

Let us check the usage of one Selector class very closely.

What is Selector flutter?

Firstly, Selector is always specific about the model data type. Since we're handling integer data type, we've mentioned it categorically.

Secondly, selector callback returns a void method that passe two parameters, context and provider.

Finally that provider parameter can access any public property of Counter model class and that listens to the change when we press the button.

Again the builder parameter gets the value from selector parameter.

As a result, we can display the changed-state quite easily.

```
Selector<Counter, int>(
            selector: (_, provider) => provider.count,
            builder: (context, firstValue, child) {
                print('Building first value');
                return Container(
                margin: const EdgeInsets.all(10),
                padding: const EdgeInsets.all(10),
                child: Text(
                    'You pressed $firstValue times',
                    style: const TextStyle(
                    fontSize: 60,
                    fontFamily: 'Allison',
                    fontWeight: FontWeight.bold,
                    color: Colors.red,
                    ),
                ),
                );
            },
            ),
```

We've learned how selector allows us to select a specific value to listen to. Consequently, the associated widget only rebuilds itself.

So far we've seen that there are different types of Selector object that we can use. In the previous section we've seen how we can use selector as an extension of context keyword in our build method.

Certainly that is useful with the latest provider package version 6. However, when we use Selector as described above, we can use the child just like the Consumer.

Later we'll discuss that topic. So stay tuned.

What is Flutter Selector child

Flutter Selector child works just like a Consumer child. But there is a difference!

Overall, from all the previous take on state management using Provider package, we've seen how to use Provider.of, Selector and Consumer. However, we've not discussed another key feature of Selector class. That is Selector child in flutter.

Although it is more or less similar to the Consumer child, yet there is a difference.

What is the basic difference?

As we have been saying, since we have started discussing state management using provider package, that Consumer and Selector has many similarities. But Selector provides some fine control over the widgets when build method is called.

To be more precise, Selector is a Consumer that allows us to define exactly which model class properties we need to handle.

In our example, we have been dealing with a Counter class that has two integer data type that keep changing with the respective button press.

If it was a Consumer class, with the change of state in one property calls the build method of the other property. But Selector class with the help of selector callback stops that extra callback and avoid unnecessary widget rebuilds.

In many Flutter app we have models that deal with many similar data type properties. In that case, it's not a good idea that one change on any property forces other properties to run associated build methods.

Selector helps us to solve that problem.

Selector is just another class like Consumer from provider package. However, there is a distinct difference between Selector and Consumer. Before we proceed, a gentle reminder; we're discussing flutter state management in a specific category, so don't forget to check that for the latest update.

In this section we'll look into the role of Selector's child parameter. It works like Consumer child. So that Selector's child may have expensive widgets with many other widgets below the tree; however, change in Selector class doesn't affect those expensive widgets.

Therefore we'll also learn how we can avoid unnecessary widget rebuilds with the help of Selector class child parameter.

Although Selector and Consumer tackles state management using different methods, yet they have one similarity. Just like Consumer, the Selector class also obtains a value using Provider.of, and after that, the Selector passes that value to the selector callback.

The selector callback then returns an object that contains the information of change, which is needed to build that widget again.

In many cases, we want to filter the updates by selecting an information related to the model class.

Not only that, this process of filtering triggers the change only on the selected value. And it also prevents widget rebuilds of the values belonging to the same data type.

As we progress, we'll see how Selector child allows us to pass expensive widgets without rebuilding when we select a specific value to listen to.

And, subsequently, we'll see only when the selected value changes, how the expensive widgets associated with the Selector child doesn't change.

Let us see the full code first, after that, we'll discuss the part of code snippets associated with Selector child.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_artisan/models/counter.dart';

import 'package:provider/provider.dart';

class ProviderSampleFive extends StatelessWidget {
const ProviderSampleFive({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: ProviderSampleFiveHome(),
    );
}
}

class ProviderSampleFiveHome extends StatelessWidget {
const ProviderSampleFiveHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    print('Home page building');
    return Scaffold(
    body: Center(
        child: ListView(
        children: const [
            FirstSelector(),
            SecondSelector(),
            SelectorFirstMethod(),
            SizedBox(
            height: 5,
            ),
            SelectorSecondMethod(),
        ],
        ),
    ),
    );
}
}

class FirstSelector extends StatelessWidget {
const FirstSelector({
    Key? key,
```

```
}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Selector<Counter, int>(
    selector: (_, provider) => provider.count,
    builder: (context, firstValue, child) {
        print('Building first value');
        return Column(
        children: [
/// calling another expensive widget
            const HumongousWidget(),
            Container(
            margin: const EdgeInsets.all(5),
            padding: const EdgeInsets.all(5),
            child: Text(
                'You pressed $firstValue times',
                style: const TextStyle(
                fontSize: 60,
                fontFamily: 'Allison',
                fontWeight: FontWeight.bold,
                color: Colors.red,
                ),
            ),
            ),
            if (child != null) child,
        ],
        );
    },
    // Build the expensive widget here.
    child: Container(
        padding: const EdgeInsets.only(top: 10),
/// calling one expensive widget as child of Selector class
        child: const ExpensiveWidget(),
    ),
    );
}
}

class SecondSelector extends StatelessWidget {
const SecondSelector({
    Key? key,
}) : super(key: key);
```

```
87
88  @override
89  Widget build(BuildContext context) {
90      return Selector<Counter, int>(
91      selector: (_, provider) => provider.num,
92      builder: (context, secondValue, child) {
93          print('Building second value');
94          return Column(
95          children: [
96  /// calling another expensive widget
97              const HumongousWidget(),
98              Container(
99              margin: const EdgeInsets.all(5),
100             padding: const EdgeInsets.all(5),
101             child: Text(
102                 'You pressed $secondValue times',
103                 style: const TextStyle(
104                 fontSize: 60,
105                 fontFamily: 'Allison',
106                 fontWeight: FontWeight.bold,
107                 color: Colors.red,
108                 ),
109             ),
110             ),
111             if (child != null) child,
112         ],
113         );
114     },
115     // Build the expensive widget here.
116     child: Container(
117         padding: const EdgeInsets.only(top: 10),
118 /// calling one expensive widget as child of Selector class
119         child: const ExpensiveWidget(),
120     ),
121     );
122 }
123 }
124
125 class SelectorSecondMethod extends StatelessWidget {
126 const SelectorSecondMethod({
127     Key? key,
128 }) : super(key: key);
129
```

```
@override
Widget build(BuildContext context) {
    print('Counter second method building');
    return ElevatedButton(
    onPressed: context.select((Counter c) => c.addOne),
    child: const Text(
        'Increment second value',
        style: TextStyle(
        fontSize: 20,
        color: Colors.white,
        ),
    ),
    );
}
}

class SelectorFirstMethod extends StatelessWidget {
const SelectorFirstMethod({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    print('Counter first method building');
    return ElevatedButton(
    onPressed: context.select((Counter c) => c.increment),
    child: const Text(
        'Increment first value',
        style: TextStyle(
        fontSize: 20,
        color: Colors.white,
        ),
    ),
    );
}
}
/// first expensive widget
class HumongousWidget extends StatelessWidget {
const HumongousWidget({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    print('Homongous widget building');
```

```
    return Center(
    /// building another humongous widget tree
    child: Wrap(
        direction: Axis.vertical,
        children: [
        Column(
            children: const [
            Text(
                'First Row of Consumer Child',
                style: TextStyle(
                fontSize: 25.0,
                color: Colors.blue,
                fontFamily: 'Allison',
                fontWeight: FontWeight.bold,
                ),
            ),
            SizedBox(
                height: 5,
            ),
            Text(
                'First Row of Consumer Child',
                style: TextStyle(
                fontSize: 25.0,
                color: Colors.blue,
                fontFamily: 'Allison',
                fontWeight: FontWeight.bold,
                ),
            ),
            ],
        ),
        ],
    ),
    );
}
}
/// second expensive widget
class ExpensiveWidget extends StatelessWidget {
const ExpensiveWidget({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    print('Expensive widget building');
    return Center(
```

```
216     /// building another humongous widget tree
217     child: Wrap(
218         direction: Axis.vertical,
219         children: [
220         Column(
221             children: const [
222             Text(
223                 'First Row inside Consumer',
224                 style: TextStyle(
225                 fontSize: 25.0,
226                 color: Colors.blue,
227                 fontFamily: 'Allison',
228                 fontWeight: FontWeight.bold,
229                 ),
230             ),
231             SizedBox(
232                 height: 5,
233             ),
234             Text(
235                 'First Row inside Consumer',
236                 style: TextStyle(
237                 fontSize: 25.0,
238                 color: Colors.blue,
239                 fontFamily: 'Allison',
240                 fontWeight: FontWeight.bold,
241                 ),
242             ),
243             ],
244         ),
245         ],
246     ),
247     );
248 }
249 }
```

The bold section of the above code will tell us the whole story of Selector child. The comments also explain how things are taking place.

Since we've already seen how Consumer child works, we can easily find out that Selector child also works on the same principle.

Now when we run the app, the associated build methods give us these print statements.

```
Home page building
Building first value
Homongous widget building
Expensive widget building
Building second value
Homongous widget building
Expensive widget building
Counter first method building
Counter second method building
```

Next we start pressing the button, and changing the state of the app. As the selected value changes the selector callback only builds that widget without touching the expensive widgets.

We've only pressed the second button and in a quite expected way it only builds the selected value.

```
Building second value
Building second value
```

Now we can take a more precise look at the Android Studio's Flutter performance. Just run the app and take a look at how widget rebuilds.

15. User Interface, Style, Theme and App Design

It really doesn't make any sense if our flutter app does not look good.

Therefore, we must know what are the basic guidelines and concepts behind flutter app designs.

How we can create a custom theme?

How we can style the user interface, so that it looks stunning?

This section is dedicated to in depth study of those design related concepts.

What are constraints in flutter

As a beginner at the very beginning we need to know what are constraints in Flutter .

When you plan to learn Flutter, it starts with Layout. Right? Now, you cannot learn or understand flutter layout without understanding constraints. Therefore, our flutter learning starts by answering this question first – what are constraints in flutter?

Firstly, let me warn you at the very beginning. Flutter layout is not like HTML layout.

Secondly, if you come from HTML or web development background, don't try to apply those CSS rules here. Why so? Because, HTML targets a large screen. Whereas, we're dealing with a Mobile screen. So, layout should not be same. And, there are other reasons too.

Finally, flutter is all about widgets. As a result, we need to understand Flutter layout keeping widgets in our mind.

Let's start with a Material App, and, start building a simple Container widget with a Text widget as its child.

```
1  import 'package:flutter/material.dart';
2
3  class ConstraintSample extends StatelessWidget {
4  const ConstraintSample({Key? key}) : super(key: key);
5
6  @override
7  Widget build(BuildContext context) {
8      return const MaterialApp(
9      title: 'Constraint Sample',
```

```
    debugShowCheckedModeBanner: false,
    home: ConstraintSampleHomme(),
    );
}
}
@override
Widget build(BuildContext context) {
    return Container(
    width: 150,
    height: 200,
    child: const Text('Constraint Sample'),
    );
}
```

Let's run this simple flutter app, and see what happens.

What is the problem here?

We've returned a Container widget mentioning its width and height.

```
return Container(
    width: 150,
    height: 200,
    child: const Text('Constraint Sample'),
    );
```

However, that didn't work at all.

Now we know that the constraint in Flutter is all about the size of the box, which is nothing but a widget.

A size always deals with width and height.

In the above case, the material app takes the width and height of the whole screen and directs its immediate child Container to take that size.

That is why, although we've mentioned the size of the Container widget, it doesn't work.

Therefore, we can conclude that any Widget gets its constraint or size from its immediate parent. After that, it passes that constraint to its immediate child.

And this practice goes on as the number of Widgets increases in the widget tree.

In Flutter, a parent widget always controls the immediate child's size. However, when the parent becomes grand-parent, it cannot affect the constraint or size of the grand-child.

Why?

Because, the grand child has its parent that inherits the size from its parent and decides what should be the size of its child.

We can compare this mechanism with wealth. If a person inherits some wealth from her parent, she is in a position to decide how much of that wealth she will give to her child or children.

And this process goes on.

One after another widget tells its children what their constraints or sizes are.

How do you use constraints in flutter?

Since we have got the idea, now we can apply and see how we can use constraints in flutter.

Our next code snippet is like the following.

```
@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        width: 300,
        height: 100,
        color: Colors.amber,
        alignment: Alignment.bottomCenter,
        child: const Text(
        'Constraint Sample',
        style: TextStyle(
            fontSize: 25,
            fontWeight: FontWeight.bold,
            color: Colors.black,
        ),
        ),

    ),
    );
}
```

Now, we've wrapped the Container widget with a Center widget. As a result, the Center widget gets the full size now. And, after that, it asks immediate child Container widget how big it wants to be.

The Container said, "I want to be 300 in width and 100 in height. Not only that, I want to place my child at the bottom Center position. And, I also want my child should be of color amber."

The Center widget says, "Okay. No problem. Get what you want because I have inherited the whole screen-size from my parent. Take what you need."

As a result, run the code and see the effect on the screen.

Next, we change our code to this:

```
@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        width: 300,
        height: 100,
        color: Colors.amber,
        alignment: Alignment.bottomCenter,
        child: Container(
        width: 200,
        height: 50,
        color: Colors.blue,
        alignment: Alignment.center,
        child: const Text(
            'Constraint Sample',
            style: TextStyle(
            fontSize: 20,
            fontWeight: FontWeight.bold,
            color: Colors.white,
            ),
        ),
        ),
    ),
    );
}
```

Let's see the effect on the screen first. Then we'll discuss the code.

The above image is displayed because the code says that the child Container with width 300, height 100 and color amber has a child, which is another Container and that has color blue, width 200 and height 50. However, the parent Container decides that it will show its child in the bottom Center position.

Align the child at the bottom Center means, we would pass this child Container a tight constraint that is bigger than the child's natural size, with an alignment of Alignment.bottomCenter.

As a result, the child Container gets the width 200 and height 50 and is placed at the bottom Center alignment. It happens smoothly because the parent Container's width and height is bigger than the child Container. Therefore, it allocates the exact size that it's asked for.

But it didn't happen, if the parent Container didn't set the alignment and said that, "Okay, child container, you can place yourself anywhere you like. Even you may decide your alignment."

So the code is like the following:

```
@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        width: 300,
        height: 100,
        color: Colors.amber,
        child: Container(
        width: 200,
        height: 50,
        color: Colors.blue,
        alignment: Alignment.center,
        child: const Text(
            'Constraint Sample',
            style: TextStyle(
            fontSize: 20,
            fontWeight: FontWeight.bold,
            color: Colors.white,
            ),
        ),
        ),
    ),
    );
}
```

As a result, the child Container decides to looks upward and tries to find if its grand-parent has any alignment already allocated for it.

While looking upward, it finds that the grand-parent is a Center widget itself. Therefore, it decides to take the Center position, as it has the Center alignment itself; and not only that, while doing so, it ignores its own width and height, and it takes the width and height of the immediate parent, which eventually is another Container.

As a result it overlaps completely the parent Container.

To sum up, constraints are basically sizes of width and height that any Widget gets from its parent. However, in case, padding is added, the constraint may change. Although we have not added that feature, still you may test that on your own and see how it affects the sizes of the child.

What are BoxConstraints in Flutter

Imagine each Widget as a box. So it has a size or constraints that defines width and height.

In the previous section we have discussed what constraints are. In Flutter, every widget is rendered by their underlying RenderBox objects. As a result, for each boxes constraints are BoxConstraints.

For a Flutter beginner we need to say one thing at the very beginning. The constraints actually represent sizes of the rendered boxes, which are nothing but widgets.

In that light of the previous discussion, we must try to understand what BoxConstraints are.

We've already seen that widgets pass their constraints, which consist of minimum and maximum width and height, to their children. Moreover, each child may vary in size.

As a result we can say that render tree actually passes a concrete geometry, which is size.

Subsequently the widget tree grows in sizes and for each boxes the constraints are BoxConstraints. And, it consists of four numbers – a minimum width minWidth, a maximum width maxWidth, a minimum height minHeight, and a maximum height maxHeight. Therefore we can set a range of width and height.

As we've said before, the geometry of boxes consists of a Size. Consequently, the Size satisfies the constraints.

Each child in the rendered widget tree, gets BoxConstraints from its parent. After that, the child picks us the size that satisfies the BoxConstraints adjusting with the parent's size.

Certainly, with the size, position changes. A child does not know its position.

Why?

Because, if the parent adds some padding the child's position changes with it.

We'll see that in a minute.

Consider a Flutter app where Scaffold widget acts as the immediate child of Material App widget.

As a result, the Scaffold takes the entire screen from its immediate parent Material App.

Next, the Scaffold passes its constraints to its immediate child Center widget. And then the Center takes the constraints or size from Scaffold. However, as the Scaffold widget allocates some space for the App Bar widget, therefore, Center doesn't get the whole screen.

Let us see the code.

```
1  import 'package:flutter/material.dart';
2
3  class BoxConstraintsSample extends StatelessWidget {
4  const BoxConstraintsSample({Key? key}) : super(key: key);
5
6  @override
7  Widget build(BuildContext context) {
8      return const MaterialApp(
9      title: 'BoxConstraints Sample',
```

```
    debugShowCheckedModeBanner: false,
    home: BoxConstraintsSampleHomme(),
    );
}
}

class BoxConstraintsSampleHomme extends StatelessWidget {
const BoxConstraintsSampleHomme({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('BoxConstraints Sample'),
    ),
    body: Center(
        child: Container(
        color: Colors.redAccent,
        padding: const EdgeInsets.all(
            20,
        ),
        child: const Text(
            'Box',
            style: TextStyle(
            fontFamily: 'Allison',
            color: Colors.black38,
            fontSize: 60,
            fontWeight: FontWeight.bold,
            ),
        ),
        constraints: const BoxConstraints(
            minHeight: 70,
            minWidth: 70,
            maxHeight: 200,
            maxWidth: 200,
        ),
        ),
    ),//Center
    );
}
}
```

If we run the code, we'll see the following effect.

The above code tells us about the Container's constraints that point to BoxConstraints, this way.

```
constraints: const BoxConstraints(
            minHeight: 70,
            minWidth: 70,
            maxHeight: 200,
            maxWidth: 200,
        ),
```

The Container widget has a constraints parameter that points to the BoxConstraints widget, which simply defines the minimum and maximum width, height. And the range is between 70px to 200px.

As a result, if we try to make the Text widget bigger, that will not display the whole text, in that case.

Why?

Because, Container passes its constraints to its child Text widget and it gets that exact value. That means, the size of Text widget must remain in between 70px to 200px.

What happens if we try to pass the same constraints to another Container, which will act as the immediate child.

Let's change our code.

How do you use box constraints in Flutter?

To use box constraints in Flutter, we must understand how BoxConstraints widget acts, maintaining the range of width and height.

Let's change our above code and it will look like the following, now.

```
import 'package:flutter/material.dart';

class BoxConstraintsSample extends StatelessWidget {
const BoxConstraintsSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'BoxConstraints Sample',
    debugShowCheckedModeBanner: false,
    home: BoxConstraintsSampleHomme(),
    );
}
}
```

```
15
16 class BoxConstraintsSampleHomme extends StatelessWidget {
17 const BoxConstraintsSampleHomme({Key? key}) : super(key: key);
18
19 @override
20 Widget build(BuildContext context) {
21     return Scaffold(
22     appBar: AppBar(
23         title: const Text('BoxConstraints Sample'),
24     ),
25     body: Center(
26         child: Container(
27         color: Colors.redAccent,
28         padding: const EdgeInsets.all(
29             20,
30         ),
31         constraints: const BoxConstraints(
32             minHeight: 170,
33             minWidth: 170,
34             maxHeight: 400,
35             maxWidth: 400,
36         ),
37         child: Container(
38             color: Colors.blueAccent[200],
39             padding: const EdgeInsets.all(
40             20,
41             ),
42             child: const Text(
43             'Box',
44             style: TextStyle(
45                 fontFamily: 'Allison',
46                 color: Colors.white,
47                 fontSize: 60,
48                 fontWeight: FontWeight.bold,
49             ),
50             ),
51             constraints: const BoxConstraints.expand(
52             height: 100,
53             width: 100,
54             ),
55         ),
56         ), //container
57     ), //Center
```

```
    );
}
}
```

In the above code, we find two Container widgets. Both have constraints defined. The first Container have constraints like the following:

```
constraints: const BoxConstraints(
            minHeight: 170,
            minWidth: 170,
            maxHeight: 400,
            maxWidth: 400,
        ),
```

And its child, the second Container has constraints like the following:

```
constraints: const BoxConstraints.expand(
            height: 100,
            width: 100,
            ),
```

The first Container's constraints define a range of width and height. However,the second Container's constraints point to BoxConstraints constructor BoxConstraints.expand.

What does this mean, as long as the size of the child Container is concerned?

We can explain it this way.

Since the child Container receives its constraints from its parent Container. Within that range it can expand its width and height up to 100px. Not more than that.

Moreover, this expansion process starts from the Center.

Understanding how these widgets or boxes handle the constraints in flutter is very important for the beginners.

In general, there are three kind of boxes that we'll encounter while we learn Flutter.

Firstly, the widgets like Center and ListView; they always try to be as big as possible.

Secondly, the widgets like Transform and Opacity always try to take the same size as their children.

And, finally, there are widgets like Image and Text that try to fit to a particular size.

As we'll progress, we'll find, how these constraints vary from widget to widget.

As example, we can remember the role of Center widget. It always maintains the maximum size. The minimum constraint does not work here.

What is widget in Flutter

In Flutter everything is Widget. But in reality it's a class and creates its instances also.

Firstly, widget in flutter is a class that describes how our flutter app should look like by creating its instances.

Secondly, the central idea behind using widgets is to build our user interface using widgets. To clarify, widgets are like boxes on the mobile, tab or desktop screen. Consequently, since each box has a size, every widget has constraints that deal with width and height.

Therefore, finally, we can define a widget as a class that builds or rebuilds its description; and, our flutter app works on that principle.

For a beginner, we need to add something more with this definition.

In Flutter everything is widget. As a result, we must think widget as a central hierarchy in a Flutter framework.

Let us see a minimalist Flutter App to understand this concept first.

```
import 'package:flutter/material.dart';
import 'widgets/first_flutter_app.dart';

void main() {
runApp(
    const FirstFlutterApp(),
);
}
```

The above code shows us that the runApp() function takes the given Widget FirstFlutterApp() and makes it the root of the widget tree.

And this is our first custom widget that will sit on the top of the tree.

With reference to the main() function we must also add that to work it properly we need to import Material App library. However, we don't want to dig deep now. Just to make it simple, let's know that we need to have a material design so we can show our widget boxes. Right?

Further, we have created a sub-directory called "widgets" in our "lib" directory and keep the code of FirstFlutterApp(). Remember that also represents a class of hierarchy. Therefore we need to import that local library too.

That is why we need to import that also.

```
import 'widgets/first_flutter_app.dart';
```

Now, we can take a look at the custom widget that we've built.

```
import 'package:flutter/material.dart';

class FirstFlutterApp extends StatelessWidget {
const FirstFlutterApp({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: Center(
        child: Text(
        'Hello, Flutter!',
        ),
    ),
    );
}
}
```

The FirstFlutterApp() extends Stateless widget. Widgets have state. But, don't worry, we'll discuss it later.

The widget tree consists of three more widgets. The MaterialApp, Center widget and its child, the Text widget.

As a consequence, the framework forces the root widget to cover the screen. It places the text "Hello, Flutter" at the Center of the screen.

Since we have used MaterialApp, we don't have to worry about the text direction. The MaterialApp will take care of that.

Since each widget is a Dart class, we need to instantiate each widget and want them to show up at the particular location. This is done through the Element class. This is nothing but an instantiation of a Widget at a particular location in the tree.

What do we see?

A text "Hello, Flutter".

However, it is displayed on the particular position in the widget tree. Now, this widget tree can be a complex one.

As a result, widgets can be inflated into more elements as the tree grows in size.

Before we close down, one thing to remember. A widget is an immutable description of part of a user interface.

We'll discuss that in a minute when we will discuss Element class in detail.

What is element in Flutter

We can locate any widget in a widget tree by its element that is an instantiation of widget.

We have been trying to understand how Flutter works from a beginner's point of view. We've already discussed how Widget, constraints and BoxConstraints are related in our previous sections. Now, we'll try to understand what's the role of Element in Flutter.

So far, we've learned that widget is a class and a root widget might have a tree of other widgets.

That's fine.

However, we haven't known so far, how the instantiation of Widget works. After all, widget is a class. Therefore, that must be instantiated.

Element is the answer. In a widget tree, we can find the instantiation of a widget in a particular location, because that is a unique element.

Suppose we reuse Text widget several times in a widget tree. As a result, Text widget is instantiated several times. But each element is unique. Moreover, we can find each Text widget by that unique Element.

We can add one more comment to the above statement. With the widget tree, an Element tree is also formed.

Since widgets are immutable, any widget can be used to configure multiple sub-trees. However, due to the presence of Element, we can easily find the widget's specific location.

Let us see a screenshot and try to understand how this concept works.

In the above screenshot we have two Text widgets, two Text Button widgets. And at the bottom, we have a Floating action button also.

Let's see the associated code.

```
import 'package:flutter/material.dart';

class UnderstandingElement extends StatelessWidget {
const UnderstandingElement({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Constraint Sample',
    debugShowCheckedModeBanner: false,
    home: UnderstandingElementHomme(),
    );
}
```

```
}

class UnderstandingElementHomme extends StatelessWidget {
const UnderstandingElementHomme({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Element Sample'),
    ),
    body: Center(
        child: Column(
        children: [
            const Text('Element One: Text Widget'),
            const Text('Element Two: : Text Widget'),
            Row(
            children: [
                TextButton(
                onPressed: () {},
                child: const Text('Element Three: TextButton Widget'),
                ),
                TextButton(
                onPressed: () {},
                child: const Text('Element Four: TextButton Widget'),
                ),
            ],
            )
        ],
        ),
    ),
    floatingActionButton: FloatingActionButton(
        onPressed: () {},
        child: const Icon(Icons.add_a_photo),
    ),
    );
}
}
```

Now, in the above widget tree, how can we find the second Text widget? It has a unique element or instantiation of that Widget, which actually points to a particular location.

Actually that element represents the rendered widget on the screen.

If the parent widget rebuilds and creates a new widget for this location, a widget associated with a given element can also change.

What is Align in Flutter

Align widget allows us to place a child widget anywhere we want inside another widget.

Align is a widget that aligns its child within itself. Moreover, based on the child's size, it optionally sizes itself.

For instance, let us think about a minimal Flutter app that aligns a Flutter logo at top right.

```
Center(
        child: Column(
        children: [
            Container(
            height: 120.0,
            width: 120.0,
            color: Colors.blue[50],
            child: const Align(
                alignment: Alignment.topRight,
                child: FlutterLogo(
                size: 60,
                ),
            ),
            ),
```

Just run the app, and see how it looks like.

The code is quite simple. It gives the Container a light blue color. Besides, it has definite width and height, which are tight constraints.

Now, as a child the Align widget places its child, a Flutter logo at the top right corner inside the Container.

However, we could have placed it with an alignment of Alignment.bottomRight.

To do that we should have given the Container a tight constraint, just like before, and that constraint should be bigger than the Flutter logo.

Next, we consider another piece of code that might place three Flutter logo at three different positions inside the same Container.

As we find when we run the code, three Flutter logos show up at three different positions.

Let's see the full code snippet first. After that, we'll discuss the code.

```dart
import 'package:flutter/material.dart';

class AlignSample extends StatelessWidget {
const AlignSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Align Sample',
    debugShowCheckedModeBanner: false,
    home: AlignSampleHomme(),
    );
}
}

class AlignSampleHomme extends StatelessWidget {
const AlignSampleHomme({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Align Sample'),
    ),
    body: Center(
        child: Column(
        children: [
            Container(
            height: 120.0,
            width: 120.0,
            color: Colors.blue[50],
            child: const Align(
                alignment: Alignment.topRight,
                child: FlutterLogo(
                size: 60,
                ),
            ),
            ),
            const SizedBox(
            height: 10,
            ),
            Container(
            height: 120.0,
```

```
            width: 120.0,
            color: Colors.yellow[50],
            child: const Align(
                alignment: Alignment(0.2, 0.6),
                child: FlutterLogo(
                size: 60,
                ),
            ),
            ),
            const SizedBox(
            height: 10,
            ),
            Container(
            height: 120.0,
            width: 120.0,
            color: Colors.red[50],
            child: const Align(
                alignment: FractionalOffset(0.2, 0.6),
                child: FlutterLogo(
                size: 60,
                ),
            ),
            ),
        ],
        ),
    ),
    floatingActionButton: FloatingActionButton(
        onPressed: () {},
        child: const Icon(Icons.add_a_photo),
    ),
    );
}
}
```

The alignment property describes a point in the child's coordinate system and a different point in the coordinate system of this widget.

After that, the Align widget positions the child, here a Flutter logo, in a way so that both points are lined up on top of each other.

In the first case, the Align widget uses one of the defined constants from Alignment, which is Alignment.topRight.

```
Container(
            height: 120.0,
            width: 120.0,
            color: Colors.blue[50],
            child: const Align(
                alignment: Alignment.topRight,
                child: FlutterLogo(
                size: 60,
                ),
            ),
            ),
```

As a result, this constant value places the FlutterLogo at the top right corner of the parent blue Container.

However, the second case is different, where the Alignment defines a single point.

```
Container(
            height: 120.0,
            width: 120.0,
            color: Colors.yellow[50],
            child: const Align(
                alignment: Alignment(0.2, 0.6),
                child: FlutterLogo(
                size: 60,
                ),
            ),
            ),
```

It calculates the position of the Flutter logo in a different way.

The formula is, the result of (0.2 * width of FlutterLogo/2 + width of FlutterLogo/2) comes to a whole number, that is 36.0.

And this is a point in the coordinate system of Flutter logo.

The next point is defined by this formula – (0.6 * height of FlutterLogo/2 + height of FlutterLogo/2), which is equal to 48.0. Moreover, it's point in the coordinate system of the Align widget.

As a result Align will place the FlutterLogo at (36.0, 48.0) according to this coordinate system.

Although in the third example, the calculation goes in the same direction; yet the result differs with the previous one.

```
alignment: FractionalOffset(0.2, 0.6)
```

Consequently, the position of Flutter logo changes.

How to use aspect ratio widget

The AspectRatio Widget tries to find the best size to maintain aspect ratio of a child widget.

The AspectRatio widget attempts to size the child to a specific aspect ratio.

Suppose we have a Container widget with width 100, and height 100. In that case the aspect ratio would be 100/100; that is, 1.0.

Now, each Widget has its own constraints. As a result, the AspectRatio Widget tries to find the best size to maintain aspect ratio. However, while doing so it respects it's layout constraints.

Let's run the code and see the effect where we have used three different types of aspect ratio.

A Container widget has an AspectRatio widget, which has a child Container in a different color.

As a result, we see different types of color combination.

Let's see the full code now.

```
import 'package:flutter/material.dart';

class AspectRatioSample extends StatelessWidget {
const AspectRatioSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'AspectRatio Sample',
    debugShowCheckedModeBanner: false,
    home: AspectRatioSampleHomme(),
    );
}
}

class AspectRatioSampleHomme extends StatelessWidget {
const AspectRatioSampleHomme({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
```

```
        title: const Text('AspectRatio Sample'),
    ),
    body: Center(
        child: Column(
        children: [
            Container(
            color: Colors.red,
            alignment: Alignment.center,
            padding: const EdgeInsets.all(10),
            width: 100.0,
            height: 100.0,
            child: AspectRatio(
                aspectRatio: 2.0,
                child: Container(
                width: 50.0,
                height: 50.0,
                color: Colors.yellow,
                ),
            ),
            ),
            const SizedBox(
            height: 10,
            ),
            Container(
            color: Colors.blue,
            alignment: Alignment.center,
            width: 100.0,
            height: 100.0,
            child: AspectRatio(
                aspectRatio: 2.0,
                child: Container(
                width: 80.0,
                height: 70.0,
                color: Colors.white,
                ),
            ),
            ),
            const SizedBox(
            height: 10,
            ),
            Container(
            color: Colors.green,
            alignment: Alignment.center,
```

```
66              width: 100.0,
67              height: 100.0,
68              child: AspectRatio(
69                  aspectRatio: 0.5,
70                  child: Container(
71                  width: 100.0,
72                  height: 50.0,
73                  color: Colors.black26,
74                  ),
75              ),
76              ),
77          ],
78          ),
79      ),
80      floatingActionButton: FloatingActionButton(
81          onPressed: () {},
82          child: const Icon(Icons.add_a_photo),
83      ),
84      );
85  }
86  }
```

Remember, in each case, the AspectRatio widget tries to find the best possible size and adjusts the child accordingly.

It comes to our help, when we try to change the size of an image on the fly.

What is Baseline in Flutter

When we try to position a child widget inside or outside of parent widget, Baseline helps us.

Baseline is a widget that positions its child according to the child widget's baseline.

Does it not make any sense?

Well, truly it didn't make any sense to me also, when I first encountered this widget.

In fact, the above statement doesn't really make any sense if we don't turn this abstraction into a concrete example.

Therefore, firstly, let's see one screenshot of a simple Flutter app where we've used Baseline widget.

Secondly, we'll look into the code and try to understand how this widget works.

In the above image we see three Baseline examples. The first one consists of two Container widgets.

Let's see the code.

```
Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Baseline(
                baseline: 0,
                baselineType: TextBaseline.alphabetic,
                child: Container(
                width: 50,
                height: 50,
                color: Colors.purple,
                ),
            ),
            ),
```

The Baseline widget has two required parameters. The baseline, and the baselineType. The second parameter means the type of the baseline.

As a result, we need to supply value to those parameters.

The baseline parameter plays the most important role, of course. It requires a double value.

In the above case, the baseline is zero.

So it sits on top of the parent Container.

How does it happen?

It happens because the Baseline widget tries to shift the Child Container's bottom or baseline by calculating the distance from the top of the parent Container. Since it's zero, it cannot shift it. So it sits on its top.

As a result, it cannot enter the parent Container.

In the second case, the Child Container's baseline is 50.

```
Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Baseline(
                baseline: 50,
                baselineType: TextBaseline.alphabetic,
                child: Container(
                width: 50,
                height: 50,
                color: Colors.purple,
```

```
12                  ),
13              ),
14              ),
```

Therefore, the baseline logical pixels below the top of the parent Container is 50px. As a result, the bottom of the child Container shifts 50px inside the parent Container.

And the parent Container contains the child in the middle.

Take a look at the screenshot above, you'll understand how it works.

How do you use baseline in flutter?

Most importantly, we use Baseline widget when we want to position the child widget's bottom according to the distance from the top of the parent widget.

When the child Container's baseline is 100px, it moves its bottom 100px exactly, from the top of the parent Container.

As a consequence, the child's bottom merges with the parent's bottom. The above screenshot displays the same thing.

```
Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Baseline(
                baseline: 100,
                baselineType: TextBaseline.alphabetic,
                child: Container(
                width: 50,
                height: 50,
                color: Colors.purple,
                ),
            ),
            ),
```

If we start increasing the value of the baseline, and make it 110px, what happens?

If we run the code we'll see that the child Container moves outside the parent Container.

Moreover, from the top of the parent Container's to the bottom of the child Container, the distance is 110px exact.

Let's take a look at the full code finally.

```
import 'package:flutter/material.dart';

class BaselineSample extends StatelessWidget {
const BaselineSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Baseline Sample',
    debugShowCheckedModeBanner: false,
    home: BaselineSampleHomme(),
    );
}
}

class BaselineSampleHomme extends StatelessWidget {
const BaselineSampleHomme({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Baseline Sample'),
    ),
    body: Center(
        child: Column(
        children: [
            const SizedBox(
            height: 100,
            ),
            Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Baseline(
                baseline: 0,
                baselineType: TextBaseline.alphabetic,
                child: Container(
                width: 50,
                height: 50,
                color: Colors.purple,
                ),
            ),
```

```
            ),
            const SizedBox(
            height: 20,
            ),
            Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Baseline(
                baseline: 50,
                baselineType: TextBaseline.alphabetic,
                child: Container(
                width: 50,
                height: 50,
                color: Colors.purple,
                ),
            ),
            ),
            const SizedBox(
            height: 20,
            ),
            Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Baseline(
                baseline: 110,
                baselineType: TextBaseline.alphabetic,
                child: Container(
                width: 50,
                height: 50,
                color: Colors.purple,
                ),
            ),
            ),
        ],
        ),
    ),
    );
}
}
```

We can provide a negative value to the baseline of the child Container. Try to make it -50.

At that instance, the bottom of the child Container moves away upward and goes out of the parent Container in the upward direction.

Try it. Happy fluttering.

How to use theme in Flutter

We can create a custom global theme and apply that across the entire flutter app.

Layout of any Flutter app greatly depends on a global theme that we can apply to any widget in the widget tree.

In this section we'll learn how we can create a global theme object that can help us to change or modify our style globally, across the whole flutter app.

To do that, we will create a new ThemeData object first.

```
final globalTheme = ThemeData(
primarySwatch: Colors.deepOrange,
textTheme: const TextTheme(
    bodyText1: TextStyle(
    fontSize: 22,
    height: 1.2,
    ),
    bodyText2: TextStyle(
    color: Colors.blue,
    fontSize: 20,
    fontWeight: FontWeight.bold,
    height: 1.0,
    ),
    caption: TextStyle(
    fontSize: 16,
    fontWeight: FontWeight.bold,
    fontStyle: FontStyle.italic,
    height: 1.2,
    ),
    headline1: TextStyle(
    color: Colors.deepOrange,
    fontFamily: 'Allison',
    fontWeight: FontWeight.bold,
    fontSize: 60,
    ),
    headline2: TextStyle(
    color: Colors.black38,
```

```
    fontSize: 30,
    fontWeight: FontWeight.bold,
    ),
),
appBarTheme: const AppBarTheme(
    backgroundColor: Colors.amber,
    // This will control the "back" icon
    iconTheme: IconThemeData(color: Colors.red),
    // This will control action icon buttons that locates on the right
    actionsIconTheme: IconThemeData(color: Colors.blue),
    centerTitle: false,
    elevation: 15,
    titleTextStyle: TextStyle(
    color: Colors.deepPurple,
    fontFamily: 'Allison',
    fontWeight: FontWeight.bold,
    fontSize: 40,
    ),
),
);
```

Usually the ThemeData class comes with a default configuration. With the help of context we can apply the Material App theme property across the flutter app.

However, we can also override the default configuration and modify it according to our choices.

Exactly that's what we've done here, in the above code.

We've defined the configuration of the overall visual Theme for a MaterialApp or a widget sub-tree within the app.

That's why we've included a Theme widget at the top of the sub-tree.

Normally, if we don't create our own theme object, we can obtain the default configuration with Theme.of. Material components that typically depend on the colorScheme and textTheme. These properties are always provided with non-null values.

As a result, with the help of context of the nearest BuildContext ancestor the static Theme.of method finds the ThemeData value.

In our case, however, we apply the ThemeData object the following way.

```
class GlobalThemeSample extends StatelessWidget {
const GlobalThemeSample({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    debugShowCheckedModeBanner: false,
    title: 'Global Theme Sample',
    theme: globalTheme,
    home: const GlobalThemeSampleHome(),
    );
}
}
```

Once we have declared overall visual Theme for our MaterialApp, we can apply the properties across the widget tree.

It starts with AppBar theme, right way.

```
class GlobalThemeSampleHome extends StatelessWidget {
const GlobalThemeSampleHome({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text(
        'Global Theme Sample',
        style: globalTheme.appBarTheme.titleTextStyle,
        ),
    ),
    body: const GlobalThemeBody(),
    );
}
}
```

Next, the rest global theme properties are applied across the widget tree.

