A Guide to Qt 6

For the fun in learning

about Qt

and the World we reside in

BenCoepp

Ben Coepp

Before we get started

Before we can get started with the book there are a few things I need to make clear, one is for legal reasons and the other is for the simple fact that I want to be honest with you how this book works and who was involved with the creation of the book.

First of I am not in any way, shape and or form linked or a part of Qt, the Qt Company. Therefore, I do not represent Qt or the Qt Company with this book, the statements I bring forth and the content I provide.

I am responsible for all the content and work in this book. If there are problems, mistakes, or misconceptions, then they are my responsibilities. If you want to contact me all my contact information can be found in Chapter 8 of this book.

Content

1 Introduction

Before we start with the book, there are a few things I like to talk about, these are not essential for buying this book, and I would even go so far as to tell you to skip this part if you are only here for the content and the tutorials. But if you want to know a little bit more about me, what we are going to do and how we are going to do it, then read along, for all others, refer to the Index and jump straight into **Chapter *2 Content***.

1.1 About Me

First off congratulations for buying this book, now that you already bought the book you might have some questions about what we are going to talk about and who the author, me is. The answers to what we are doing will be provided later on, but the question of who I am I can answer now.

My name is Ben Cöppicus, currently I am 20 years old living in Germany and working as a software engineer. I am still young so I already know this might turn some people off the topic and off reading the book immediately, and I am terribly sorry for this. I am just young, and there is nothing I can really do about that. But Qt is dear to my heart and I use it nearly every day for my own work, my own development projects, unfortunately not for the company I work for they currently do not use Qt.

My focus and stuff I generally do is in mobile and desktop development until now my focus was on creating applications

for office specific purpose like financing and general management as well as task management.

Because I am still young, there will be some people that question my knowledge and what I even know about specific topics, that I talk about and I can fully understand this. And I will be completely honest with you I do not know or understand everything. I have specific knowledge in some extremely specific parts about Qt. I talk about these parts I want to share with you and want to make more people aware of and more people understand how stuff like this works but this knowledge is extremely limited. I am not a university professor, I am not a teacher and for lack of a better word, I will never pretend to be one. So, prepare yourself for reading a lot of this is my opinion, and this is stuff I do, so always take the stuff you read in this book with a grain of salt. I will try to be as accurate and descriptive and generally as best of an author and a teacher as I can be, but Qt is an exceptionally large and complicated topic with a lot of people that have a far greater understanding and more rounded out knowledge of what Qt is and how it functions.

1.2 What is this Book about?

I really like Qt and the development experience using it. But it lacks tutorials, videos, and books about it, at least if you look at the last two years, there was not so much new content provided. Learning it can be done throw just trying out all the features and experimentation but that is not a good way to learn Qt in my opinion.

You could also go through the examples provided by Qt, but these are also not the greatest[1]. But that only motivated me more for writing this book. This books purpose can be boiled down to the fact that by the end of this book you should be able to make your own applications with Qt. May they be mobile or desktop applications. I am not all knowing, or the only voice on matters in Qt. But I know how to use it and how to best get started using Qt, and that is in my opinion the most important part. I am not trying to push a specific way of using Qt or programming onto you, just showing mine. I have my own way of doing it and sometimes this is not the best way or the way that you would most likely find elsewhere.

My way of doing it can be boiled down to, I want to achieve the things I want, and I am willing to do that with every way available to me. This just means that I do not always use the right way but always the way that best achieves the results I want.

1.3 What are we doing in this Book?

This book will take you on a ride throw a lot of features Qt has to offer. We will be starting with writing our first few applications and the concepts behind them. From there we will be going over more complicated concepts and principles behind making good applications. We will also cover most of the components that are relevant to making applications, at least those which you will

[1] This does not mean it is bad, but there are some parts which are out of date, but they are still extremely great showcases and can really help you learn and understand specific topics in Qt

use on a regular basis. And finally, we are going to create some real applications that you would use in real life.

What do I mean by that? Well, what you would find in a lot of books out there are applications that might be considered good educational content, but not actual good applications. This, I want to alleviate with bringing in my own work-related projects that I needed to do. These in my opinion will best represent the type of work you will need to do. They are still fun applications that I constructed in such a way as to make them as enjoyable as possible. Overall, this book is supposed to teach you Qt, and all the underlying concepts, principles, and elements. As well as general programming and development know-how. This will also be a large focus later in the book. The middle part that goes over the components that Qt provide can also be used as documentation and a place to look for specific solution or an example. But remember, I do not want to replace the Qt Docs, they might be outdated at some places but overall, they are compared to a lot of other software documentation extremely good. But what do I even mean with outdated? Well, if you are someone new learning Qt it might come to you as that some part of the documentation is better presented and polished then others. This comes down to how important the subject or component is and how frequently it is updated and used. And with outdated I specifically mean documentations like the Qml Local Storage, or QML Calendar that are still used and new users might want to implement, but they are not really that well supported anymore or are somewhat hard to understand or get into. And Chapter 3 of this book will be all about that, shining a light on some of the components that you will tend to use quit a lot throw your development years, but that might not be explained enough on the Qt Docs for a complete beginner.

And if you are someone who is a little bit more experienced you might find some useful information none the less.

1.4 What are we using?

As the title of this book suggests we are going to mainly use Qt. And specifically, Qt 6.0[2] and above. This also comes with the Qt Creator and Qt Design Studio which we are also going to use for all the application development we are going to do. If you already have some knowledge of how Qt works, and you want to use your own desired IDE then feel free to do so. But remember that all the screenshots as well as descriptions will be based on Qt Creator and Design Studio.

We also need Android Studio for the Android SDK, SDK Tools and the NDK. This is essential for our Android development. Without it we cannot really do anything. Other than that, I would also recommend getting something like Visual Studio Code or Atom as a Text Editor. We are not going to use this, because we can do everything, we need in Qt Creator and Design Studio. But down the line it would be helpful for creating specific files or writing code that is not highlighted as good as QML[2] for instance JavaScript. Which has little to no highlight at all. But you can choose if you want to use this, in this book we are not going to use it, and I will only mention it when it could be used. But I will not be covering the installation of Visual Studio Code or any other Text Editor or IDE, you can use them if you want to, but they are not covered in this book, which means I cannot help you if something goas wrong.

If you wanted to develop applications for IOS devices well then, I am sorry to tell you that this will not be covered in this book.

[2] For the projects we are doing it makes no difference which Qt version you use, as long as it is above 5.12

1.5 Signals, Warnings, and the Context

As with all books that teach you something, there will come the time where I want to add a bit more information. And for that purpose, I will give a signal.

- **This might be done using the footnotes[1]**
- **A different text colour or *cursive* writing**
- **Image / Screenshot**
    - **And the description below the screenshots**

These should be readable and easy to find. But I want to minimise the amount of use I make of them. This is for the reason of how I structure the book. I only want to use them in specific places where there is a true need for them and there is no other way. Also remember to check out the page about the book on my website. There you will find a few more passages as well as links to Git Hub where I have a full repository with all the code and resources we use in the project. You do not need this per say but if you have questions that are not answered in the book or are not as clear as you want them to be. Then you can check these resources.

But all these signals and warnings are easy to understand and difficult to miss. And as you can see this book is not visually stunning. This was never the point of what I wanted to create, the content and the information I want to teach is the most important part.

You will see that I tend to use screenshots on a very regular basis, all screenshots were created using Qt 6 as it is the current version, but if you are running any current version of Qt then all screenshots still hold true.

Sometimes I will also use code snippets if I want to show you a lot of code at ones, because using screenshot in that instance would be quit the undertaking and be really confusing at times.

2 Content

The content I provide in this book can generally be read in any way you want, but I would strongly recommend you read it in order so as not to lose focus and miss something important.

But later on, you might want to just jump straight to a specific topic or subject that interests you, this is also possible but refer to the Index of this book to figure out where you need to go.

2.1 Setting up the Tools we need

Now we come to the least interesting part of this book, downloading, and installing all the tools, software, and SDKs as well as NDKs[3] we need. I highly recommend you follow along and install anything you are missing. Except for Android Studio and Qt as well as Qt Creator you can use whatever tool, IDE, or text editor you will not. Just please remember that if you choose your own tools, you might get problems, warnings that this book cannot help you with.

But that does not mean that I will leave you hanging, when you stumble across problems like this you can always write me an

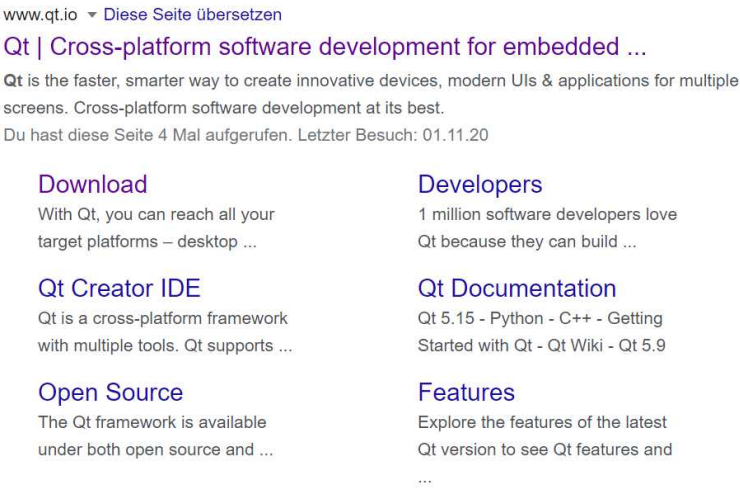
[3] SDKs and NDKs are fundamental in any development project, they include all the files and data you need to develop your applications

email or leave a question on the Amazon page for this book, I will try to help you with any question or problem you might have.