```
class GlobalThemeBody extends StatelessWidget {
const GlobalThemeBody({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    DateTime now = DateTime.now();
    String stringDate = DateFormat('yyyy-MM-dd – kk:mm').format(now);
    return Center(
    child: Column(
        children: [
        const SizedBox(
            height: 10,
        ),
        Container(
            height: 70,
            width: 70,
            color: Colors.blue[50],
            child: const Align(
            alignment: Alignment.topCenter,
            child: FlutterLogo(
                size: 60,
            ),
            ),
        ),
        const SizedBox(
            height: 10,
        ),
        Text(
            'Headline 1',
            style: globalTheme.textTheme.headline1,
        ),
        const SizedBox(
            height: 10,
        ),
        Text(
            'Headline 2',
            style: globalTheme.textTheme.headline2,
        ),
        Container(
            margin: const EdgeInsets.all(5),
            padding: const EdgeInsets.all(5),
            child: Text(
            'Body Text 1: Here goes some introduction about yourself.',
```

```
            style: globalTheme.textTheme.bodyText1,
            ),
        ),
        Container(
            margin: const EdgeInsets.all(5),
            padding: const EdgeInsets.all(5),
            child: Text(
            'Body Text 2: Here goes some more information regarding your works.',
            style: globalTheme.textTheme.bodyText2,
            ),
        ),
        Container(
            margin: const EdgeInsets.all(5),
            padding: const EdgeInsets.all(5),
            child: Text(
            stringDate,
            style: globalTheme.textTheme.caption,
            ),
        ),
        ],
    ),
    );
}
}
```

Now we can run the app and see the output.

Certainly, we can apply more style. Moreover, we can create a custom Theme class and with the help of Provider package we can give user a chance to change theme anytime.

We'll discuss that in a separate section.

- All related code snippets - https://github.com/sanjibsinha/build_blog_with_flutter/tree/how_-provider_works[82]

How to use theme with Provider on flutter

Using a custom theme across the Flutter app, can also be done through Provider package.

In our previous section we've discussed how we can use a global theme in Flutter and apply that style across the Widget tree. However, there are better ways to do that.

Certainly, using Provider package to pass ThemeData object is one of them.

[82]https://github.com/sanjibsinha/build_blog_with_flutter/tree/how_provider_works

Moreover, it makes our flutter app more performant.

However, we can use Provider to share a Theme across an entire app in many ways. One of them is to provide a unique ThemeData to the MaterialApp constructor.

In this section, we'll see how to use class Constructors to pass the provided value or ThemeData object. The next section will show you a more robust and easy way to use Provider in providing ThemeData object with less line of code.

Sharing colors, and a unique font style throughout a Flutter app, is always a great idea. However, using the Provider package to use that ThemeData object across the app, will be more interesting.

Therefore, let's jump in and start building our global theme in a different way.

Let us create three folders in our lib folder first. Model, View and Controller.

In model, we create a class first.

```
import 'package:flutter/material.dart';

class GlobalTheme with ChangeNotifier {
final globalTheme = ThemeData(
    primarySwatch: Colors.deepOrange,
    textTheme: const TextTheme(
    bodyText1: TextStyle(
        fontSize: 22,
    ),
    bodyText2: TextStyle(
        color: Colors.blue,
        fontSize: 18,
        fontWeight: FontWeight.bold,
    ),
    caption: TextStyle(
        fontSize: 16,
        fontWeight: FontWeight.bold,
        fontStyle: FontStyle.italic,
    ),
    headline1: TextStyle(
        color: Colors.deepPurple,
        fontSize: 50,
        fontFamily: 'Allison',
    ),
    headline2: TextStyle(
        color: Colors.deepOrange,
        fontSize: 30,
        fontWeight: FontWeight.bold,
```

```
    ),
    ),
    appBarTheme: const AppBarTheme(
    backgroundColor: Colors.amber,
    // This will control the "back" icon
    iconTheme: IconThemeData(color: Colors.red),
    // This will control action icon buttons that locates on the right
    actionsIconTheme: IconThemeData(color: Colors.blue),
    centerTitle: false,
    elevation: 15,
    titleTextStyle: TextStyle(
        color: Colors.deepPurple,
        fontWeight: FontWeight.bold,
        fontFamily: 'Allison',
        fontSize: 40,
    ),
    ),
);
}
```

We've created a globalTheme property that defines a unique global theme for us, using the ThemeData widget.

Next, we'll add the dependency in our pubspec.yaml file, so that we can use Provider package to provide that model class object throughout our app.

```
dependencies:
cupertino_icons: ^1.0.2
flutter:
    sdk: flutter
intl: ^0.17.0
provider: ^6.0.1
```

Now, we should keep the Provider above the root folder and use multi provider and ChangeNotifier-Provider, so that we can later use other Providers as well.

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'model/global_theme.dart';
import 'view/home.dart';

void main() {
runApp(
    /// Providers are above [Root App] instead of inside it, so that tests
    /// can use [Root App] while mocking the providers
    MultiProvider(
    providers: [
        ChangeNotifierProvider(create: (_) => GlobalTheme()),
    ],
    child: const Home(),
    ),
);
}
```

In our Home widget, we can get the unique and custom ThemeData object using Provider.of(context).

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '/model/global_theme.dart';
import 'home_page.dart';

class Home extends StatelessWidget {
const Home({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    final ThemeData globalTheme = Provider.of<GlobalTheme>(context).globalTheme;
    return MaterialApp(
    title: 'Flutter Demo',
    theme: globalTheme,
    home: HomePage(
        homeTheme: globalTheme,
    ),
    );
}
}
```

Next step is much easier. Just pass the same ThemeData object, to child widgets in the Widget tree through the class constructors.

In fact, app-wide themes are now just ThemeData object provided by the Provider. Once the theme parameter of Material App takes that unique ThemeData object, we can use that across the app. Because Themes widgets created at the root of an app by the MaterialApp can be used anywhere.

Since we've defined our custom Theme, we can use it within our own widgets. Flutter's Material widgets also use our custom Theme to set the background colors and font styles for AppBars, Buttons and more.

Now we'll pass the custom ThemeData object quite easily and use in our child widget.

```
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';

class HomePage extends StatelessWidget {
final ThemeData homeTheme;
const HomePage({Key? key, required this.homeTheme}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text(
        'Testing Global Theme with Provider',
        style: homeTheme.appBarTheme.titleTextStyle,
        ),
    ),
    body: HomeBody(
        homeTheme: homeTheme,
    ),
    );
}
}

class HomeBody extends StatelessWidget {
final ThemeData homeTheme;
const HomeBody({Key? key, required this.homeTheme}) : super(key: key);

@override
Widget build(BuildContext context) {
    DateTime now = DateTime.now();
    String stringDate = DateFormat('yyyy-MM-dd – kk:mm').format(now);
    return Center(
    child: Column(
        children: [
```

```
35          Container(
36              margin: const EdgeInsets.all(5),
37              padding: const EdgeInsets.all(5),
38              child: Text(
39              'Headline 2 theme style provided by provider',
40              style: homeTheme.textTheme.headline2,
41              ),
42          ),
43          Container(
44              margin: const EdgeInsets.all(5),
45              padding: const EdgeInsets.all(5),
46              child: Text(
47              'Headline 1 theme style provided by provider',
48              style: homeTheme.textTheme.headline1,
49              ),
50          ),
51          Container(
52              margin: const EdgeInsets.all(5),
53              padding: const EdgeInsets.all(5),
54              child: Text(
55              'Body Text 2: Here goes some introduction about yourself. Theme by provi\
56  der.',
57              style: homeTheme.textTheme.bodyText2,
58              ),
59          ),
60          Container(
61              margin: const EdgeInsets.all(5),
62              padding: const EdgeInsets.all(5),
63              child: Text(
64              'Body Text 1: Here goes some more information regarding your works. Them\
65  e by provider.',
66              style: homeTheme.textTheme.bodyText1,
67              ),
68          ),
69          Container(
70              margin: const EdgeInsets.all(5),
71              padding: const EdgeInsets.all(5),
72              child: Text(
73              'Datetime theme style provided by provider: $stringDate',
74              style: homeTheme.textTheme.caption,
75              ),
76          ),
77          ],
```

```
78      ),
79      );
80  }
81  }
```

As a result, our app uses that custom theme from AppBar to the body.

- All related code snippets - https://github.com/sanjibsinha/build_blog_with_flutter/tree/theme-with-provider[83]

[83] https://github.com/sanjibsinha/build_blog_with_flutter/tree/theme-with-provider

16. Flutter 2.8, Future, await, async and Database

Flutter 2.8 has added a major performance booster to our Flutter Applications.

What is new in Flutter 2.8

Using Flutter's single codebase we can create applications for Android, iOS, Windows, Linux, Web and many more. Flutter 2.8 has added many interesting features that makes this development much easier. In addition it's added a major performance booster.

If you've been reading my blog, you may have noticed that I have written about the earlier Flutter revisions and releases.

However, Flutter 2.8 has overtaken 2.5 and improving the performance of Flutter applications on mobile devices.

We can simply upgrade to the new version by issuing this command on our terminal.

```
flutter upgrade
```

It takes some time to upgrade depending on the speed of your internet.

Then you can check the Flutter and Dart version.

```
Flutter 2.8.0 • channel stable • https://github.com/flutter/flutter.git
Framework • revision cf44000065 (3 days ago) • 2021-12-08 14:06:50 -0800
Engine • revision 40a99c5951
Tools • Dart 2.15.0
....
Dart SDK version: 2.15.0 (stable) (Fri Dec 3 14:23:23 2021 +0100) on "linux_x64"
```

As a result, our Flutter applications open faster and consume less memory.

As some of Google's core apps like Google Play and Stadia uses Flutter, it's quite expected that Google will keep improving Flutter core.

Flutter 2.8 ha made it much easier to connect to Firebase. The good news is Firebase plugins for Flutter have been upgraded from "Beta" to "Stable."

Now, sign-in has also become easier with a Widget.

We've just seen that with the revision and new release of Flutter 2.8, Dart programming language SDK has also been updated to 2.15.

We'll discuss in a separate post how we can utilise Dart's new features. Some features will definitely boost the development of Flutter User Interfaces.

Dart 2.15 also brings concurrency improvements, enhanced enumerations, and optimisations that provide a 10 percent reduction in memory utilisation.

However, I personally like the web view part.

The webview_flutter plugin has been upgraded to 3.0 and that means provides preliminary support for a new platform: the web.

To use it, we can add the following line to your pubspec.yaml:

```
dependencies:
webview_flutter: ^3.0.0
webview_flutter_web: ^0.1.0
```

To sum up, Flutter 2.8 promises faster startup and lower resource requirements for mobile apps. We can also easily connect with the back-end services.

Let us try webview_flutter plugin in a simple Flutter 2.8 application.

```
import 'package:flutter/material.dart';
import 'package:webview_flutter/webview_flutter.dart';

void main() {
runApp(const MyApp());
}

class MyApp extends StatelessWidget {
const MyApp({Key? key}) : super(key: key);

// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(
        primarySwatch: Colors.blue,
    ),
    home: const MyHomePage(title: 'Flutter Demo Home Page'),
    );
}
```

```
}

class MyHomePage extends StatefulWidget {
const MyHomePage({Key? key, required this.title}) : super(key: key);

final String title;

@override
State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
@override
Widget build(BuildContext context) => Scaffold(
        appBar: AppBar(title: const Text('Flutter WebView example')),
        body: const WebView(initialUrl: 'https://sanjibsinha.com'),
    );
}
```

Just run the code and you'll find that web view has brought the web pages as we want.

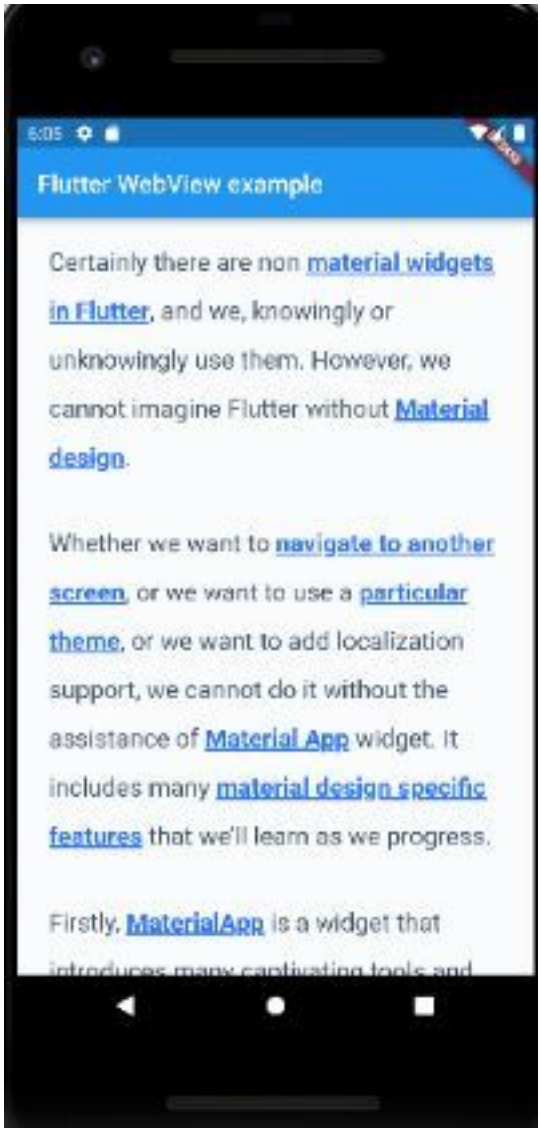

Figure 16.1 – Flutter 2.8 and web view 3.0 plugin

We can always check out the new webview codelab, which might give us a proper guide to host web content in our Flutter app.

Future, await and async

Future in Flutter or Dart gives us a promise token and returns a value at in future.

Flutter is mainly single thread. Why so? Because Dart language is a single threaded language. However, Flutter uses several threads to do its work.

Does it sound confusing?

Don't worry. Basically, in Flutter all of our Dart code runs on User Interface thread. In spite of that, it uses other threads besides that. Just to name a few, there are Platform thread, I/O thread, etc.

On the contrary Android applications are not single threaded. Besides the main thread Android applications can run the heavy jobs in worker or background thread.

What do we mean by heavy jobs?

Well, let me explain. In any mobile application, users need to accomplish many tasks simultaneously. Consider an event like button click. That might do some small logical operations, such as get the result of a small addition. The main thread can handle such small operations and doesn't take much time.

However, an image to be opened using a network, or downloading a file isn't a small operation. As a result, the main thread needs to do the heavy lifting. And, that might halt the whole program.

However, Dart and Flutter have its answer. They together perform long-running operations with the help of Future API, async, await keywords, and then functions.

Together they perform asynchronous programming in Flutter.

Asynchronous doesn't mean multi-threaded. Basically it means the code shouldn't run at the same time and the job of main thread shouldn't be burdensome.

In case of Flutter, asynchronous means that an operation is scheduled to be run on the same thread after other tasks have finished.

With reference to asynchronous programming we need to understand isolate also. In a separate section, we'll discuss isolate.

When the conception of isolate comes into the picture, Dart no longer remains single threaded language. Because it can create separate isolate. Although within an isolate Dart again runs on a single thread.

Future in Flutter or Dart gives us a promise token and says that a value will be returned at some point in future.

It never says in which thread it will have its job done at that certain point.

This part is little tricky. However, we need to see how we can write a simple Dart code to understand this concept.

```dart
import 'dart:async';

void main(List<String> args) {
print('main thread starts >>>>>');
print('main process starts >>>>>');

openImage();

print('main process ends and starts counting seconds... >>>>>');
}

void openImage() async {
var imageFile = await downloadImage();
print('The downloaded file is --> $imageFile');
print('main thread ends after 10 seconds....');
}

Future<String> downloadImage() {
var image = Future<String>.delayed(Duration(seconds: 10), () {
    return 'Here is an imaginative image downloaded.';
});
return image;
}
```

Let's try to understand the above code.

Depending on that, as a consequence, we can later create a database operations in Flutter. Remember, database queries takes time and we must do using Future.

Future works on <T>, or type. In our case, we've used String type and returns a text after 10 seconds.

However, when we run the program, the default process starts the main thread. And it prints, main thread starts and main process starts.

Then after 10 seconds we get the result that our Future object returns. And at the same time, the main thread closes down.

Let's run and watch the output, so that everything makes sense.

```
main thread starts >>>>>
main process starts >>>>>
main process ends and starts counting seconds... >>>>>
The downloaded file is --> Here is an imaginative image downloaded.
main thread ends after 10 seconds....
Exited
```

To sum up, here a Future object has produced a value of type String. A Future object belongs to any one of the states – either it's completed, or it's uncompleted.

When the main thread starts and we call a Future, it queues up work and returns an uncompleted state.

But we don't see that part in our bare eyes.

When the Future's operation is finished, it returns a completed state with a value of same type.

If it cannot, Future completes with an error.

Which database we use in Flutter

SQLite database with future, await, and async create, retrieve, update, or delete data in Flutter.

We can use SQLite database in Flutter. However, we need to use a special package or plugin sqflite which is available in pub.dev. We also need to use Future API, async, await keywords, and then functions to make it successful.

We've discussed this feature for absolute beginners in previous section, is Flutter single thread?

Anyway, as a result, the sqflite package provides classes and functions to interact with a SQLite database. Moreover, using SQLite database is better than using a local file, or key-value store.

There are reasons to do that.

SQLite database provides faster CRUD. That is, we can create, retrieve, update and delete data. And, it's always better than the local persistent solutions.

In this section we'll see how we can create a users table in our SQLite database, and retrieve that data on our Flutter application.

It will be a gentle introduction to SQLite database with Flutter. We'll see how we can create a database, insert some data and after that, retrieve them.

Later as we progress, we'll see how we can improve the other functionalities. To get the code snippets used in this section, we have a respective GitHub repository. If you have interest, please download and run the code.

Besides sqflite package, we need to use another package path, that will define the location for storing the database on the disk.

For the beginners, here is a guide what SQLite database is.

What is SQLite database?

SQLite is a C-language library that implements many features at one go. It is small, fast, self-contained, high-reliability, full-featured, SQL database engine.

By the way, SQLite is the most used database engine in the world. Besides, SQLite database file format is stable, cross-platform, and backwards compatible.

There are over 1 trillion SQLite databases in active use at present.

Therefore, let's go ahead and make our first Flutter Application with SQLite database.

How do you make a database on Flutter app?

Let's add the dependencies in our pubspec.yaml file first.

```
dependencies:
flutter:
    sdk: flutter
sqflite:
path:
```

After that, create a model folder, inside lib folder, and create one user class and a database helper class there.

Firstly, let's take a look at the user model class.

```
class User {
final int? id;
final String name;
final String location;

User({
    this.id,
    required this.name,
    required this.location,
});

User.fromMap(Map<String, dynamic> res)
    : id = res["id"],
        name = res["name"],
        location = res["location"];

Map<String, Object?> toMap() {
```

```
    return {
    'id': id,
    'name': name,
    'location': location,
    };
}
}
```

To get data stored in SQLite database, we need to convert them to a map. The reason is simple. After all, in our Flutter Application we need to convert them to a list of items.

That's why we have created a named constructor User.fromMap() and a method toMap().

Secondly, we'll create a table with the help of the helper class.

```
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';

import 'user.dart';

class DatabaseHandler {
Future<Database> initializeDB() async {
    String path = await getDatabasesPath();
    return openDatabase(
    join(path, 'usersfirst.db'),
    onCreate: (database, version) async {
        await database.execute(
        "CREATE TABLE usersfirst(id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT NOT\
 NULL, location TEXT NOT NULL)",
        );
    },
    version: 1,
    );
}

Future<int> insertUser(List<User> users) async {
    int result = 0;
    final Database db = await initializeDB();
    for (var user in users) {
    result = await db.insert('usersfirst', user.toMap());
    }
    return result;
}

```

```
Future<List<User>> retrieveUsers() async {
    final Database db = await initializeDB();
    final List<Map<String, Object?>> queryResult = await db.query('usersfirst');
    return queryResult.map((e) => User.fromMap(e)).toList();
}
}
```

The method getDatabasePath() of sqflite package will get the default database location. However, the join() method is inside the package path that will join the given path into a single path.

As a matter of fact, two packages sqflite and path are necessary for this reason.

In addition, to keep our first Flutter SQLite database application simple, we're going to create, insert, and retrieve the users. We'll add the list of users manually in our main method.

Now, we're going to display the users data.

```
import 'package:flutter/material.dart';

import 'model/database_handler.dart';
import 'model/user.dart';

void main() {
runApp(const MyApp());
}

/// adding first branch
class MyApp extends StatelessWidget {
const MyApp({Key? key}) : super(key: key);

// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Flutter smimple database',
    theme: ThemeData(
        primarySwatch: Colors.blue,
    ),
    home: const MyHomePage(title: 'Flutter smimple database'),
    );
}
}

class MyHomePage extends StatefulWidget {
```

```
28  const MyHomePage({Key? key, required this.title}) : super(key: key);
29
30  final String title;
31
32  @override
33  State<MyHomePage> createState() => _MyHomePageState();
34  }
35
36  class _MyHomePageState extends State<MyHomePage> {
37  late DatabaseHandler handler;
38
39  @override
40  void initState() {
41      Future<int> addUsers() async {
42      User firstUser = User(name: "Mana", location: "Nabagram");
43      User secondUser = User(name: "Babu", location: "Nabagram");
44      User thirdUser = User(name: "Pata", location: "Nabagram");
45      List<User> listOfUsers = [
46          firstUser,
47          secondUser,
48          thirdUser,
49      ];
50      return await handler.insertUser(listOfUsers);
51      }
52
53      super.initState();
54      handler = DatabaseHandler();
55      handler.initializeDB().whenComplete(() async {
56      await addUsers();
57      setState(() {});
58      });
59  }
60
61  @override
62  Widget build(BuildContext context) {
63      return Scaffold(
64      appBar: AppBar(
65          title: Text(widget.title),
66      ),
67      body: FutureBuilder(
68          future: handler.retrieveUsers(),
69          builder: (BuildContext context, AsyncSnapshot<List<User>> snapshot) {
70          if (snapshot.hasData) {
```

```
71              return ListView.builder(
72              itemCount: snapshot.data?.length,
73              itemBuilder: (BuildContext context, int index) {
74                  return Card(
75                  child: ListTile(
76                      key: ValueKey<int>(snapshot.data![index].id!),
77                      contentPadding: const EdgeInsets.all(8.0),
78                      title: Text(
79                      snapshot.data![index].name,
80                      style: Theme.of(context).textTheme.headline3,
81                      ),
82                      subtitle: Text(
83                      snapshot.data![index].location.toString(),
84                      style: Theme.of(context).textTheme.headline5,
85                      ),
86                  ),
87                  );
88              },
89              );
90          } else {
91              return const Center(child: CircularProgressIndicator());
92          }
93          },
94      ),
95      );
96  }
97  }
```

Next, we create an instance of class DatabaseHandler first. With the help of the database handler object we can call initalizeDb() method to create the SQLite database.

We know that Future in Flutter or Dart gives us a promise token and says that a value will be returned at some point in future. Therefore, when Future is completed, addUsers() method is called.

Consequently, the addUsers() method calls insertUsers() method to insert the list of users to the SQLite database.

Next, the FutureBuilder widget builds itself based on the latest snapshot of interaction with a Future. Moreover, unless the Future is completed, it gives us the uncompleted state which shows a circular progress indicator.

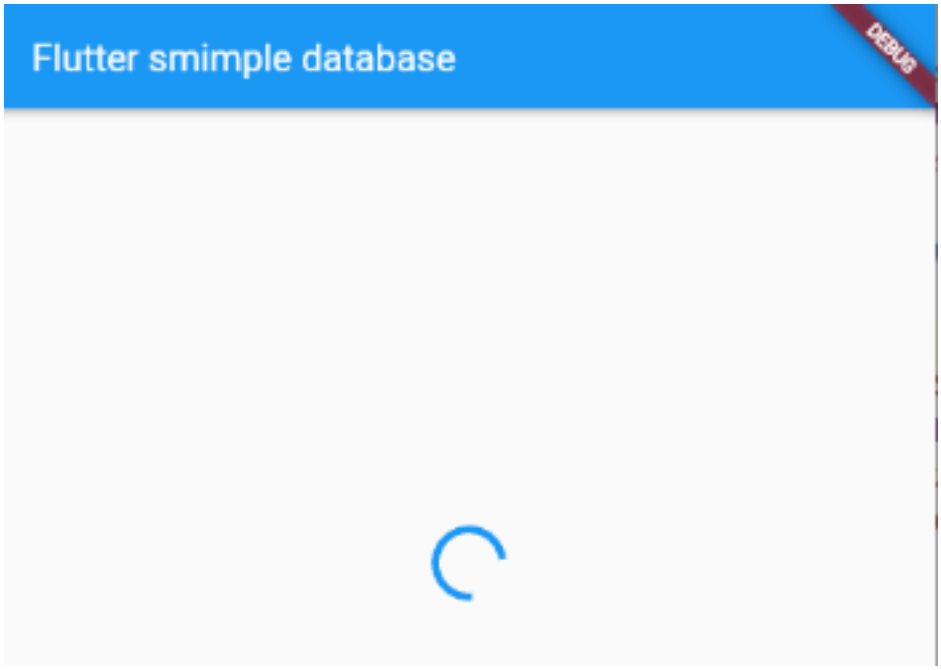

Figure 16.2 – Flutter application tries to retrieve data from SQLite database with FutureBuilder

However, when the value is returned in Future, it retrieves data from SQLite database successfully.

Figure 16.3 – FutureBuilder retrieves data from SQLite database in Flutter

By the way, we've added this list of users manually.

```
Future<int> addUsers() async {
    User firstUser = User(name: "Mana", location: "Nabagram");
    User secondUser = User(name: "Babu", location: "Nabagram");
    User thirdUser = User(name: "Pata", location: "Nabagram");
    List<User> listOfUsers = [
        firstUser,
        secondUser,
        thirdUser,
    ];
    return await handler.insertUser(listOfUsers);
    }
```

In the next section, we'll try to implement other features of CRUD in our Flutter Application.

With the help of SQLite database in accordance with future, await, and async we can insert, retrieve, update, or delete data in Flutter.

SQLite Database and Flutter

In this section, we'll take a look at how we can insert data to SQL database and display them. As we progress, we'll learn the other techniques to update and delete data.

It is preferable to use SQLite database in Flutter, because it is faster than local file. However, we need to use a special package or plugin sqflite which is available in pub.dev. We also need to use Future API, async, await keywords, and then functions to make it successful.

We've discussed Future, await and async for absolute beginners in previous section, is Flutter single thread?

Anyway, as a result, the sqflite package provides classes and functions to interact with a SQLite database.

What is SQLite database?

SQLite is a C-language library that implements many features at one go. It is small, fast, self-contained, high-reliability, full-featured, SQL database engine.

By the way, SQLite is the most used database engine in the world. Besides, SQLite database file format is stable, cross-platform, and backwards compatible.

There are over 1 trillion SQLite databases in active use at present.

Therefore, let's go ahead and make our first Flutter Application with SQLite database.

How to insert data in SQL database in Flutter?

Firstly, we need a Text Controller to type on the screen. Right?

Then, we need a Text Button to press, so that that piece of data will be inserted into the SQLite database.

Secondly, we need to add the dependency.

```
dependencies:
cupertino_icons: ^1.0.2
flutter:
    sdk: flutter
intl: ^0.17.0
path_provider: ^2.0.8
provider: ^6.0.1
sqflite:
```

Next, we need a model data class and Database Handler helper class that will connect our SQLite database to the model data class.

First, data model class.

```
class User {
final int? id;
final String name;

User({
    this.id,
    required this.name,
});

User.fromMap(Map<String, dynamic> res)
    : id = res["id"],
        name = res["name"];

Map<String, Object?> toMap() {
    return {
    'id': id,
    'name': name,
    };
}
}
```

We need to map that data model class so that later we can work on it in Flutter.

To get data stored in SQLite database, we need to convert them to a map. The reason is simple. After all, in our Flutter Application we need to convert them to a list of items.

That's why we have created a named constructor User.fromMap() and a method toMap().

Secondly, we'll create a table with the help of the helper class.

Why?

Because, the database helper class will provide the methods that will create the database table, and help us to insert and retrieve data from SQLite database.

```
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';

import 'user.dart';

class DatabaseHandler {
Future<Database> initializeDB() async {
    String path = await getDatabasesPath();
    return openDatabase(
    join(path, 'usersix.db'),
    onCreate: (database, version) async {
```

```
12          await database.execute(
13          "CREATE TABLE usersix(id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT NOT NU\
14 LL)",
15          );
16      },
17      version: 1,
18      );
19  }
20
21  Future<int> insertUser(List<User> users) async {
22      int result = 0;
23      final Database db = await initializeDB();
24      for (var user in users) {
25      result = await db.insert('usersix', user.toMap());
26      }
27      return result;
28  }
29
30  Future<List<User>> retrieveUsers() async {
31      final Database db = await initializeDB();
32      final List<Map<String, Object?>> queryResult = await db.query('usersix');
33      return queryResult.map((e) => User.fromMap(e)).toList();
34  }
35  }
```

It's a good practice that we break down these code snippets and keep this data model and helper class in our model folder inside lib folder.

The method getDatabasePath() of sqflite package will get the default database location. However, the join() method is inside the package path that will join the given path into a single path.

As a matter of fact, two packages sqflite and path are necessary for this reason.

In addition, to keep our first Flutter SQLite database application simple, we're going to create, insert, and retrieve the users. We'll add the list of users manually in our main method.

Our next challenge is to show a text field to the user so that she can type any text and press the button.

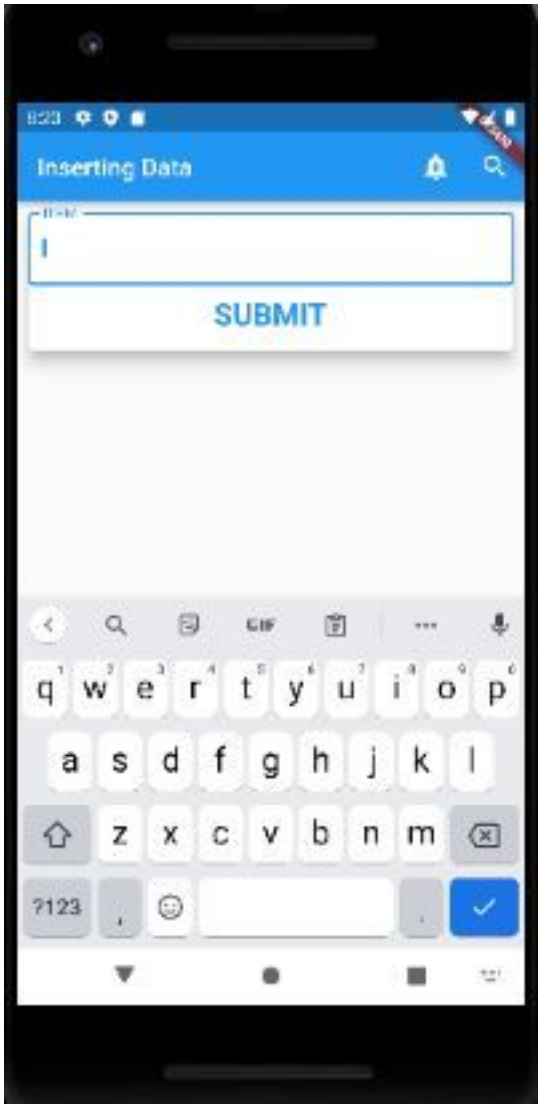

Figure 16.4 – Inserting data in SQLite database in Flutter

At the same page we need to show the Navigate button, that will take us to another screen where the inserted data will be displayed.

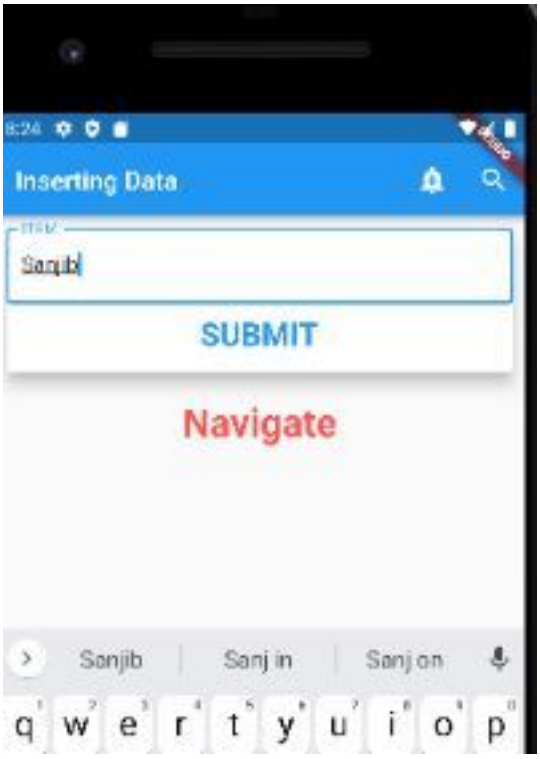

Figure 16.5 – Data is being inserted in SQLite database in Flutter

```dart
import 'package:flutter/material.dart';
import 'package:flutter_data_and_backend/view/future_dark.dart';

import 'model/user.dart';

void main() {
runApp(const MyApp());
}

/// we're now in branch six
///
class MyApp extends StatelessWidget {
const MyApp({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'data',
    home: MyAppHome(),
    );
}
```

```
}

class MyAppHome extends StatefulWidget {
const MyAppHome({Key? key}) : super(key: key);

@override
State<MyAppHome> createState() => _MyAppHomeState();
}

class _MyAppHomeState extends State<MyAppHome> {
final List<User> usersList = [];

final nameController = TextEditingController();

void addName(String name) {
    final user = User(
    name: name,
    );
    setState(() {
    usersList.add(user);
    });
}

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Inserting Data'),
        actions: <Widget>[
        IconButton(
            icon: const Icon(Icons.add_alert),
            tooltip: 'Show Snackbar',
            onPressed: () {
            ScaffoldMessenger.of(context).showSnackBar(
                const SnackBar(
                content: Text('A SnackBar'),
                ),
            );
            },
        ),
        IconButton(
            icon: const Icon(Icons.search_outlined),
            tooltip: 'Search',
```

```
            onPressed: () {
            // our code
            },
        ),
        ],
    ),
    body: Center(
        child: Column(
        children: [
            Container(
            padding: const EdgeInsets.all(5),
            child: Card(
                elevation: 10,
                child: Column(
                children: [
                    TextField(
                    decoration: const InputDecoration(
                        border: OutlineInputBorder(),
                        labelText: 'ITEM',
                        suffixStyle: TextStyle(
                        fontSize: 50,
                        fontWeight: FontWeight.bold,
                        ),
                    ),
                    controller: nameController,
                    ),
                    TextButton(
                    onPressed: () {
                        addName(
                        nameController.text,
                        );
                    },
                    child: const Text(
                        'SUBMIT',
                        style: TextStyle(
                        fontSize: 25,
                        fontWeight: FontWeight.bold,
                        ),
                    ),
                    ),
                    const SizedBox(
                    height: 5,
                    ),
```

```
                ],
                ),
            ),
            ),
            NavigationWidget(usersList: usersList),
        ],
        ),
    ),
    );
}
}

class NavigationWidget extends StatelessWidget {
const NavigationWidget({
    Key? key,
    required this.usersList,
}) : super(key: key);

final List<User> usersList;

@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        padding: const EdgeInsets.all(5),
        height: 150,
        width: 350,
        child: Column(
        children: usersList.map((e) {
            return Column(
            children: [
                TextButton(
                onPressed: () {
                    Navigator.push(
                    context,
                    MaterialPageRoute(
                        builder: (context) => FutureDark(
                        name: e.name,
                        ),
                    ),
                    );
                },
                child: const Text(
```

```
                    'Navigate',
                    style: TextStyle(
                    fontSize: 30.0,
                    fontWeight: FontWeight.bold,
                    color: Colors.redAccent,
                    ),
                ),
                ),
            ],
            );
        }).toList(),
        ),
    ),
    );
}
}
```

The above code could be broken down to more pages or screens using more custom widgets.

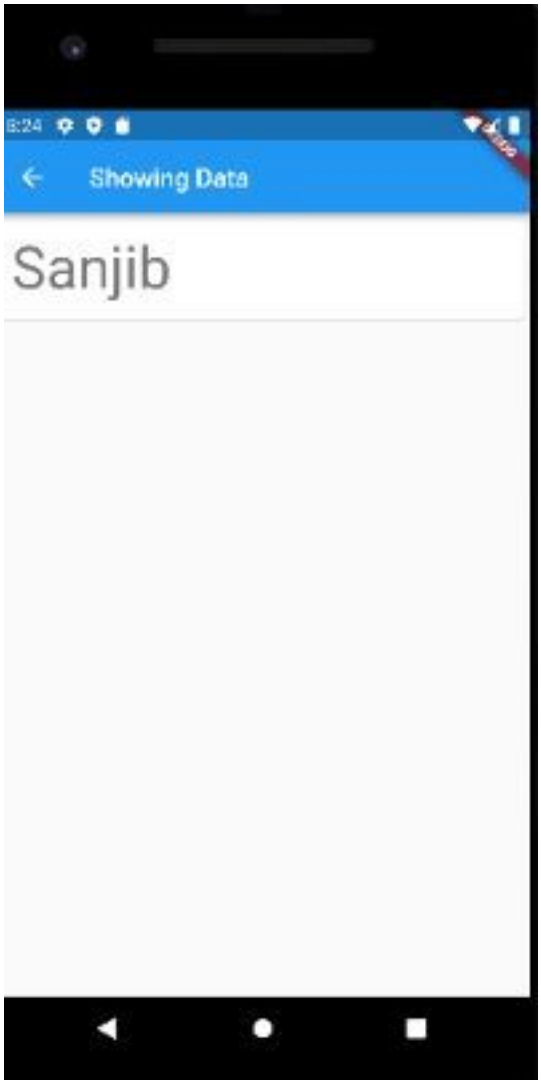

Figure 16.6 – Data from SQLite database is being shown on Flutter screen

However, in one place will help us to understand the mechanism of how we have used Text Controller, Text Button and a Material Page Route to insert data and after that, we can see them.

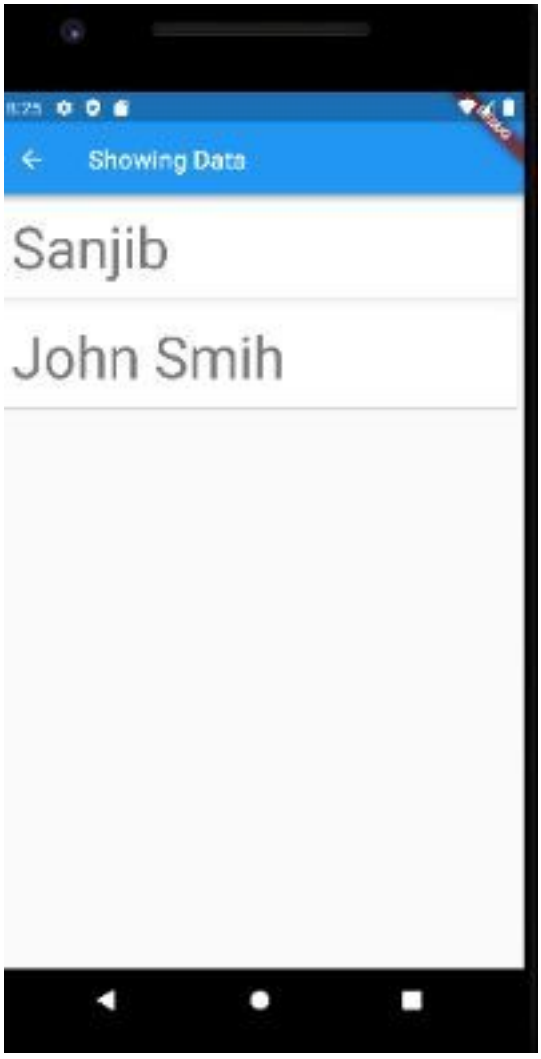

Figure 16.7 – Second Data from SQLite database is being shown on Flutter screen

Second Data from SQLite database is being shown on Flutter screen How to retrieve data from SQLite database in Flutter?

Retrieving data from SQLite database is much easier than inserting data.

With the help of Future API, async, await keywords, and then functions and FutureBuilder widget, we can do that.

At the same screen we also catch the data or list item that we've sent from the home page.

```dart
import 'package:flutter/material.dart';

import 'package:flutter_data_and_backend/model/database_handler.dart';
import 'package:flutter_data_and_backend/model/user.dart';

class FutureDark extends StatefulWidget {
const FutureDark({
    Key? key,
    required this.name,
}) : super(key: key);

final String name;

@override
State<FutureDark> createState() => _FutureDarkState();
}

class _FutureDarkState extends State<FutureDark> {
DatabaseHandler? handler;
@override
void initState() {
    List<User> users = [
    User(name: widget.name.toString()),
    ];
    Future<int> addUsers() async {
    return await handler!.insertUser(users);
    }

    super.initState();
    handler = DatabaseHandler();
    handler!.initializeDB().whenComplete(() async {
    await addUsers();
    setState(() {});
    });
}

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Showing Data'),
    ),
    body: FutureBuilder(
```

```
44          future: handler!.retrieveUsers(),
45          builder: (BuildContext context, AsyncSnapshot<List<User>> snapshot) {
46          if (snapshot.hasData) {
47              return ListView.builder(
48              itemCount: snapshot.data?.length,
49              itemBuilder: (BuildContext context, int index) {
50                  return Card(
51                  child: ListTile(
52                      key: ValueKey<int>(snapshot.data![index].id!),
53                      contentPadding: const EdgeInsets.all(8.0),
54                      title: Text(
55                      snapshot.data![index].name,
56                      style: Theme.of(context).textTheme.headline3,
57                      ),
58                  ),
59                  );
60              },
61              );
62          } else {
63              return const Center(child: CircularProgressIndicator());
64          }
65          },
66      ),
67      );
68  }
69  }
```

The Future Builder is a widget that builds itself based on the latest snapshot of interaction with a Future.

We've obtained the future must have been obtained earlier, during State.initState, State.didUpdateWidget, or State.didChangeDependencies.

FutureBuilder must not be created during the State.build or StatelessWidget.build method call when constructing the FutureBuilder.

17. Create, Retrieve, Update and Delete with SQLite Database: Build A Blog and My Diary Application in Flutter

In the previous chapter, we have had a gentle introduction on SQLite database and Flutter. We have learned how to create a SQLite database with the help of “sqflite” package.

In addition we’ve also seen how to use “path” package to define a local path to create the database.

In the following two sections, we’ll learn how we can plan, create and modify a SQLite database in Flutter, so that we can successfully do the CRUD, or Create, Retrieve, Update and Delete data and build a Blog Application using that knowledge.

And in the last section we’ll convert the existing Blog Application and refactor it to a “My Diary” Aplication.

SQLite Blog in Flutter: First Part

This is the first part of building a Blog application with SQLite database in Flutter.

We’re going to learn how we can build a SQLite Blog Application in Flutter. As it sounds, a Blog App must fulfill some criteria.

What are they?

The SQLite database should allow us to insert, update or delete data. Moreover, we need to retrieve data as well.

For better understanding, let’s break this flutter tutorial in a few parts.

So, in this first part will teach us to learn a couple of things.

Firstly, we’ll use a Flutter package or plugin, sqflite which is available in pub.dev.

Secondly, we also need to use Future API, async, await keywords, and then functions to make it successful.

We’ve discussed Future, await and async for absolute beginners in previous section, is Flutter single thread? Therefore, if you’re a beginner, you might take a look before we start.

Which database is best for Flutter?

The sqflite package provides classes and functions to interact with a SQLite database.

And, finally, we need a Text Controller to type on the screen. Right?

We also need a Text Button or Elevated Button to press, so that that piece of data will be inserted into the SQLite database.

To begin with, we need to add the dependencies in pubspec.yaml.

```
dependencies:
cupertino_icons: ^1.0.2
flutter:
    sdk: flutter
flutter_staggered_grid_view: ^0.4.1
intl: ^0.17.0
path:
provider: ^6.0.1
sqflite:
```

After that, we need to use await and async in our entry point.

```
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'view/all_pages.dart';

Future main() async {
WidgetsFlutterBinding.ensureInitialized();
await SystemChrome.setPreferredOrientations([
    DeviceOrientation.portraitUp,
    DeviceOrientation.portraitDown,
]);

runApp(const MyApp());
}

class MyApp extends StatelessWidget {
static const String title = 'Blogs';

const MyApp({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) => MaterialApp(
```

```
        debugShowCheckedModeBanner: false,
        title: title,
        themeMode: ThemeMode.light,
        theme: ThemeData(
        primaryColor: Colors.pink.shade200,
        scaffoldBackgroundColor: Colors.pink.shade600,
        appBarTheme: const AppBarTheme(
            backgroundColor: Colors.transparent,
            elevation: 0,
        ),
        ),
        home: const AllPages(),
    );
}
```

In our Material App widget, we've defined the global theme. In addition, we pass that theme to our Home page AllPages.

Next, we need to define couple of things in this page, AllPages. If there is no data in our SQLite database, then it will show a page like below.

Figure 17.1 – Home page of the Blog App in Flutter

As a result, we can start adding blog items to this Flutter Application. However, in our Home page, AllPages stateful Widget, we must define that.

Let's take a look at the AllPages code snippet.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_staggered_grid_view/flutter_staggered_grid_view.dart';
import '/model/blogs.dart';
import '/model/blog.dart';
import 'edit.dart';
import 'detail.dart';
import '/controller/blog_card.dart';

class AllPages extends StatefulWidget {
const AllPages({Key? key}) : super(key: key);

@override
_AllPagesState createState() => _AllPagesState();
}

class _AllPagesState extends State<AllPages> {
late List<Blog> blogs;
bool isLoading = false;

@override
void initState() {
    super.initState();

    refreshingAllBogs();
}

@override
void dispose() {
    BlogDatabaseHandler.instance.close();

    super.dispose();
}

Future refreshingAllBogs() async {
    setState(() => isLoading = true);

    blogs = await BlogDatabaseHandler.instance.readAllBlogs();

    setState(() => isLoading = false);
}

@override
Widget build(BuildContext context) => Scaffold(
```

```
        appBar: AppBar(
        title: const Text(
            'Blogs',
            style: TextStyle(fontSize: 24),
        ),
        actions: [
            Padding(
            padding: const EdgeInsets.symmetric(vertical: 8, horizontal: 12),
            child: ElevatedButton(
                style: ElevatedButton.styleFrom(
                onPrimary: Colors.white,
                primary: Colors.pink.shade900,
                ),
                onPressed: () async {
                await Navigator.of(context).push(
                    MaterialPageRoute(builder: (context) => const EditPage()),
                );

                refreshingAllBogs();
                },
                child: const Text('Add Blog'),
            ),
            )
        ],
        ),
        body: Center(
        child: isLoading
            ? const CircularProgressIndicator()
            : blogs.isEmpty
                ? const Text(
                    'No Blogs in the beginning...',
                    style: TextStyle(color: Colors.white, fontSize: 60),
                    )
                : buildingAllBlogs(),
        ),
    );

Widget buildingAllBlogs() => StaggeredGridView.countBuilder(
        padding: const EdgeInsets.all(8),
        itemCount: blogs.length,
        staggeredTileBuilder: (index) => const StaggeredTile.fit(2),
        crossAxisCount: 4,
        mainAxisSpacing: 4,
```

```
87          crossAxisSpacing: 4,
88          itemBuilder: (context, index) {
89          final blog = blogs[index];
90
91          return GestureDetector(
92              onTap: () async {
93              await Navigator.of(context).push(MaterialPageRoute(
94                  builder: (context) => DetailPage(blogId: blog.id!),
95              ));
96
97              refreshingAllBogs();
98              },
99              child: BlogCard(blog: blog, index: index),
100         );
101         },
102     );
103 }
```

The above Widget plays a very important role in our Flutter SQLite Blog Application.

Firstly, it would let us allow to add a new blog. To do that, this widget takes us to a different Widget, EditPage.

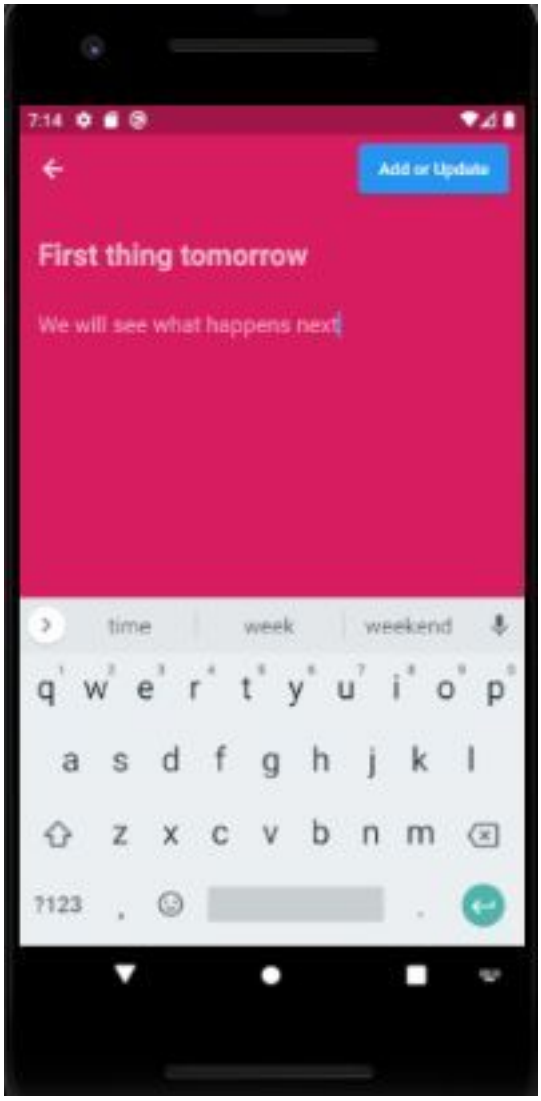

Figure 17.2 – Inserting data into Blog App in Flutter

After that, once we are done in the EditPage, that again sends us back to this Widget, AllPages. And, here, we start seeing all the blogs.

Figure 17.3 – Retrieving data in Blog App in Flutter

Retrieving data from SQLite database is much easier than inserting data. However, we'll learn everything from scratch here while building this Blog app in Flutter.

Meanwhile, with the help of Future API, async, await keywords, and then functions and Future-Builder widget, we can do that.

At the same screen we also catch the data or list item that we've sent from the home page.

In the above code, we've found a couple of interesting thing.

Firstly, the home page imports couple of other pages and two model classes. The model classes provide the data models.

Secondly, the model class also serves as database handler utilities.

And, finally, we needed the edit page and the detail page to serve other purposes, such as updating

and deleting items.

At the end, after finishing all tasks, we come back to the home page again.

This is the flow of logic that runs this SQLite Blog application in Flutter.

In the next section, we'll take a look at how we can build the data model and database utilities with the help of sqflite package.

By that time, if you have interest, please visit the respective GitHub repository.

- All related Code Snippet in this GitHub repository[84]

SQLite Blog, Flutter: Second Part

This is the second part of building a Blog application with SQLite database in Flutter.

We've already started building the SQLite Blog application in Flutter. The previous section has discussed the application structure. In this second part, we'll concentrate on database connection, data model classes.

As we proceed, we'll also learn how we can develop the same application thematically. Therefore, we'll add more functionalities. We'll keep in mind that our application should look great and should be user friendly.

First thing first. Let's recapitulate a few things about SQLite database and Flutter.

Firstly, we'll use a Flutter package or plugin, sqflite which is available in pub.dev.

Secondly, we also need to use Future API, async, await keywords, and then functions to make it successful.

We've discussed Future, await and async for absolute beginners in previous section, is Flutter single thread? Therefore, if you're a beginner, you might take a look before we start.

Which DB is best for Flutter?

SQLite database is one of the best DB for Flutter. Although there are couple more. Like Hive, Firebase, etc.

However we're using SQLite database for this Blog application in Flutter, because it's local, fast and easy to maintain.

First of all, we need a Blog data model.

Let's keep that class in model folder.

[84] https://github.com/sanjibsinha/flutter_data_and_backend/tree/test-blog

```dart
const String tableOfBlogs = 'Blogs';

class BlogFields {
static final List<String> values = [
    /// Adding all fields
    id, title, description, time
];

static const String id = '_id';
static const String title = 'title';
static const String description = 'description';
static const String time = 'time';
}

class Blog {
final int? id;
final String title;
final String description;
final DateTime createdTime;

const Blog({
    this.id,
    required this.title,
    required this.description,
    required this.createdTime,
});

Blog copy({
    int? id,
    String? title,
    String? description,
    DateTime? createdTime,
}) =>
    Blog(
        id: id ?? this.id,
        title: title ?? this.title,
        description: description ?? this.description,
        createdTime: createdTime ?? this.createdTime,
    );

static Blog fromJson(Map<String, Object?> json) => Blog(
        id: json[BlogFields.id] as int?,
        title: json[BlogFields.title] as String,
```

```
        description: json[BlogFields.description] as String,
        createdTime: DateTime.parse(json[BlogFields.time] as String),
    );

Map<String, Object?> toJson() => {
        BlogFields.id: id,
        BlogFields.title: title,
        BlogFields.description: description,
        BlogFields.time: createdTime.toIso8601String(),
    };
}
```

We need to Map the data model object. It's because Flutter wants a list to display.

Figure 17.4 – Retrieving blog items on Home page

Second thing we need a database handler class that will create the table, and at the same time, it'll do the CRUD.

We've learned what CRUD is. Create, retrieve, update and delete.

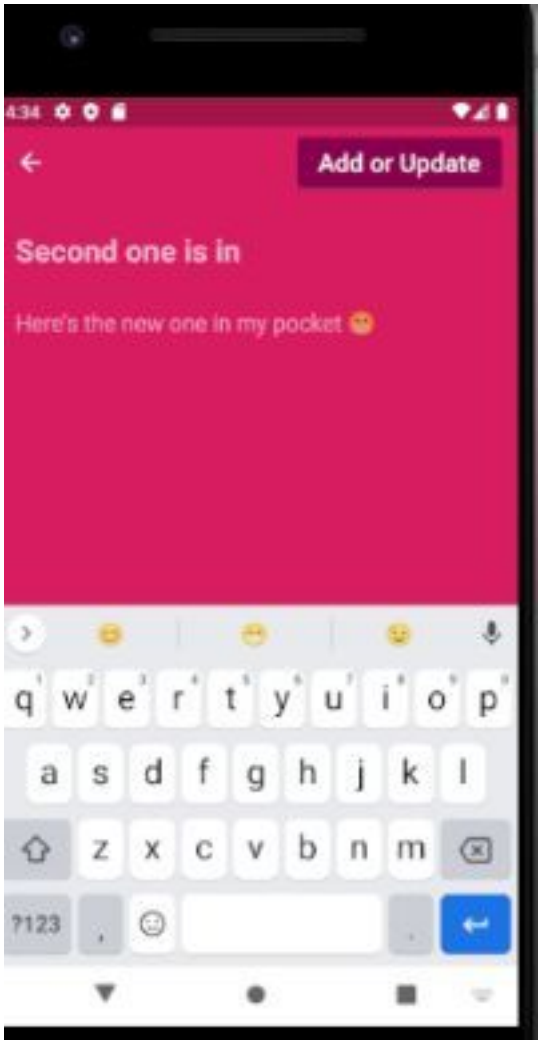

Figure 17.5 – Adding Blog items to SQLite database in Flutter

In this SQLite Blog application we'll also do the same. However, we've tweaked the previous home page code to accommodate a floating action button.

```
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';
import '../model/blog.dart';

class BlogDatabaseHandler {
static final BlogDatabaseHandler instance = BlogDatabaseHandler._init();

static Database? _database;

BlogDatabaseHandler._init();

```

```
Future<Database> get database async {
    if (_database != null) return _database!;

    _database = await _initDB('newblogs.db');
    return _database!;
}

Future<Database> _initDB(String filePath) async {
    final dbPath = await getDatabasesPath();
    final path = join(dbPath, filePath);

    return await openDatabase(path, version: 1, onCreate: _createDB);
}

Future _createDB(Database db, int version) async {
    const idType = 'INTEGER PRIMARY KEY AUTOINCREMENT';
    const textType = 'TEXT NOT NULL';

    await db.execute('''
CREATE TABLE $tableOfBlogs (
${BlogFields.id} $idType,
${BlogFields.title} $textType,
${BlogFields.description} $textType,
${BlogFields.time} $textType
)
''');
}

Future<Blog> create(Blog blog) async {
    final db = await instance.database;

    final id = await db.insert(tableOfBlogs, blog.toJson());
    return blog.copy(id: id);
}

Future<Blog> readBlog(int id) async {
    final db = await instance.database;

    final maps = await db.query(
    tableOfBlogs,
    columns: BlogFields.values,
    where: '${BlogFields.id} = ?',
    whereArgs: [id],
```

```
55      );
56
57      if (maps.isNotEmpty) {
58      return Blog.fromJson(maps.first);
59      } else {
60      throw Exception('ID $id not found');
61      }
62  }
63
64  Future<List<Blog>> readAllBlogs() async {
65      final db = await instance.database;
66
67      const orderBy = '${BlogFields.time} ASC';
68
69      final result = await db.query(tableOfBlogs, orderBy: orderBy);
70
71      return result.map((json) => Blog.fromJson(json)).toList();
72  }
73
74  Future<int> update(Blog blog) async {
75      final db = await instance.database;
76
77      return db.update(
78      tableOfBlogs,
79      blog.toJson(),
80      where: '${BlogFields.id} = ?',
81      whereArgs: [blog.id],
82      );
83  }
84
85  Future<int> delete(int id) async {
86      final db = await instance.database;
87
88      return await db.delete(
89      tableOfBlogs,
90      where: '${BlogFields.id} = ?',
91      whereArgs: [id],
92      );
93  }
94
95  Future close() async {
96      final db = await instance.database;
97
```

```
    db.close();
}
}
```

In the above code, everything is very verbose.

Firstly, we've mentioned a path so the database gets created locally and stores data locally.

Secondly, we've created the database and table with fields. As we see, we've kept it simple.

Thirdly, we've created methods that will read, update, delete items.

Finally, we've closed the database.

At the same time, we've changed the code of our home page slightly. As a result, we can now add or update the blog items either from AppBar, or through the floating action button.

```
import 'package:flutter/material.dart';
import 'package:flutter_staggered_grid_view/flutter_staggered_grid_view.dart';
import '/model/blogs.dart';
import '/model/blog.dart';
import 'edit.dart';
import 'detail.dart';
import '/controller/blog_card.dart';

class AllPages extends StatefulWidget {
const AllPages({Key? key}) : super(key: key);

@override
_AllPagesState createState() => _AllPagesState();
}

class _AllPagesState extends State<AllPages> {
late List<Blog> blogs;
bool isLoading = false;

@override
void initState() {
    super.initState();

    refreshingAllBogs();
}

@override
void dispose() {
    BlogDatabaseHandler.instance.close();
```

```
30
31      super.dispose();
32  }
33
34  Future refreshingAllBogs() async {
35      setState(() => isLoading = true);
36
37      blogs = await BlogDatabaseHandler.instance.readAllBlogs();
38
39      setState(() => isLoading = false);
40  }
41
42  @override
43  Widget build(BuildContext context) => Scaffold(
44          appBar: AppBar(
45          title: const Text(
46              'Blogs',
47              style: TextStyle(fontSize: 24),
48          ),
49          actions: [
50              Padding(
51              padding: const EdgeInsets.symmetric(vertical: 8, horizontal: 12),
52              child: ElevatedButton(
53                  style: ElevatedButton.styleFrom(
54                  onPrimary: Colors.white,
55                  primary: Colors.pink.shade900,
56                  ),
57                  onPressed: () async {
58                  await Navigator.of(context).push(
59                      MaterialPageRoute(builder: (context) => const EditPage()),
60                  );
61
62                  refreshingAllBogs();
63                  },
64                  child: const Text(
65                  'Add or Update Blog',
66                  style: TextStyle(
67                      fontSize: 20,
68                  ),
69                  ),
70              ),
71              )
72          ],
```

```
        ),
        body: Center(
        child: isLoading
            ? const CircularProgressIndicator()
            : blogs.isEmpty
                ? const Text(
                    'No Blogs in the beginning...',
                    style: TextStyle(color: Colors.white, fontSize: 60),
                    )
                : buildingAllBlogs(),
        ),
        floatingActionButton: FloatingActionButton.extended(
        tooltip: 'Add or Update Blog',
        foregroundColor: Colors.white,
        backgroundColor: Colors.pink.shade900,
        onPressed: () async {
            await Navigator.of(context).push(
            MaterialPageRoute(builder: (context) => const EditPage()),
            );

            refreshingAllBogs();
        },
        label: const Text(
            'Add or Update Blog',
            style: TextStyle(
            fontSize: 30,
            ),
        ),
        ),
    );

Widget buildingAllBlogs() => StaggeredGridView.countBuilder(
        padding: const EdgeInsets.all(8),
        itemCount: blogs.length,
        staggeredTileBuilder: (index) => const StaggeredTile.fit(2),
        crossAxisCount: 4,
        mainAxisSpacing: 4,
        crossAxisSpacing: 4,
        itemBuilder: (context, index) {
        final blog = blogs[index];

        return GestureDetector(
            onTap: () async {
```

```
116             await Navigator.of(context).push(MaterialPageRoute(
117                 builder: (context) => DetailPage(blogId: blog.id!),
118             ));
119 
120             refreshingAllBogs();
121             },
122             child: BlogCard(blog: blog, index: index),
123         );
124         },
125     );
126 }
```

As a result, when there is no blog items, the home page looks like the following one.

Figure 17.6 – Adding a Floating action button to SQLite Blog Home page in Flutter

In the next part, or section we'll take a look at the edit page, where actual action takes place.

After that, we'll also see how we can use the Card widget to display all inserted data.

As we're changing the previous code, we keep them in separate GitHub branch. For full code for this section, please visit the respective GitHub repository.

- All related Code Snippet in this GitHub repository[85]

SQLite Blog, Flutter: Final Part

This is the final part of our Blog application in Flutter where we use SQLite database and CRUD.

In this final part of building SQLite Blog application in Flutter we will accomplish the basic principle of CRUD. As a consequence, we'll create, retrieve, update and delete data in SQLite database.

We've already built a significant part of a SQLite Blog Application in Flutter. We might also see the progress of the initial phase in this section – SQLite Blog application in Flutter.

The previous section has discussed the application structure.

In the second part, we'd concentrated on database connection, data model classes.

In this final part we'll take a look at the logic flow and see how we can convert this Blog application to a My Diary application.

We've also changed the layout in a significant way.

Therefore, before we jump in, let's recapitulate a few things about SQLite database and Flutter.

Firstly, we'll use a Flutter package or plugin, sqflite which is available in pub.dev.

Secondly, we also need to use Future API, async, await keywords, and then functions to make it successful.

Finally, we've discussed Future, await and async for absolute beginners in previous section, is Flutter single thread?

Therefore, if you're a beginner, you might take a look before we proceed towards the final section.

What packages we need for SQLite database in Flutter?

We'll start with the pubspec.yaml file, where we must add all the dependencies.

[85] https://github.com/sanjibsinha/flutter_data_and_backend/tree/one-blog

```
dependencies:
cupertino_icons: ^1.0.2
flutter:
    sdk: flutter
flutter_staggered_grid_view: ^0.4.1
intl: ^0.17.0
path:
provider: ^6.0.1
sqflite:
```

To give this SQLite Blog Application in Flutter a final touch, we need some packages or plugins.

The “sqflite”, “path”, “intl”, and “flutter_staggered_grid_view” packages will help us in many ways.

The “sqflite”, and “path” packages work at tandem. They help each other as we’ll find later. With the help of “path” package we define the path of the local database.

We need the packages “intl”, and “flutter_staggered_grid_view” for different purposes.

The package “intl” helps us to format the date in our Blog. And, the “flutter_staggered_grid_view” package helps us in building the layout while we display the blog or diary’s contents.

Does flutter need a backend?

Yes, Flutter needs a backend. Moreover, we can choose between two. Either we can go with a server, or we store our data locally with SQLite database.

In our case, we have decided to use SQLite database.

Consequently, we need a service or utility class that will create a database in local path first. Next, it will help us to use the SQL language to create a table with required fields.

```
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';
import '../model/blog.dart';

class BlogDatabaseHandler {
static final BlogDatabaseHandler instance = BlogDatabaseHandler._init();

static Database? _database;

BlogDatabaseHandler._init();

Future<Database> get database async {
    if (_database != null) return _database!;

```

```
    _database = await _initDB('newblogs.db');
    return _database!;
}

Future<Database> _initDB(String filePath) async {
    final dbPath = await getDatabasesPath();
    final path = join(dbPath, filePath);

    return await openDatabase(path, version: 1, onCreate: _createDB);
}

Future _createDB(Database db, int version) async {
    const idType = 'INTEGER PRIMARY KEY AUTOINCREMENT';
    const textType = 'TEXT NOT NULL';

    await db.execute('''
CREATE TABLE $tableOfBlogs (
${BlogFields.id} $idType,
${BlogFields.title} $textType,
${BlogFields.description} $textType,
${BlogFields.time} $textType
)
''');
}

Future<Blog> create(Blog blog) async {
    final db = await instance.database;

    final id = await db.insert(tableOfBlogs, blog.toJson());
    return blog.copy(id: id);
}

Future<Blog> readBlog(int id) async {
    final db = await instance.database;

    final maps = await db.query(
    tableOfBlogs,
    columns: BlogFields.values,
    where: '${BlogFields.id} = ?',
    whereArgs: [id],
    );

    if (maps.isNotEmpty) {
```

```
58      return Blog.fromJson(maps.first);
59      } else {
60      throw Exception('ID $id not found');
61      }
62  }
63
64  Future<List<Blog>> readAllBlogs() async {
65      final db = await instance.database;
66
67      const orderBy = '${BlogFields.time} ASC';
68
69      final result = await db.query(tableOfBlogs, orderBy: orderBy);
70
71      return result.map((json) => Blog.fromJson(json)).toList();
72  }
73
74  Future<int> update(Blog blog) async {
75      final db = await instance.database;
76
77      return db.update(
78      tableOfBlogs,
79      blog.toJson(),
80      where: '${BlogFields.id} = ?',
81      whereArgs: [blog.id],
82      );
83  }
84
85  Future<int> delete(int id) async {
86      final db = await instance.database;
87
88      return await db.delete(
89      tableOfBlogs,
90      where: '${BlogFields.id} = ?',
91      whereArgs: [id],
92      );
93  }
94
95  Future close() async {
96      final db = await instance.database;
97
98      db.close();
99  }
100 }
```

Firstly, we've mentioned a path so the database gets created locally and stores data locally.

Secondly, we've created the database and table with fields. As we see, we've kept it simple.

Thirdly, we've created methods that will read, update, delete items.

Finally, we've closed the database.

At the same time, we've changed the code of our home page slightly. As a result, we can now add or update the blog items either from AppBar, or through the floating action button.

Subsequently, to help the utility class we need a data model class.

```
const String tableOfBlogs = 'Blogs';

class BlogFields {
static final List<String> values = [
    /// Adding all fields
    id, title, description, time
];

static const String id = '_id';
static const String title = 'title';
static const String description = 'description';
static const String time = 'time';
}

class Blog {
final int? id;
final String title;
final String description;
final DateTime createdTime;

const Blog({
    this.id,
    required this.title,
    required this.description,
    required this.createdTime,
});

Blog copy({
    int? id,
    String? title,
    String? description,
    DateTime? createdTime,
}) =>
```

```
    Blog(
        id: id ?? this.id,
        title: title ?? this.title,
        description: description ?? this.description,
        createdTime: createdTime ?? this.createdTime,
    );

static Blog fromJson(Map<String, Object?> json) => Blog(
        id: json[BlogFields.id] as int?,
        title: json[BlogFields.title] as String,
        description: json[BlogFields.description] as String,
        createdTime: DateTime.parse(json[BlogFields.time] as String),
    );

Map<String, Object?> toJson() => {
        BlogFields.id: id,
        BlogFields.title: title,
        BlogFields.description: description,
        BlogFields.time: createdTime.toIso8601String(),
    };
}
```

We've kept these files in our "model" sub-folder.

After that, we have built three pages and to keep them we have created a "view" sub-folder.

Which database is used for flutter?

As we've been discussing the topic we find, the Flutter team also recommends to use SQLite database. They say, "Flutter apps can make use of the SQLite databases via the sqflite plugin available on pub."

Why?

The reason is simple. And it's explained below.

If our app needs to persist and query large amounts of data on the local device, it's always better to use a database instead of a local file or key-value store.

In general, databases provide faster inserts, updates, and queries compared to other local persistence solutions, and SQLite is the best choice.

Let's proceed with our code.

First, we have a home page, that will handle the layout and backend at the same time.

If there is no entry it shows us a blank page and we start adding items.

If not, it shows like the following screenshot.

Figure 17.7 – Home page of My Diary SQLite database app in flutter

We've already added three entries. As a result it shows like the above screenshot.

Next, we'll see the code snippet of the home page.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_staggered_grid_view/flutter_staggered_grid_view.dart';
import '/model/blogs.dart';
import '/model/blog.dart';
import 'edit.dart';
import 'detail.dart';
import '/controller/blog_card.dart';

class AllPages extends StatefulWidget {
const AllPages({Key? key}) : super(key: key);

@override
_AllPagesState createState() => _AllPagesState();
}

class _AllPagesState extends State<AllPages> {
late List<Blog> blogs;
bool isLoading = false;

@override
void initState() {
    super.initState();

    refreshingAllBogs();
}

@override
void dispose() {
    BlogDatabaseHandler.instance.close();

    super.dispose();
}

Future refreshingAllBogs() async {
    setState(() => isLoading = true);

    blogs = await BlogDatabaseHandler.instance.readAllBlogs();

    setState(() => isLoading = false);
}

@override
Widget build(BuildContext context) => Scaffold(
```

```
        appBar: AppBar(
        title: const Text(
            'My Diary',
            style: TextStyle(fontSize: 24),
        ),
        ),
        body: Center(
        child: isLoading
            ? const CircularProgressIndicator()
            : blogs.isEmpty
                ? const Text(
                    'No Entry in the beginning...',
                    style: TextStyle(color: Colors.white, fontSize: 60),
                    )
                : buildingAllBlogs(),
        ),
        floatingActionButton: FloatingActionButton.extended(
        tooltip: 'Write Diary...',
        foregroundColor: Colors.white,
        backgroundColor: Colors.pink.shade900,
        onPressed: () async {
            await Navigator.of(context).push(
            MaterialPageRoute(builder: (context) => const EditPage()),
            );

            refreshingAllBogs();
        },
        label: const Text(
            'Write Diary...',
            style: TextStyle(
            fontSize: 30,
            ),
        ),
        ),
    );

Widget buildingAllBlogs() => StaggeredGridView.countBuilder(
        padding: const EdgeInsets.all(8),
        itemCount: blogs.length,
        staggeredTileBuilder: (index) => const StaggeredTile.fit(2),
        crossAxisCount: 4,
        mainAxisSpacing: 4,
        crossAxisSpacing: 4,
```

```
        itemBuilder: (context, index) {
        final blog = blogs[index];

        return GestureDetector(
            onTap: () async {
            await Navigator.of(context).push(MaterialPageRoute(
                builder: (context) => DetailPage(blogId: blog.id!),
            ));

            refreshingAllBogs();
            },
            child: BlogCard(blog: blog, index: index),
        );
        },
    );
}
```

We have a list of all blog entries. If it's empty, it will show an empty page with no contents. If not, it will take us to a controller.

```
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import '../model/blog.dart';

/// these shades of colors will appear on
/// the display screen and it will appear
/// based on the index of the list
final shadeOfColors = [
Colors.pink.shade100,
Colors.purple.shade100,
Colors.teal.shade200,
Colors.orange.shade200,
Colors.white10,
];

class BlogCard extends StatelessWidget {
const BlogCard({
    Key? key,
    required this.blog,
    required this.index,
}) : super(key: key);

final Blog blog;
```

```
final int index;

@override
Widget build(BuildContext context) {
    final color = shadeOfColors[index % shadeOfColors.length];
    final time = DateFormat.yMMMd().format(blog.createdTime);
    final minHeight = getMinHeight(index);

    return Card(
    color: color,
    child: Container(
        constraints: BoxConstraints(minHeight: minHeight),
        padding: const EdgeInsets.all(8),
        child: Column(
        mainAxisSize: MainAxisSize.min,
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
            Text(
            time,
            style: TextStyle(color: Colors.grey.shade700),
            ),
            const SizedBox(height: 4),
            Text(
            blog.title,
            style: const TextStyle(
                color: Colors.black,
                fontSize: 20,
                fontWeight: FontWeight.bold,
            ),
            ),
            const SizedBox(
            height: 5,
            ),
            Text(
            blog.description,
            style: const TextStyle(
                color: Colors.blueAccent,
                fontSize: 16,
                fontWeight: FontWeight.w300,
            ),
            ),
        ],
        ),
```

```
    ),
    );
}

double getMinHeight(int index) {
    switch (index % 4) {
    case 0:
        return 100;
    case 1:
        return 150;
    case 2:
        return 150;
    case 3:
        return 100;
    default:
        return 100;
    }
}
}
```

The each Card widget has been defined here to show all the items.

Moreover, we can also start writing the content also. In that case, the home page will take us to the Edit page.

```
import 'package:flutter/material.dart';
import '../model/blogs.dart';
import '../model/blog.dart';
import '../controller/blog_form.dart';

class EditPage extends StatefulWidget {
final Blog? blog;

const EditPage({
    Key? key,
    this.blog,
}) : super(key: key);
@override
_EditPageState createState() => _EditPageState();
}

class _EditPageState extends State<EditPage> {
final _formKey = GlobalKey<FormState>();
```

```
late bool isImportant;
late int number;
late String title;
late String description;

@override
void initState() {
    super.initState();

    title = widget.blog?.title ?? '';
    description = widget.blog?.description ?? '';
}

@override
Widget build(BuildContext context) => Scaffold(
        appBar: AppBar(
        actions: [buildButton()],
        ),
        body: Form(
        key: _formKey,
        child: BlogForm(
            title: title,
            description: description,
            onChangedTitle: (title) => setState(() => this.title = title),
            onChangedDescription: (description) =>
                setState(() => this.description = description),
        ),
        ),
    );

Widget buildButton() {
    final isFormValid = title.isNotEmpty && description.isNotEmpty;

    return Padding(
    padding: const EdgeInsets.symmetric(vertical: 8, horizontal: 12),
    child: ElevatedButton(
        style: ElevatedButton.styleFrom(
        onPrimary: Colors.white,
        onSurface: Colors.pink.shade900,
        shadowColor: Colors.grey.shade600,
        primary: isFormValid ? Colors.pink.shade900 : Colors.pink.shade900,
        ),
        onPressed: addOrUpdateBlog,
```

```
62          child: const Text(
63          'Add or Update',
64          style: TextStyle(
65              fontSize: 20,
66          ),
67          ),
68      ),
69      );
70  }
71
72  void addOrUpdateBlog() async {
73      final isValid = _formKey.currentState!.validate();
74
75      if (isValid) {
76      final isUpdating = widget.blog != null;
77
78      if (isUpdating) {
79          await updateBlog();
80      } else {
81          await addBlog();
82      }
83
84      Navigator.of(context).pop();
85      }
86  }
87
88  Future updateBlog() async {
89      final blog = widget.blog!.copy(
90      title: title,
91      description: description,
92      );
93
94      await BlogDatabaseHandler.instance.update(blog);
95  }
96
97  Future addBlog() async {
98      final blog = Blog(
99      title: title,
100     description: description,
101     createdTime: DateTime.now(),
102     );
103
104     await BlogDatabaseHandler.instance.create(blog);
```

```
105 }
106 }
```

Here we can start writing the fresh content just like the following screenshot.

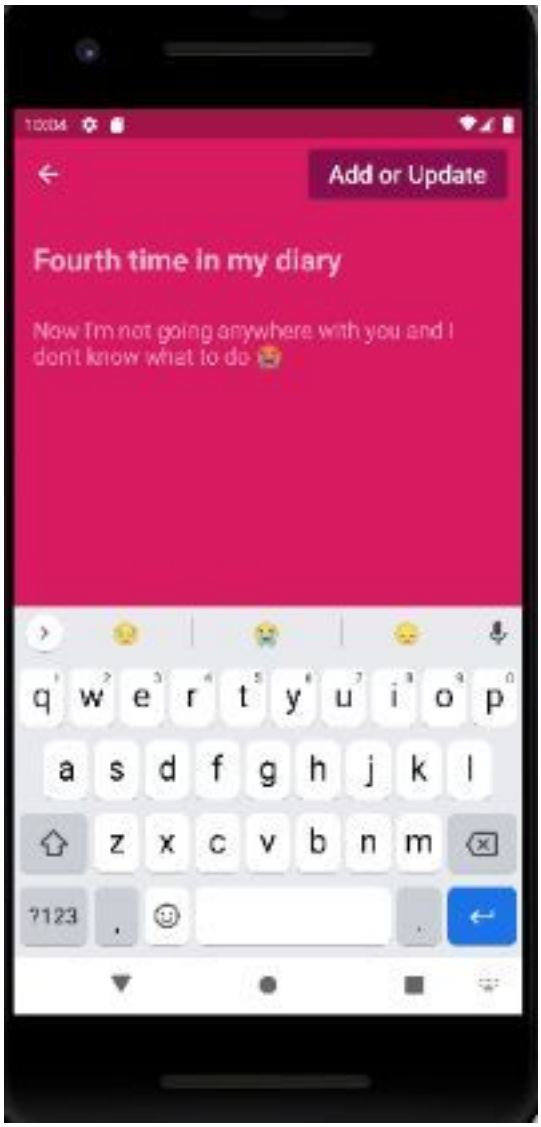

Figure 17.8 – Writing content for my diary SQLite database in Flutter

After the writing is over, we can press the "Add or Update" button and insert the data to SQLite database.

How Do we insert data to SQLite Database?

Well, we need a text controller or a Form that will take care of it? Right?

The user must be able to write her posts in this application. Whether, she is writing a blog post or adding her diary entry, that does not matter.

To make it happen, we need another blog form controller.

```
import 'package:flutter/material.dart';

class BlogForm extends StatelessWidget {
final String? title;
final String? description;

final ValueChanged<String> onChangedTitle;
final ValueChanged<String> onChangedDescription;

const BlogForm({
    Key? key,
    this.title = '',
    this.description = '',
    required this.onChangedTitle,
    required this.onChangedDescription,
}) : super(key: key);

@override
Widget build(BuildContext context) => SingleChildScrollView(
        child: Padding(
        padding: const EdgeInsets.all(16),
        child: Column(
            mainAxisSize: MainAxisSize.min,
            children: [
            buildTitle(),
            const SizedBox(height: 8),
            buildDescription(),
            const SizedBox(height: 16),
            ],
        ),
        ),
    );

Widget buildTitle() => TextFormField(
        maxLines: 1,
        initialValue: title,
        style: const TextStyle(
        color: Colors.white70,
```

```
        fontWeight: FontWeight.bold,
        fontSize: 24,
        ),
        decoration: const InputDecoration(
        border: InputBorder.none,
        hintText: 'Here Title...',
        hintStyle: TextStyle(color: Colors.white70),
        ),
        validator: (title) =>
            title != null && title.isEmpty ? 'Title cannot be empty' : null,
        onChanged: onChangedTitle,
    );

Widget buildDescription() => TextFormField(
        maxLines: 5,
        initialValue: description,
        style: const TextStyle(color: Colors.white60, fontSize: 18),
        decoration: const InputDecoration(
        border: InputBorder.none,
        hintText: 'Here description...',
        hintStyle: TextStyle(color: Colors.white60),
        ),
        validator: (title) => title != null && title.isEmpty
            ? 'Description cannot be empty'
            : null,
        onChanged: onChangedDescription,
    );
}
```

We need two text form field to accept the data from user.

Once we are over, the added items show on the screen.

Figure 17.9 – Added item show on Home page

If we want to edit any of the items, we can just tap the item and gesture detector will take us to detail page where we find the edit and delete buttons.

However, in this time, we can edit them. In addition, we can also delete them as well.

Figure 17.10 – Edit or delete any item in SQLite database

This page has been taken care of by another page, that basically assists our edit page to maintain the state and allow us to edit or delete.