2.1.1 Downloading + Installing Qt

If you already know how to install Qt or you want to figure it out yourself, you do not need to follow the steps down below.

First of we need to get Qt from somewhere and the best way is through their website. Just type Qt into google.com and it will take you from there.



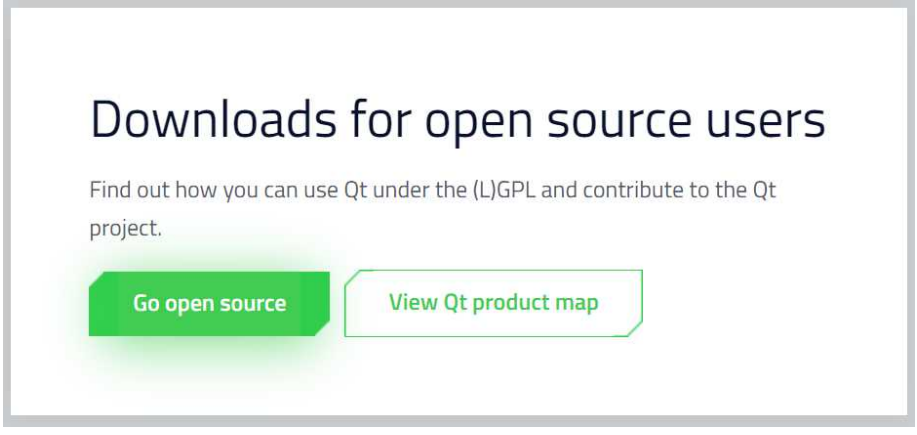
Google Search of Qt

Click the first link that says qt.io. Here you will be presented with the homepage of the Qt Company. Here you can find a lot about the product and the services that the Qt Company provides. But for us the only important part is up top.

Qt Website Top-Bar

The *Download. Try.* button. Clicking this will bring us to the next part. If you want to buy Qt, then you can also click the green *Buy Now*. Button, but for our needs and what I want to teach you it is not needed. And Qt has a good Open-Source base which has all the features needed for most developers. And I never needed anything else so, we surely do not need to buy Qt. Also, there are only a few features that can be bought with the commercial license. And you only need it if you want to make money selling and providing an application.

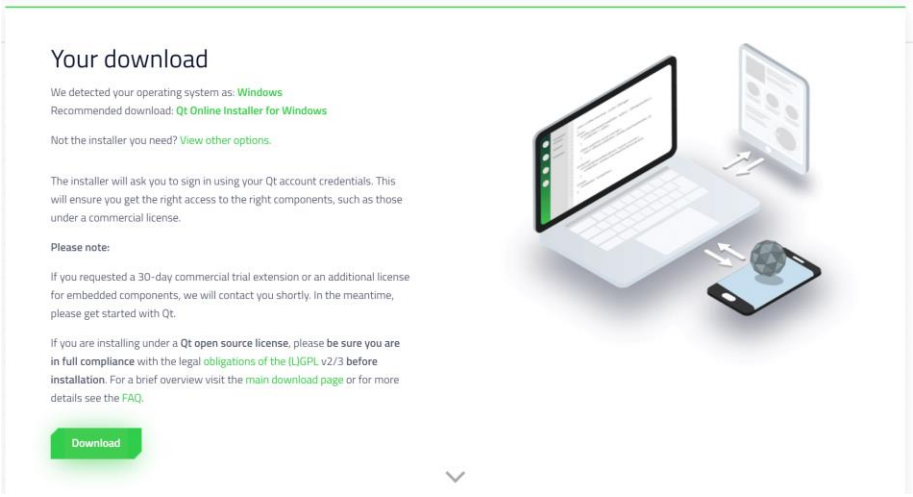
Next scroll down to the different download links and click on the green button that says *Go open source*. This will take you to the next section.



Download Card for Open-Source Users

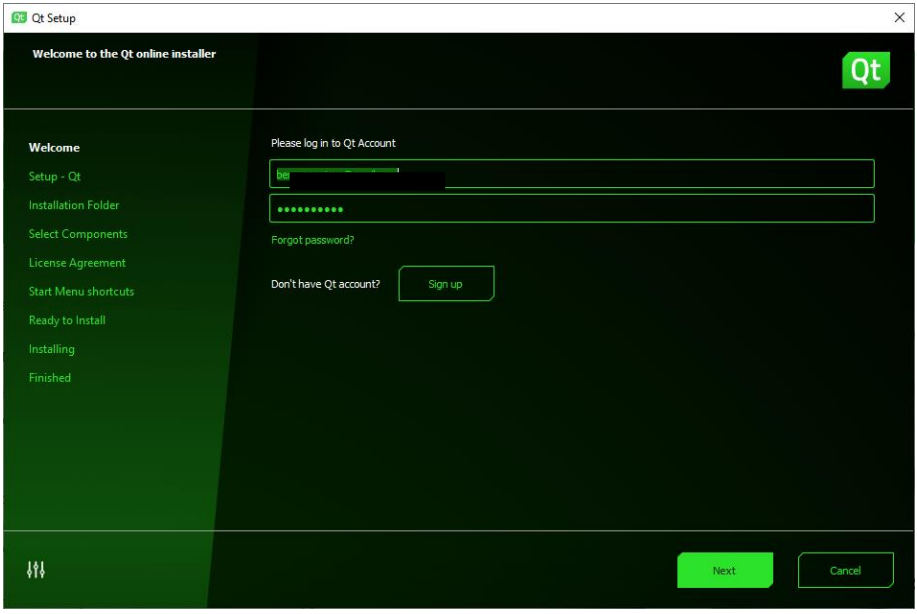
Here you also need to scroll down near the bottom of the page and click on *Download the Qt Online Installer*. This will bring you to the final page we need to visit on their website.

On the final page you only need to click on the *Download* Button. This will download the installer.



For now, you can close the browser and start the installer.

The installation is going to take a while so remember that. But other than this just follow the screenshots down below.



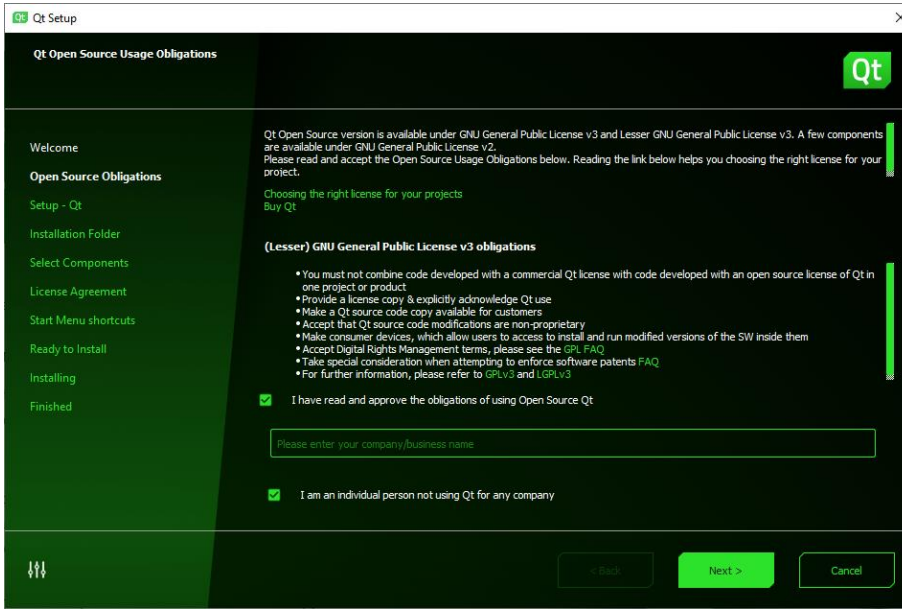
Login Form of the Qt Installer

Here you need to just put in your Qt account name and password. If you do not already have an account, then you can click the Sign-Up link and create an account. Creating an account is mandatory, so if you do not have one click create or go to the Qt website and create one for yourself.

If you already have an account, then you fill out the two input forms and then click next. As always keep your password and email a secret, otherwise someone might try stealing your data.

This is also why I blacked out my email. Not because I do not want to receive emails from you, but to many emails is also not great.

Qt open source runs under the GNU General Public License v3[4]. Therefore, you cannot use this version of Qt we downloaded, the Installer for If it is for commercial use.