```
1  import 'package:flutter/material.dart';
2  import 'package:intl/intl.dart';
3  import '../model/blogs.dart';
4  import '../model/blog.dart';
5  import 'edit.dart';
6
7  class DetailPage extends StatefulWidget {
8  final int blogId;
9
```

```
const DetailPage({
    Key? key,
    required this.blogId,
}) : super(key: key);

@override
_DetailPageState createState() => _DetailPageState();
}

class _DetailPageState extends State<DetailPage> {
late Blog blog;
bool isLoading = false;

@override
void initState() {
    super.initState();

    refreshBlog();
}

Future refreshBlog() async {
    setState(() => isLoading = true);

    blog = await BlogDatabaseHandler.instance.readBlog(widget.blogId);

    setState(() => isLoading = false);
}

@override
Widget build(BuildContext context) => Scaffold(
        appBar: AppBar(
        actions: [
            const Text(' ... '),
            editButton(),
            const Text(' ... '),
            deleteButton(),
        ],
        ),
        body: isLoading
            ? const Center(child: CircularProgressIndicator())
            : Padding(
                padding: const EdgeInsets.all(12),
                child: ListView(
```

```
                padding: const EdgeInsets.symmetric(vertical: 8),
                children: [
                    const SizedBox(height: 10),
                    Text(
                    blog.title,
                    style: const TextStyle(
                        color: Colors.white,
                        fontSize: 22,
                        fontWeight: FontWeight.bold,
                    ),
                    ),
                    const SizedBox(height: 10),
                    Text(
                    DateFormat.yMMMd().format(blog.createdTime),
                    style: const TextStyle(color: Colors.white38),
                    ),
                    const SizedBox(height: 10),
                    Text(
                    blog.description,
                    style:
                        const TextStyle(color: Colors.white70, fontSize: 18),
                    )
                ],
                ),
            ),
    );

Widget editButton() => ElevatedButton(
        style: ElevatedButton.styleFrom(
        onPrimary: Colors.white,
        onSurface: Colors.pink.shade900,
        shadowColor: Colors.grey.shade600,
        primary: Colors.pink.shade900,
        ),
        onPressed: () async {
        if (isLoading) return;

        await Navigator.of(context).push(
            MaterialPageRoute(
            builder: (context) => EditPage(blog: blog),
            ),
        );

```

```
96          refreshBlog();
97          },
98          child: const Text(
99          'Edit',
100         style: TextStyle(
101             fontSize: 20,
102         ),
103         ),
104     );
105
106 Widget deleteButton() => ElevatedButton(
107         style: ElevatedButton.styleFrom(
108         onPrimary: Colors.white,
109         onSurface: Colors.pink.shade900,
110         shadowColor: Colors.grey.shade600,
111         primary: Colors.pink.shade900,
112         ),
113         onPressed: () async {
114         await BlogDatabaseHandler.instance.delete(widget.blogId);
115
116         Navigator.of(context).pop();
117         },
118         child: const Text(
119         'Delete',
120         style: TextStyle(
121             fontSize: 20,
122         ),
123         ),
124     );
125 }
```

As we can see in the above code, we have edit and delete buttons both in our AppBar.

Now. we've accomplished the task of building a Blog SQLite database application in Flutter. Of course, we can use the same application as a My Diary.

For the full code snippet for this SQLite database application in Flutter, please visit the respective GitHub repository.

- All related code with this section[86]

[86] https://github.com/sanjibsinha/flutter_data_and_backend

18. Scoped Model, Provider, SQLite Database and FutureBuilder

In this section we will delve deep into SQLite database and Flutter. However, we change the State using Scoped Model and Provider.

SQLite with Provider in Flutter

Without a stateful widget can we use SQLite database in Flutter? Yes, with Provider.

Can we use SQLite database with Provider package in Flutter? The answer is, yes! We can.

Not only that, we can also reduce pressure on system resource while we store persistent data with provider package.

Most importantly, we always want to make our Flutter app faster and performant. Since storing persistent data requires a lot of state management, the Provider package always helps us to achieve that target.

As a result, in this section, we try to use SQLite database with Provider.

No stateful widget. No extra widget rebuilding. We've kept things quite simple. However, if you are a beginner, please learn Future, await and async first. As we've discussed Future, await and async for absolute beginners in previous section- is Flutter single thread? Therefore, if you're a beginner, you might take a look before we proceed towards the final section.

Before using provider package, we've built an entire Blog, or My Diary application using SQLite Database in Flutter.

Not only that, before doing that, in a step by step process we've built the SQLite Blog Application in Flutter. We might also see the progress of the initial phase in this section – SQLite Blog application in Flutter.

The previous section has discussed the application structure.

In the second part, we'd concentrated on database connection, data model classes.

Firstly, we'll use a Flutter package or plugin, sqflite which is available in pub.dev.

Secondly, we also need to use Future API, async, await keywords, and then functions to make it successful.

And finally, we are going to store persistent data in our local SQLite database, using provider package.

What is Sqflite flutter?

The sqflite is a very useful SQLite plugin for Flutter. It supports iOS, Android and MacOS.

For any type of complex CRUD operations, we get support from this plugin, or package. Moreover, this plugin supports transactions and batches.

Therefore, we have helpers for insert, query, update and delete queries. Above all, the DB operation executed in a background thread on iOS and Android. As a result, we get a much faster Flutter application than any other backend operation.

We need to add the dependencies first to our pubspec.yaml file.

```
dependencies:
cupertino_icons: ^1.0.2
flutter:
    sdk: flutter
path: ^1.8.0
provider: ^6.0.2
sqflite: ^2.0.1
```

The three packages in bold, are necessary to build our first Name-Keeper Flutter Application using SQLite database and Provider.

How do I get data from SQLite database in flutter?

Our next challenge is to create a helper class. It will not only create the SQLite database in a given path, but also create the table. Moreover, it will insert and retrieve data.

Besides the helper class, we need a User data model, and a User Provider class that will notify the listeners.

We'll take a look at the classes separately and try to understand how they work together. We have kept three classes in our model folder.

Firstly, the helper class.

```
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';

import 'user.dart';

class DatabaseHandler {
Future<Database> initializeDB() async {
    String path = await getDatabasesPath();
    return openDatabase(
    join(path, 'usereleven.db'),
    onCreate: (database, version) async {
        await database.execute(
        "CREATE TABLE usereleven(id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT NOT\
 NULL, location TEXT NOT NULL)",
        );
    },
    version: 1,
    );
}

Future<int> insertUser(List<User> users) async {
    int result = 0;
    final Database db = await initializeDB();
    for (var user in users) {
    result = await db.insert('usereleven', user.toMap());
    }
    return result;
}

Future<List<User>> retrieveUsers() async {
    final Database db = await initializeDB();
    final List<Map<String, Object?>> queryResult = await db.query('usereleven');
    return queryResult.map((e) => User.fromMap(e)).toList();
}
}
```

The above code is quite verbose and meaningful. The database handler class will first define a path where SQLite database gets created.

After that, it will create a table with ID auto increment, and two columns where we store the name and location.

Finally, it defines two methods to insert and retrieve data from the local database.

However, the Future object wants a list of Users that it can map to list so that we can finally get

them on screen after the insertion is over.

Therefore, let's take a look at the User class, next.

```
class User {
final int? id;
final String name;
final String location;

User({
    this.id,
    required this.name,
    required this.location,
});

User.fromMap(Map<String, dynamic> res)
    : id = res["id"],
        name = res["name"],
        location = res["location"];

Map<String, Object?> toMap() {
    return {
    'id': id,
    'name': name,
    'location': location,
    };
}
}
```

Now we need a user provider class that will notify the listeners when we press the button "Add Users" like the following screenshot.

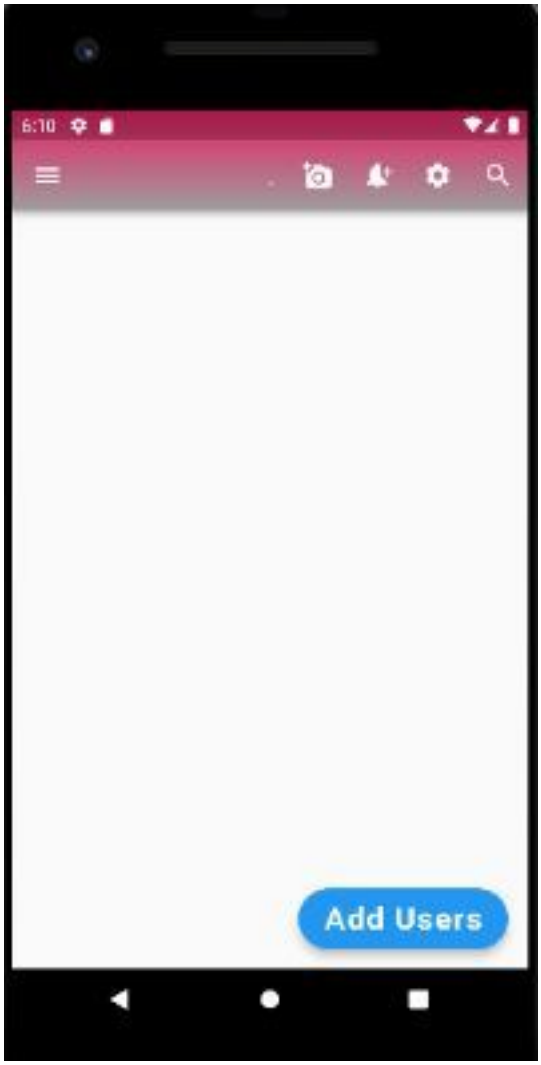

Figure 18.1 – SQLite database and Provider in Flutter first screen

At the same time, we'll also keep the Provider at the top of the root widget.

```
import 'package:flutter/material.dart';
import '/model/user_provider.dart';
import 'view/my_app.dart';

void main() {
Provider.debugCheckInvalidValueType = null;
runApp(
    MultiProvider(
    providers: [
        ChangeNotifierProvider(create: (_) => UserProvider()),
    ],
    child: const MyApp(),
```

```
    ),
);
}
```

The User Provider class plays the most important role in this SQLite database and Flutter application.

```
import 'package:flutter/material.dart';

import 'database_handler.dart';
import 'user.dart';

final handler = DatabaseHandler();

class UserProvider with ChangeNotifier {
User _userOne = User(name: 'Hagudu', location: 'Japan');
User get userOne => _userOne;
/*
User _userTwo = User(name: 'Mutudu', location: 'Hokkaidu');
User get userTwo => _userTwo;

User _userThree = User(name: 'John Smith', location: 'East Coast');
User get userThree => _userThree;
*/

void addingUsers() {
    _userOne = _userOne;
    //_userTwo = userTwo;
    //_userThree = userThree;

    notifyListeners();
}
}
```

We've commented out other users as we want to insert one user at a time. We could have added them at once of course.

Is SQLite persistent?

We're going to see how the SQLite database persistently stores data, so we can keep adding one user name and location one after another.

The Change Notifier Provider works with Future builder here, in the my home page widget.

```dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '/model/user_provider.dart';
import '/model/database_handler.dart';
import '/model/user.dart';

class MyHomePage extends StatelessWidget {
const MyHomePage({Key? key}) : super(key: key);

static const String title = 'Database Handling';

@override
Widget build(BuildContext context) {
    final userProvider = Provider.of<UserProvider>(context);

    final handler = DatabaseHandler();
    Future<int> addUsers() async {
    User firstUser = User(
        name: userProvider.userOne.name,
        location: userProvider.userOne.location,
    );

    /*
User secondUser = User(
        name: userProvider.userTwo.name,
        location: userProvider.userTwo.location,
    );

    User thirddUser = User(
        name: userProvider.userThree.name,
        location: userProvider.userThree.location,
    );
*/
    List<User> listOfUsers = [
        firstUser,
        //secondUser,
        //thirddUser,
    ];
    return await handler.insertUser(listOfUsers);
    }

    return Scaffold(
    appBar: customAppBar(title),
```

```
44      body: FutureBuilder(
45          future: handler.retrieveUsers(),
46          builder: (BuildContext context, AsyncSnapshot<List<User>> snapshot) {
47          if (snapshot.hasData) {
48              return ListView.builder(
49              itemCount: snapshot.data?.length,
50              itemBuilder: (BuildContext context, int index) {
51                  return Card(
52                  child: ListTile(
53                      key: ValueKey<int>(snapshot.data![index].id!),
54                      contentPadding: const EdgeInsets.all(8.0),
55                      title: Text(
56                      snapshot.data![index].name,
57                      style: const TextStyle(
58                          fontSize: 30,
59                          color: Colors.red,
60                      ),
61                      ),
62                      subtitle: Text(
63                      snapshot.data![index].location,
64                      style: const TextStyle(
65                          fontSize: 20,
66                          color: Colors.red,
67                      ),
68                      ),
69                  ),
70                  );
71              },
72              );
73          } else {
74              return const Center(child: CircularProgressIndicator());
75          }
76          },
77      ),
78      floatingActionButton: FloatingActionButton.extended(
79          onPressed: () {
80          handler.initializeDB().whenComplete(() async {
81              await addUsers();
82          });
83
84          userProvider.addingUsers();
85          },
86          label: const Text(
```

```
87          'Add Users',
88          style: TextStyle(
89              fontSize: 25,
90              fontWeight: FontWeight.bold,
91          ),
92          ),
93      ),
94      );
95  }
96
97  AppBar customAppBar(String title) {
98      return AppBar(
99      centerTitle: true,
100     //backgroundColor: Colors.grey[400],
101     flexibleSpace: Container(
102         decoration: const BoxDecoration(
103         gradient: LinearGradient(
104             colors: [
105             Colors.pink,
106             Colors.grey,
107             ],
108             begin: Alignment.topRight,
109             end: Alignment.bottomRight,
110         ),
111         ),
112     ),
113     //elevation: 20,
114     titleSpacing: 80,
115     leading: const Icon(Icons.menu),
116     title: Text(
117         title,
118         textAlign: TextAlign.left,
119     ),
120     actions: [
121         buildIcons(
122         const Icon(Icons.add_a_photo),
123         ),
124         buildIcons(
125         const Icon(
126             Icons.notification_add,
127         ),
128         ),
129         buildIcons(
```

```
130         const Icon(
131             Icons.settings,
132         ),
133         ),
134         buildIcons(
135         const Icon(Icons.search),
136         ),
137     ],
138     );
139 }
140 
141 IconButton buildIcons(Icon icon) {
142     return IconButton(
143     onPressed: () {},
144     icon: icon,
145     );
146 }
147 }
```

The flow of logic is quite simple. Inside our build method, we've got the Provider of Type User Provider and its context.

Next, we have instantiated the Database handler object. Without this handler we cannot initiate the process of inserting and retrieving data.

As a result, we can press the "Add Users" button that fires the event of inserting and retrieving data from the SQLite database.

Figure 18.2 – SQLite database and Provider in Flutter with first user

Once we have inserted the first user name and location, we can comment out the first user in User and User Handler class.

Then we can insert the second user's name and location.

Figure 18.3 – SQLite database and Provider in Flutter with second user

We can clearly see that how SQLite database persists data. However, we don't have to use stateful widget to manage state. The provider package helps us to notify listeners which is a Future builder.

Now, we can add as many user's name and location.

Figure 18.4 – SQLite database and Provider in Flutter with third user

- For the full code snippet please visit the respective GitHub repository.[87]

What is Scoped Model in flutter

Scoped Model is the simplest version of Inherited Widget, managing State in Flutter.

As the name suggests, the Scoped Model in Flutter examines the scope and passes data downwards. We can create the scope at the top first with a type. And, after that in the descendant widgets, we can pass that data type.

If you have already learned Provider, then you might sense the similarity. However, provider is more versatile and might be complex. Moreover, provider package uses ChangeNotifier of Flutter.

[87] https://github.com/sanjibsinha/flutter_artisan/tree/final-provider-sqflite

We'll come to that point later.

Before that, let's see what is Scoped Model, and how it works. In addition, whether we can use scoped model with other packages or not.

The best way to understand any topic is to view images first. Therefore, let's view the Flutter application we've built using Scoped Model.

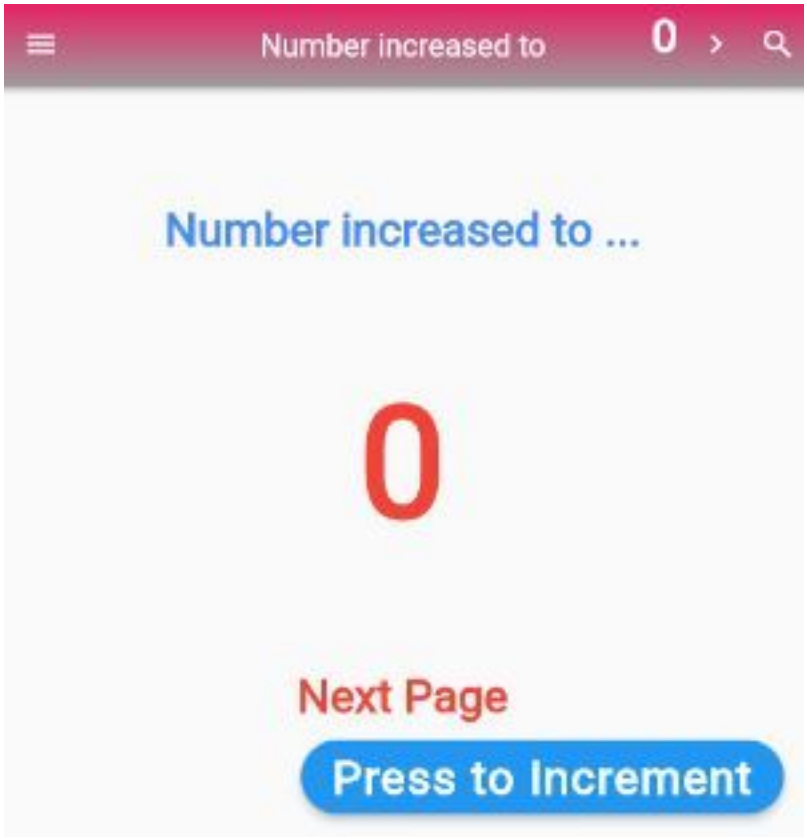

Figure 18.5 – Scoped model example one

While we press the button, the state of the Text widget which display the number changes. As a result, the number increases.

The increment reflects also on the AppBar, and on the screen facing us.

Let's see the next stage.

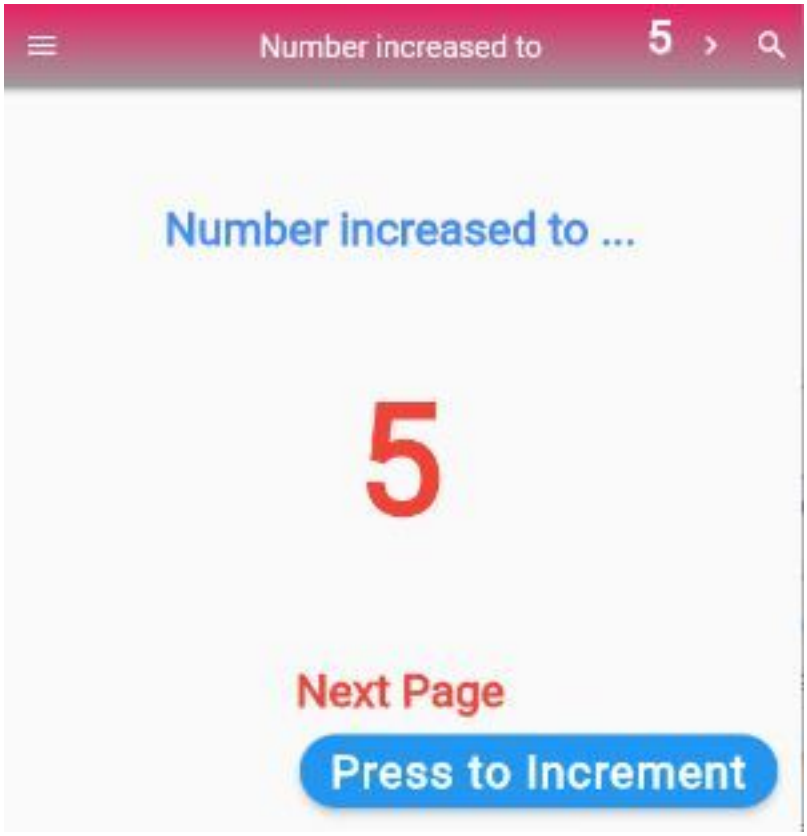

Figure 18.6 – Scoped model example two

We've pressed the button five times, and the number changes in two places.

Now, we're going to press the Next Page button to navigate to another page.

Why?

Because the Next Page is also another descendant of Scoped Model. Therefore, in that case, the number should increase there too.

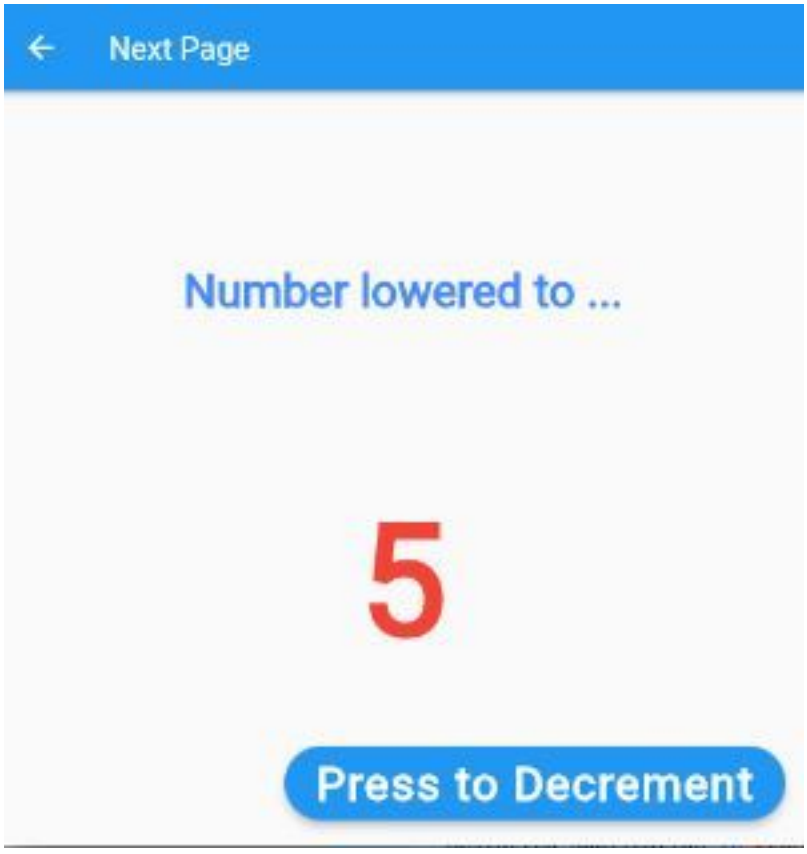

Figure 18.7 – Scoped model example three

Voila! It works.

The number in the Next Page changes simultaneously along with the home page.

Now, in this page, we will press the decrement button. As a result, the number will lower down. But, will that change the number in home page?

Let's see. First, let's press the decrement button.

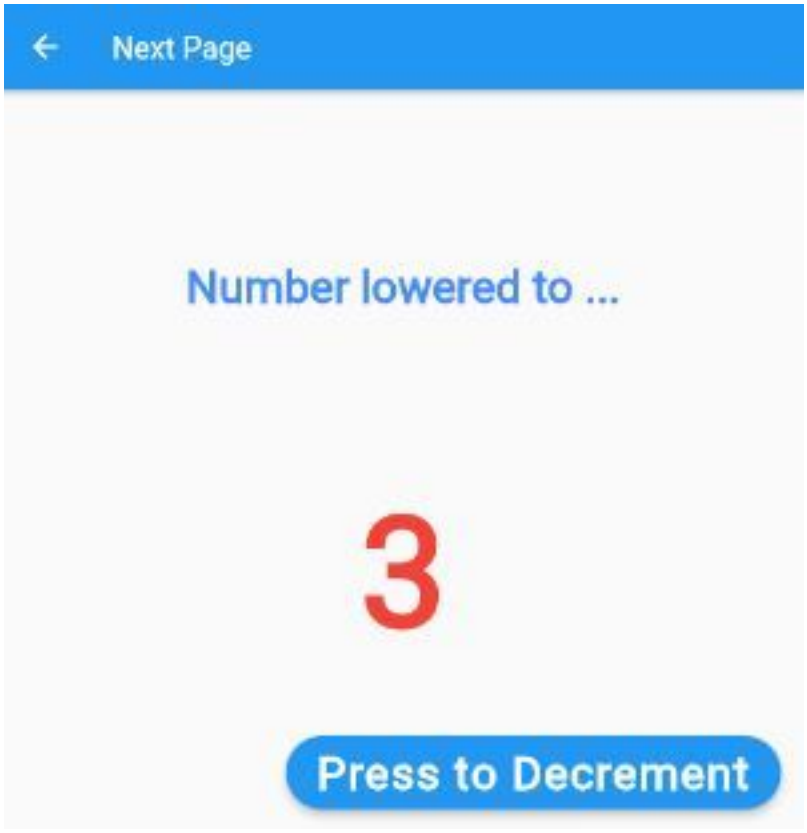

Figure 18.8 – Scoped model example four

We've pressed the decrement button twice and the number lowers down to 3.

Fine.

Next, we move back to the home page.

What do we see?

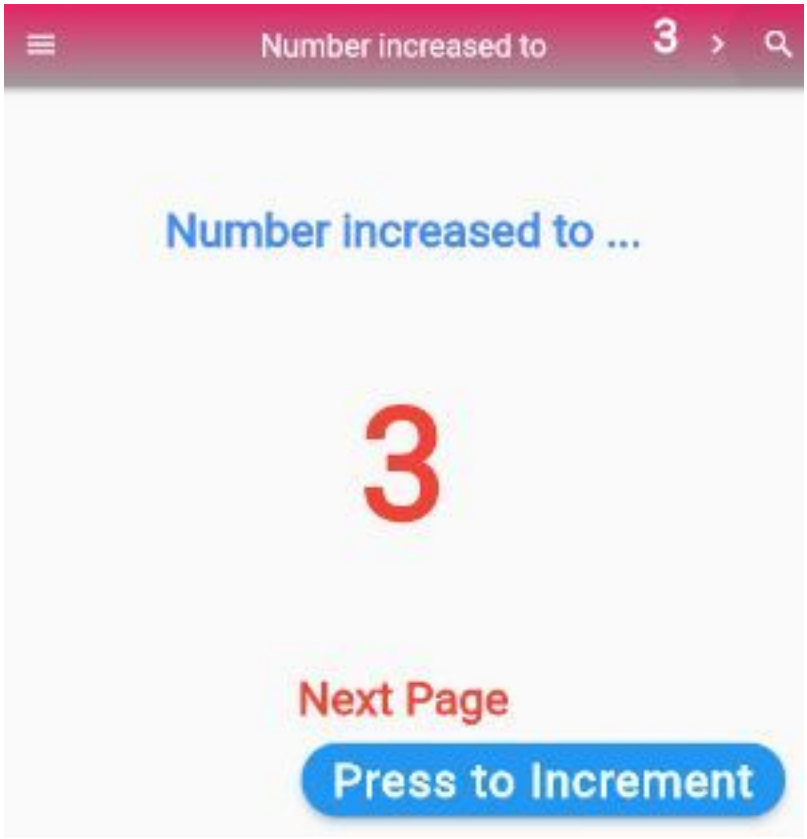

Figure 18.9 – Scoped model example five

The number reduces to 3.

Consequently, we've successfully managed state with the help of Scoped Model.

To sum up, Scoped Model helps us to manage state in a very simple way.

Now, the time has come to inspect the code.

How do you use the scoped model in Flutter?

Firstly, we need to add the dependency.

```
dependencies:
cupertino_icons: ^1.0.4
flutter:
    sdk: flutter

scoped_model: ^2.0.0-nullsafety.0
```

We've made it sure that the scoped_model package we're using must adhere to the null safety.

Next, we should create a model class of Counter.

```
import 'package:scoped_model/scoped_model.dart';

class Counter extends Model {
int _counter = 0;
int get counter => _counter;
void increment() {
    _counter++;
    notifyListeners();
}

void decrement() {
    _counter--;
    notifyListeners();
}
}
```

Our Counter class extends the Model class provided by the package. As a result, the Counter object can notify all the components that subscribe to all the public properties and methods of Counter type.

Therefore, we need to declare the scope at the top and mention the type which is Counter here.

```
import 'package:flutter/material.dart';
import 'package:provider_et_sqflite/model/counter.dart';
import 'package:scoped_model/scoped_model.dart';

import 'my_home_page.dart';

class MyApp extends StatelessWidget {
const MyApp({
    Key? key,
    required this.counter,
}) : super(key: key);
final Counter counter;

// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return ScopedModel<Counter>(
    model: Counter(),
    child: MaterialApp(
        debugShowCheckedModeBanner: false,
        title: 'Scoped Model Simple',
```

```
        theme: ThemeData(
        primarySwatch: Colors.blue,
        ),

        /// child widgets are now under its scope
        /// and we can use this model anywhere below
        ///
        home: const MyHomePage(),
    ),
    );
}
}
```

As a result, we can pass the Counter object to all the descendants.

First see the Home page code.

```
import 'package:flutter/material.dart';
import 'package:provider_et_sqflite/model/counter.dart';
import 'package:scoped_model/scoped_model.dart';

import 'next_page.dart';

class MyHomePage extends StatelessWidget {
const MyHomePage({Key? key}) : super(key: key);

static const String title = 'Number increased to';

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: customAppBar(title),

    /// the child widget below can use the scoped model
    ///
    floatingActionButton: ScopedModelDescendant<Counter>(
        builder: (context, child, model) => FloatingActionButton.extended(
        onPressed: () {
            model.increment();
        },
        label: const Text(
            'Press to Increment',
            style: TextStyle(
```

```
27              fontSize: 30,
28              fontWeight: FontWeight.w600,
29              ),
30          ),
31          ),
32      ),
33
34      /// the child widget below can use the scoped model
35      ///
36      body: ScopedModelDescendant<Counter>(
37          builder: (context, child, model) => Column(
38                  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
39                  children: [
40                  const Text(
41                      'Number increased to ...',
42                      style: TextStyle(
43                      fontSize: 30,
44                      fontWeight: FontWeight.w600,
45                      color: Colors.blueAccent,
46                      ),
47                  ),
48                  Center(
49                      child: Text(
50                      model.counter.toString(),
51                      style: const TextStyle(
52                          fontSize: 100,
53                          fontWeight: FontWeight.w600,
54                          color: Colors.red,
55                      ),
56                      ),
57                  ),
58                  TextButton(
59                      onPressed: () {
60                      Navigator.push(
61                          context,
62                          MaterialPageRoute(
63                          builder: (context) => const NextPage(),
64                          ),
65                      );
66                      },
67                      child: const Text(
68                      'Next Page',
69                      style: TextStyle(
```

```
                        fontSize: 30,
                        fontWeight: FontWeight.w600,
                        color: Colors.red,
                    ),
                    ),
                )
                ],
            )),
    );
}

AppBar customAppBar(String title) {
    return AppBar(
    centerTitle: true,
    //backgroundColor: Colors.grey[400],
    flexibleSpace: Container(
        decoration: const BoxDecoration(
        gradient: LinearGradient(
            colors: [
            Colors.pink,
            Colors.grey,
            ],
            begin: Alignment.topRight,
            end: Alignment.bottomRight,
        ),
        ),
    ),
    //elevation: 20,
    titleSpacing: 80,
    leading: const Icon(Icons.menu),
    title: Text(
        title,
        textAlign: TextAlign.left,
    ),
    actions: [
        ScopedModelDescendant<Counter>(
        builder: (context, child, model) => Container(
            padding: const EdgeInsets.all(5),
            child: Text(
            model.counter.toString(),
            style: const TextStyle(
                fontSize: 30,
                fontWeight: FontWeight.w900,
```

```
                color: Colors.white,
            ),
            ),
        ),
        ),
        buildIcons(
        const Icon(
            Icons.navigate_next,
        ),
        ),
        buildIcons(
        const Icon(Icons.search),
        ),
    ],
    );
}

IconButton buildIcons(Icon icon) {
    return IconButton(
    onPressed: () {},
    icon: icon,
    );
}
}
```

Now, as we go down the widget tree, we can use the Scoped Model Descendant with the type. So the descendant widgets can access that type with the help of model.

```
floatingActionButton: ScopedModelDescendant<Counter>(
        builder: (context, child, model) => FloatingActionButton.extended(
        onPressed: () {
            model.increment();
        },
....
```

In the same vein, the Next Page is the another descendant of Scoped Model. Consequently, at that page we can access all properties and methods of Counter class.

Moreover, the same state prevails across all the descendant widgets.

Let's take a look at the Next Page code.

```dart
import 'package:flutter/material.dart';
import 'package:provider_et_sqflite/model/counter.dart';
import 'package:scoped_model/scoped_model.dart';

class NextPage extends StatelessWidget {
const NextPage({Key? key}) : super(key: key);

static const String title = 'Next Page';

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text(title),
    ),

    /// the child widget below can use the scoped model
    ///
    floatingActionButton: ScopedModelDescendant<Counter>(
        builder: (context, child, model) => FloatingActionButton.extended(
        onPressed: () {
            model.decrement();
        },
        label: const Text(
            'Press to Decrement',
            style: TextStyle(
            fontSize: 30,
            fontWeight: FontWeight.w600,
            ),
        ),
        ),
    ),

    /// the child widget below can use the scoped model
    ///
    body: ScopedModelDescendant<Counter>(
        builder: (context, child, model) => Column(
                mainAxisAlignment: MainAxisAlignment.spaceEvenly,
                children: [
                const Text(
                    'Number lowered to ...',
                    style: TextStyle(
                    fontSize: 30,
```

```
                    fontWeight: FontWeight.w600,
                    color: Colors.blueAccent,
                    ),
                ),
                Center(
                    child: Text(
                    model.counter.toString(),
                    style: const TextStyle(
                        fontSize: 100,
                        fontWeight: FontWeight.w600,
                        color: Colors.red,
                    ),
                    ),
                ),
                ],
            )),
    );
}

AppBar customAppBar(String title) {
    return AppBar(
    centerTitle: true,
    //backgroundColor: Colors.grey[400],
    flexibleSpace: Container(
        decoration: const BoxDecoration(
        gradient: LinearGradient(
            colors: [
            Colors.pink,
            Colors.grey,
            ],
            begin: Alignment.topRight,
            end: Alignment.bottomRight,
        ),
        ),
    ),
    //elevation: 20,
    titleSpacing: 80,
    leading: const Icon(Icons.menu),
    title: Text(
        title,
        textAlign: TextAlign.left,
    ),
    actions: [
```

```
87          ScopedModelDescendant<Counter>(
88          builder: (context, child, model) => Container(
89              padding: const EdgeInsets.all(5),
90              child: Text(
91              model.counter.toString(),
92              style: const TextStyle(
93                  fontSize: 30,
94                  fontWeight: FontWeight.w900,
95                  color: Colors.white,
96              ),
97              ),
98          ),
99          ),
100         buildIcons(
101         const Icon(
102             Icons.navigate_next,
103         ),
104         ),
105         buildIcons(
106         const Icon(Icons.search),
107         ),
108     ],
109     );
110 }
111
112 IconButton buildIcons(Icon icon) {
113     return IconButton(
114     onPressed: () {},
115     icon: icon,
116     );
117 }
118 }
```

As a result, we've seen how we can manage the state across the whole flutter application.

Passing data across the Flutter Application

A Flutter User Interface can pass data all the way down the tree from parent to child. We can always pass them through constructors, or use Inherited widget.

But, doing that manually makes it cumbersome as our Flutter application gets bigger.

To solve this issue, we can use Scoped Model, which is a simplified version of Inherited widget. Of course, Provider works on the same principle, although having a lot more options.

Moreover, using Provider is not as easy as using Scoped Model. In both cases, we need to make a new Context.

Why?

Because the exposed data is included in the Context. Now any widget that uses that Context can access that data.

Now it's always a good practice to place that access point as low as possible. In fact, we should use Scoped Model, or Provider as closest as possible to the widget that uses that exposed data.

The biggest advantage of using Scoped Model is it separates the User Interface and Business Logic. And that too in a very simple way.

To get the full code please visit the respective GitHub Repository.

- For the full code snippet please visit the respective GitHub repository.[88]

Scoped Model and SQLite in Flutter

Just like Provider, we can use Scoped Model with SQLite database in Flutter.

In the previous section we've seen how we can use Scoped Model in Flutter. In a couple of previous sections, we've also examined the scope of using SQLite Database in Flutter. In this section we'll discuss how we can join Scoped Model and SQLite database, so that they work at tandem in Flutter.

We're going to build a Note-keeper app where we'll store Name and Location of users. Likewise, we've already built the same application with Provider and SQLite Database. If you have interest, please check it.

Firstly, we can always use SQLite database in Flutter. However, we need to use a special package or plugin sqflite which is available in pub.dev. We also need to use Future API, async, await keywords, and then functions to make it successful.

We've discussed this feature for absolute beginners in previous section, is Flutter single thread? If you're a complete beginner searching to know about Future in Flutter, please check it.

First thing first, to use SQLite Database in Flutter, we use the sqflite package.

Why?

Because this package provides classes and functions to interact with a SQLite database. There are other reasons too, using SQLite database is better than using a local file, or key-value store.

In addition, SQLite database provides faster CRUD. That is, we can create, retrieve, update and delete data. And, it's always better than any other local persistent solutions.

Besides sqflite package, we need to use another package path, that will define the location for storing the database on the disk.

For the beginners, here is a guide what SQLite database is.

[88] https://github.com/sanjibsinha/provider_et_sqflite/tree/scoped-model-first

What is SQLite database and how it works?

SQLite is a C-language library that implements many features at one go. It is small, fast, self-contained, high-reliability, full-featured, SQL database engine.

By the way, SQLite is the most used database engine in the world. Besides, SQLite database, file format is stable, cross-platform, and backwards compatible.

There are over 1 trillion SQLite databases in active use at present.

Therefore, let's go ahead and make our first Flutter Application with SQLite database.

Besides, using Scoped Model, and SQLite database, we'll also use Future Builder widget. We'll discuss Future Builder in detail later, meanwhile let's learn a few key points about Future Builder.

Scoped Model, SQLite database and Future Builder

How about getting a gentle introduction to Future Builder?

Well, to understand the whole mechanism behind building a Note-keeper application in Flutter, this initiation might help us.

Future Builder is a widget that builds itself based on the latest snapshot of interaction with a Future.

According to the documentation we must obtain the future object during State.initState, State.didUpdateWidget, or State.didChangeDependencies.

However, while using with Provider or Scoped Model, we use the Future Builder in a different way. We'll see that in a minute.

Next, let's have another gentle introduction to "scoped_model" package also. Because without this very useful package, we couldn't use Inherited Widget in the simplest way in Flutter.

How do you use scoped_model in flutter?

To start with we need to add all the dependencies first.

```
1  dependencies:
2  cupertino_icons: ^1.0.4
3  flutter:
4      sdk: flutter
5  path: ^1.8.0
6  scoped_model: ^2.0.0-nullsafety.0
7  sqflite: ^2.0.1
```

After that, with the help of Scoped Model we can easily pass a data model from a parent Widget to a child Widget.

Moreover, it also rebuilds all of the children that use the model when the model is updated.

The Scoped Model package provides three main classes.

Let's see what they are.

Firstly, the Model class as we've used in the User Model, like the following.

```
import 'package:scoped_model/scoped_model.dart';

import 'database_handler.dart';
import 'user.dart';

class UserModel extends Model {
User _userOne = User(name: 'Json Web', location: 'Detroit');
User get userOne => _userOne;

void addingUsers() {
    _userOne = userOne;

    notifyListeners();
}
}
```

We need another User class that will define the behaviour of user object and map the items to give an output in a List of items.

```
class User {
final int? id;
final String name;
final String location;

User({
    this.id,
    required this.name,
    required this.location,
});

User.fromMap(Map<String, dynamic> res)
    : id = res["id"],
        name = res["name"],
        location = res["location"];

Map<String, Object?> toMap() {
```

```
    return {
    'id': id,
    'name': name,
    'location': location,
    };
}
}
```

As a result, we can construct User object inside User model class.

Secondly, we need two more widgets.

The first one is the Scoped Model Widget where we tell the parent widget to pass the Model class properties and methods to its descendant children widgets.

Since we need to pass a Model deep down your Widget hierarchy, you can wrap our Model in a ScopedModel Widget. This will make the Model available to all descendant Widgets.

```
import 'package:flutter/material.dart';

import 'package:provider_et_sqflite/model/user_model.dart';
import 'package:scoped_model/scoped_model.dart';

import 'my_home_page.dart';

class MyApp extends StatelessWidget {
const MyApp({
    Key? key,
    required this.user,
}) : super(key: key);
final UserModel user;

// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    debugShowCheckedModeBanner: false,
    title: 'Scoped Model Simple',
    theme: ThemeData(
        primarySwatch: Colors.blue,
    ),

    /// child widggets are now under its scope
    /// and we can use this model anywhere below
```

```
    ///
    home: ScopedModel<UserModel>(
        model: UserModel(),
        child: const MyHomePage(),
    ),
    );
}
}
```

As a result, the Home Page can now make the model available to all descendant widgets.

Finally, we need the Scoped Model Descendant Widget that can listen to the Models for any changes.

```
import 'package:flutter/material.dart';
import 'package:scoped_model/scoped_model.dart';

import '../model/user_model.dart';
import '/model/database_handler.dart';
import '/model/user.dart';

class MyHomePage extends StatelessWidget {
const MyHomePage({Key? key}) : super(key: key);

static const String title = 'Database Handling';

@override
Widget build(BuildContext context) {
    final userModel = ScopedModel.of<UserModel>(context);

    final handler = DatabaseHandler();
    Future<int> addUsers() async {
    User firstUser = User(
        name: userModel.userOne.name,
        location: userModel.userOne.location,
    );
    List<User> listOfUsers = [
        firstUser,
    ];
    return await handler.insertUser(listOfUsers);
    }

    return ScopedModelDescendant<UserModel>(
    builder: (context, child, model) => Scaffold(
```

```
31          appBar: customAppBar(title),
32          body: FutureBuilder(
33          future: handler.retrieveUsers(),
34          builder: (BuildContext context, AsyncSnapshot<List<User>> snapshot) {
35              if (snapshot.hasData) {
36              return ListView.builder(
37                  itemCount: snapshot.data?.length,
38                  itemBuilder: (BuildContext context, int index) {
39                  return Card(
40                      child: ListTile(
41                      key: ValueKey<int>(snapshot.data![index].id!),
42                      contentPadding: const EdgeInsets.all(8.0),
43                      title: Text(
44                          snapshot.data![index].name,
45                          style: const TextStyle(
46                          fontSize: 30,
47                          color: Colors.red,
48                          ),
49                      ),
50                      subtitle: Text(
51                          snapshot.data![index].location,
52                          style: const TextStyle(
53                          fontSize: 20,
54                          color: Colors.blue,
55                          ),
56                      ),
57                      ),
58                  );
59                  },
60              );
61              } else {
62              return const Center(child: CircularProgressIndicator());
63              }
64          },
65          ),
66          floatingActionButton: FloatingActionButton.extended(
67          onPressed: () {
68              handler.initializeDB().whenComplete(() async {
69              await addUsers();
70              });
71
72              model.addingUsers();
73          },
```

```
        label: const Text(
            'Add Users',
            style: TextStyle(
            fontSize: 25,
            fontWeight: FontWeight.bold,
            ),
        ),
        ),
    ),
    );
}

AppBar customAppBar(String title) {
    return AppBar(
    centerTitle: true,
    //backgroundColor: Colors.grey[400],
    flexibleSpace: Container(
        decoration: const BoxDecoration(
        gradient: LinearGradient(
            colors: [
            Colors.pink,
            Colors.grey,
            ],
            begin: Alignment.topRight,
            end: Alignment.bottomRight,
        ),
        ),
    ),
    //elevation: 20,
    titleSpacing: 80,
    leading: const Icon(Icons.menu),
    title: Text(
        title,
        textAlign: TextAlign.left,
    ),
    actions: [
        buildIcons(
        const Icon(Icons.add_a_photo),
        ),
        buildIcons(
        const Icon(
            Icons.notification_add,
        ),
```

```
        ),
        buildIcons(
        const Icon(
            Icons.settings,
        ),
        ),
        buildIcons(
        const Icon(Icons.search),
        ),
    ],
    );
}

IconButton buildIcons(Icon icon) {
    return IconButton(
    onPressed: () {},
    icon: icon,
    );
}
}
```

The Model has been passed down from the parent to the child Widget tree using an InheritedWidget. When an InheritedWidget is rebuilt, it will rebuild all of the Widgets that depend on its data.

As a result, when we press the "Add Users" button, one User with name and location is added to the SQLite Database. We've already defined the insert and retrieve methods in a Database helper class, like the following.

```
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';

import 'user.dart';

class DatabaseHandler {
Future<Database> initializeDB() async {
    String path = await getDatabasesPath();
    return openDatabase(
    join(path, 'userthirteen.db'),
    onCreate: (database, version) async {
        await database.execute(
        "CREATE TABLE userthirteen(id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT N\
OT NULL, location TEXT NOT NULL)",
        );
```

```
    },
    version: 1,
    );
}

Future<int> insertUser(List<User> users) async {
    int result = 0;
    final Database db = await initializeDB();
    for (var user in users) {
    result = await db.insert('userthirteen', user.toMap());
    }
    return result;
}

Future<List<User>> retrieveUsers() async {
    final Database db = await initializeDB();
    final List<Map<String, Object?>> queryResult =
        await db.query('userthirteen');
    return queryResult.map((e) => User.fromMap(e)).toList();
}
}
```

Now, we need to add users through our User Model class, in two steps, like the following.

```
class UserModel extends Model {
User _userOne = User(name: 'Json Web', location: 'Detroit');
User get userOne => _userOne;
...
Widget build(BuildContext context) {
    final userModel = ScopedModel.of<UserModel>(context);

    final handler = DatabaseHandler();
    Future<int> addUsers() async {
    User firstUser = User(
        name: userModel.userOne.name,
        location: userModel.userOne.location,
    );
    List<User> listOfUsers = [
        firstUser,
    ];
    return await handler.insertUser(listOfUsers);
...
```

Therefore, we get our first user added to our Name-keeper Flutter application as press the "Add Users" button.

Figure 18.10 – Scoped Model and SQLite database example in Flutter

Now we can keep added them through User Model class by pressing the button. However, we need to manually add them in User Model class which is not advisable.

Yet, to understand the mechanism it's okay for the time being.

Later we'll pass that Model class to the Scoped Model Descendant Widget to initialise SQLite Database handling process by taking user inputs.

So stay tuned.

- For full code snippet please visit the respective GitHub Repository - [89]

[89] https://github.com/sanjibsinha/provider_et_sqflite/tree/sqflite-scoped-first

What is future builder in Flutter

The FutureBuilder widget obtains Future by change of state, or by dependencies.

Future Builder is a widget that builds itself. But, it requires a snapshot of interaction with a Future.

How will we get the Future?

We must have obtained the Future in three ways.

They are State.initState, State.didUpdateWidget, or State.didChangeDependencies.

What does it mean?

We may have obtained Future either through change of state, or by change in dependencies.

It means a lot.

Why?

The reason is, we can use FutureBuilder using Scoped Model or Provider. In other Words using the Inherited Widget.

In fact, we are going to do the same thing here.

We will use both Scoped Model and Provider to insert data to a SQLite database. As a result, the FutureBuilder widget will rebuild itself and retrieve the data.

Actually we will execute asynchronous functions. Consequently the asynchronous function will make the User Interface update.

By default FutureBuilder is stateful in nature. It maintains its own state.

How does FutureBuilder work?

The FutureBuilder shows messages like “loading”. It does that based on connection state. And, after that based on new “data”, or “snapshot”, it updates the UI. As a consequence we get a new view.

The advantage of FutureBuilder is, it does not use two “state variables”. When the new “data” arrives, it updates the “view”.

To sum up, the FutureBuilder is a wrapper or boilerplate of what we do.

What is the difference between FutureBuilder and StreamBuilder?

We’ll discuss #StreamBuilder class in a separate article. However, the StreamBuilder class has some similarities with the FutureBuilder class.

Firstly, they listen to changes that occur on their respective objects.

Secondly, after that, they trigger a new construction when they get the notification.

This is the basic functionality that they both practice.

Then, what is the difference?

The object makes the difference. This is the object that they are listening to.

As we told before, #Future is a representation of asynchronous function. As a result, the #Future has one and only answer.

If the #Future is completed successfully, we get the result. If not, we get the error.

On the other hand, the #Stream will get each new value. Even if it had an error, we would get the last value.

Therefore, the main difference is a #Future cannot listen to a change that varies. The #Future has one single answer always.

Enough talking.

Let us view the screenshots first. Viewing the images of our Flutter Applications will clarify the concept.

After that we will discuss the respective code.

When should I use FutureBuilder?

In our first Flutter Application we are going to use the Scoped Model and the Provider with a single FutureBuilder widget. As a result, it does not work properly.

However, the data has been inserted to the SQLite database properly.

Firstly, we will try to insert data to a SQLite database by a Provider object.

Let us press the Button. According to our code, it should insert two User objects. One from the Provider and the other from the Scoped Model class.

Because the FutureBuilder listens to both objects it should update the UI and display two user names.

But that did not happen.

The second user's name will only be displayed when we click the Button below.

If we click the Button "Add Users by Provider" again, only one name is displayed below the two Users. Although the second user is inserted, it does not reflect on the UI.

Let us check the code where we have mixed the Scoped Model and the Provider.

```dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import 'package:scoped_model/scoped_model.dart';

import '/model/user_model.dart';
import '/model/user_prvider.dart';
import '/model/database_handler.dart';
import '/model/user.dart';

class MyHomePage extends StatelessWidget {
const MyHomePage({Key? key}) : super(key: key);

static const String title = 'Provider, Scoped Model, SQLite';

@override
Widget build(BuildContext context) {
    final userModel = ScopedModel.of<UserModel>(context);
    final userProvider = Provider.of<UserProvider>(context);

    final handler = DatabaseHandler();
    Future<int> addUsers() async {
    User firstUser = User(
        name: userModel.userOne.name,
        location: userModel.userOne.location,
    );
    User secondUser = User(
        name: userProvider.userTwo.name,
        location: userProvider.userTwo.location,
    );
    List<User> listOfUsers = [
        firstUser,
        secondUser,
    ];
    return await handler.insertUser(listOfUsers);
    }

    return ScopedModelDescendant<UserModel>(
    builder: (context, child, model) => Scaffold(
        appBar: customAppBar(title),
        body: FutureBuilder(
        future: handler.retrieveUsers(),
        builder: (BuildContext context, AsyncSnapshot<List<User>> snapshot) {
```

```
44              if (snapshot.hasData) {
45              return ListView.builder(
46                  itemCount: snapshot.data?.length,
47                  itemBuilder: (BuildContext context, int index) {
48                  return Column(
49                      children: [
50                      Card(
51                          child: ListTile(
52                          key: ValueKey<int>(snapshot.data![index].id!),
53                          contentPadding: const EdgeInsets.all(8.0),
54                          title: Text(
55                              snapshot.data![index].name,
56                              style: const TextStyle(
57                              fontSize: 30,
58                              color: Colors.red,
59                              ),
60                          ),
61                          subtitle: Text(
62                              snapshot.data![index].location,
63                              style: const TextStyle(
64                              fontSize: 20,
65                              color: Colors.blue,
66                              ),
67                          ),
68                          ),
69                          elevation: 20,
70                      ),
71                      TextButton(
72                          onPressed: () {
73                          handler.initializeDB().whenComplete(() async {
74                              await addUsers();
75                          });
76                          model.addingUsers();
77                          },
78                          child: const Text(
79                          'Add Users by Scoped Model',
80                          style: TextStyle(
81                              fontSize: 20,
82                              fontWeight: FontWeight.w600,
83                          ),
84                          ),
85                      )
86                      ],
```

```
                );
                },
            );
            } else {
            return const Center(child: CircularProgressIndicator());
            }
        },
        ),
        floatingActionButton: FloatingActionButton.extended(
        onPressed: () {
            handler.initializeDB().whenComplete(() async {
            await addUsers();
            });
            userProvider.addingUsers();
        },
        label: const Text(
            'Add Users by Provider',
            style: TextStyle(
            fontSize: 25,
            fontWeight: FontWeight.bold,
            ),
        ),
        ),
    ),
    );
}

AppBar customAppBar(String title) {
    return AppBar(
    centerTitle: true,
    //backgroundColor: Colors.grey[400],
    flexibleSpace: Container(
        decoration: const BoxDecoration(
        gradient: LinearGradient(
            colors: [
            Colors.pink,
            Colors.grey,
            ],
            begin: Alignment.topRight,
            end: Alignment.bottomRight,
        ),
        ),
    ),
```

```
    elevation: 20,
    titleSpacing: 80,
    title: Text(
        title,
        textAlign: TextAlign.left,
    ),
    );
}
}
```

For full code please visit the respective GitHub Repository given at the end of this section.

However, we could have separated them. And that is what we have done for the second Flutter Application.

As a result, the FutureBuilder and our Flutter Application works perfectly.

How do you reset the future builder Flutter?

This time, we will add the first user by using the Scoped Model.

If we press the Button below it will add the user and it also displays the name and location of User. Moreover, below the display of the newly created User object, we can see the “Navigation Button”.

Let us see the code first. After that we will click the “Next Page” Button to add another User by the Provider.

Firstly, we see the User Model class that extends the Model class from the Scoped Model dependency.

```
import 'package:scoped_model/scoped_model.dart';

import 'user.dart';

class UserModel extends Model {
User _userModel = User(name: 'John Smith', location: 'Back East');
User get userModel => _userModel;

void addingUsers() {
    _userModel = userModel;

    notifyListeners();
}
}
```

Secondly we will see the FutureBuilder widget that uses the Dependent Scoped User Model to add the Users.

```dart
import 'package:flutter/material.dart';

import 'package:scoped_model/scoped_model.dart';

import '/model/user_model.dart';

import '/model/database_handler.dart';
import '/model/user.dart';
import 'provider_home_page.dart';

class ModelHomePage extends StatelessWidget {
const ModelHomePage({Key? key}) : super(key: key);

static const String title = 'Adding by Scoped Model';

@override
Widget build(BuildContext context) {
    final userModel = ScopedModel.of<UserModel>(context);

    final handler = DatabaseHandler();
    Future<int> addUsers() async {
    User firstUser = User(
        name: userModel.userModel.name,
        location: userModel.userModel.location,
    );

    List<User> listOfUsers = [
        firstUser,
    ];
    return await handler.insertUser(listOfUsers);
    }

    return ScopedModelDescendant<UserModel>(
    builder: (context, child, model) => Scaffold(
        appBar: customAppBar(title),
        body: FutureBuilder(
        future: handler.retrieveUsers(),
        builder: (BuildContext context, AsyncSnapshot<List<User>> snapshot) {
            if (snapshot.hasData) {
            return ListView.builder(
                itemCount: snapshot.data?.length,
                itemBuilder: (BuildContext context, int index) {
                return Column(
```

```
                    children: [
                    Card(
                        child: ListTile(
                        key: ValueKey<int>(snapshot.data![index].id!),
                        contentPadding: const EdgeInsets.all(8.0),
                        title: Text(
                            snapshot.data![index].name,
                            style: const TextStyle(
                            fontSize: 30,
                            color: Colors.red,
                            ),
                        ),
                        subtitle: Text(
                            snapshot.data![index].location,
                            style: const TextStyle(
                            fontSize: 20,
                            color: Colors.blue,
                            ),
                        ),
                        ),
                        elevation: 20,
                    ),
                    TextButton(
                        onPressed: () {
                        Navigator.push(
                            context,
                            MaterialPageRoute(
                            builder: (context) => const ProviderHomePage(),
                            ),
                        );
                        },
                        child: const Text(
                        'Next Page, Add by Provider',
                        style: TextStyle(
                            fontSize: 20,
                            fontWeight: FontWeight.w600,
                        ),
                        ),
                    )
                    ],
                );
                },
            );
```

```
87              } else {
88              return const Center(child: CircularProgressIndicator());
89              }
90          },
91          ),
92          floatingActionButton: FloatingActionButton.extended(
93          onPressed: () {
94              handler.initializeDB().whenComplete(() async {
95              await addUsers();
96              });
97              userModel.addingUsers();
98          },
99          label: const Text(
100             'Add Users by Model',
101             style: TextStyle(
102             fontSize: 25,
103             fontWeight: FontWeight.bold,
104             ),
105         ),
106         ),
107     ),
108     );
109 }
110
111 AppBar customAppBar(String title) {
112     return AppBar(
113     centerTitle: true,
114     //backgroundColor: Colors.grey[400],
115     flexibleSpace: Container(
116         decoration: const BoxDecoration(
117         gradient: LinearGradient(
118             colors: [
119             Colors.pink,
120             Colors.grey,
121             ],
122             begin: Alignment.topRight,
123             end: Alignment.bottomRight,
124         ),
125         ),
126     ),
127     elevation: 20,
128     titleSpacing: 80,
129     title: Text(
```

```
        title,
        textAlign: TextAlign.left,
    ),
    );
}
}
```

Next, we go the next page where we can add the second User by using the Provider.

The next page correctly shows that one User has already been added to the SQLite database.

Now, when we add the Button “Add Users by Provider”, another user is added. Moreover, two users are displayed correctly on the UI.

Let us take a look at the code now.

Firstly we will take a look at the User Provider class that extends the ChangeNotifier. After that it notifies the listeners.

```
import 'package:flutter/material.dart';

import 'user.dart';

class UserProvider with ChangeNotifier {
User _userProvider = User(name: 'Json Web', location: 'Detroit');
User get userProvider => _userProvider;

void addingUsers() {
    _userProvider = userProvider;

    notifyListeners();
}
}
```

Secondly we will see the FutureBuilder that rebuilds when the User object changes its state.

We have created another FutureBuilder for this page, so that we can add users by Provider.

```dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import '/model/user_prvider.dart';
import '/model/database_handler.dart';
import '/model/user.dart';

class ProviderHomePage extends StatelessWidget {
const ProviderHomePage({Key? key}) : super(key: key);

static const String title = 'Adding By Provider';

@override
Widget build(BuildContext context) {
    final userProvider = Provider.of<UserProvider>(context);

    final handler = DatabaseHandler();
    Future<int> addUsers() async {
    User secondUser = User(
        name: userProvider.userProvider.name,
        location: userProvider.userProvider.location,
    );
    List<User> listOfUsers = [
        secondUser,
    ];
    return await handler.insertUser(listOfUsers);
    }

    return Scaffold(
    appBar: AppBar(
        title: const Text(title),
    ),
    body: FutureBuilder(
        future: handler.retrieveUsers(),
        builder: (BuildContext context, AsyncSnapshot<List<User>> snapshot) {
        if (snapshot.hasData) {
            return ListView.builder(
            itemCount: snapshot.data?.length,
            itemBuilder: (BuildContext context, int index) {
                return Column(
                children: [
                    Card(
                    child: ListTile(
```

```
                        key: ValueKey<int>(snapshot.data![index].id!),
                        contentPadding: const EdgeInsets.all(8.0),
                        title: Text(
                        snapshot.data![index].name,
                        style: const TextStyle(
                            fontSize: 30,
                            color: Colors.red,
                        ),
                        ),
                        subtitle: Text(
                        snapshot.data![index].location,
                        style: const TextStyle(
                            fontSize: 20,
                            color: Colors.blue,
                        ),
                        ),
                    ),
                    elevation: 20,
                    ),
                ],
                );
            },
            );
        } else {
            return const Center(child: CircularProgressIndicator());
        }
        },
    ),
    floatingActionButton: FloatingActionButton.extended(
        onPressed: () {
        handler.initializeDB().whenComplete(() async {
            await addUsers();
        });
        userProvider.addingUsers();
        },
        label: const Text(
        'Add Users by Provider',
        style: TextStyle(
            fontSize: 25,
            fontWeight: FontWeight.bold,
        ),
        ),
    ),
```

```
87      );
88  }
89  }
```

In the second Flutter Application we have successfully used The Scoped Model and the Provider. However, we have implemented the FutureBuilder widget separately.

For the full code snippet for this Flutter Application, please visit the respective GitHub Repository.

- For full code snippet please visit the respective GitHub Repository - [90]

[90] https://github.com/sanjibsinha/provider_et_sqflite/tree/experiment-provider-scoped-one

19. Future then, aync, await, API, JSON: Let's build a Current Weather Tracker App

There are plugins like "Geolocator" and "http" that will easily connect through API and track live current Weather data.

However, we need to understand a few important concepts such as Future then, aync, await, and API, JSON in Flutter.

We will build the Current Weather Tracker App step-by-step. And, finally, we will see the output at the end.

However, you can clone each step from the respective GitHub repoitory branches.

Let's name the App - "WithHer".

Future Flutter: WithHer App – Step 1

We are going to build a Weather App. As before, we'll build it step by step. And this is our first step where we will understand a key concept – Future in Flutter.

But to start with let me assure you one thing first.

It does not take much effort to build a weather app. Let's name it "WithHer".

Why it is not difficult?

Because we will build this app with a Flutter Geolocator plugin. In this first step, first we'll see how we can get the location. As a result, we will print the latitude and longitude.

After that, we will discuss the role of Future in Flutter. By the way, the Future class returns the result of an asynchronous programme.

Why we need to understand the role of Future in getting the location?

We'll see it in a minute.

However, besides, we need to understand what is asynchronous programming. In addition, we will have to understand two keywords – async and await.

How to the get the location in Flutter? Firstly, we need to add the dependency of the Geolocator plugin in our "pubspec.yaml" file.

```
dependencies:
flutter:
    sdk: flutter

cupertino_icons: ^1.0.2
geolocator: ^8.2.0
```

Secondly, in our Flutter app, we need to import the plugin where we need it.

To make it simple, we will test our weather app in the top-level main() file.

```
import 'package:geolocator/geolocator.dart';
```

How do we know a location?

Certainly the latitude and the longitude help us to know the exact location.

And the Geolocator plugin helps us to control how exact we want to be. However, we will keep our priority at the lowest level.

Why?

Because if we want to get the exact location of the user, it may take more time. For that reason, we will make it the lowest. So that, within a radius of 500 meters we can get the weather update.

With reference to that, we must remember one thing. The Geolocator plugin gets the data from internet.

Therefore, it may take an uncertain time. Moreover, the plugin has curtailed our job. If we want to do the low level plumbing and write the whole code we have to write long lines of code.

In addition, we need a certain level of expertise.

But with a few lines of code, the Geolocator plugin has done the same task.

```
void getLocation() async {
    Position position = await Geolocator.getCurrentPosition(
        desiredAccuracy: LocationAccuracy.lowest);
    print(position);
}
```

Actually we're riding the Geolocator plugin's back and shoulders and curtail the heavy task. Right?

Geolocator plugin makes our life easier

Let us see the full code snippet now.

```dart
import 'package:flutter/material.dart';
import 'package:geolocator/geolocator.dart';

void main() {
runApp(const MyApp());
}

class MyApp extends StatelessWidget {
const MyApp({Key? key}) : super(key: key);

// This widget is the root of your application.
@override
Widget build(BuildContext context) {
return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(
    primarySwatch: Colors.blue,
    ),
    home: const MyHomePage(title: 'Flutter Demo Home Page'),
);
}
}

class MyHomePage extends StatefulWidget {
const MyHomePage({Key? key, required this.title}) : super(key: key);

final String title;

@override
State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
void getLocation() async {
Position position = await Geolocator.getCurrentPosition(
    desiredAccuracy: LocationAccuracy.lowest);
print(position);
}

@override
Widget build(BuildContext context) {
return Scaffold(
    appBar: AppBar(
```

```
    title: Text(widget.title),
    ),
    body: Center(
    child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
        const Text(
            'We\'re trying to find the Lat and Lang:',
        ),
        Text(
            'Location',
            style: Theme.of(context).textTheme.headline4,
        ),
        ],
    ),
    ),
    floatingActionButton: FloatingActionButton(
    onPressed: () {
        getLocation();
    },
    tooltip: 'Get Location',
    child: const Icon(Icons.location_on),
    ), // This trailing comma makes auto-formatting nicer for build methods.
);
}
}
```

As we see, the getLocation() method uses two keywords "async" and "await". Later we have called that method through the onPressed property of the floating action button.

If you want to follow the steps, please clone this branch of GitHub repository.

As we see the above code we find that the two keywords "async" and "await" play an important role here.

We'll come back to that point in a minute. Before that let's run the app.

Let's run the app first.

If we click the floating action button, we will get the latitude and longitude.

However, without the user's permission we cannot get her location. Right?

Therefore, if we click the floating action button to get the location, the app will seek permission of the user.

Click allow to proceed. And in return it will print the latitude and longitude in the terminal.

```
Latitude: 22.5892652, Longitude: 88.3056119
```

Our WithHer App is working perfectly. Now, the time has come to understand what is Future in Flutter. In addition, what is the relation between the Future class and asynchronous programming.

What is asynchronous programming?

Firstly, there are two types of programming. Synchronous and asynchronous.

We need to understand why we need asynchronous programming?

Take a look at the following code which represents synchronous programming.

```
/// an example of synchronous programme
///

void main() {
callEveryTask();
}

void callEveryTask() {
doThisFirst();
doThisSecond();
doThisThird();
}

void doThisFirst() {
print('Doing it first');
}

void doThisSecond() {
print('Doing it second');
}

void doThisThird() {
print('Doing it third');
}
```

When we run the above code, the execution will take synchronously. That means it will execute the first function. Then the second, and lastly it executes the third function.

```
Output:

Doing it first
Doing it second
Doing it third
```

The code execution is sequential. But what will happen, if the second function takes much time to execute?

In that case, the third function will have to wait until the second function finishes its task. Right?

Now, consider a case, where the second function is downloading a big file, or gets an image from the internet.

In such cases, user will have to wait to get the result of the third function.

But we don't want this to happen.

Why?

It does not go with our principle. Because we want to give the user a good experience, we should allow the third function to go ahead and after that the second function may execute.

What is Future in Flutter?

We can achieve this with the help of the Future class in Flutter.

The Future class has a named constructor Future.delayed(). It runs the computation after a certain delay.

Let's use this Future constructor to delay the second function for 2 seconds.

```
/// an example of asynchronous programme
///

void main() {
callEveryTask();
}

void callEveryTask() {
doThisFirst();
doThisSecond();
doThisThird();
}

void doThisFirst() {
```

```
String result = 'First task completed.';
print('Doing it first');
}

String doThisSecond() {
String result = 'Second task completed.';
Duration duration = const Duration(seconds: 2);
Future.delayed(duration, () {
    print('Doing it second');
    result;
});
return result;
}

void doThisThird() {
String result = 'Third task completed.';
print('Doing it third');
}
```

As a result, when we run the code, the third function will execute after the first function executes.

Then, after 2 seconds the second function will execute.

```
output ##

Doing it first
Doing it third
Doing it second
```

This time, we have coded asynchronously. Although we have established a relation between the Future class and asynchronous programming, yet we need to know two keywords – "async" and "await".

What are async and await?

Just like any other TYPE in Dart, the Future is also a TYPE. As we have learned before that every class is a TYPE of an object, or instance of that class, Future is also a TYPE.

Why?

The reason is simple. It's a class. Right?

Therefore, we can change the TYPE of the second function from String to Future. But to do that, we have to use two keywords – async and await.

So our previous code changes as follows.

```
void main() {
callEveryTask();
}

void callEveryTask() async {
doThisFirst();
String secondTask = await doThisSecond();
doThisThird(secondTask);
}

void doThisFirst() {
String result = 'First task completed.';
print('Doing it first');
}

Future doThisSecond() async {
String result = '..second task completed..';
Duration duration = const Duration(seconds: 2);
await Future.delayed(duration, () {
    result;

    print('Please Wait ....');
});
return result.toString();
}

void doThisThird(String secondTask) {
String result = 'Third task completed.';
print('Doing it third with $secondTask');
}
```

We have tweaked the previous code a little bit. Now, we have forced the third function to take the output from the second function as its input.

As a result, we need to tell the second function that another function is waiting for your output. Therefore, if you have anything to do, do it now, However, after that, pass the output to the third function.

As a result, we get the following output.

```
output ##

Doing it first
Please Wait ....
Doing it third with ..second task completed..
```

Now, as the second function stops for 2 seconds, the user gets a message like "Please Wait ....".

Certainly it gives the user a better experience. Because the user knows that she has to wait, she will wait patiently.

This is an introduction to Future and asynchronous programming. Later while we build the weather app, we will discuss it more.

Flutter State: WithHer App – Step 2

We have been building a weather App. In the first step, the "WithHer" App has used the Geolocator plugin. In addition, we have also used a Stateful Widget.

Why we have used a Stateful Widget? We will realise as we progress.

But before that, we need to understand a few important concepts that revolve around the State class.

Firstly, the State class is an abstract class that defines the logic and internal state for a Stateful Widget.

Secondly, it does not act like a Stateless Widget.

What is the main difference? During the lifetime of a Stateless Widget, it does not change. On the contrary, it destroys the old instance and creates a new instance.

But in its lifetime, a Stateful Widget may change its information.

Let us consider a Flutter App that has two pages. We can use the "routes" property to mention the path.

```dart
import 'package:flutter/material.dart';
import 'first_page.dart';
import 'my_app_home.dart';
import 'second_page.dart';

class MyApp extends StatelessWidget {
const MyApp({Key? key}) : super(key: key);

// This widget is the root of your application.
@override
Widget build(BuildContext context) {
```

```
    return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(
        primarySwatch: Colors.blue,
    ),
    //home: const MyAppHome(title: 'Flutter Demo Home Page'),
    initialRoute: '/',
    routes: {
        '/': (context) => const FirstPage(),
        SecondPage.routeName: (context) => const SecondPage(),
    },
    );
}
}
```

Certainly the First page is a Stateless widget which has a Text Button that redirects us to the second page.

```
import 'package:flutter/material.dart';
import 'package:with_her/view/second_page.dart';

class FirstPage extends StatelessWidget {
const FirstPage({Key? key}) : super(key: key);

static const routeName = '/first-page';

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('First Page'),
    ),
    body: Center(
        child: Container(
        margin: const EdgeInsets.all(20.0),
        width: 400.0,
        height: 150.0,
        child: TextButton(
            onPressed: () {
            Navigator.push(
                context,
                MaterialPageRoute(
                builder: (context) => const SecondPage(),
```

```
                ),
            );
            },
            child: const Text('Go to Second Page'),
        ),
        ),
    ),
    );
}
}
```

On the other hand, the second page is a Stateful Widget with a Text Button that on pressing takes us back to the First page again.

The second page has the following code.

```
import 'package:flutter/material.dart';

class SecondPage extends StatefulWidget {
const SecondPage({Key? key}) : super(key: key);

static const routeName = '/second-page';

@override
State<SecondPage> createState() => _SecondPageState();
}

class _SecondPageState extends State<SecondPage> {
@override
void initState() {
    // TODO: implement initState
    super.initState();
    print('I am called first, after each State object has been created.');
}

@override
void deactivate() {
    // TODO: implement deactivate
    super.deactivate();
    print('The state object removed from the tree');
}

@override
```

```
Widget build(BuildContext context) {
    print('I am called after initState() method.');
    return Scaffold(
    appBar: AppBar(
        title: const Text('Second Page'),
    ),
    body: Center(
        child: Container(
        margin: const EdgeInsets.all(20.0),
        width: 400.0,
        height: 150.0,
        child: TextButton(
            onPressed: () {
            Navigator.pop(context);
            },
            child: const Text('Go to Frist Page'),
        ),
        ),
    ),
    );
}
}
```

How Flutter State works

As we come to the second page, two methods fire immediately.

In our terminal we have got two outputs.

```
I am called first, after each State object has been created.
I am called after initState() method.
```

The first output comes from the initState() method. And the second output comes from the build() method.

However, when we press the back button to go back to the first page, another output comes in.

```
The state object removed from the tree
This output comes from the deactivate() method.
```

As a result, we have a clear picture how the life cycle of the Stateful Widget starts and ends.

As we progress with our weather app, we will discuss this topic again. But now we have a basic understanding of how the Stateful widget works.

API Flutter: WithHer App – Step 3

How can we use API in Flutter? In fact, only with API we can build the weather App "WithHer". Without the API? We cannot build it.

Therefore, firstly, we need to know what is API? Secondly, how we can use the API so that we can build the weather App.

As a result, we would get the current weather of any city in the world.

Till now, we have been progressing a little bit. We have learned how we can use Future in Flutter.

After that, we have also discussed the life cycle of Stateful Widget.

Why?

Because Stateful Widget will play an important role in using the API that will help us to get the current weather.

What is an API?

An API is a short name for Application Programming Interface.

But, to clarify, an API is actually a type of software.

What kind of software is it?

API is a kind of software that acts as an intermediary between two applications. In short, an API can connect two applications.

In a wider sense, an API can connect two computers so that they can communicate.

If we go to read the Flutter documentation, it says as follows:

Welcome to the Flutter API reference documentation!

In fact, when we use Flutter framework, we use API all the time. But we do that without knowing.

For example, Flutter and Dart gives us plenty of classes, methods, libraries to use.

Why?

The reason is simple. We do not have to write everything from scratch. Instead, we can use them and build any app in a very short period of time.

The same thing happens in our weather app, we will get the data from the open weather map website.

We can go to that website anytime, and check the current weather of any place.

Our "WithHer" App will fetch the weather data from there. And, certainly, with the help of the API that they provide.

Without that API, our App cannot display the current data.

How can we use API in Flutter?

We will see in a minute how we can use API in Flutter. But, before that, we would like to see what we are going to build.

In the first screenshot, we will see two Text Widgets. They display a static text.

But below the Text Widgets, we have a Text Button. Once we press the Button, the Text Widgets display the name of the city, and the current weather there.

In our previous section, we have seen how we can get the latitude and longitude.

For that we used the Geolocator plugin. Right?

Actually, in this case, we get the city name and current weather with the help of latitude and longitude. But, with that, we need the API key.

So that, we can get the weather as follows.

By the way, you may wonder about the name of the city.

Kara-Kulja is the center of Kara-Kulja District in Osh Region of Kyrgyzstan.

In the morning, there was nice weather. Clear blue sky. I had seen in my mind-eyes.

API and Format Exception error

Firstly, we need to register at the open weather map website to get the API key, or app ID.

Secondly, we need to use the "http" plugin and add the dependency in our "puspec.yaml" file.

```
dependencies:
flutter:
    sdk: flutter

cupertino_icons: ^1.0.2
geolocator: ^8.2.0
http: ^0.13.4
```

Finally, we need to import the package in our file.

Then, firstly you register at the open weather map website, and get your own API key. After that you will find that as we request for data, the API responds and sends data in JSON format.

We will discuss JSON in our next section. Otherwise, we will not understand why many people get the format exception error.

At present, we will take a look at the full code.

```dart
import 'dart:convert';
import 'dart:ui';
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

class MyAppHome extends StatefulWidget {
const MyAppHome({Key? key, required this.title}) : super(key: key);

final String title;

@override
State<MyAppHome> createState() => _MyAppHomeState();
}

class _MyAppHomeState extends State<MyAppHome> {
String apiKey = 'Your Key';
double lat = 40.7128;
double lon = 74.0060;

Future<String> getCityName() async {
    final httpsUri = Uri.http('api.openweathermap.org', '/data/2.5/weather', {
    'lat': '$lat',
    'lon': '$lon',
    'appid': apiKey,
    });

    var request = await http.get(httpsUri);
    if (request.statusCode == 200) {
    String data = request.body.toString();
    var city = jsonDecode(data)['name'];

    return city;
    } else {
    return '${request.statusCode}';
    }
}

Future<String> getWeather() async {
    final httpsUri = Uri.http('api.openweathermap.org', '/data/2.5/weather', {
    'lat': '$lat',
    'lon': '$lon',
    'appid': apiKey,
    });
```

```
44
45      var request = await http.get(httpsUri);
46      if (request.statusCode == 200) {
47      String data = request.body.toString();
48      var description = jsonDecode(data)['weather'][0]['description'];
49
50      return description;
51      } else {
52      return '${request.statusCode}';
53      }
54  }
55
56  String city = 'Your ';
57  String description = 'Good Weather ';
58
59  @override
60  Widget build(BuildContext context) {
61      //var city = getData();
62      return Scaffold(
63      appBar: AppBar(
64          title: const Text('Get City and Weather'),
65      ),
66      body: Center(
67          child: Column(
68          mainAxisAlignment: MainAxisAlignment.center,
69          children: <Widget>[
70              Text(
71              'Welcome to ${city.toString()} city!',
72              style: Theme.of(context).textTheme.headline6,
73              ),
74              Text(
75              description.toString(),
76              style: Theme.of(context).textTheme.headline4,
77              ),
78              const SizedBox(
79              height: 20.0,
80              ),
81              TextButton(
82              onPressed: () {
83                  getCityName().then((String result) {
84                  setState(() {
85                      city = result;
86                  });
```

```
                });
                getWeather().then((String result) {
                setState(() {
                    description = result;
                });
                });
            },
            child: const Text(
                'Get City and Current Weather',
                style: TextStyle(
                color: Colors.white,
                fontSize: 30.0,
                fontWeight: FontWeight.bold,
                ),
            ),
            style: ButtonStyle(
                backgroundColor: MaterialStateProperty.all(Colors.red),
                shape: MaterialStateProperty.all(
                RoundedRectangleBorder(
                    borderRadius: BorderRadius.circular(30.0),
                ),
                ),
            ),
            )
        ],
        ),
    ),
    // This trailing comma makes auto-formatting nicer for build methods.
    );
}
}
```

We have hard coded the latitude, longitude and the app ID.

As we progress, we will take the inputs and pass them dynamically.

But, till now, we have fetched the data from the open weather map website with the help of API.

Without API, we cannot use the data as follows.

```
final httpsUri = Uri.http('api.openweathermap.org', '/data/2.5/weather', {
    'lat': '$lat',
    'lon': '$lon',
    'appid': apiKey,
    });
```

We cannot request the data as follows.

```
var request = await http.get(httpsUri);
```

After that, with the help of Dart convert package, we have converted the JSON data to a human readable format.

```
import 'dart:convert';
...
String data = request.body.toString();
    var description = jsonDecode(data)['weather'][0]['description'];
```

Overall, we have progressed a lot to display the current weather data. In the next sections, we will progress more.

Stay tuned.

JSON Flutter: WithHer App – Step 4

As we progress, we find that we need to format structured data to run the weather app. In fact, in many Flutter app, at some point we need to format JSON data.

In our "WithHer" App, we have done that.

It proves that our weather app is working. As a result, we have been able to format the JSON data that comes from the open weather map website API.

In our previous section we have discussed the role of API in Flutter.

And before that, we have discussed why we need a Stateful Widget and what is Future in Flutter.

We need to understand them for the sake of our weather app. For example, without API, Stateful Widget, and Future, we cannot build this weather App.

Now, we need to know how we format the JSON data.

Firstly, we will know what does the term JSON mean? Secondly, we will learn how we have formatted the JSON data.

What is JSON in Flutter?

Firstly, let us see a small JSON data which will give us an idea about how it looks.