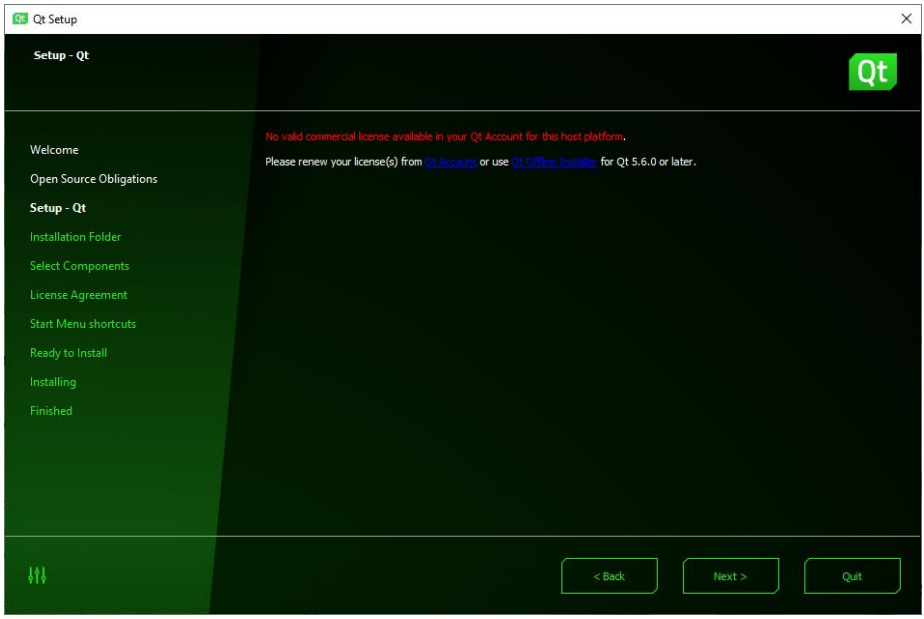
Qt License Agreement

You need to check that you have read and approved of the license agreements and ether type in your company's name or leave the checkbox by I am an individual. If you have done both then Next will be enabled and we can continue.

A little anecdote to this, Qt is not that large of a company, compared to other companies like Amazon or Microsoft. So, if you want to make money selling an app or providing some other kind of monetary service, please buy a Qt License. It helps the company finance the development, and you will not get in legal problems.

[4] The GNU General Public License v3 is a special License that allows you to use Qt and its Components and Features for non-commercial use https://www.qt.io/licensing/

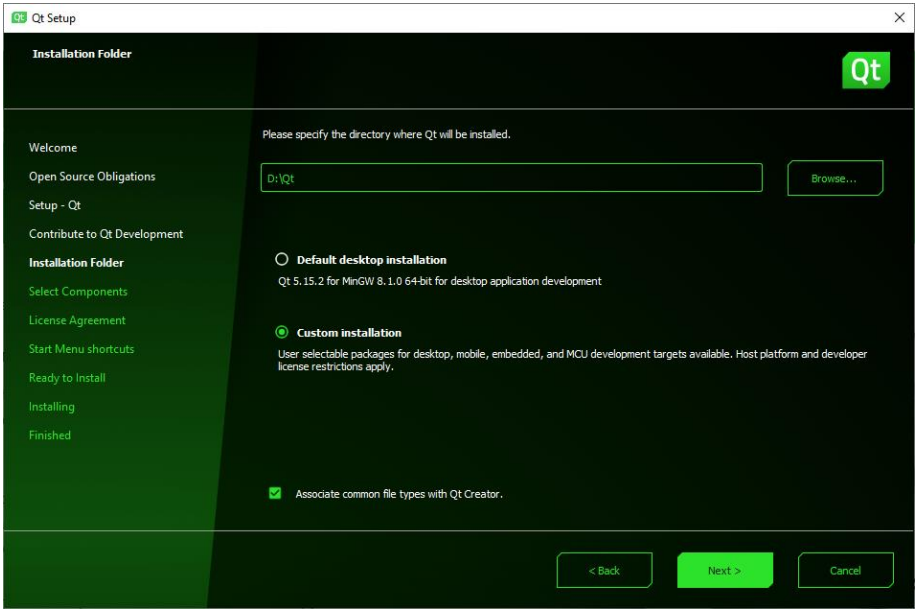
This page can be ignored. We do not have a commercial license and we do not want one, and we do not need one for what we are doing. So, hit Next and continue. If you ask me why I good this page, well I do not know myself and a lot of people online were not able to figure that one out.



Expired License Page, you can ignore this here

When you did this the installer will retrieve some metainformation and then download it.

Depending on your internet connection this can take a while. Also, I had the problem on some machines that even while having a good connection the download took a long time so be aware of that.



Location Page as well as installation type

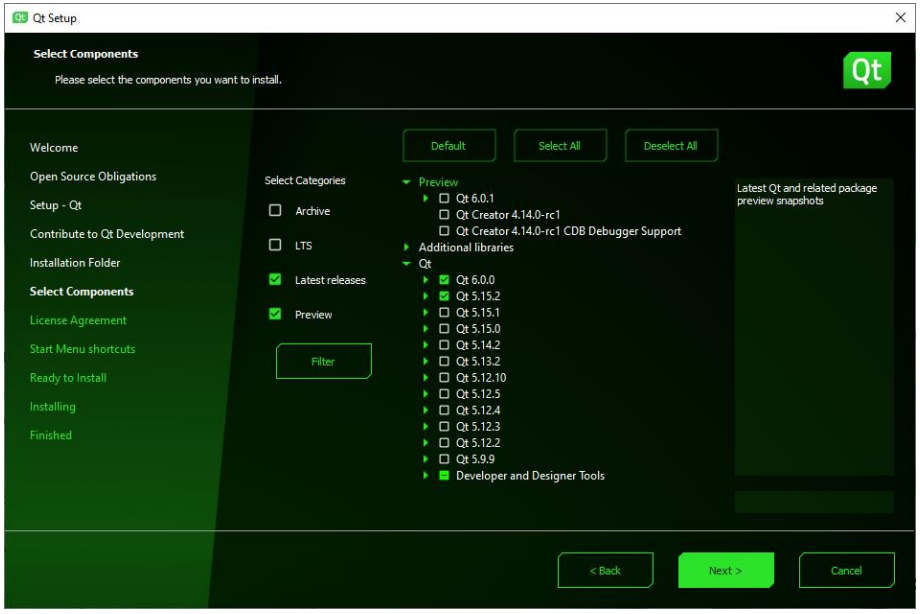
Next the installer asks if you want to provide usage data, crash reports, and general statistics to the Qt Company. I myself check the first Box and let them use my data. But this choice is up to you. After you selected your choice hit Next.

If you done your choice, but later want to change this, then you can lunch the Qt Maintenance Tool. There you get another chance to change this setting.

On the next page the installer wants you to select the installation path and if you want a custom installation or the default desktop installation. We will leave everything as it is and hit next. If you want to you can chose your own installation folder and even select the default desktop installation. Just leave your hands of the checkbox in the bottom left. With everything done click Next.

This is the most important step in the installer and the one that can screw up your development environment so read carefully. We will not be using a preview version of Qt but a stable

release. For that you need to click on the Qt drop down and expend it.

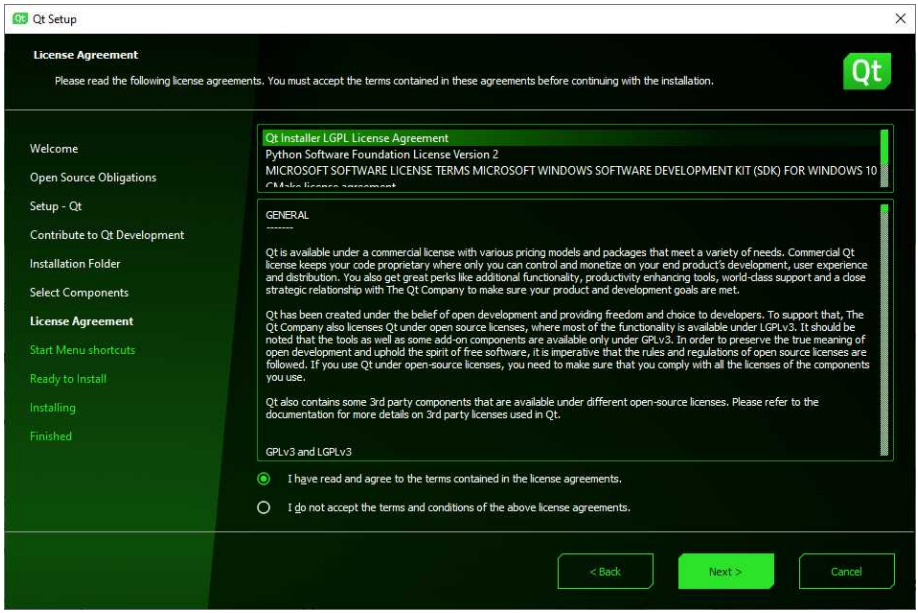


Here you can choose the kits and tools

This will pop open a drop down with a lot of different Qt versions. For us only the newest version is of any use. Check Qt 6 and open the Qt 6 drop down and let us have a look inside.