```dart
import 'dart:convert';

void main() {
var userProfile = {
"name": "Json Web",
"profession": "Spy",
"location": "Unknown"
};

String jsonString = jsonEncode(userProfile);


Map<String, dynamic> user = jsonDecode(jsonString);
print('${user['name']} is a ${user['profession']} and his location'
    ' is ${user['location']}.');

}
// // Json Web is a Spy and his location is Unknown.
```

The above code shows that JSON data is a structured data that we can encode into a String and later decode.

But we can always make this code better.

Therefore, we can use a User model to avoid the compile time error.

```dart
import 'dart:convert';

class User {
final String? name;
final String? profession;
final String? location;


User(this.name, this.profession, this.location);

User.fromJson(Map<String, dynamic> json)
    : name = json['name'],
        profession = json['profession'],
        location = json['location'];

Map<String, dynamic> toJson() => {
        'name': name,
        'profession': profession,
```

```
        'location': location,
    };
}

void main() {
User userProfile = User('Json Web', 'Spy', 'Unknown');

String json = jsonEncode(userProfile);

Map<String, dynamic> userMap = jsonDecode(json);
var user = User.fromJson(userMap);


print('${user.name} is a ${user.profession} and his location'
    ' is ${user.location}.');

}

// Json Web is a Spy and his location is Unknown.
```

In that case, our code will be more type safe which we want. But in our weather App, we can decode the JSON data.

The weather map JSON data and user location model class In our weather App, we have added the dependency of the "http" plugin in our "pubspec.yaml" file.

Why?

Because we will request the open weather map API to get the data in JSON format.

After that, we have used the Dart convert package to decode JSON data.

Let us see the user location model class firstly.

```
import 'dart:convert';
import 'package:http/http.dart' as http;

class UserLocation {
String apiKey = 'Secret Key';
double lat = 40.7128;
double lon = 74.0060;

Future<String> getCityName() async {
    final httpsUri = Uri.http('api.openweathermap.org', '/data/2.5/weather', {
    'lat': '$lat',
    'lon': '$lon',
```

```
    'appid': apiKey,
    });

    var request = await http.get(httpsUri);
    if (request.statusCode == 200) {
    String data = request.body.toString();
    var city = jsonDecode(data)['name'];

    return city;
    } else {
    return '${request.statusCode}';
    }
}

Future<String> getWeather() async {
    final httpsUri = Uri.http('api.openweathermap.org', '/data/2.5/weather', {
    'lat': '$lat',
    'lon': '$lon',
    'appid': apiKey,
    });

    var request = await http.get(httpsUri);
    if (request.statusCode == 200) {
    String data = request.body.toString();
    var description = jsonDecode(data)['weather'][0]['description'];

    return description;
    } else {
    return '${request.statusCode}';
    }
}
}
```

Secondly, we will take a look at how the open weather map website sends us the current weather data in JSON format.

For example, we have typed any city name, then the website will show the sample JSON data as follows.

```
{
"coord": {
    "lon": -122.08,
    "lat": 37.39
},
"weather": [
    {
    "id": 800,
    "main": "Clear",
    "description": "clear sky",
    "icon": "01d"
    }
],
"base": "stations",
"main": {
    "temp": 282.55,
    "feels_like": 281.86,
    "temp_min": 280.37,
    "temp_max": 284.26,
    "pressure": 1023,
    "humidity": 100
},
"visibility": 10000,
"wind": {
    "speed": 1.5,
    "deg": 350
},
"clouds": {
    "all": 1
},
"dt": 1560350645,
"sys": {
    "type": 1,
    "id": 5122,
    "message": 0.0139,
    "country": "US",
    "sunrise": 1560343627,
    "sunset": 1560396563
},
"timezone": -25200,
"id": 420006353,
"name": "Mountain View",
"cod": 200
```

```
}
```

As a result, when we request the data with our APP key, or app ID, it will send us the data in the above format.

Certainly, we need to provide the latitude and longitude also.

As we progress, we will make it simple.

But at present it shortens the view page. Because we have defined the properties and methods in the model class.

The Object Oriented Approach

Now we can create a user location object. And, after that, we can access the methods.

```
UserLocation location = UserLocation();
```

It makes our code more object oriented than before.

There is one problem that we need to solve later.

What is that?

We still hard code the data. Instead we can make them dynamic.

Let's see the landing page.

```
import 'package:flutter/material.dart';

import '../model/location.dart';

/// expleining json
///
class MyAppHome extends StatefulWidget {
const MyAppHome({Key? key, required this.title}) : super(key: key);

final String title;

@override
State<MyAppHome> createState() => _MyAppHomeState();
}

class _MyAppHomeState extends State<MyAppHome> {
var city = 'Your ';
var description = 'Good Weather ';
```

```
19
20  @override
21  Widget build(BuildContext context) {
22      UserLocation location = UserLocation();
23
24      return Scaffold(
25      appBar: AppBar(
26          title: const Text('Get City and Weather'),
27      ),
28      body: Center(
29          child: Column(
30          mainAxisAlignment: MainAxisAlignment.center,
31          children: <Widget>[
32              Text(
33              'Welcome to ${city.toString()} city!',
34              style: Theme.of(context).textTheme.headline6,
35              ),
36              Text(
37              description.toString(),
38              style: Theme.of(context).textTheme.headline4,
39              ),
40              const SizedBox(
41              height: 20.0,
42              ),
43              TextButton(
44              onPressed: () {
45                  location.getCityName().then((result) {
46                  setState(() {
47                      city = result;
48                  });
49                  });
50                  location.getWeather().then((result) {
51                  setState(() {
52                      description = result;
53                  });
54                  });
55              },
56              child: const Text(
57                  'Get City and Current Weather',
58                  style: TextStyle(
59                  color: Colors.white,
60                  fontSize: 30.0,
61                  fontWeight: FontWeight.bold,
```

```
                ),
            ),
            style: ButtonStyle(
                backgroundColor: MaterialStateProperty.all(Colors.red),
                shape: MaterialStateProperty.all(
                RoundedRectangleBorder(
                    borderRadius: BorderRadius.circular(30.0),
                ),
                ),
            ),
            )
        ],
        ),
    ),
    // This trailing comma makes auto-formatting nicer for build methods.
    );
}
}
// you can clone this Step
```

The code repositories for this branch[91]

In the above code, we have used a special Future method.

In the next section, we will dig deeper to know that.

Future Then: WithHer App – Step 5

We told before that we would come back and discuss the role of the "Future then" method in Flutter. Therefore, here we are.

In this step, we have accomplished many tasks. Certainly, we have done that with a special Future method – then().

Future, then, async, and await

In our App, why do we need the "Future then" method? What is the difference with the keywords – "Future async, and await"?

In this section we will find the answers. In addition, we have also changed the location class in our weather App.

[91] https://github.com/sanjibsinha/with_her/tree/fourth-step-json-explained

As a result, our weather app is now getting the latitude and longitude from the Geolocator plugin. We're no longer hard coding the value.

Now our "WithHer" App becomes dynamic. We will discuss that part in the next section.

In this part we will concentrate on Future class.

Why?

Because the Future class plays an important role in our App.

What is Future.then() method?

We have learned that Flutter and Dart is single thread. It does all the work on its main thread. It has no worker thread that runs parallel.

The main thread handles small tasks. When it simultaneously performs heavy task, it may hang, or freeze.

That is why the multi thread concept comes up.

Now, the question is, how Flutter manages this heavy task?

The answer is asynchronous programming. In synchronous programming, everything goes step by step. One after another.

Consequently, if a heavy task falls in the middle, or in the beginning, the whole program halts.

Asynchronous programming solves that issue. Flutter and Dart lets the small tasks to perform before the heavy task. And the heavy task is performed in the background.

Why the Future class is important?

To understand the importance of Future class, let us see a Dart program first where we are downloading a dummy weather data.

The time to download the data takes 8 seconds.

Therefore when we call this function it takes 8 seconds and after that it prints the output.

```
void main() {
performAnotherHevyTaskWithThen();
print('main thread starts....');
print('main thread ends....');
}

void performAnotherHevyTaskWithThen() {
Future<String> result = getWeatherDataUsingAnotherAPI();
result.then((value) {
    print('The content of the file with Future then - $value');
});
}

Future<String> getWeatherDataUsingAnotherAPI() {
Future<String> result = Future.delayed(const Duration(seconds: 8), () {
    return 'Downloading File completed after 8 seconds.';
});
return result;
}

/**
OUTPUT:

main thread starts....
main thread ends....
The content of the file with Future then - Downloading File completed after 8 second\
s.

*
*
*/
```

In the above code, we have used the "Future then" method. But before that we have told Dart to perform a heavy task that takes 8 seconds to finish.

```
Future<String> getWeatherDataUsingAnotherAPI() {
Future<String> result = Future.delayed(const Duration(seconds: 8), () {
    return 'Downloading File completed after 8 seconds.';
});
return result;
}
```

After that, we use the "Future then" method to call the above function.

Watch this part.

```
void performAnotherHevyTaskWithThen() {
Future<String> result = getWeatherDataUsingAnotherAPI();
result.then((value) {
    print('The content of the file with Future then - $value');
});
}
```

Lastly, when we call this function in our main thread it executes after 8 seconds. However, Dart allows the small tasks to perform first.

```
void main() {
performAnotherHevyTaskWithThen();
print('main thread starts....');
print('main thread ends....');
}

/**
OUTPUT::::

main thread starts....
main thread ends....
The content of the file with Future then - Downloading File completed after 8 second\
s.

*
*
*/
```

Before the "Future async and await", we used the "Future then".

But the "Future async and await" makes it more simple.

Let us see the code where we use the "Future async and await" instead of "Future then".

```
void main() {
performAHevyTaskWithAsyncAndAwait();
print('main thread starts....');
print('main thread ends....');
}
void performAHevyTaskWithAsyncAndAwait() async {
String result = await getWeatherDataUsingAnAPI();
print('The content of the file with Future, async and await - $result');
}

Future<String> getWeatherDataUsingAnAPI() {
Future<String> result = Future.delayed(const Duration(seconds: 6), () {
    return 'Downloading File completed after 6 seconds.';
});
return result;
}
/**
OUTPUT:::

main thread starts....
main thread ends....
The content of the file with Future, async and await - Downloading File completed af\
ter 6 seconds.

*
*
*/
```

The difference between Future then and async, await

As such, both type of Future functions program asynchronously. But there is a major difference. In "Future then" method, although we return a String value, yet we cannot explicitly declare it. As a result, we have to declare it as the Future<String> value.

```
void performAnotherHevyTaskWithThen() {
Future<String> result = getWeatherDataUsingAnotherAPI();
result.then((value) {
...
```

But, in case of "Future async, await" we confirm that we are getting a String value.

```
void performAHevyTaskWithAsyncAndAwait() async {
String result = await getWeatherDataUsingAnAPI();
...
```

Certainly that makes our task easier than the "Future then" method.

But in some cases, we still prefer to use the "Future then" method. We have done the same thing in our weather App. Take a look at the following image.

The page displays a static Text output that will change as we press the button.

The Geolocator plugin gets the current latitude and longitude dynamically. After that, based on that value, the user gets the current weather data.

The Text Button onPressed property updates the data using the "Future then" method.

```
TextButton(
            onPressed: () {
                location.getCitynameWithGeolocator().then((result) {
                setState(() {
                    city = result;
                });
                });
                location.getWeatherDescriptionWithGeolocator().then((result) {
                setState(() {
                    description = result;
                });
                });
            },
...
// code is incomplete for brevity
// please clone the project from this branch of GitHub Repository
```

To sum up, the Future object acts as a promise token. When Flutter finds that some heavy task is waiting it promises the user that in Future you will get it. So please wait.

What is the advantage?

Many.

Because the Future object can return any data type. In the above code it returns String data type. But we can pass and return any data type with the help of the Future object.

The code repositories for this branch[92]

[92]https://github.com/sanjibsinha/with_her/tree/fifth-step-future-then

Future in Flutter: WithHer App – Step 6

While we've building the weather app, we have learned a few key concepts in Flutter. Whenever we get data from outside resource, we need to use Future in Flutter.

In the "WithHer" App, which we have been building, we have used Future. In fact, in the first step, we have introduced the concept of Future first.

After that, in the fifth step, we have discussed Future then, async and await in great detail.

To sum up, we define Future as a function in Flutter and Dart. But, instead of void, we use Future. When we return a value from Future, we pass it a Type.

In this section, we will finally show how we have organized our code in a model weather class with Future and return weather data in our view page.

What is Future in Flutter?

We have requested the data from open weather map website using their API.

After that, the website responded and posted data in JSON format.

As a result, in our location model class, we have defined all the properties and methods.

```
import 'dart:convert';
import 'package:geolocator/geolocator.dart';
import 'package:http/http.dart' as http;

/// this is sixth step
///

class UserLocation {
Future<String> getCitynameWithGeolocator() async {
    try {
    Position position = await Geolocator.getCurrentPosition(
        desiredAccuracy: LocationAccuracy.lowest);
    double lat = position.latitude;
    double lon = position.longitude;
    return getCityName(lat, lon, 'Secret');
    } catch (e) {
    return e.toString();
    }
}

```

```
Future<String> getWeatherDescriptionWithGeolocator() async {
    try {
    Position position = await Geolocator.getCurrentPosition(
        desiredAccuracy: LocationAccuracy.lowest);
    double lat = position.latitude;
    double lon = position.longitude;
    return getWeatherDescription(lat, lon, 'Secret');
    } catch (e) {
    return e.toString();
    }
}

Future<String> getCityName(double lat, double lon, String apiKey) async {
    final httpsUri = Uri.http('api.openweathermap.org', '/data/2.5/weather', {
    'lat': '$lat',
    'lon': '$lon',
    'appid': apiKey,
    });

    var request = await http.get(httpsUri);
    if (request.statusCode == 200) {
    String data = request.body.toString();
    var city = jsonDecode(data)['name'];

    return city;
    } else {
    return '${request.statusCode}';
    }
}

Future<String> getWeatherDescription(
    double lat, double lon, String apiKey) async {
    final httpsUri = Uri.http('api.openweathermap.org', '/data/2.5/weather', {
    'lat': '$lat',
    'lon': '$lon',
    'appid': apiKey,
    });

    var request = await http.get(httpsUri);
    if (request.statusCode == 200) {
    String data = request.body.toString();
    var description = jsonDecode(data)['weather'][0]['description'];

```

```
64      return description;
65      } else {
66      return '${request.statusCode}';
67      }
68  }
69
70  Future getWeatherWithGeolocator() async {
71      try {
72      Position position = await Geolocator.getCurrentPosition(
73          desiredAccuracy: LocationAccuracy.lowest);
74      double lat = position.latitude;
75      double lon = position.longitude;
76      var data = getWeatherData(lat, lon, 'Secret');
77      return data;
78      } catch (e) {
79      e;
80      }
81  }
82
83  Future getWeatherData(double lat, double lon, String apiKey) async {
84      final httpsUri = Uri.http('api.openweathermap.org', '/data/2.5/weather', {
85      'lat': '$lat',
86      'lon': '$lon',
87      'appid': apiKey,
88      });
89
90      var request = await http.get(httpsUri);
91      if (request.statusCode == 200) {
92      String data = request.body;
93      var decodedData = jsonDecode(data);
94      return decodedData;
95      } else {
96      '${request.statusCode}';
97      }
98  }
99  }
```

As we see, in one method we have passed the latitude, longitude and API key with the "http" plugin.

Then finally we have called that method in another method that uses live data using the Geolocator plugin.

Consequently, we have got the city name and description in two separate methods.

However, in one method, we have got the whole JSON Map data.

```
Future getWeatherWithGeolocator() async {
    try {
    Position position = await Geolocator.getCurrentPosition(
        desiredAccuracy: LocationAccuracy.lowest);
    double lat = position.latitude;
    double lon = position.longitude;
    var data = getWeatherData(lat, lon, 'Secret');
    return data;
    } catch (e) {
    e;
    }
}

Future getWeatherData(double lat, double lon, String apiKey) async {
    final httpsUri = Uri.http('api.openweathermap.org', '/data/2.5/weather', {
    'lat': '$lat',
    'lon': '$lon',
    'appid': apiKey,
    });

    var request = await http.get(httpsUri);
    if (request.statusCode == 200) {
    String data = request.body;
    var decodedData = jsonDecode(data);
    return decodedData;
    } else {
    '${request.statusCode}';
    }
}
```

Since in this Future method we have not passed any Type, it automatically assumes that the Type would be dynamic.

In fact, we will get the data in JSON Map format.

So far, we have learned that JSON is a structured data that consists a Map. In addition, that Map has many Map or List inside.

Therefore, we can get the value by accessing key as follows.

```
var city = jsonDecode(data)['name'];
```

But in some cases, it could be complex as follows.

```
var description = jsonDecode(data)['weather'][0]['description'];
```

Display data with Future in Flutter

A Future is a promise token that we pass it to the user and says, please wait,you will get the data in Future.

We get the Future data either with the "then" method, or with "async, await".

Let us take a look at the code below that uses both to display current weather data based on user's location.

```
import 'package:flutter/material.dart';

import '../model/location.dart';

/// this is sixth step and last one on Future
///
class MyAppHome extends StatefulWidget {
const MyAppHome({Key? key, required this.title}) : super(key: key);

final String title;

@override
State<MyAppHome> createState() => _MyAppHomeState();
}

class _MyAppHomeState extends State<MyAppHome> {
UserLocation location = UserLocation();
var city = '';
var description = '';

@override
void initState() {
    super.initState();
    var description;
    location.getWeatherWithGeolocator().then((value) {
    description = value;
    print(description);
    });
}

```

```
@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text(widget.title),
    ),
    body: Center(
        child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
            Text(
            city.toString(),
            style: Theme.of(context).textTheme.headline6,
            ),
            Text(
            description.toString(),
            style: Theme.of(context).textTheme.headline4,
            ),
            const SizedBox(
            height: 20.0,
            ),
            TextButton(
            onPressed: () async {
                city = await location.getCitynameWithGeolocator();
                setState(() {
                city = city;
                });
                description =
                    await location.getWeatherDescriptionWithGeolocator();
                setState(() {
                description = description;
                });
            },
            child: const Text(
                'Get City and Current Weather',
                style: TextStyle(
                color: Colors.white,
                fontSize: 30.0,
                fontWeight: FontWeight.bold,
                ),
            ),
            style: ButtonStyle(
                backgroundColor: MaterialStateProperty.all(Colors.red),
```

```
                shape: MaterialStateProperty.all(
                RoundedRectangleBorder(
                    borderRadius: BorderRadius.circular(30.0),
                ),
                ),
            ),
            )
        ],
        ),
    ),
    // This trailing comma makes auto-formatting nicer for build methods.
    );
}
}
```

In the above code, we have first get the whole weather data in JSON Map format through init() method.

In the second step, we discussed the State object.

Therefore, once we run the app, we get the weather data in JSON Map format on our console.

```
{coord: {lon: xx.3163, lat: xx.6066}, weather: [{id: 721, main: Haze, description: h\
aze, icon: 50d}], base: stations,
main: {temp: 301.14, feels_like: 305.69, temp_min: 301.14, temp_max: 301.14, pressur\
e: 1008, humidity: 83}, visibility:
2800, wind: {speed: 4.12, deg: 180}, clouds: {all: 75}, dt: 1648860766, sys: {type: \
1, id: 9114, country: XX, sunrise:
1648857516, sunset: 1648902114}, timezone: 19800, id: 1348747, name: xxxx, cod: 200}
```

As we see in the above code, the weather key has a value of List that again consists of a Map. In that map, we have a description keyword that has a value "haze".

What does that mean?

The Geolocator and the http plugins have fetched the current weather description of the place where the user is now at present.

Therefore, we can get that description key as follows.

```
var description = jsonDecode(data)['weather'][0]['description'];
```

And finally, we have displayed the description in our Text Widget.

```
Text(
            description.toString(),
            style: Theme.of(context).textTheme.headline4,
),
```

What we have requested, we have got in clear text. The open weather map website responded in JSON format which is also clear text.

For that reason, the Future didn't take much time and returned data in a fast manner.

It doesn't happen all the time.

Why?

Because sometimes the API may have to handle a large file, or open an image which is big.

Anyway, we have progressed a lot.

Now we can design our app, take input from the user and show the weather data.

The code repositories for this branch[93]

Pass data to State Flutter: Final Weather App

How do we pass data to State object in Flutter? In other words, how we can pass data to a Stateful Widget? Is it same as we do in the Stateless Widget?

The answer is no. Not exactly the same way we do that.

But in a similar vein, we pass data through class constructor first. And, after that, we use the "widget" property of the State class to access that data.

Firstly, in this section we will learn how we can accomplish that task. Secondly, at the same way, we will also finish our weather App "WithHer" that we have been building for some time.

Therefore, the first thing first.

Let us create a model class location. And keep that file in our model folder.

[93] https://github.com/sanjibsinha/with_her/tree/sixth-step-future-in-flutter

```dart
import 'dart:convert';
import 'package:geolocator/geolocator.dart';
import 'package:http/http.dart' as http;

/// this is seventh step
///

class UserLocation {
Future<dynamic> getWeatherWithGeolocator() async {
    try {
    Position position = await Geolocator.getCurrentPosition(
        desiredAccuracy: LocationAccuracy.lowest);
    double lat = position.latitude;
    double lon = position.longitude;
    var data = getWeatherData(lat, lon, 'Secret Key');
    return data;
    } catch (e) {
    e;
    }
}

Future<dynamic> getWeatherData(double lat, double lon, String apiKey) async {
    final httpsUri = Uri.http('api.openweathermap.org', '/data/2.5/weather', {
    'lat': '$lat',
    'lon': '$lon',
    'appid': apiKey,
    });

    var request = await http.get(httpsUri);
    if (request.statusCode == 200) {
    String data = request.body;
    var decodedData = jsonDecode(data);
    return decodedData;
    } else {
    '${request.statusCode}';
    }
}
}
```

We have used three packages. The Dart convert library will help us to convert the JSON data to a Future dynamic type.

As a result, later we can use this location object to get the current weather data in our App through the API of the open weather map website.

Passing data from a Stateful Widget

First, we need a Stateful Widget to load the current weather Data from the API. Right?

Next, we will pass that dynamic Future data to a loading screen where we will display the current weather of the location where the user is staying at present.

For example, in this weekend, I had come to the Howrah City. Therefore, my weather app should show that result.

However, since I have kept my location accuracy at the lowest point, it might vary. Consequently, the App may display some other place around the Howrah city.

```
desiredAccuracy: LocationAccuracy.lowest);
```

Anyway, let us see the first page where we have created a location object.

```
import 'package:flutter/material.dart';

import '../model/location.dart';
import 'weather_page.dart';

/// this is sixth step and last one on Future
///
class MyAppHome extends StatefulWidget {
const MyAppHome({Key? key, required this.title}) : super(key: key);

final String title;

@override
State<MyAppHome> createState() => _MyAppHomeState();
}

class _MyAppHomeState extends State<MyAppHome> {
UserLocation location = UserLocation();

@override
void initState() {
    super.initState();
    updateWeather();
}

void updateWeather() {
    var weather;
```

```
    location.getWeatherWithGeolocator().then((value) {
    weather = value;
    Navigator.push(context, MaterialPageRoute(builder: (context) {
        return WeatherPage(
        weather: weather,
        title: 'Get City and Weather',
        );
    }));
    });
}

@override
Widget build(BuildContext context) {
    return Scaffold(
    backgroundColor: Colors.grey.shade600,
    );
}
}
```

Very simple logic that we have followed. Firstly, we have created a location object.

```
UserLocation location = UserLocation();
```

Secondly, we have got the necessary weather data as a dynamic Future object. After that we have sent that weather data to the weather page where we will display the result.

```
void updateWeather() {
    var weather;
    location.getWeatherWithGeolocator().then((value) {
    weather = value;
    Navigator.push(context, MaterialPageRoute(builder: (context) {
        return WeatherPage(
        weather: weather,
        title: 'Get City and Weather',
        );
    }));
    });
}
```

Finally, we have called the above method in our "initState()" method.

```
@override
void initState() {
    super.initState();
    updateWeather();
}
```

As the first screen loads it asks for the user permission first.

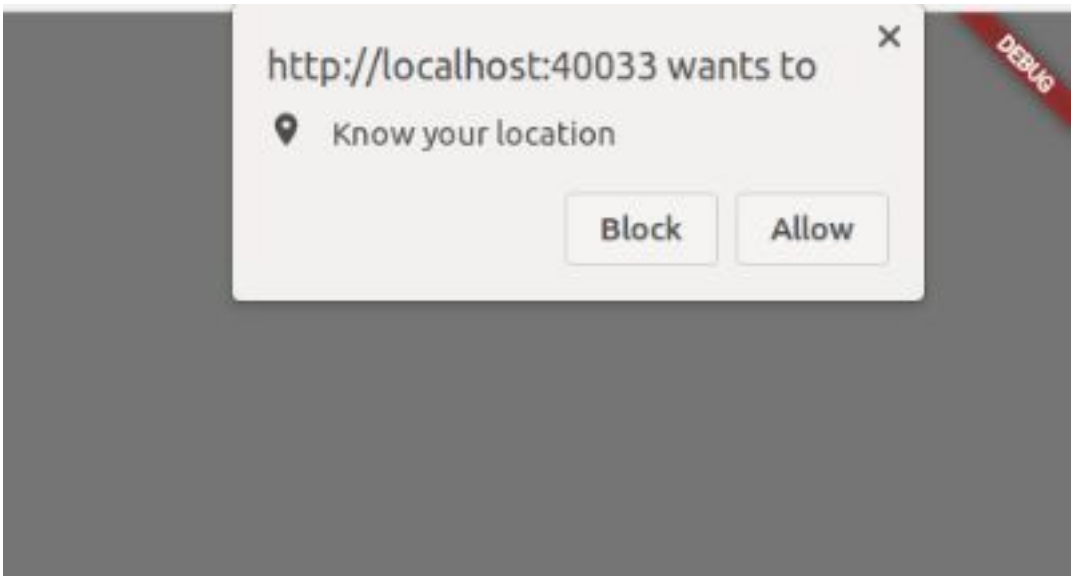

Figure 19.1 – Assking for User's permision to display location on User's screen

To get the current weather, we need to give the permission by clicking the allow button.

As a result, it will pass the weather data as a dynamic Future object.

Pass data to a State object in Flutter

Getting data in a Stateless Widget does not make things tricky. Rather it's simple.

We have received the data in final variables and display them. Right?

But in a Stateful Widget, we need to use the State "widget" property.

Certainly, we first receive that data in final variables just as we do in a Stateless Widget.

But after that, we need to access that data through the "widget" property.

Let us see the code. After that, we will discuss how it works.

```
import 'package:flutter/material.dart';

class WeatherPage extends StatefulWidget {
const WeatherPage({Key? key, required this.weather, required this.title})
    : super(key: key);

final dynamic weather;
final String title;

@override
State<WeatherPage> createState() => _WeatherPageState();
}

class _WeatherPageState extends State<WeatherPage> {
@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text(widget.title),
    ),
    body: Center(
        child: Container(
        decoration: const BoxDecoration(
            image: DecorationImage(
            image: NetworkImage(
                'https://cdn.pixabay.com/photo/2022/02/19/22/48/forest-7023487_960_7\
20.jpg'),
            fit: BoxFit.cover,
            ),
        ),
        constraints: const BoxConstraints.expand(),
        child: SafeArea(
            child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
                Text(
                'Welcome to ${widget.weather['name']}',
                style: Theme.of(context).textTheme.headline3,
                ),
                Text(
                'Current temperature: ${(widget.weather['main']['temp'] - 273.15).to\
StringAsFixed(2)} Celsius',
                style: Theme.of(context).textTheme.headline6,
```

```
                ),
                Text(
                'Current condition: ${widget.weather['weather'][0]['description']}',
                style: Theme.of(context).textTheme.headline4,
                ),
                const SizedBox(
                height: 20.0,
                ),
            ],
            ),
        ),
        // This trailing comma makes auto-formatting nicer for build methods.
        ),
    ),
    );
}
}
```

As we can see we have received two data from the parent widget. One is the "weather" and the other is the "title". As a result, we have displayed the title as follows.

```
appBar: AppBar(
        title: Text(widget.title),
    ),
```

Here the "widget" property of the State class can access the properties of the Stateful Widget in this way.

For example, we have displayed the city name as follows.

```
Text(
                'Welcome to ${widget.weather['name']}',
                style: Theme.of(context).textTheme.headline3,
    ),
```

Now we can display the current weather data of the user.

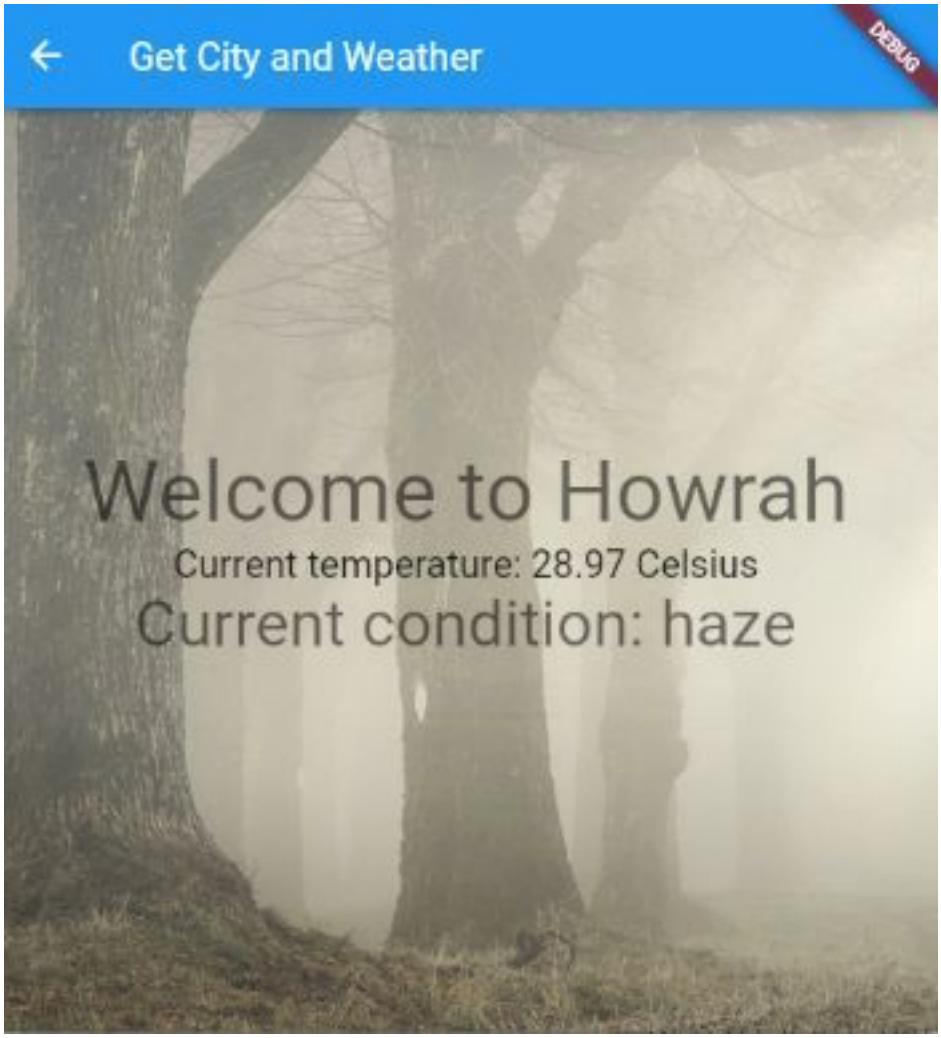

Figure 19.2 – Current Weather displayed on User's screen

Remember, in our previous steps we have shown that the weather data is actually a JSON Map as follows.

```
{coord: {lon: 88.3163, lat: 22.6066}, weather: [{id: 721, main: Haze, description: h\
aze, icon: 50d}], base: stations,
main: {temp: 301.14, feels_like: 305.69, temp_min: 301.14, temp_max: 301.14, pressur\
e: 1008, humidity: 83}, visibility:
2800, wind: {speed: 4.12, deg: 180}, clouds: {all: 75}, dt: 1648860766, sys: {type: \
1, id: 9114, country: IN, sunrise:
1648857516, sunset: 1648902114}, timezone: 19800, id: 1348747, name: Howrah, cod: 20\
0}
```

Here the "name" acts as a key. So we have accessed the value of the city name by using the key.

If you want to clone the whole project and test in your machine, please clone the GitHub repository.

But you need to create and pass your own API key as app ID.

The code repositories for the final branch[94]

[94] https://github.com/sanjibsinha/with_her

20. Building Two Flutter Chat Apps with Firebase, and Firestore - First app with Stateful Widget, Second App with Provider

In this chapter we're going to learn how to use Firebase authentication and Firestore database by building two Flutter Chat apps.

The first Flutter Chat App we will build with Stateful Widget.

The second Flutter Chat App will be build using Provider package, and pattern.

However, in both cases, we have used the Firebase authentication and Firestore database. At the same time we will learn how to maintain the Firebase authentication and Firestore database rules.

Chat App with Stateful widget – Step One

With Flutter Firebase we can build a Chat App. In this section, we will learn the rules and see how a user can register and login by email and password.

For example, to register and login process we use the Firebase authentication.

After that, we will use the Firestore database to save our Chat messages that user send to each other.

By the way, in previous sections, we have been building a News App where Flutter acts as the frontend. But the WordPress acts as the backend.

Although we have not finished that app yet in between we can take a break to build a small Chat app with Flutter Firebase.

Why we use the Firebase?

Because just like the WordPress we can also use Firebase as the backend to Flutter.

Before discussing the code and other rules of Firebase, let us see how we can authenticate users through the Flutter App.

Figure 20.1 – Flutter Firebase welcome page

This is our welcome page where the new user can register. And the existing user can login to join

the Chat page.

The registration page looks as follows.

Figure 20.2 – Flutter Firebase registration page

We see that in the registration page a user is registering with email and password.

Once the user gets registered, the data of the user will be available in the Firebase authentication page.

Let's see the screenshot. That will explain.

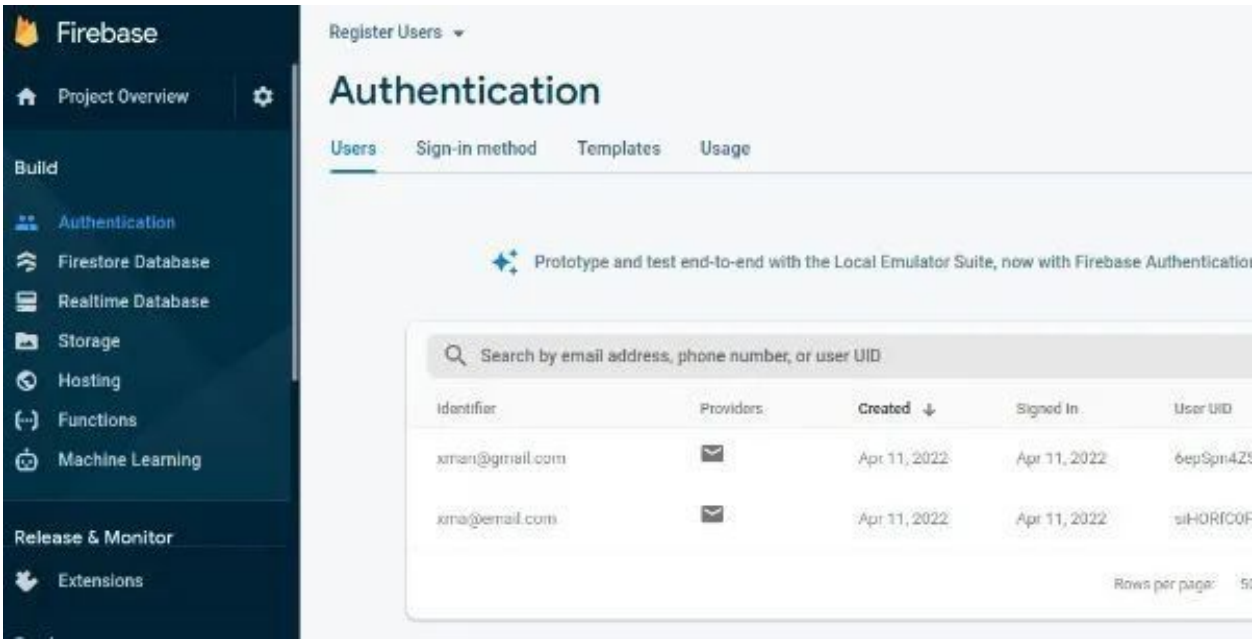

Figure 20.3 – Firebase authentication page shows list of users.

Just like the registration page, the login page also looks the same.

For instance, the login page takes inputs from the users. And as a result, the registered-user can type in the email and password.

The Firebase authentication instantly checks the credential and sends the user to the Chat page.

Let's see how the login page looks like.

Figure 20.4 – Flutter Firebase login page

Is Flutter and Firebase full stack?

Yes, Flutter and Firebase can work together to build a full stack App. In fact, the Chat App we are going to create will be an example of full stack app.

However, we must remember that Flutter is a frontend UI toolkit that wants a backend support to work with any database.

In that scenario, Firebase acts as the backend database.

In case of the News App, we have seen how WordPress gives the backend database support.

The same thing happens in case of Firebase.

Is Firebase easy to learn?

Certainly, for a beginner, using the Firebase as a backend to the Flutter may look tough.

But at one time, we all have started as a beginner. Right?

Therefore, just follow the instructions and code along with these steps. You will find that neither Firebase, nor the Firestore looks tough anymore.

So, first thing first.

We need to login to the Firebase home page with our Gmail account. After that we need to click the "Go to Console" tab.

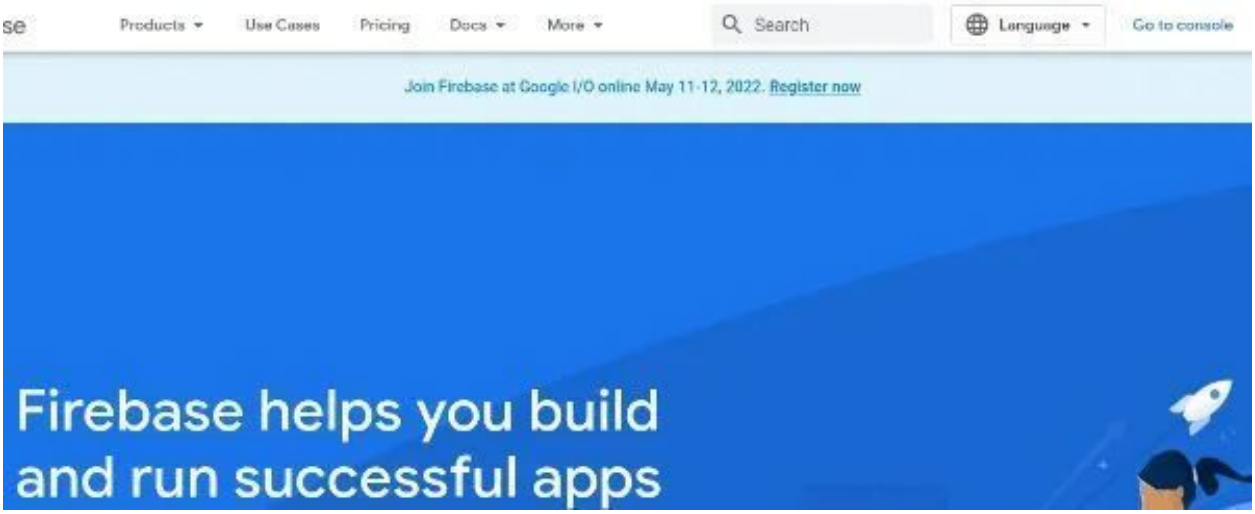

Figure 20.5 – Firebase *Go to Console*

You will find it on the top right hand corner of the page.

Once you click the tab, it will take you to the "Add a Project" page.

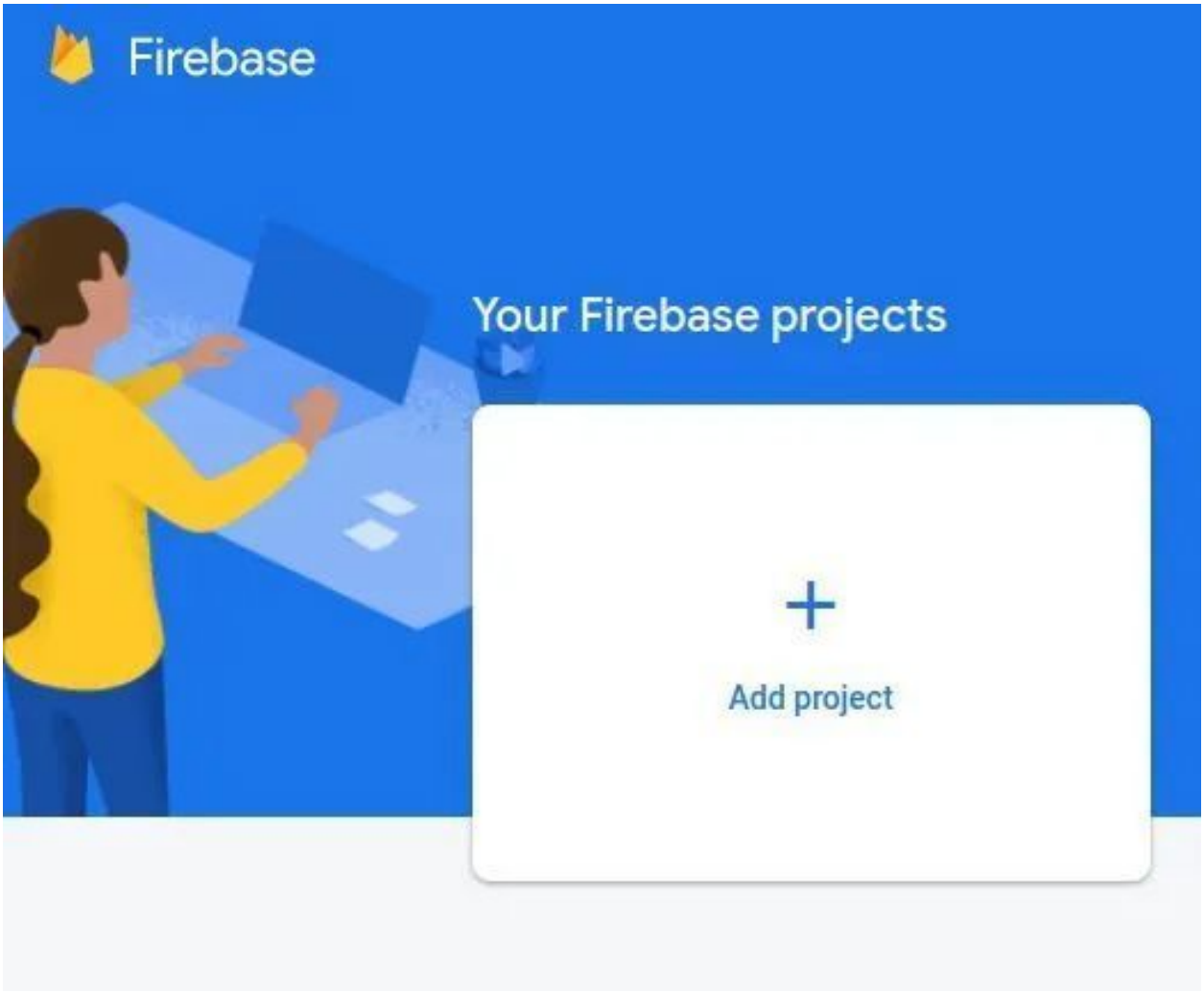

Figure 20.6 – Firebase Add Project

The Firebase Add project will guide you to do the rest of the things.

It will ask you to download a Google services JSON file. After downloading the file, we need to place it inside the project's "android/app" folder.

Meanwhile we should also add a few lines in our "android/app/build.gradle" file. Besides, in our "android/build.gradle" file we need to add a line.

From where we will get those lines?

Firebase instructs us to copy the line and place them where they are needed.

We also need to add a few necessary packages to our "pubspec.yaml" file dependency section.