Here you can find a lot of different Packages that Qt ships with a normal installation. For a beginner I would always recommend to just leave everything as it is and continue. A little side note, Qt takes up a lot of space, above 50 GB so if you do not have that much space on your Hard drive then you might want to remove some of the libraries and or packages. Thinks we will not need are Web Assembly, MSVC and UWP. Everything else is needed and should be included. I know this is a little bit outrages how much space it will take up, but it will be worth it in the end.

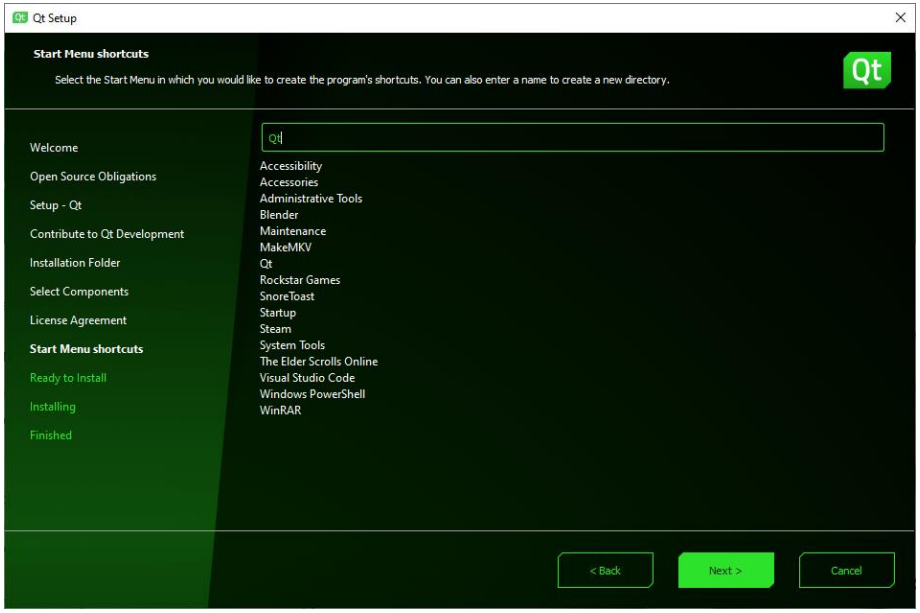
At least in my opinion it is worth it, having all the available resources as well as tools and features right from the get-go can help you start out immediately with a new project. You do not need to search for the right tool or something like this. You can just start.



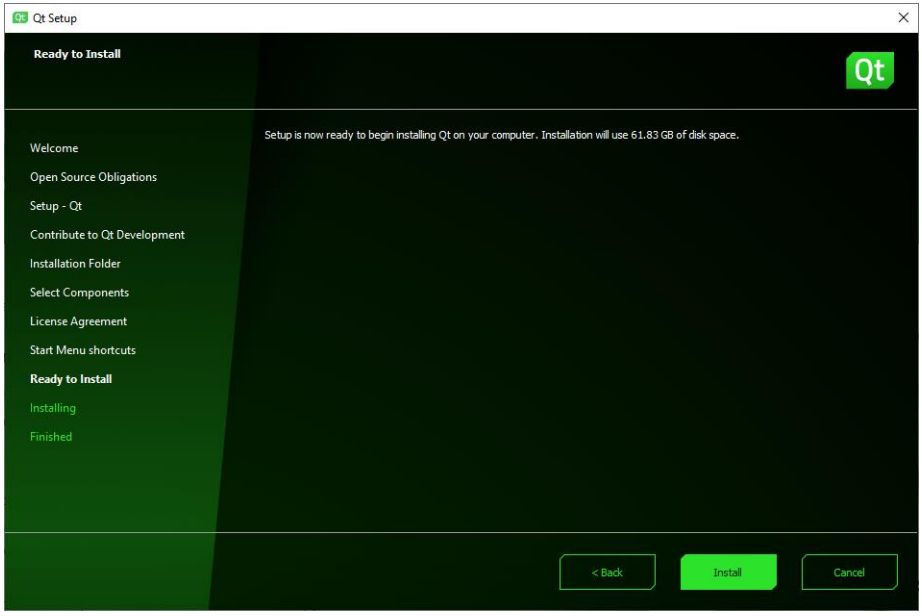
Qt License Agreement Wizard Page

Here you need to agree to the license agreements. No real choice here, you might want to read the agreement but if you are finished continuing.

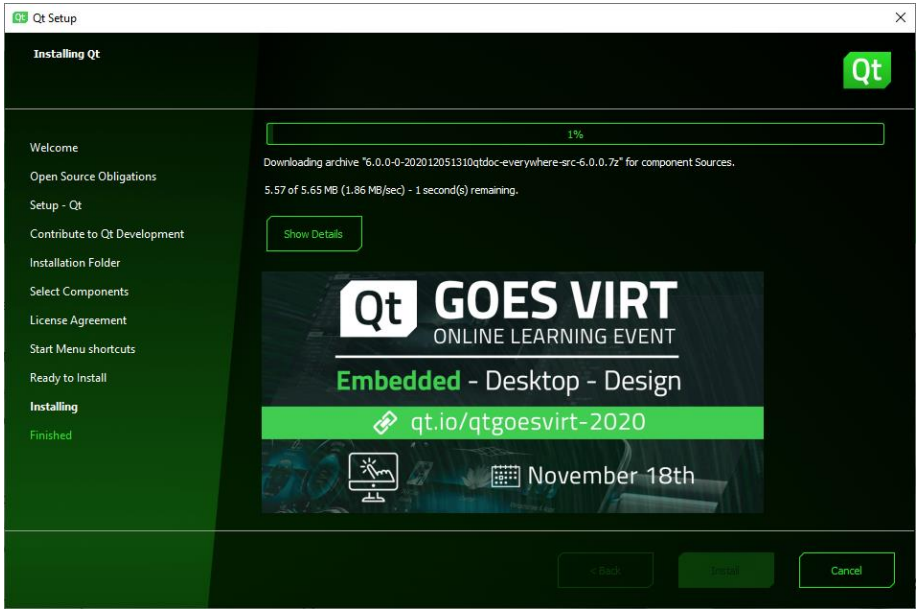
If you want to publish your application or code you should read this, it will tell you a little bit about what is ok, what you can and cannot do with the Open-Source license. Also, my recommendation is to always go and contact the Qt Company when you are unsure about your plans, or product. It is better to ask then break the rules.



Here you can see that Qt will create a shortcut and integrate it in the window menu, so it can be selected from there.



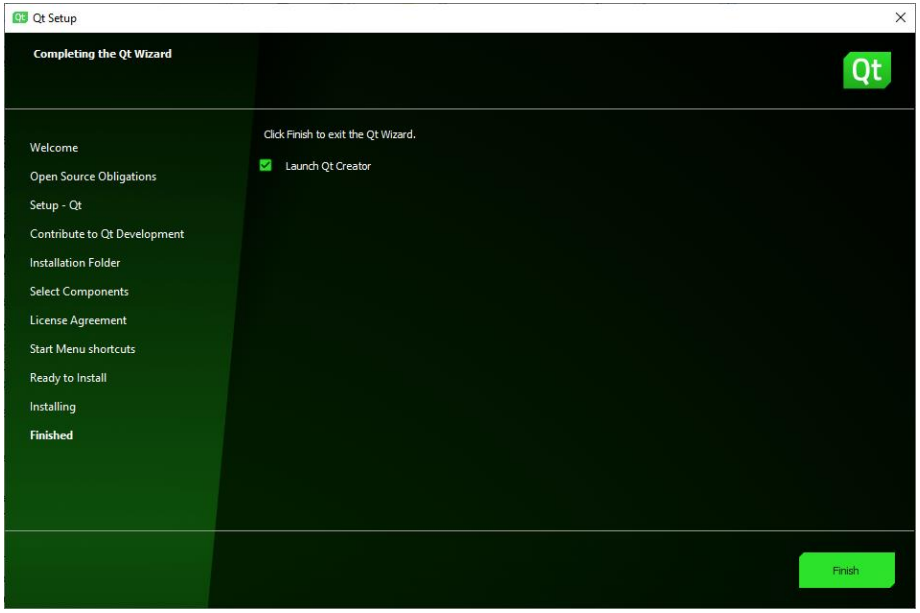
Here Qt tells you how large the Installation is going to be. If it is too large for you or you do not have that much space go back and deselect some of the packages.



Installation page, you can follow the installation here

This you will need to let run for quite a while. When done with this installation you will most likely never revisit this installer other than to update Qt and that is it. It will install everything you need and when finished the install button in the bottom right will be enabled.

For the time that this can take, what I mean with a while is that it can take up to an hour or even more depending on your internet connection and speed. In my case this takes around 30 minutes. But on other devices this took me around two hours so be aware that this might take a while.



After everything is installed and setup you will be brought to this last Page, here you can choose to either open Qt Creator right away or not. With your choice made hit the Finish button.

Now having installed Qt and all the tools belonging to it you might be wandering how updating and maintaining the software and tools is handled or if you can download other packages or versions later. And yes, you can, mainly throw the Maintenance Tool Qt provides. It is basically the installer, throw which you can manage all the versions and packages on your machine. This also extends to the fact that Qt requires a lot of space and this will only add open time when you install more and more packages and versions. And depending on the size of the install drive you might run out of space not long after. For us and in the aspect of this book we will not be touching this again but remember that if you need it than there is an option to do it.

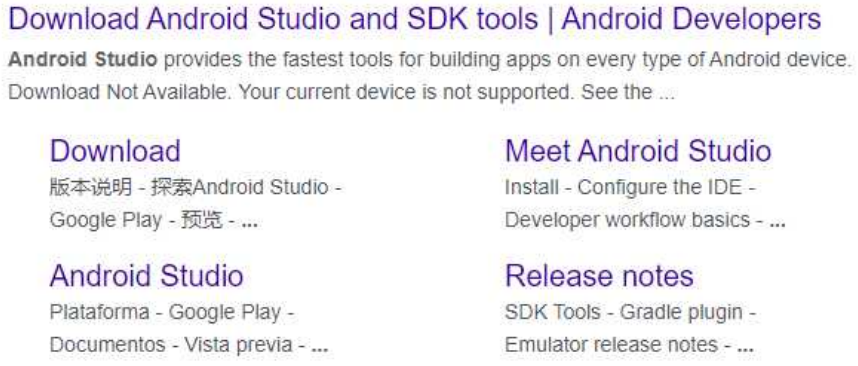
For a side note, which is not important to the subject but still worth maintaining is the subject of having multiple different version on your machine. In my opinion it is strictly speaking not

necessary. If you are only working on one specific software or only with this book, you will only work with one version. But if you need to work and maintain different software and projects that were made with a different version than yes you will need to download and install multiple versions. It would still be my recommendation to update the version of the software or project, but this might not be possible depending on the specific type of software or project you have. So, take this as my suggestion and thing about if you need this.

21.3 Downloading + Installing Android Studio

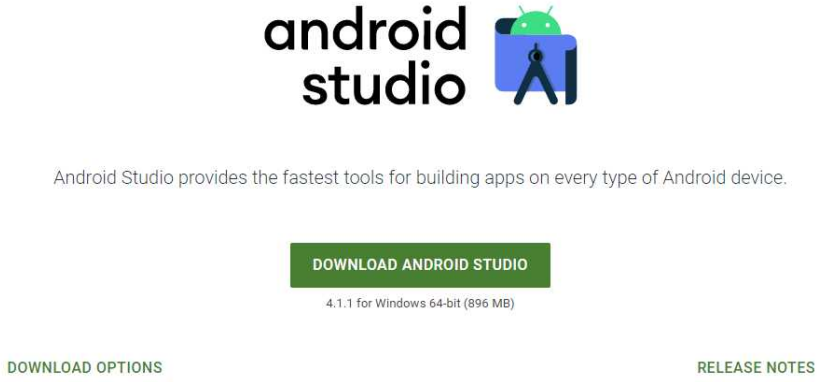
Next, we are going to download and install Android Studio, we are going to do this now as we are already installing everything else, but we are not going to use Android Studio, or the SDKs and NDKs for a long time. But please follow along or if you already want to jump back to this chapter when you need it.

As with the other two getting this is not that hard. Google Android Studio and you will find it as the first few search results.



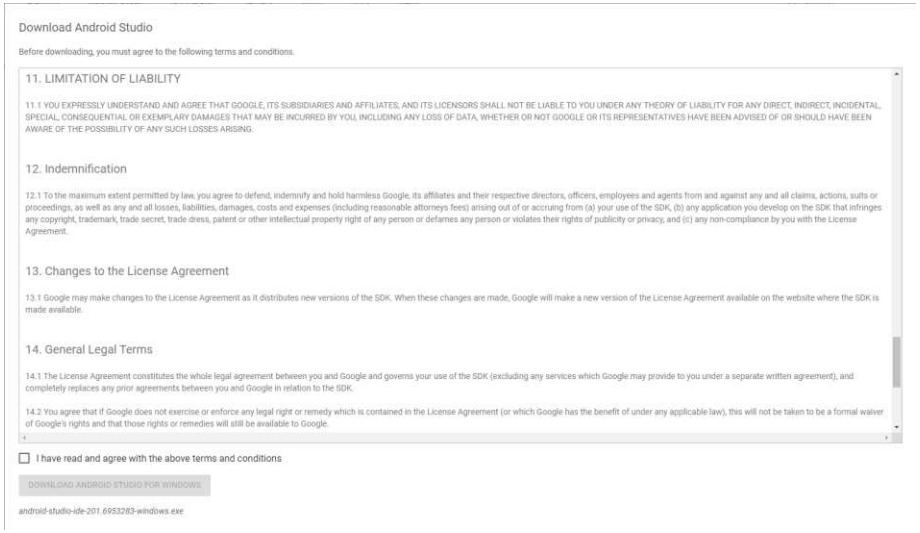
Google Search of Android Studio

Click it and it will bring you to the homepage of Android Studio for Android Developers. This is not only the place to download the installer we need but also to read up on newest features as well as the docs. These provide a lot of great examples and guides to using Android Studio which are essential in my opinion if you want to use this as your main development environment. For us it can be remarkably interesting but is not essential, because we are just using it for the SDK, NDK and Dev-Tools.



Download button for Android Studio

For us just hit the Download Android Studio Button. This will open a popup where you can read the license agreement and other information, click the checkbox and the installer will be downloaded.



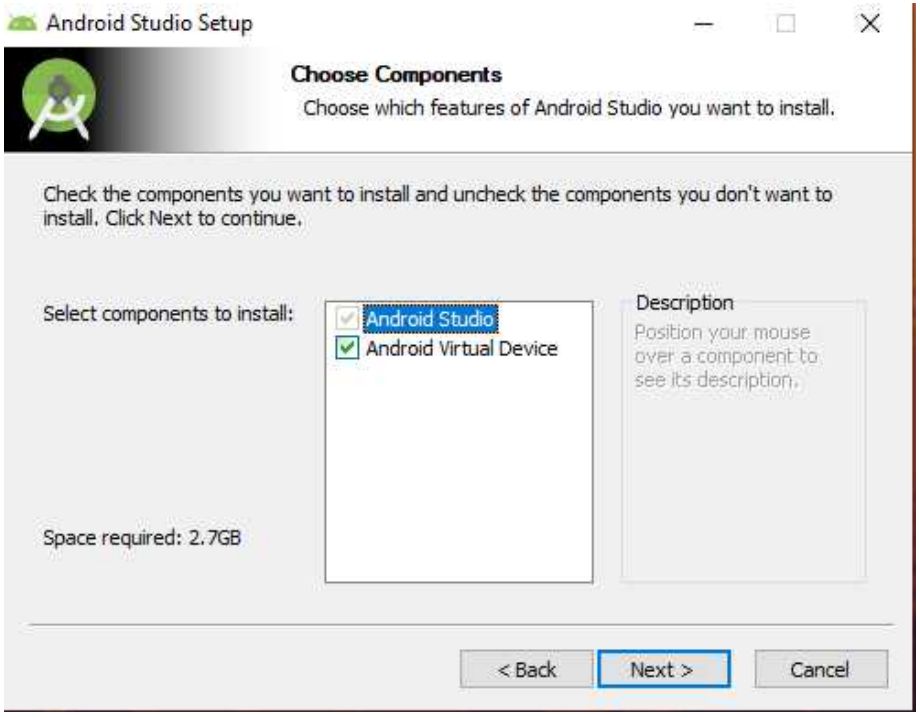
License Agreement of Android Studio and its Tools, especially important that you read this.

If you have done this the 600+ MB installer will be downloaded. This not only seems large for an installer, but it is. And even after having downloaded it several times I still cannot figure it out what causes this servery large size.

I know this is a bit of a tangent but I hope you still can keep track with me. This is also a good point to warn you about my habit of rambling about topics and subject. I hope this will not take you completely out of the book, and if you want you can skip these parts whenever you want.