```
dependencies:
flutter:
    sdk: flutter

cupertino_icons: ^1.0.2
firebase_core: ^1.14.0
firebase_auth: ^3.3.13
cloud_firestore: ^3.1.11
```

We need to add the firebase_core, firebase_auth, and the cloud_firestore packages.

As we progress, we will see how these packages work together when we build the Chat app.

So stay tuned and we will meet again in the next section.

How to Initialise Chat App and Avoid Errors

In the previous section we have learned that Flutter and Firebase can work together. In fact, Flutter works as the frontend. And Firebase acts as backend.

As a result, together they work as a full stack app.

However, before using the Firebase, we need to know a few basic rules.

For a small Flutter app, Firebase is free. In that case, we should limit ourselves within 100 connections.

For example, more than 100 users cannot use the Chat app that we've been building.

But for a big app where many users can participate, we can pay and increase our storage capacity.

What is Firebase in Flutter?

Beginners want to know the answer. So we need to clarify.

When we clarify, it makes sense. Right?

Firebase gives different types of backend services to Flutter.

What does the backend service mean? In fact, it means many things.

```
real time database
cloud storage
authentication
crash reporting
machine learning
and many more…
```

Since Flutter is a frontend UI toolkit, we need a backend service.

In the previous sections we have seen how we have built a News App with WordPress as the backend.

Not only that, we have built a personal Blog App with SQLite database as our backend.

Therefore, we can do many things and use Firebase in the similar vein.

Initialise App and avoid errors

Firebase has changed its rules and developed gradually. Therefore, please read the documentation to get the updated news.

One of them is we need to initialise Firebase at the entry point of our Flutter app.

As an outcome, our top level main() function looks as follows.

```
void main() async {
WidgetsFlutterBinding.ensureInitialized();
await Firebase.initializeApp();
runApp(const FirebaseLoginRegister());
}
```

But we need to import the “firebase_core” package which we’ve added as a dependency.

```
import 'package:firebase_core/firebase_core.dart';
```

We also need some other packages as well to run the Chat App. We have discussed that topic in our previous section.

```
dependencies:
flutter:
    sdk: flutter

cupertino_icons: ^1.0.2
firebase_core: ^1.14.0
firebase_auth: ^3.3.13
cloud_firestore: ^3.1.11
```

Not only that, we also have to add another Firebase method before we initialise the State of the Widget.

```
void initState() {
    super.initState();
    Firebase.initializeApp().whenComplete(() {
    setState(() {});
    });
}
```

We'll discuss this topic in detail. With that we'll also discuss other important concepts.

Flutter and Firebase can act together. But we should know the rules to use them as the full stack.

So stay tuned. We'll meet in the next section.

Firebase Chat App authentication

In the previous sections we have learned how we can avoid errors while we use Firebase and Flutter.

Moreover, we have learned how to customise the button.

Now we'll finish the authentication section pf our Chat App. To do that we need Firebase and Flutter. Most importantly, they should work together.

To sum up, the Firebase gives different types of backend services to Flutter.

Since Flutter is a frontend UI toolkit, therefore we need a backend service.

Certainly we can use other backend service. As we have built a News App with WordPress as the backend.

Let us see the top-level main function first.

```dart
import 'package:flutter/material.dart';
import 'package:firebase_core/firebase_core.dart';

import 'view/chat.dart';
import 'view/login.dart';
import 'view/register.dart';
import 'view/welcome.dart';

void main() async {
WidgetsFlutterBinding.ensureInitialized();
await Firebase.initializeApp();
runApp(const FirebaseLoginRegister());
}

class FirebaseLoginRegister extends StatelessWidget {
const FirebaseLoginRegister({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    debugShowCheckedModeBanner: false,
    theme: ThemeData.dark().copyWith(
        textTheme: const TextTheme(
        bodyText1: TextStyle(color: Colors.black54),
        ),
    ),
    initialRoute: Welcome.id,
    routes: {
        Welcome.id: (context) => const Welcome(),
        Login.id: (context) => const Login(),
        Register.id: (context) => const Register(),
        Chat.id: (context) => const Chat(),
    },
    );
}
}
```

Firstly, we need to initialise the Firebase service. After that, we have used the"routes" property of the Material App Widget to define various routes.

We need a welcome page where the new user either register or the existing user will login so that they can chat.

How Firebase authentication works

In Firebase, there are several ways to authenticate users. The most popular one is, of course, the email authentication.

First we need to enable that service.

After that, in the welcome page we use two buttons so that users can either register, or login.

```
import 'package:flutter/material.dart';
import 'package:firebase_core/firebase_core.dart';
import '../model/rounded_button.dart';
import 'register.dart';

import 'login.dart';

class Welcome extends StatefulWidget {
const Welcome({Key? key}) : super(key: key);
static const String id = 'welcome';

@override
_WelcomeState createState() => _WelcomeState();
}

class _WelcomeState extends State<Welcome> {
@override
void initState() {
    super.initState();
    Firebase.initializeApp().whenComplete(() {
    setState(() {});
    });
}

@override
Widget build(BuildContext context) {
    return Scaffold(
    backgroundColor: Colors.redAccent,
    body: Padding(
        padding: const EdgeInsets.symmetric(horizontal: 24.0),
        child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        crossAxisAlignment: CrossAxisAlignment.stretch,
        children: <Widget>[
```

```
            Row(
            children: <Widget>[
                Expanded(
                child: Hero(
                    tag: 'logo',
                    child: Container(
                    padding: const EdgeInsets.all(8.0),
                    child: Image.network(
                        'https://cdn.pixabay.com/photo/2015/05/19/07/44/browser-7732\
15_960_720.png'),
                    height: 160.0,
                    width: 160.0,
                    ),
                ),
                ),
                const Expanded(
                child: Text(
                    'Chatting Friends: Register or Login',
                    style: TextStyle(
                    fontSize: 45.0,
                    color: Colors.black,
                    fontWeight: FontWeight.w900,
                    ),
                ),
                ),
            ],
            ),
            const SizedBox(
            height: 18.0,
            ),
            RoundedButton(
            title: 'Log In',
            colour: Colors.black45,
            onPressed: () {
                Navigator.pushNamed(context, Login.id);
            },
            ),
            RoundedButton(
            title: 'Register',
            colour: Colors.black45,
            onPressed: () {
                Navigator.pushNamed(context, Register.id);
            },
```

```
78              ),
79          ],
80          ),
81      ),
82      );
83  }
84  }
```

When Flutter initialises the State, we also initialise the Firebase service.

Certainly inside the setState() method, we can do some more tasks.

```
1  class _WelcomeState extends State<Welcome> {
2  @override
3  void initState() {
4      super.initState();
5      Firebase.initializeApp().whenComplete(() {
6      setState(() {});
7      });
8  }
9  ...
```

The welcome page looks as follows.

Figure 20.7 – Flutter and Firebase Chatting App welcome page

In the register page we need two Firebase packages that we have already added in our “pubspec.yaml”

file.

```
dependencies:
flutter:
    sdk: flutter

cupertino_icons: ^1.0.2
firebase_core: ^1.14.0
firebase_auth: ^3.3.13
cloud_firestore: ^3.1.11
```

To clarify, the “firebase_auth” and the “firebase_core” packages help us to create the Firebase authentication instance.

final _auth = FirebaseAuth.instance; As a result, we can use that instance to create users email and password.

```
final user = await _auth.createUserWithEmailAndPassword(
                    email: email!,
                    password: password!,
                );
```

If we take a look at the register page, that will clarify how we use Firebase and Flutter.

It’s simple.

Above all, the Firebase packages take care of the authentication which we want. In addition, later the Flutter app authenticate users based on that email and password.

Let’s see the code of register page.

```
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:firebase_core/firebase_core.dart';

import 'chat.dart';

import '../model/rounded_button.dart';

class Register extends StatefulWidget {
const Register({Key? key}) : super(key: key);
static const String id = 'register';

@override
```

```
14  _RegisterState createState() => _RegisterState();
15  }
16
17  class _RegisterState extends State<Register> {
18  final _auth = FirebaseAuth.instance;
19
20  String? email;
21  String? password;
22
23  User? loggedInUser;
24
25  @override
26  void initState() {
27      super.initState();
28      Firebase.initializeApp().whenComplete(() {
29      setState(() {});
30      });
31      getUser();
32  }
33
34  void getUser() {
35      try {
36      final user = _auth.currentUser;
37      if (user != null) {
38          loggedInUser = user;
39      }
40      } catch (e) {
41      throw e.toString();
42      }
43  }
44
45  @override
46  Widget build(BuildContext context) {
47      return Scaffold(
48      backgroundColor: Colors.redAccent,
49      body: Padding(
50          padding: const EdgeInsets.symmetric(horizontal: 24.0),
51          child: Column(
52          mainAxisAlignment: MainAxisAlignment.center,
53          crossAxisAlignment: CrossAxisAlignment.stretch,
54          children: <Widget>[
55              Flexible(
56              child: Hero(
```

```
                tag: 'logo',
                child: Container(
                padding: const EdgeInsets.all(8.0),
                child: Image.network(
                    'https://cdn.pixabay.com/photo/2015/05/19/07/44/browser-773215_9\
60_720.png'),
                height: 160.0,
                width: 160.0,
                ),
            ),
            ),
            const SizedBox(
            height: 18.0,
            ),
            TextField(
            decoration: const InputDecoration(
                border: OutlineInputBorder(),
                hintText: 'Enter Email',
            ),
            keyboardType: TextInputType.emailAddress,
            textAlign: TextAlign.center,
            onChanged: (value) {
                email = value;
            },
            style: const TextStyle(
                fontSize: 35.0,
                fontWeight: FontWeight.bold,
                color: Colors.white,
            ),
            ),
            const SizedBox(
            height: 8.0,
            ),
            TextField(
            decoration: const InputDecoration(
                border: OutlineInputBorder(),
                hintText: 'Enter Password',
            ),
            obscureText: true,
            textAlign: TextAlign.center,
            onChanged: (value) {
                password = value;
            },
```

```
100             style: const TextStyle(
101                 fontSize: 35.0,
102                 fontWeight: FontWeight.bold,
103                 color: Colors.white,
104             ),
105             ),
106             const SizedBox(
107             height: 24.0,
108             ),
109             RoundedButton(
110             title: 'Register',
111             colour: Colors.black45,
112             onPressed: () async {
113                 setState(() {});
114                 try {
115                 final user = await _auth.createUserWithEmailAndPassword(
116                     email: email!,
117                     password: password!,
118                 );
119                 if (user != null) {
120                     Navigator.pushNamed(context, Chat.id);
121                 }
122 
123                 setState(() {});
124                 } catch (e) {
125                 throw e.toString();
126                 }
127             },
128             ),
129         ],
130         ),
131     ),
132     );
133 }
134 }
```

In the above code, the highlighted sections play the important role.

Is Firebase a backend or database? As we have said earlier, Firebase is a backend service that supports many things. And, of course, NoSQL database is one of them.

Moreover, it keeps data in JSON format, and that makes it faster than other database.

When the user registers we can see the data in our Firebase console.

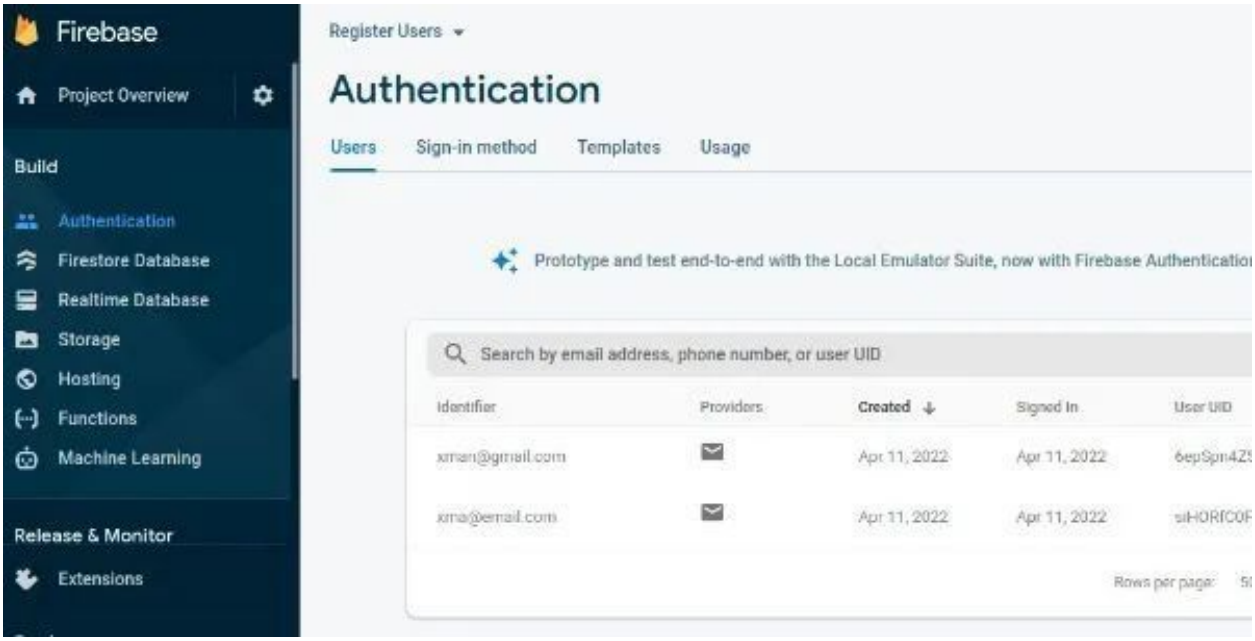

Figure 20.8 – Flutter Firebase users have been added

As an outcome now a registered user can login and take part in chatting with other registered users. The code of login page is almost identical as the register page.

```
import 'package:flutter/material.dart';
import 'package:firebase_core/firebase_core.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'chat.dart';

import '../model/rounded_button.dart';

class Login extends StatefulWidget {
const Login({Key? key}) : super(key: key);
static const String id = 'login';

@override
_LoginState createState() => _LoginState();
}

class _LoginState extends State<Login> {
final _auth = FirebaseAuth.instance;
String? email;
String? password;

@override
```

```
22  void initState() {
23      super.initState();
24      Firebase.initializeApp().whenComplete(() {
25      setState(() {});
26      });
27  }
28
29  @override
30  Widget build(BuildContext context) {
31      return Scaffold(
32      backgroundColor: Colors.redAccent,
33      body: Padding(
34          padding: const EdgeInsets.symmetric(horizontal: 24.0),
35          child: Column(
36          mainAxisAlignment: MainAxisAlignment.center,
37          crossAxisAlignment: CrossAxisAlignment.stretch,
38          children: <Widget>[
39              Flexible(
40              child: Hero(
41                  tag: 'logo',
42                  child: Container(
43                  padding: const EdgeInsets.all(8.0),
44                  child: Image.network(
45                      'https://cdn.pixabay.com/photo/2015/05/19/07/44/browser-773215_9\
46  60_720.png'),
47                  height: 160.0,
48                  width: 160.0,
49                  ),
50              ),
51              ),
52              const SizedBox(
53              height: 48.0,
54              ),
55              TextField(
56              decoration: const InputDecoration(
57                  border: OutlineInputBorder(),
58                  hintText: 'Enter Email',
59              ),
60              keyboardType: TextInputType.emailAddress,
61              textAlign: TextAlign.center,
62              onChanged: (value) {
63                  email = value;
64              },
```

```
65              style: const TextStyle(
66                  color: Colors.white,
67                  fontSize: 30.0,
68                  fontWeight: FontWeight.bold,
69              ),
70              ),
71              const SizedBox(
72              height: 8.0,
73              ),
74              TextField(
75              decoration: const InputDecoration(
76                  border: OutlineInputBorder(),
77                  hintText: 'Enter Password',
78              ),
79              obscureText: true,
80              textAlign: TextAlign.center,
81              onChanged: (value) {
82                  password = value;
83              },
84              style: const TextStyle(
85                  color: Colors.white,
86                  fontSize: 30.0,
87                  fontWeight: FontWeight.bold,
88              ),
89              ),
90              const SizedBox(
91              height: 24.0,
92              ),
93              RoundedButton(
94              title: 'Log In',
95              colour: Colors.black45,
96              onPressed: () async {
97                  setState(() {});
98                  try {
99                  final user = await _auth.signInWithEmailAndPassword(
100                     email: email!,
101                     password: password!,
102                 );
103                 if (user != null) {
104                     Navigator.pushNamed(
105                     context,
106                     Chat.id,
107                     );
```

```
                }

                setState(() {});
                } catch (e) {
                throw e.toString();
                }
            },
            ),
        ],
        ),
    ),
    );
}
}
```

Flutter checks whether the user's email and password match with the Firebase authentication service.

If it matches, it takes the user to the Chat page.

Figure 20.9 – Flutter and Firebase Chatting App login page

For the Chat page we will use the Firestore database. In the next section, we will discuss that and

see how Firestore and Flutter work together.

So stay tuned, and we will meet in the next section.

Stateful Chat App Final Step

What is the relation between Flutter and Firestore? In this final section we'll discuss that.

Basically, when we use Firebase with Flutter, we use the database of Firestore as a service.

So Firebase and Firestore give us a few different type of backend services.

We need to understand it first.

Therefore, our target will be to connect our Flutter Chat app with the Firestore database. Right?

Why?

Because, authenticated users can can sign in and chat with other registered users.

Meanwhile we discuss this topic, we will also finish our chat app that we have been building for a while.

The previous section let's us know how to use authentication in Firebase.

Besides, we have learned how we can avoid errors while we use Firebase and Flutter. Moreover, we have also learned how to customise the button to give our Chat app a unique look.

As we have seen earlier, we need Firebase and Flutter.

Most importantly, they should work together.

Certainly we can use other backend service for Flutter. As we have built a News App with WordPress as the backend.

How does Firestore work in Flutter?

Firstly, we need a Firebase membership that we can get by signing in with the Gmail account.

Secondly, we should enable Email as our authentication service.

Finally, we can create a collection in Firestore. On the dashboard we can click the database option and create a collection.

It looks as follows.

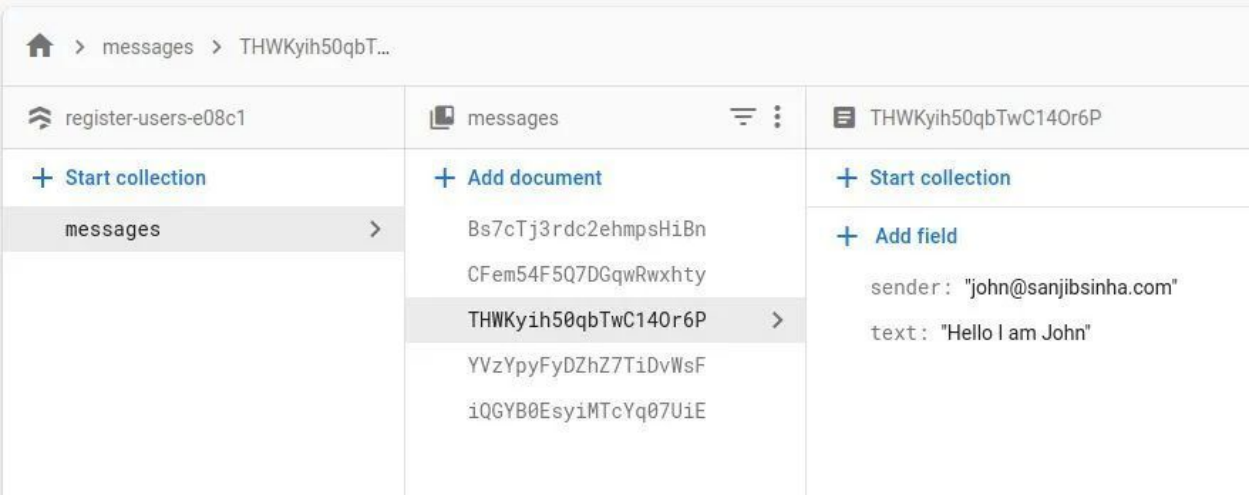

Figure 20.10 – Flutter Firestore collection

As we see the above collection requires only two fields. Both are texts. The first one represents the sender's name.

Who is the sender?

Any authenticated user who have either registered or signed in.

As a result, in our Chat page we see all the messages. Besides we can show the registered users on the Firebase page.

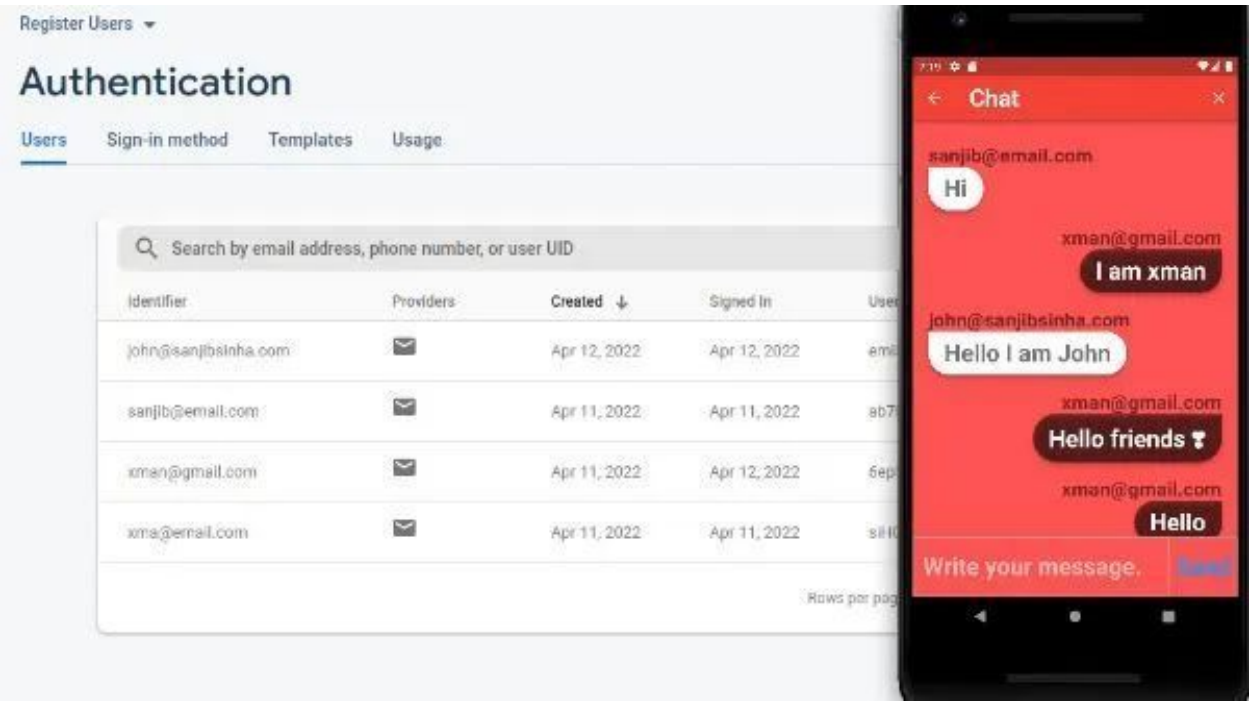

Figure 20.11 – Flutter Firestore Chat Page shows messages

Now our Chat App is perfectly working. As an outcome, new users can register, or the registered users can sign in.

After that, they can start chatting.

The chatting does not start automatically. To chat, we need to initialise the Firebase instance first.

```
@override
void initState() {
    super.initState();
    Firebase.initializeApp().whenComplete(() {
    setState(() {});
    });
    getCurrentUser();
}
```

After that, we can call the current user who can chat. How do we get the current user?

To get the current user we use the Firebase authentication property as follows.

```
final _auth = FirebaseAuth.instance;
```

As a result, we can define logged in user as the current user.

How do we get the logged in user?

It's simple.

```
final _firestore = FirebaseFirestore.instance;
User? loggedInUser;
```

With the logged in user we declare the Firestore instance as global value.

What is the advantage?

For example, in the later part we can add messages to the Firestore database collection.

```
onPressed: () {
                    messageTextController.clear();
                    _firestore.collection('messages').add({
                        'text': messageText,
                        'sender': loggedInUser!.email,
                    });
                    },
```

Consequently, when the user press the button, her message and email get added to the collection.

How do you fetch data from Firestore database in Flutter?

We can create two separate Widgets to fetch data from Firestore database.

Firstly, we need a Widget that will display the data from the Firestore database. But it cannot display data if we don't have a Stream Builder widget as follows.

```
class MessageStreamBuilder extends StatelessWidget {
const MessageStreamBuilder({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return StreamBuilder<QuerySnapshot>(
    stream: _firestore.collection('messages').snapshots(),
    builder: (context, snapshot) {
        if (!snapshot.hasData) {
        return const Center(
            child: CircularProgressIndicator(
            backgroundColor: Colors.lightBlueAccent,
            ),
        );
        }
        final messages = snapshot.data!.docChanges.reversed;
        List<DisplayMessages> displayMessages = [];
        for (var message in messages) {
        final messageText = message.doc['text'];
        final messageSender = message.doc['sender'];

        final currentUser = loggedInUser!.email;

        final displayMessage = DisplayMessages(
            sender: messageSender,
            text: messageText,
            isMe: currentUser == messageSender,
        );

        displayMessages.add(displayMessage);
        }
        return Expanded(
        child: ListView(
            reverse: true,
```

```
            padding:
                const EdgeInsets.symmetric(horizontal: 10.0, vertical: 20.0),
            children: displayMessages,
        ),
        );
    },
    );
}
}
```

Only after that, we can change the style of the Container Widget that will display the data from Firestore.

Therefore, we can also define the widget which will display the message on the screen.

```
class DisplayMessages extends StatelessWidget {
const DisplayMessages({
    Key? key,
    required this.sender,
    required this.text,
    required this.isMe,
}) : super(key: key);

final String sender;
final String text;
final bool isMe;

@override
Widget build(BuildContext context) {
    return Padding(
    padding: const EdgeInsets.all(10.0),
    child: Column(
        crossAxisAlignment:
            isMe ? CrossAxisAlignment.end : CrossAxisAlignment.start,
        children: <Widget>[
        Text(
            sender,
            style: const TextStyle(
            fontSize: 25.0,
            color: Colors.black54,
            fontWeight: FontWeight.bold,
            ),
        ),
```

```
        Material(
            borderRadius: isMe
                ? const BorderRadius.only(
                    topLeft: Radius.circular(30.0),
                    bottomLeft: Radius.circular(30.0),
                    bottomRight: Radius.circular(30.0))
                : const BorderRadius.only(
                    bottomLeft: Radius.circular(30.0),
                    bottomRight: Radius.circular(30.0),
                    topRight: Radius.circular(30.0),
                ),
            elevation: 5.0,
            color: isMe ? Colors.black54 : Colors.white,
            child: Padding(
            padding:
                const EdgeInsets.symmetric(vertical: 10.0, horizontal: 20.0),
            child: Text(
                text,
                style: TextStyle(
                color: isMe ? Colors.white : Colors.black54,
                fontSize: 30.0,
                fontWeight: FontWeight.bold,
                ),
            ),
            ),
        ),
        ],
    ),
    );
}
}
```

Once we have defined these two widgets which will display data, we can call them in the Chat widget as follows.

```
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_core/firebase_core.dart';

final _firestore = FirebaseFirestore.instance;
User? loggedInUser;

class Chat extends StatefulWidget {
static const String id = 'chat';

const Chat({Key? key}) : super(key: key);
@override
_ChatState createState() => _ChatState();
}

class _ChatState extends State<Chat> {
final messageTextController = TextEditingController();
final _auth = FirebaseAuth.instance;

String? messageText;

@override
void initState() {
    super.initState();
    Firebase.initializeApp().whenComplete(() {
    setState(() {});
    });
    getCurrentUser();
}

void getCurrentUser() {
    try {
    final user = _auth.currentUser;
    if (user != null) {
        loggedInUser = user;
    }
    } catch (e) {
    throw e.toString();
    }
}

@override
```

```
Widget build(BuildContext context) {
    return Scaffold(
    backgroundColor: Colors.redAccent,
    appBar: AppBar(
        leading: null,
        actions: <Widget>[
        IconButton(
            icon: const Icon(Icons.close),
            onPressed: () {
                _auth.signOut();
                Navigator.pop(context);
            }),
        ],
        title: const Text(
        'Chat',
        style: TextStyle(
            color: Colors.white,
            fontSize: 30.0,
            fontWeight: FontWeight.bold,
        ),
        ),
        backgroundColor: Colors.red,
    ),
    body: SafeArea(
        child: Column(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        crossAxisAlignment: CrossAxisAlignment.stretch,
        children: <Widget>[
            const MessageStreamBuilder(),
            Container(
            width: double.infinity,
            decoration: BoxDecoration(
                border: Border.all(
                color: Colors.grey,
                ),
            ),
            child: Row(
                crossAxisAlignment: CrossAxisAlignment.center,
                children: <Widget>[
                Expanded(
                    child: TextField(
                    controller: messageTextController,
                    onChanged: (value) {
```