Android Studio Installer

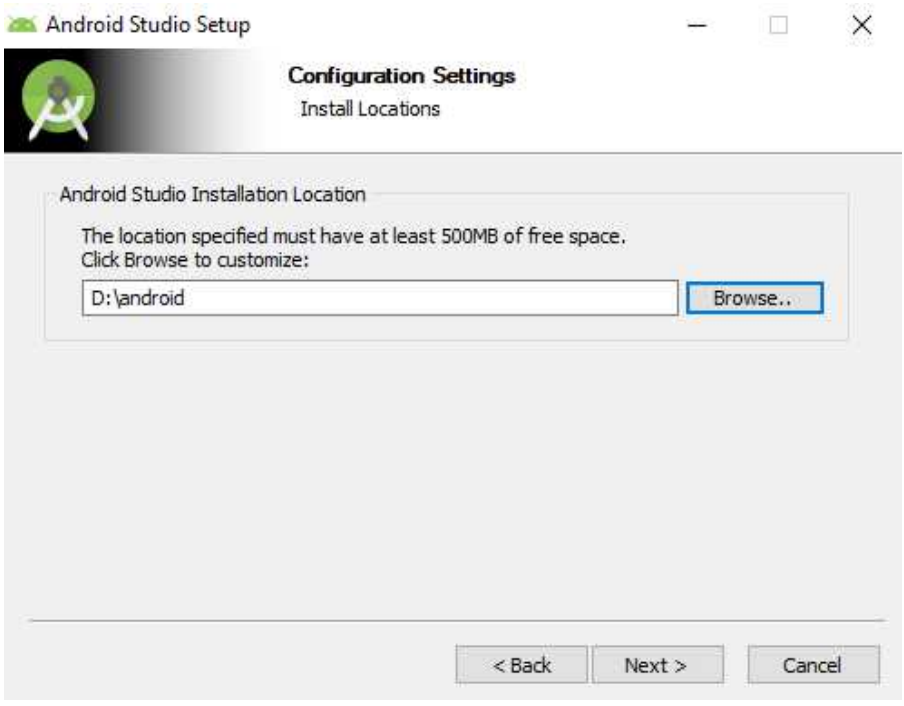
If you open the installer and start the installation you will be greeted by an opening page with some text about the installer and what is going to be installed. But this is of no concern to us.



Here the only real choice is to install both. I suspect that there are going to be some people that do not want the Virtual Device, but in my opinion, it is one of the best feature's Android Studio comes with and it is one which we are going to use regularly in our Android development.

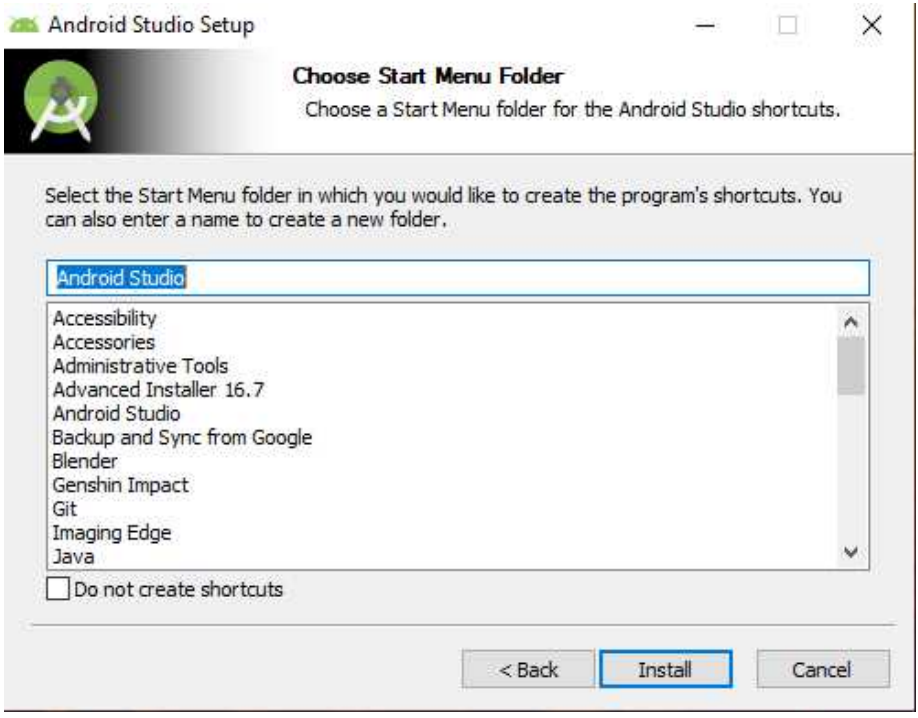
Basically, it is an emulator for Android. And a powerful one. I would even go so far as to call it the Android emulator, there are other ones out there, but they lack the support and the feature spec of this one. If your choice which checkboxes to check then hit next and continue.

Choosing the installation path is next up, here you are free to install it wherever you want. I would recommend installing it in a place where you will know where to find it. For me this is my D:\\ Drive with a folder on it. Remember that Android Studio must be installed in an empty folder. Why can't they just create their own empty folder? Well, I do not know, but we must provide one. And for me this is going to be just android.



Done with the installation folder, then hit next and continue.

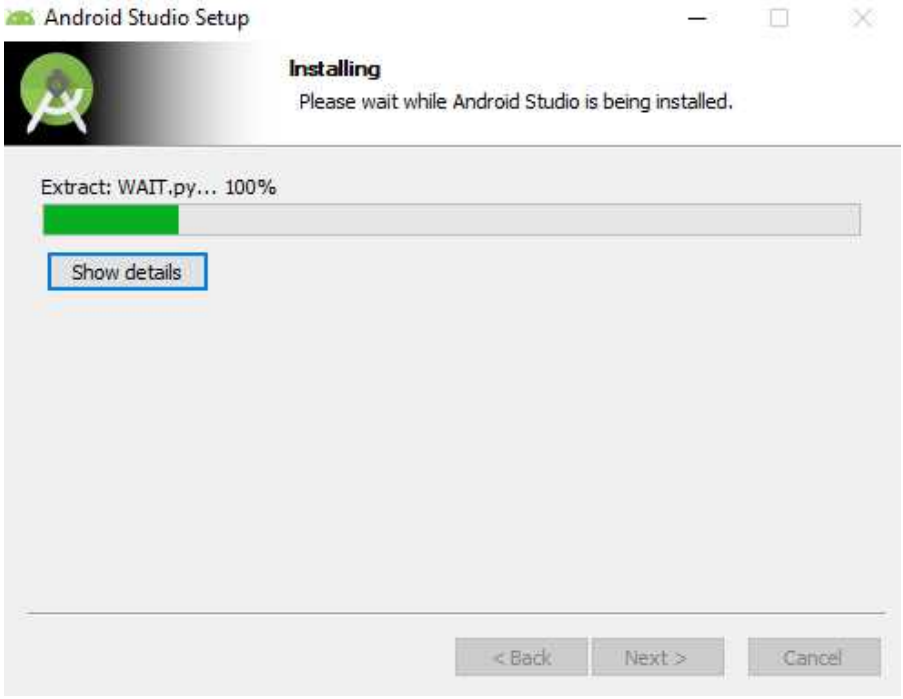
Next up we will have the choice if we do not want to create a desktop shortcut. I want one so I leave it as it is, but if you do not want one then to check the box. Done with this click next.



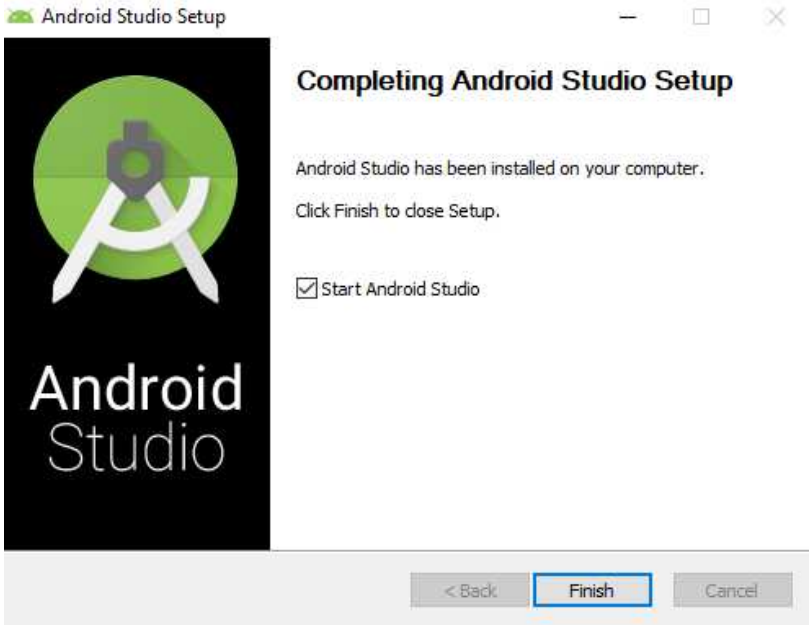
Android Wizard Start Menu and Shortcuts

This will bring us to the progress bar where we can watch the installation take place. You can open the show details button, to have a better overview of what is being installed at what time.

This might not be so interesting as this installation is rather quick. And when it is done hit next and we will be bought to the last page of the installation.



When you see this page, you are done with the installation and you can open Android Studio if you want to, but it is not needed.

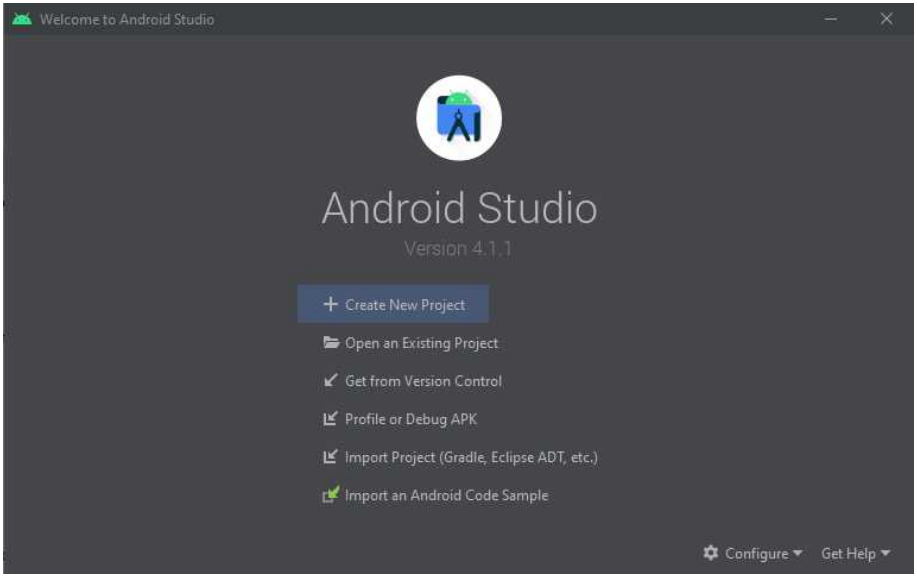


Compared to the Qt installation this is quick and straight forward. We were not even presented with a lot of options to choose from. And the actual size needed to be installed was not too big.

Unfortunately, we are not done with installing everything related to Android. So, we need to do that now.

21.4 Configuring the Android SDK, NDK and Development Tools

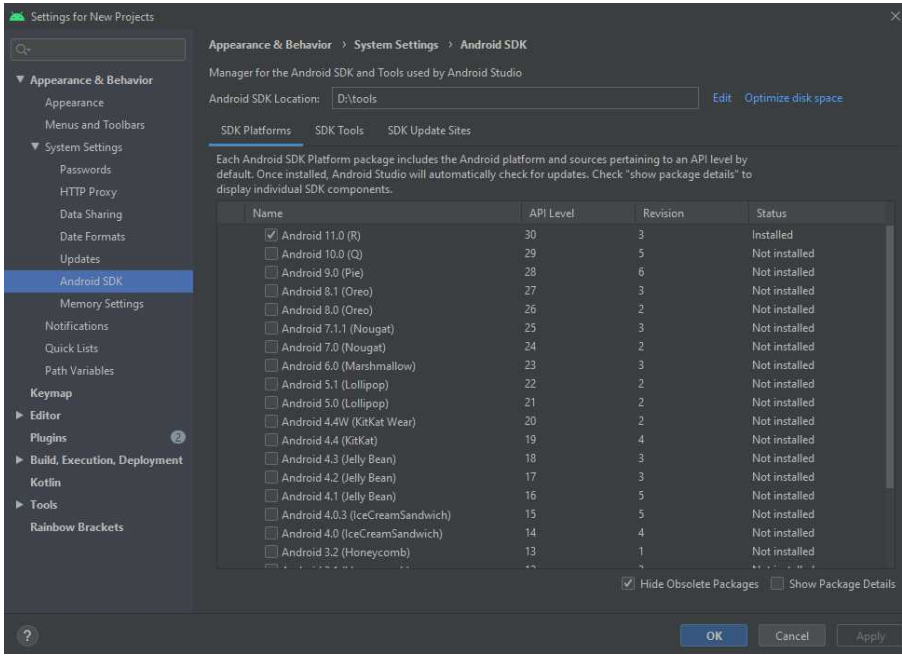
For this we need to open Android Studio[5]. If this is your first-time opening Android Studio you will be presented by this page.



Welcome Page to Android Studio

If you see anything else, or it requires you to install something then follow the on-screen information. When you are all court up with where we are now, you want to open the SDK Manager. This can be done a multitude of ways. One being on the screen right here when you open configure and from there open the SDK Manager. There are also ways to get there if you have a project open.

[5] In this book I used Android Studio 4.1.2, but if you have any newer version you should use that one

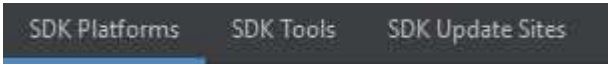


SDK Manager, here you can manage your SDK Platforms and Tools

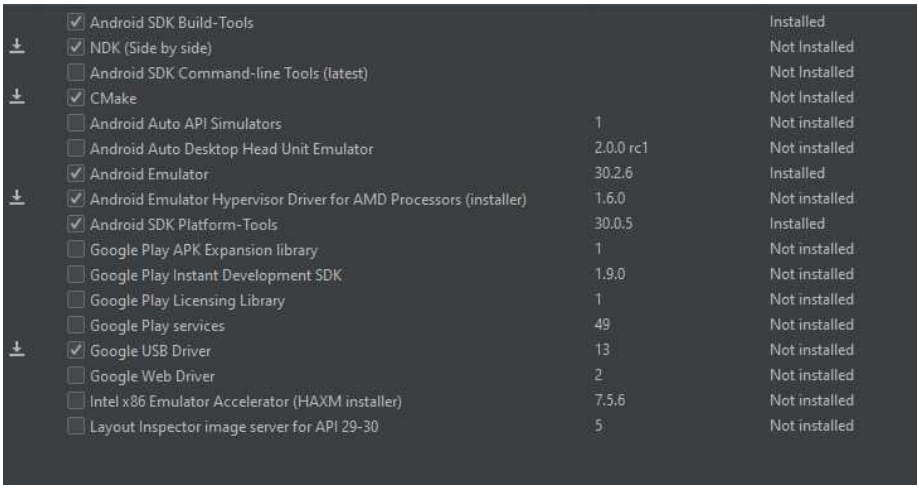
When you opened the SDK Manger[6] you will be presented with this view. Here we need to do a few things. First of if there is no checkbox active for the newest or any of the SDK Platforms you see open right now, you need to check one. I would always recommend opening the newest one that is released, as it is the most supported, feature right version out there. It is also recommended to choose an incredibly old Version, so you can test how low your Application can get version wise.

But choose a Version that is supported by Qt, and that is not so old. I tend to use one or two versions behind the newest version available. That is at least a good way of doing it in my opinion. Next, check that and move to the SDK Tools tap up at the top

[6] The SDK Manger changes from time to time so do not feel confused when the Screenshot does not match what you see



Here we need check a few things to listen carefully.



The tools we need, select all of them

The tools selected are needed, so check them, if some of them are not visible you might want to have a look at if you disable hide Obsolete Packages, sometimes Android Studio puts old packages there. But to explain why we need these packages.

- **Android Studio SDK Build-Tools**
  Obvious as to why we need it. We are going to build Android apps and for that we need the Build tools.

- **NDK (Side by Side)**
  This is a bit trickier to explain, if already worked with Android Studio you might be wandering why we need this, well Qt requires this to build for Android. In the future this might be alleviated, but I strongly suggest just always installing it. It is not so large, so it will not eat up a lot of space.

- **CMake**
  As we are building applications using C++ this is also a no brainer. Qt does not require it to function, but it will come in handy later, so download this

- **Android Emulator**
  The nicest thing Android Studio provides. This is basically like having your own Android phone on your desktop it is extremely powerful and great to use. There are problems with it when you want to use it on an PC that has an AMD Processor, but we will get to that.

- **Android Emulator Hypervisor Driver for AMD Processors (installer)**
  As already mentioned, Android Emulator has its problems with AMD CPUs, these are very unpleasant and, in my opinion, a complete oversight from Androids part. You basically cannot run Android Emulator if you have an AMD CPU. You can only run it when you followed a guide from Google that explains how you need to turn some features on and some of for the AMD CPUs. If you do not have an AMD CPU then you do not need to select this, for me it is essential as I run an AMD CPU in my machine.

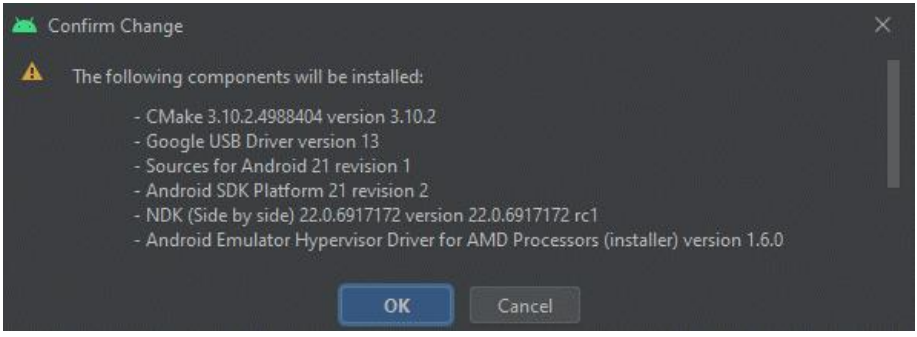
- **Android SDK Platform-Tools**
  These are the basic tools need by Android Studio to build run and deploy Android applications, they are needed also for Qt so select them.

- **Google USB Driver**
  If cannot use the emulator or you do not want to you can also deploy your applications written in Qt or Android

Studio to your phone using a USB connection. We will be needing this so select it.