```
                        messageText = value;
                    },
                    decoration: const InputDecoration(
                        border: OutlineInputBorder(),
                        hintText: 'Write your message.',
                    ),
                    style: const TextStyle(
                        fontSize: 30.0,
                        fontWeight: FontWeight.bold,
                    ),
                    ),
                ),
                TextButton(
                    onPressed: () {
                    messageTextController.clear();
                    _firestore.collection('messages').add({
                        'text': messageText,
                        'sender': loggedInUser!.email,
                    });
                    },
                    child: const Text(
                    'Send',
                    style: TextStyle(
                        fontSize: 30.0,
                        fontWeight: FontWeight.bold,
                    ),
                    ),
                ),
                ],
            ),
            ),
        ],
        ),
    ),
    );
}
}
```

Now users can write and send messages as well as they can view their messages.

We have highlighted some sections in the above code to understand the work flow. If you want to clone this repository and run in your local machine, please use this GitHub repository.

But we can certainly make this Chat app much better and faster with the help of the Provider

package.

In a series of sections we will try to build a better Chat app with Flutter Firebase, Firestore and Provider.

The code repositories for the Stateful Flutter Chat App[95]

Flutter Chat app with Provider, Firebase and Firestore

We have built a Flutter Chat app in our previous sections. That was a step by step guide. A small app although. Now we're going to build a chat app with Provider. Just like before we're also going to use Firebase.

However this app will not be like the previous one.

Why?

Because the previous chat app manages state with Stateful widgets. And this chat app will manage state with the provider package.

As a result, the application logic and work flow will not work like before.

In addition, the design will not remain same. Most importantly, the provider package helps us to make this chat app work much faster.

As an outcome the app will rebuild less widgets than before.

In this section, we will not discuss the code but will see how our new chat app looks like. Moreover, how it works.

There will be a welcome page.

[95] https://github.com/sanjibsinha/register_users

Figure 20.12 – Flutter Chat app first page

Do you want to chat with your friends? You must either register or sign in. If you have already

registered, you will sign in.

If you are new, you'll register.

Once you type the email, the chat app will detect whether you are new or registered.

How does that happen? Well, we will discuss that part and many more as we proceed.

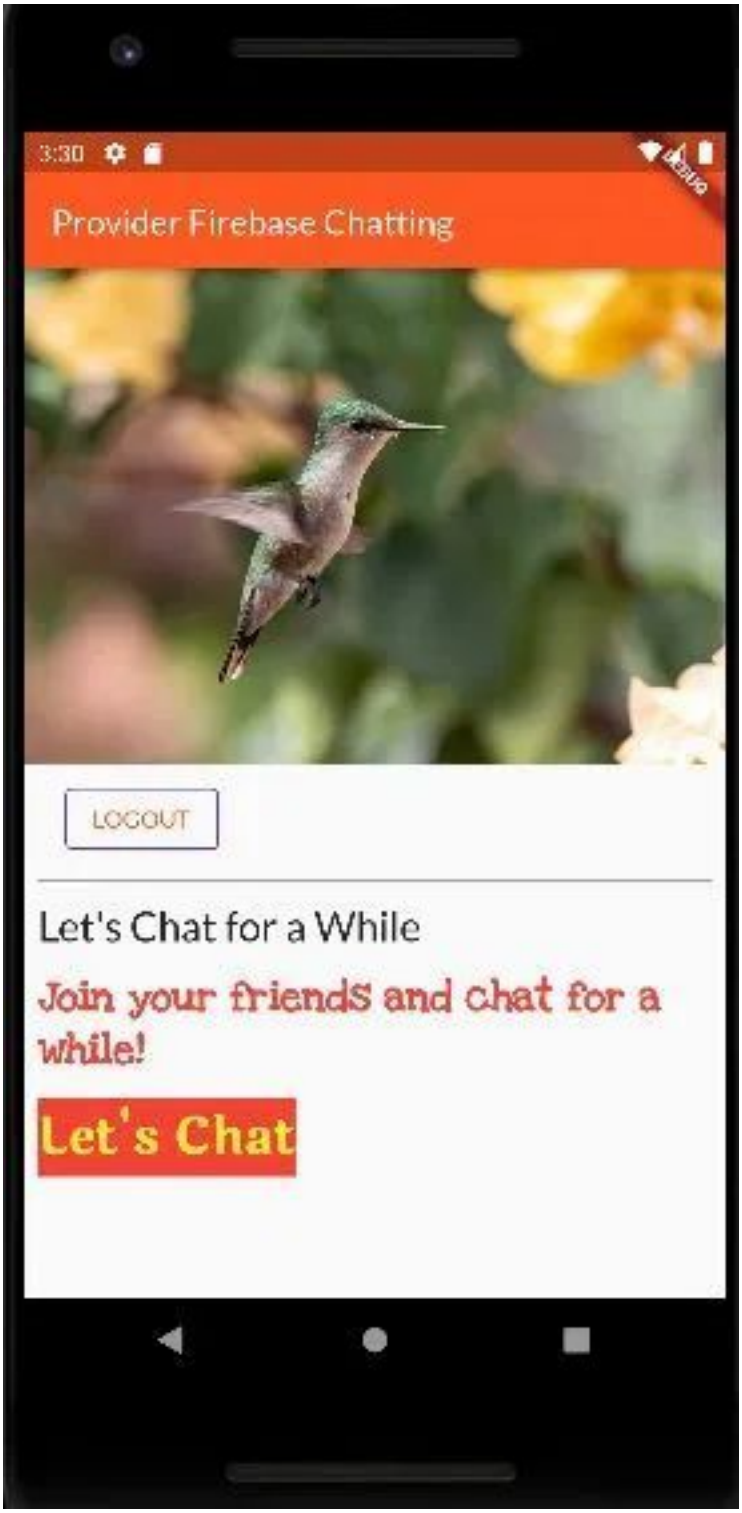

Figure 20.13 – Flutter Chat App new look

When you have signed in, the chat app will give you two options. Either you can start chatting. For that, the Button “Let’s Chat” will help.

If you change your decision? No problem, you can logout. The above image shows everything.

How to Design the Chat app?

In the first go, we have designed the “Let’s Chat” button in a different way.

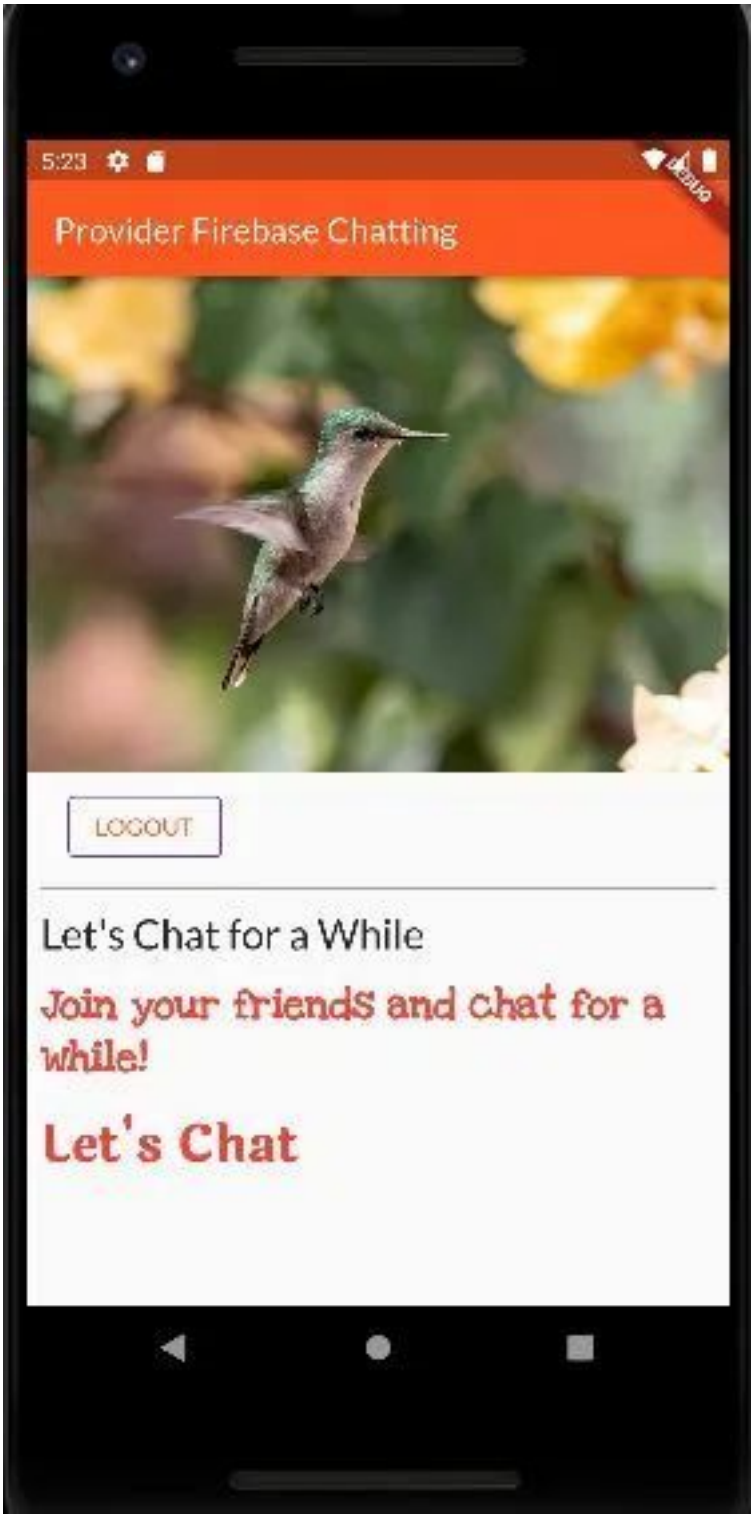

Figure 20.14 – Flutter Chat app member's page

As a result, when you have signed in, you see the Text Button in red color. Later we have added yellow as the background color. You can modify your design.

Certainly you can choose your color. We love pluralism.

How does the registration take place?

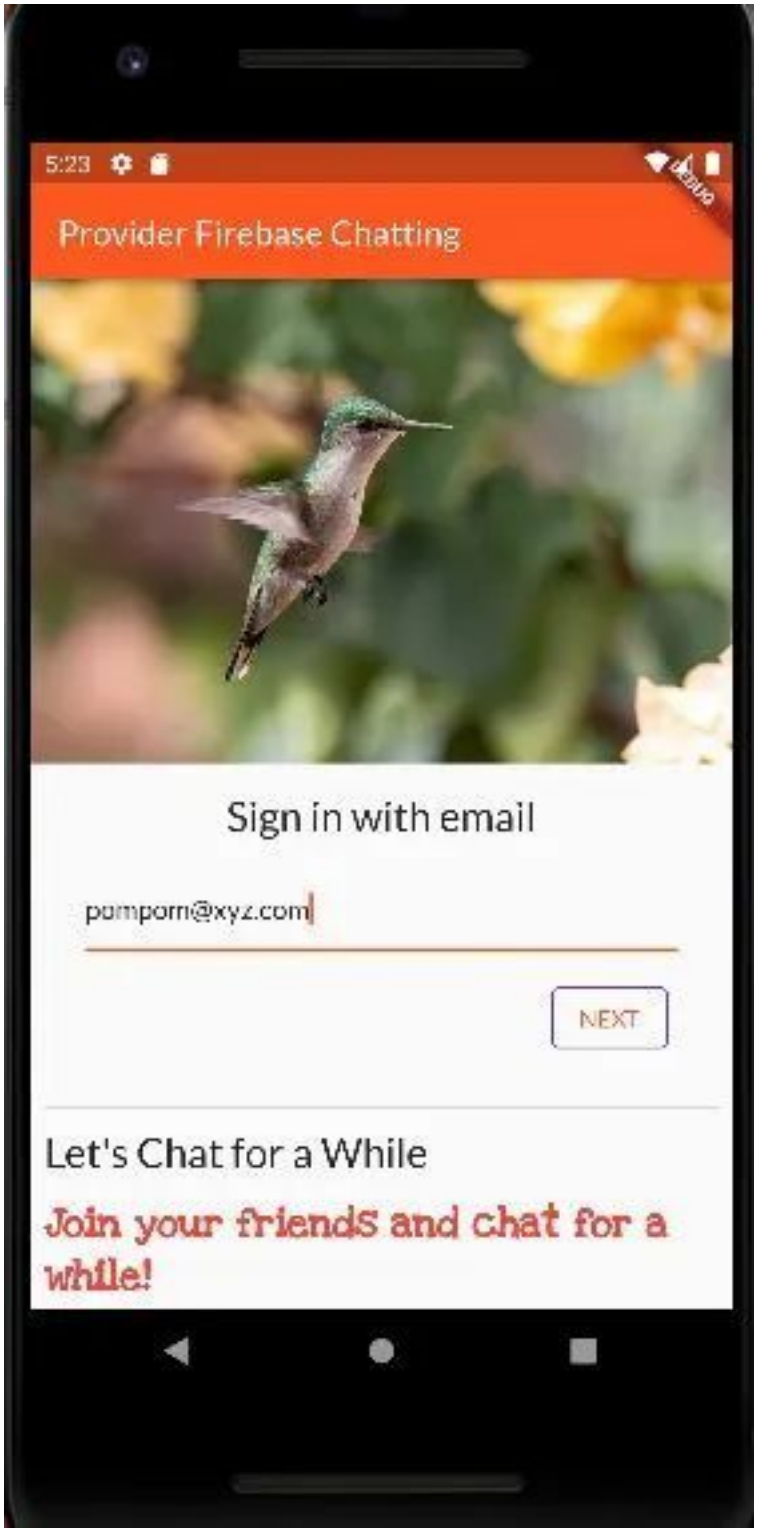

Figure 20.15 – Flutter Chat app new user registration

Suppose the user is registering with this email ID: pompom@xyz.com.

Our chat app is intelligent enough. It detects whether this email, which acts as the username, is in the Firebase authentication store or not.

If the user is new, it asks for her First and last name with the password.

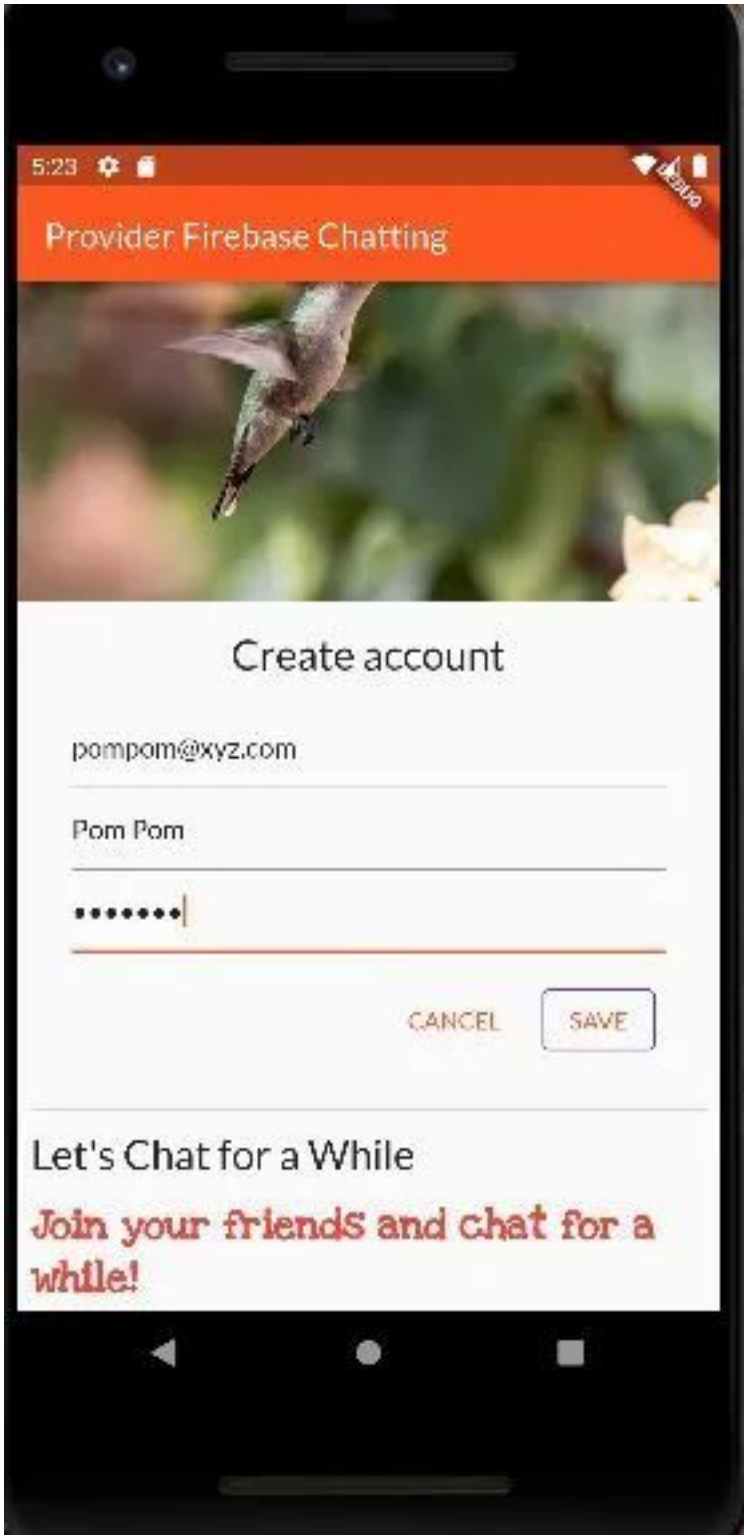

Figure 20.16 – Flutter Chat app New user registration process

If you want, you can avoid that part when you build your app. You may skip to ask for First name

and last name.

However, if you have already registered, the chat app will ask you to sign in with the password.

Figure 20.17 – Flutter Chat app _ sign in page

When the registered user comes back, she will sign in. And, as an outcome she will see the member's

page where she can chat with other members.

Now according to the database, the chat app will show the output.

While building the app, we have used two separate Firestore databases.

The output from the first one as follows.

Figure 20.18 – Flutter Chat page displays message

At the same time we can see these outputs on our Firestore console.

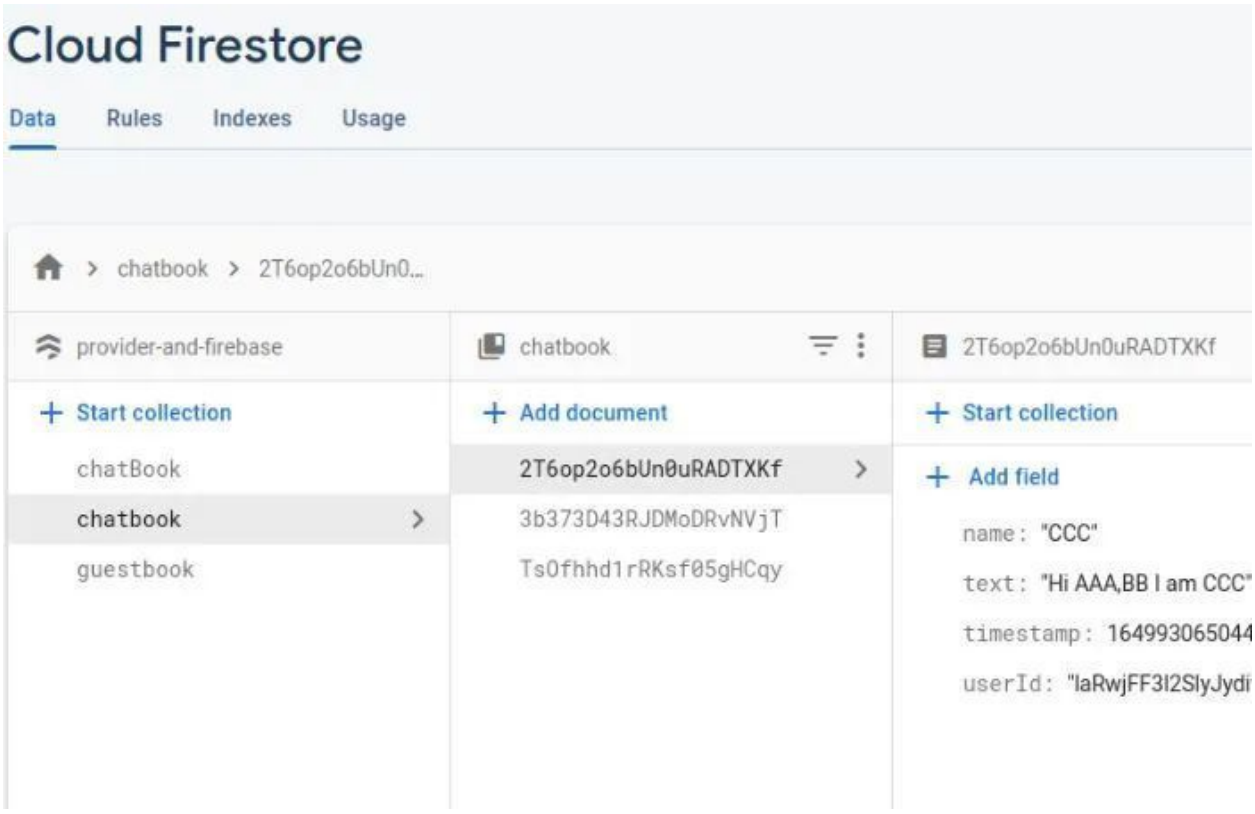

Figure 20.19 – Firestore updates itself

Once a user sends a post to the chat app, the Firebase database updates itself.

The same way as new users register, the Firebase authentication updates itself showing the users names on the console.

What is the main difference with the previous chat app?

In the previous chat app, we have added database in Firestore console before we start. But in this chat app, we can mention any database name in our application logic.

As a result, when we run the chat app it creates the database.

Certainly it makes our chat app more dynamic.

Suppose we run the chat app with a different database name. The output also changes.

A different output

That's it.

In the next section, we will learn how to build this chat app. It will be a step by step guide.

So keep reading. We'll meet soon.

Firebase, Firestore rules for Flutter Chat App

As a backend service Firebase acts fine for Flutter. But we need to follow a certain rules. It also applies to the Firestore collection that acts as the database to our Flutter Chat app.

In our previous section we have seen how good they are. For example, we have already built a Flutter Chat app.

However, in that case also, we maintained the same rules. Now, as we have been trying to make the old Flutter Chat app more dynamic and robust, we should know these rules.

What are these rules?

In addition, how they influence our Flutter Chat app?

We'll discuss these topic in this section.

Firstly we'll start with Firebase email authentication. We've already seen how we have used the email service to authenticate our users.

As our user grow the number will increase. That's fine. But how did we allow the users to register or sign in?

Well, let's see that part secondly.

After signing in to Firebase let's click the Authentication section.

What do we see?

What does it display on the screen?

It shows an edit icon on the left part. For example, at this time it is in a "disabled" mode. By clicking the edit icon, we can make it "enabled" though.

If you plan to take your Flutter Chat app to the production mode, it should remain "enabled".

But here is the caveat. If we just test our Flutter chat app, then we should keep them "disabled" after we have tested our app.

Finally, we will take a look at the Firestore collection rules. And that is really important.

Firestore collection rules playground

Most importantly this rule varies from test app to production app.

Why?

Because in test app, we should keep it in "read" mode. We cannot allow anybody to mess up with my database. Right?

If we have left it in "write" mode, any user can malign the collection with a lots of dummy data. However, in production mode, we should allow them to write or delete.

In that case, the rules will change.

Firstly, let's look at the Firestore database page.

At present we have two collections. The same name, but one of them is in the camel case.

Just above any collection, we will find a link, it says "Rules".

That link will take us to the "Rules Playground". Most importantly we should adjust everything here.

As we take a close look at the rules it reads like the following.

```
rules_version = '2';
service cloud.firestore {
match /databases/{database}/documents {
    match /{document=**} {
    allow read: if
        request.time < timestamp.date(2022, 5, 13);
    }
}
}
```

Certainly we can change it, adding the "write" functionality. In addition, we can increase the date also.

```
rules_version = '2';
service cloud.firestore {
match /databases/{database}/documents {
    match /{document=**} {
    allow read, write: if
        request.time < timestamp.date(2023, 1, 1);
    }
}
}
```

But be careful. In production mode in production mode, you cannot leave it like this. As we progress, we'll discuss that part in detail. So read every step.

And after testing is over, you should keep it in "read" mode and also change the date.

Another option is also good. You can delete the entire collection after the testing is over.

In the coming section we will start building the Flutter chat app with Provider, Firebase and Firestore.

So keep reading. We'll meet soon.

How to deal with the Business logic

In this section we'll build the Flutter Chat app UI. In addition we'll also write the Business logic with Provider package.

Therefore we need the latest provider package and we need to add that dependency to our "pubspec.yaml" file.

But at the same time we also need to fasten a few packages more.

Firstly, take a look at the packages that we have added.

```
dependencies:
flutter:
    sdk: flutter

cupertino_icons: ^1.0.2
firebase_core: ^1.14.0
firebase_auth: ^3.3.13
cloud_firestore: ^3.1.11
google_fonts: ^2.3.1
provider: ^6.0.2
```

In total we have glued together five packages. They are firebase_core, firebase_auth, cloud_firestore, google_fonts, and finally the provider.

As we have added the Firebase to our project, the user will either sign in or register.

As a result, we have to bind a button that will help users with Firebase authentication.

Now we need to add the business logic to make the authentication flow work.

The Business Logic and Provider

Let's start with the top level main() function which redirects us to the ChatApp() widget.

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'model/state_of_application.dart';

import 'view/chat_app.dart';

void main() {
runApp(
    ChangeNotifierProvider(
    create: (context) => StateOfApplication(),
    builder: (context, _) => const ChatApp(),
    ),
);
}
```

Provider is basically a wrapper around InheritedWidget to make them easier to use and more reusable.

Now other classes will consume these InheritedWidgets.

Therefore let's take a look at the model class which defines the business logic.

```
import 'package:flutter/material.dart';
import 'dart:async';
import 'package:firebase_core/firebase_core.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_auth/firebase_auth.dart';

import '../controller/authenticate_to_firebase.dart';
import '../firebase_options.dart';

import '../view/let_us_chat.dart';

class StateOfApplication extends ChangeNotifier {
StateOfApplication() {
    init();
}

Future<void> init() async {
    await Firebase.initializeApp(
    options: DefaultFirebaseOptions.currentPlatform,
    );

    FirebaseAuth.instance.userChanges().listen((user) {
```

```
23      if (user != null) {
24          _loginState = UserStatus.loggedIn;
25          _chatBookSubscription = FirebaseFirestore.instance
26              .collection('chatbook')
27              .orderBy('timestamp', descending: true)
28              .snapshots()
29              .listen((snapshot) {
30          _chatBookMessages = [];
31          for (final document in snapshot.docs) {
32              _chatBookMessages.add(
33              LetUsChatMessage(
34                  name: document.data()['name'] as String,
35                  message: document.data()['text'] as String,
36              ),
37              );
38          }
39          notifyListeners();
40          });
41      } else {
42          _loginState = UserStatus.loggedOut;
43          _chatBookMessages = [];
44      }
45      notifyListeners();
46      });
47  }
48
49  UserStatus _loginState = UserStatus.loggedOut;
50  UserStatus get loginState => _loginState;
51
52  String? _email;
53  String? get email => _email;
54
55  StreamSubscription<QuerySnapshot>? _chatBookSubscription;
56  StreamSubscription<QuerySnapshot>? get chatBookSubscription =>
57      _chatBookSubscription;
58  List<LetUsChatMessage> _chatBookMessages = [];
59  List<LetUsChatMessage> get chatBookMessages => _chatBookMessages;
60
61  void startLoginFlow() {
62      _loginState = UserStatus.emailAddress;
63      notifyListeners();
64  }
65
```

```
Future<void> verifyEmail(
    String email,
    void Function(FirebaseAuthException e) errorCallback,
) async {
    try {
    var methods =
        await FirebaseAuth.instance.fetchSignInMethodsForEmail(email);
    if (methods.contains('password')) {
        _loginState = UserStatus.password;
    } else {
        _loginState = UserStatus.register;
    }
    _email = email;
    notifyListeners();
    } on FirebaseAuthException catch (e) {
    errorCallback(e);
    }
}

Future<void> signInWithEmailAndPassword(
    String email,
    String password,
    void Function(FirebaseAuthException e) errorCallback,
) async {
    try {
    await FirebaseAuth.instance.signInWithEmailAndPassword(
        email: email,
        password: password,
    );
    } on FirebaseAuthException catch (e) {
    errorCallback(e);
    }
}

void cancelRegistration() {
    _loginState = UserStatus.emailAddress;
    notifyListeners();
}

Future<void> registerAccount(
    String email,
    String displayName,
    String password,
```

```
    void Function(FirebaseAuthException e) errorCallback) async {
    try {
    var credential = await FirebaseAuth.instance
        .createUserWithEmailAndPassword(email: email, password: password);
    await credential.user!.updateDisplayName(displayName);
    } on FirebaseAuthException catch (e) {
    errorCallback(e);
    }
}

void signOut() {
    FirebaseAuth.instance.signOut();
}

Future<DocumentReference> addMessageToChatBook(String message) {
    if (_loginState != UserStatus.loggedIn) {
    throw Exception('Must be logged in');
    }

    return FirebaseFirestore.instance
        .collection('chatbook')
        .add(<String, dynamic>{
    'text': message,
    'timestamp': DateTime.now().millisecondsSinceEpoch,
    'name': FirebaseAuth.instance.currentUser!.displayName,
    'userId': FirebaseAuth.instance.currentUser!.uid,
    });
}
}
```

Firstly, this business logic makes more sense because now we can use the "StateOfApplication" object throughout our app because of Provider.

Secondly, we have imported the necessary Firebase packages as follows.

```
import 'package:firebase_core/firebase_core.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_auth/firebase_auth.dart';
```

As an outcome, we can now check whether the user has already registered or not.

How do we test that?

Simple.

Once the user her email address and press the “Next” button, it will check whether that email address has already been stored in the Firebase authentication section or not.

Here the “Next” button is the encapsulating the business logic and the authentication flow starts from here.

How does it work?

We’ll see in a minute.

The Authentication flow

To make the authentication flow work, we need another controller that will test the credential of the users.

According to the user status it takes the user to the registration section, or to the sign in section.

```
import 'package:flutter/material.dart';
import 'package:google_fonts/google_fonts.dart';

import 'all_types_of_custom_forms.dart';
import 'all_widgets.dart';

enum UserStatus {
loggedOut,
emailAddress,
register,
password,
loggedIn,
}

class AuthenticationForFirebase extends StatelessWidget {
const AuthenticationForFirebase({
    required this.loginState,
    required this.email,
    required this.startLoginFlow,
    required this.verifyEmail,
    required this.signInWithEmailAndPassword,
    required this.cancelRegistration,
    required this.registerAccount,
    required this.signOut,
});

final UserStatus loginState;
```

```
final String? email;
final void Function() startLoginFlow;
final void Function(
    String email,
    void Function(Exception e) error,
) verifyEmail;
final void Function(
    String email,
    String password,
    void Function(Exception e) error,
) signInWithEmailAndPassword;
final void Function() cancelRegistration;
final void Function(
    String email,
    String displayName,
    String password,
    void Function(Exception e) error,
) registerAccount;
final void Function() signOut;

@override
Widget build(BuildContext context) {
    switch (loginState) {
    case UserStatus.loggedOut:
        return Row(
        children: [
            Padding(
            padding: const EdgeInsets.only(left: 24, bottom: 8),
            child: StyledButton(
                onPressed: () {
                startLoginFlow();
                },
                child: Text(
                'SignIn/Register',
                style: GoogleFonts.laila(
                    fontSize: 30.0,
                    fontWeight: FontWeight.bold,
                ),
                ),
            ),
            ),
        ],
        );
```

```
71      case UserStatus.emailAddress:
72          return CutomEmailForm(
73              callback: (email) => verifyEmail(
74                  email, (e) => _showErrorDialog(context, 'Invalid email', e)));
75      case UserStatus.password:
76          return CustomPasswordForm(
77          email: email!,
78          login: (email, password) {
79              signInWithEmailAndPassword(email, password,
80                  (e) => _showErrorDialog(context, 'Failed to sign in', e));
81          },
82          );
83      case UserStatus.register:
84          return CustomRegistrationForm(
85          email: email!,
86          cancel: () {
87              cancelRegistration();
88          },
89          registerAccount: (
90              email,
91              displayName,
92              password,
93          ) {
94              registerAccount(
95                  email,
96                  displayName,
97                  password,
98                  (e) =>
99                      _showErrorDialog(context, 'Failed to create account', e));
100         },
101         );
102     case UserStatus.loggedIn:
103         return Row(
104         children: [
105             Padding(
106             padding: const EdgeInsets.only(left: 24, bottom: 8),
107             child: StyledButton(
108                 onPressed: () {
109                 signOut();
110                 },
111                 child: const Text('LOGOUT'),
112             ),
113             ),
```

```
114         ],
115         );
116     default:
117         return Row(
118         children: const [
119             Text("Internal error, this shouldn't happen..."),
120         ],
121         );
122     }
123 }
124 
125 void _showErrorDialog(BuildContext context, String title, Exception e) {
126     showDialog<void>(
127     context: context,
128     builder: (context) {
129         return AlertDialog(
130         title: Text(
131             title,
132             style: const TextStyle(fontSize: 24),
133         ),
134         content: SingleChildScrollView(
135             child: ListBody(
136             children: <Widget>[
137                 Text(
138                 '${(e as dynamic).message}',
139                 style: const TextStyle(fontSize: 18),
140                 ),
141             ],
142             ),
143         ),
144         actions: <Widget>[
145             StyledButton(
146             onPressed: () {
147                 Navigator.of(context).pop();
148             },
149             child: const Text(
150                 'OK',
151                 style: TextStyle(color: Colors.deepPurple),
152             ),
153             ),
154         ],
155         );
156     },
```

```
157       );
158   }
159   }
```

Basically we have instantiated application state object with the ChangeNotifierProvider first. After that, in our model folder we have written the business logic.

What does the business logic do?

The application state object extends ChangeNotifier.

Why?

To make the provider aware so that it can re-display the widgets which depend on it.

Now the new User will register and create her new account. Once she has registered, she can sign in after that and chat with other users.

But, we will discuss that in the next section.

Flutter Chat app UI: designing the pages

In the last section we have seen how to handle the business logic and control the application flow. Let's design our app in a way so that it synchronises with the logic flow.

Typically we always see that a Flutter chat app uses Stateful widget. But in our case, we have used the Provider package to create the State object and later used it.

As the backend we have used Firebase and Firestore database. However, as our Flutter app runs, the Firestore database creates the collection.

By the way, we've also discussed the Firebase and Firestore rules. Please read the rules and maintain them as you progress.

If you feel curious to learn how the end product will look, you may read the previous section.

In the previous section we've also seen how our model class extends ChangeNotifier and notifies the listener.

As a result, our top level main() function looks as follows.

```dart
import 'package:flutter/material.dart';

import 'package:google_fonts/google_fonts.dart';

import 'chat_home_page.dart';

class ChatApp extends StatelessWidget {
const ChatApp({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Provider Firebase Chatting',
    theme: ThemeData(
        buttonTheme: Theme.of(context).buttonTheme.copyWith(
            highlightColor: Colors.black45,
            ),
        primarySwatch: Colors.deepOrange,
        textTheme: GoogleFonts.latoTextTheme(
        Theme.of(context).textTheme,
        ),
        visualDensity: VisualDensity.adaptivePlatformDensity,
    ),
    home: const ChatHomePage(),
    );
}
}
```

Now we should update the home page's build method to integrate with the application State.

It will check whether the user has registered already, or not. If she has registered, she can sign in as we see in the above image.

Therefore the code of our home page will look like the following.

```dart
import 'package:flutter/material.dart';
import 'package:google_fonts/google_fonts.dart';

import 'package:provider/provider.dart';
import 'package:provider_and_firebase/view/let_us_chat.dart';
import '../model/state_of_application.dart';
import '../controller/authenticate_to_firebase.dart';
import '../controller/all_widgets.dart';

class ChatHomePage extends StatelessWidget {
const ChatHomePage({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Provider Firebase Chatting'),
    ),
    body: ListView(
        children: <Widget>[
        Image.network(
            'https://cdn.pixabay.com/photo/2022/04/07/11/45/bird-7117346_960_720.jpg\
'),
        const SizedBox(height: 8),
        Consumer<StateOfApplication>(
            builder: (context, appState, _) => AuthenticationForFirebase(
            email: appState.email,
            loginState: appState.loginState,
            startLoginFlow: appState.startLoginFlow,
            verifyEmail: appState.verifyEmail,
            signInWithEmailAndPassword: appState.signInWithEmailAndPassword,
            cancelRegistration: appState.cancelRegistration,
            registerAccount: appState.registerAccount,
            signOut: appState.signOut,
            ),
        ),
        const Divider(
            height: 8,
            thickness: 1,
            indent: 8,
            endIndent: 8,
            color: Colors.grey,
        ),
```

```
44          const Header("Let's Chat for a While"),
45          const Paragraph(
46              'Join your friends and chat for a while!',
47          ),
48          Consumer<StateOfApplication>(
49              builder: (context, appState, _) => Column(
50              crossAxisAlignment: CrossAxisAlignment.start,
51              children: [
52                  if (appState.loginState == UserStatus.loggedIn) ...[
53                  TextButton(
54                      onPressed: () {
55                      Navigator.push(
56                          context,
57                          MaterialPageRoute(
58                          builder: (context) => LetUsChat(
59                              addMessage: (message) =>
60                                  appState.addMessageToChatBook(message),
61                              messages: appState.chatBookMessages,
62                          ),
63                          ),
64                      );
65                      },
66                      child: Text(
67                      'Let\'s Chat',
68                      style: GoogleFonts.laila(
69                          fontSize: 30.0,
70                          fontWeight: FontWeight.bold,
71                          color: Colors.yellow,
72                          backgroundColor: Colors.red,
73                      ),
74                      ),
75                  ),
76                  ],
77              ],
78              ),
79          ),
80          ],
81      ),
82      );
83  }
84  }
```

The highlighted part clearly shows that, if the user has registered already, in that case what will

happen.

It will navigate to another view page.

How Flutter Chat app UI works

This page will show the messages that the users write one after another.

Yet let us take a look at the code snippet firstly.

```
import 'package:flutter/material.dart';

import 'package:google_fonts/google_fonts.dart';

import 'chat_home_page.dart';

class ChatApp extends StatelessWidget {
const ChatApp({Key? key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Provider Firebase Chatting',
    theme: ThemeData(
        buttonTheme: Theme.of(context).buttonTheme.copyWith(
            highlightColor: Colors.black45,
            ),
        primarySwatch: Colors.deepOrange,
        textTheme: GoogleFonts.latoTextTheme(
        Theme.of(context).textTheme,
        ),
        visualDensity: VisualDensity.adaptivePlatformDensity,
    ),
    home: const ChatHomePage(),
    );
}
}
```

We've already defined that the registered user can start chatting and add messages in this part of code.

```
Navigator.push(
                        context,
                        MaterialPageRoute(
                        builder: (context) => LetUsChat(
                            addMessage: (message) =>
                                appState.addMessageToChatBook(message),
                            messages: appState.chatBookMessages,
                        ),
                        ),
                    );
```

If you wonder how we have got the State of the Flutter Chat app, then we need to take a look at this part of the code in our home page.

```
Consumer<StateOfApplication>(
            builder: (context, appState, _) => AuthenticationForFirebase(
            email: appState.email,
            loginState: appState.loginState,
            startLoginFlow: appState.startLoginFlow,
            verifyEmail: appState.verifyEmail,
            signInWithEmailAndPassword: appState.signInWithEmailAndPassword,
            cancelRegistration: appState.cancelRegistration,
            registerAccount: appState.registerAccount,
            signOut: appState.signOut,
            ),
        ),
```

Here the Consumer uses the Type of “StateOfApplication” which is our model class.

After that, it returns the controller “StateOfApplication” where we have defined each property.

As a result, the user can chat with other users and we can see the chats on the screen.

Certainly, it is worth noting that we need to customise the font, with Google font plugin. Apart from that, we also have different types of widgets like header or paragraph.

In the next section we will discuss that. In addition we’ll also see how Firebase and Firestore add users and messages.

Chat Apps for Android with Flutter and Firebase

In this final section we’ll make sure that our Chat apps will work for Android. In the same vein it could have worked for iOS or web. We have that options also.

To make that happen, we have made our flutter chat app platform specific.

As a result, user can choose the specific platforms and issue her own API key, ID etc.

```
import 'package:firebase_core/firebase_core.dart' show FirebaseOptions;
import 'package:flutter/foundation.dart'
    show defaultTargetPlatform, kIsWeb, TargetPlatform;
/// we need to specify the associated values according to the platform
/// we're using, like in this case, we have chosen Android platform
/// in Firebase console
///
class DefaultFirebaseOptions {
static FirebaseOptions get currentPlatform {
    if (kIsWeb) {
    return web;
    }
    // ignore: missing_enum_constant_in_switch
    switch (defaultTargetPlatform) {
    case TargetPlatform.android:
        return android;
    case TargetPlatform.iOS:
        return ios;
    case TargetPlatform.macOS:
        return macos;
    }

    throw UnsupportedError(
    'DefaultFirebaseOptions are not supported for this platform.',
    );
}

static const FirebaseOptions web = FirebaseOptions(
    apiKey: '',
    appId: '',
    messagingSenderId: '',
    projectId: '',
);

static const FirebaseOptions android = FirebaseOptions(
    apiKey: '',
    appId: '',
    messagingSenderId: '',
    projectId: '',
);

static const FirebaseOptions ios = FirebaseOptions(
    apiKey: '',
```

```
    appId: '',
    messagingSenderId: '',
    projectId: '',
);

static const FirebaseOptions macos = FirebaseOptions(
    apiKey: '',
    appId: '',
    messagingSenderId: '',
    projectId: '',
);
}
```

Since we have decided to make the flutter chat app android specific, we've worked on this part particularly.

```
static const FirebaseOptions android = FirebaseOptions(
    apiKey: '',
    appId: '',
    messagingSenderId: '',
    projectId: '',
);
```

Besides, we have also made it sure that users can write and send their messages to the chat page where other users can see them.

As a result, in our model folder, where we have define the state, we have added this section.

```
Future<void> init() async {
    await Firebase.initializeApp(
    options: DefaultFirebaseOptions.currentPlatform,
    );

    FirebaseAuth.instance.userChanges().listen((user) {
    if (user != null) {
        _loginState = UserStatus.loggedIn;
        _chatBookSubscription = FirebaseFirestore.instance
            .collection('chatbook')
            .orderBy('timestamp', descending: true)
            .snapshots()
            .listen((snapshot) {
        _chatBookMessages = [];
        for (final document in snapshot.docs) {
```

```
            _chatBookMessages.add(
            LetUsChatMessage(
                name: document.data()['name'] as String,
                message: document.data()['text'] as String,
            ),
            );
        }
        notifyListeners();
        });
    } else {
        _loginState = UserStatus.loggedOut;
        _chatBookMessages = [];
    }
    notifyListeners();
    });
}
```

We have used Future async await to notify the listeners once the chat app adds the message.

And, after that, we have also mentioned in our view section, the user can write and view the message at the same time.

```
StyledButton(
                    onPressed: () async {
                    if (_formKey.currentState!.validate()) {
                        await widget.addMessage(_controller.text);
                        _controller.clear();
                    }
                    },
                    child: Row(
                    children: const [
                        Icon(Icons.send),
                        SizedBox(width: 6),
                        Text('SEND'),
                    ],
                    ),
...
for (var message in widget.messages)
            Paragraph('${message.name}: ${message.message}'),
        const SizedBox(height: 10.0),
...
```

Certainly Flutter doesn't have any Paragraph widget. Therefore, we have built our own Widgets that manage the styling.

Style and Theme of Flutter chat app In the controller folder we have separate widgets that handle these widgets. Above all we also have have another file that defines various Google fonts and styling.

Let's see how they look at a glance.

```
class Paragraph extends StatelessWidget {
const Paragraph(this.content);
final String content;
@override
Widget build(BuildContext context) => Padding(
        padding: const EdgeInsets.symmetric(horizontal: 8, vertical: 4),
        child: Text(content,
            style: GoogleFonts.loveYaLikeASister(
            fontSize: 25.0,
            fontWeight: FontWeight.bold,
            color: Colors.red,
            )),
    );
}
...
class StyledButton extends StatelessWidget {
const StyledButton({required this.child, required this.onPressed});
final Widget child;
final void Function() onPressed;

@override
Widget build(BuildContext context) => OutlinedButton(
        style: OutlinedButton.styleFrom(
            side: const BorderSide(color: Colors.deepPurple)),
        onPressed: onPressed,
        child: child,
    );
}
...
```

Of course, you can choose your own font style here.

In the last section we have seen how to handle the business logic and control the application flow. Let's design our app in a way so that it synchronises with the logic flow.

Typically we always see that a Flutter chat app uses Stateful widget. But in our case, we have used the Provider package to create the State object and later used it.

As the backend we have used Firebase and Firestore database. However, as our Flutter app runs, the Firestore database creates the collection.

By the way, we've also discussed the Firebase and Firestore rules. Please read the rules and maintain them as you progress.

In total we have glued together five packages. They are firebase_core, firebase_auth, cloud_firestore, google_fonts, and finally the provider.

In conclusion, you can clone this flutter chat app in your local machine and run. However before using this GitHub repository please use your Firebase API key and other credentials.

Otherwise it will not work. Moreover, please maintain the Firebase and Firestore rules that I have discussed earlier.

The code repositories for the Stateful Flutter Chat App[96]

The code repositories for the Flutter Chat App with Provider, Firebase and Firestore[97]

[96] https://github.com/sanjibsinha/register_users
[97] https://github.com/sanjibsinha/provider_and_firebase

21. What Next?

If you find these information useful, I am happy. For any Flutter related query, you may send an email to

1. sanjib12sinha@gmail.com.
2. sanjib@sanjibsinha.com

And at the same time don't forget to visit the website where I write regularly on Flutter only.

You'll get the updated Flutter tips and tricks. So how about take the trips? :)

For more Flutter related Articles and Resources[98]

The code repositories for this book[99]

[98] https://sanjibsinha.com
[99] https://github.com/sanjibsinha/