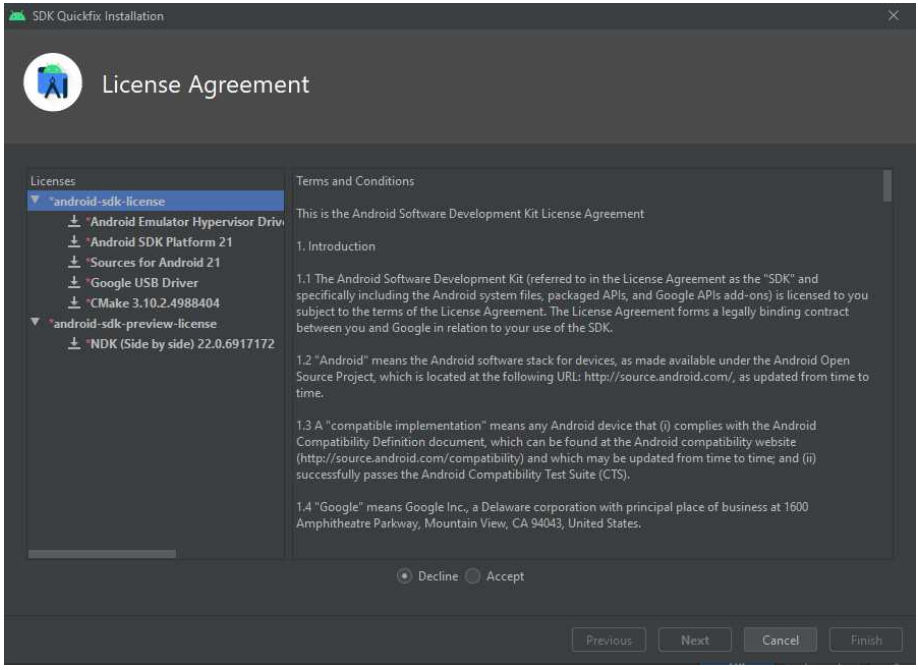
When you have selected all the packages you want you to have only one thing you need. And that is to change where Android Studio downloads and Installs these packages to. We need to link Qt to these packages, so best practise is to pick a drive and a folder that you can remember, and one that does not have any spaces or special characters in it. If you have these it can create problems later in Qt when you want to deploy your application.

If you followed everything we did so far and selected everything I mentioned and selected a suitable folder you can click OK. This will open a popup that lists all the components you want to install. Check if everything is as it should be and then click OK again.



Popup with the changes we want

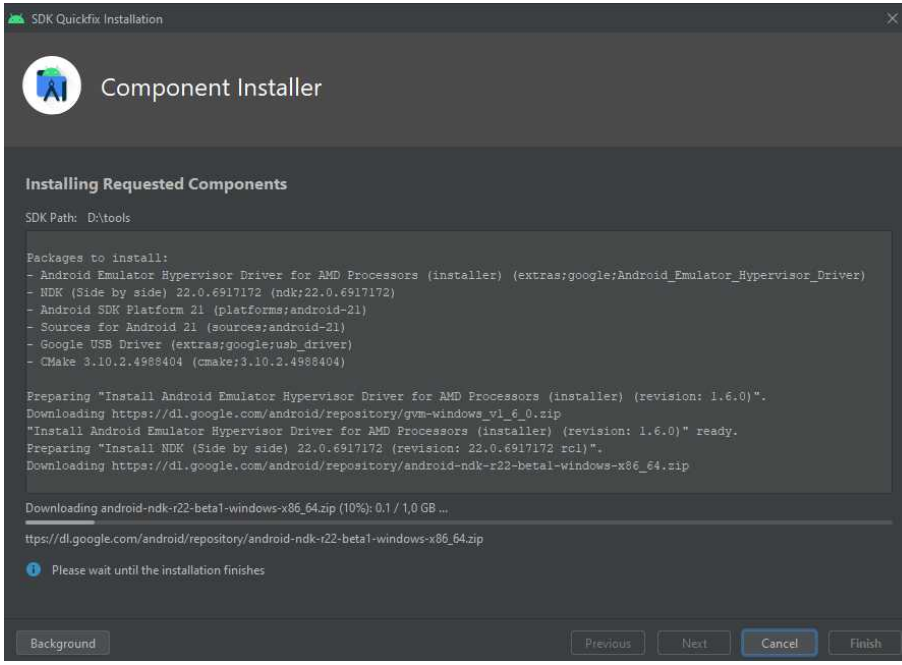
Next you will be bombarded with license agreements you need to check all of them. You can only accept these license agreements and terms of use. You do not really have a choice here so you can only accept them.



Important License Agreements

There are some interesting points in these agreements and if you have the time and patience you may want to read up on them. But for what we are doing, and what we are going to use Android Studio for, you should not need to read this.

But as always you might want to read up on them when you want to publish an application and you are not sure if what you want to do is ok.

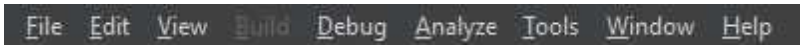


Progress bar of the installation process

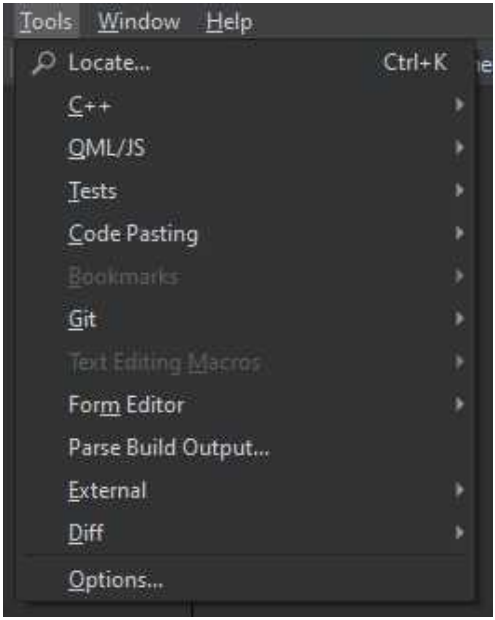
This will then start the installer which will download all the files you need and install them. You can just leave this running. Depending on how good your internet connection is this can take a little bit of time.

When the installation is finished you can hit finished and the SDK Manger will close. And with that you are done with the installing of the Android Tools, SDKs and NDKs we need.

Now we need to link them to Qt to use them in our development. So, open Qt Creator, and go with the mouse to the Top Bar.



When you opened Qt Creator you need to go to Tools, if you hover over it you will be presented with a Menu, from there you can select the last option which will be the Options. Open the Options up.

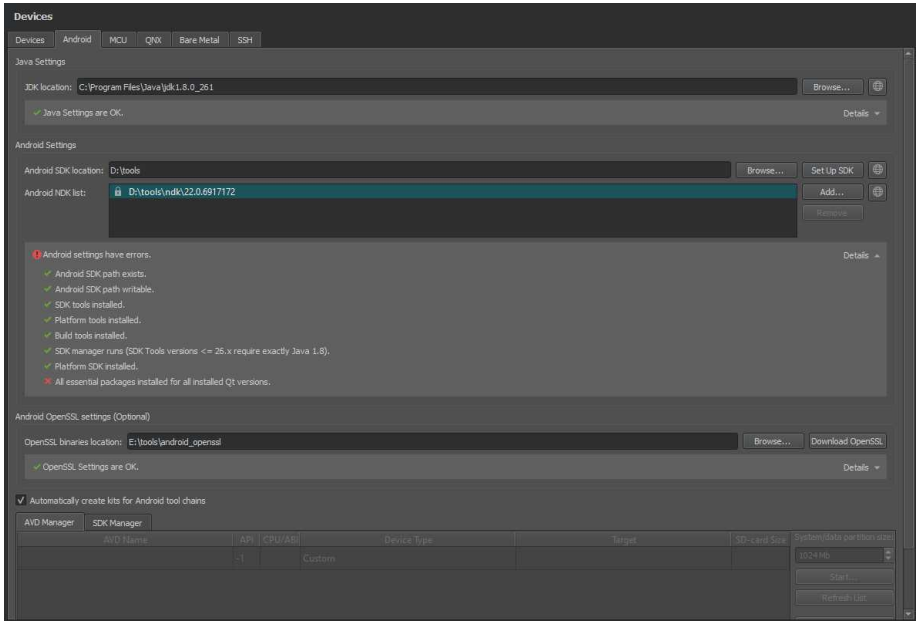


Here you need to go to the Devices tap on the left and open it up.

Also under Tools are a lot of good tools that you can use in your development.

For instance, the Git tap is one that I use quit a lot from time to time. We are not going to use this in this book.

But we are going to setup and use Git, and to be exact Git Bash.



Device configuration in Qt Creator Settings

Then you need to go to the Android tap. Here you need to put in the following. First of at the top you need to add the JDK location. If you do not have JDK on your device follow one of the tutorials out there, or by the guide from my website.

After that you need to specify the SDK location of our Android SDK. If you successful set up everything all the items in the drop down will be checked green, and that will be it for the installation and linking of the SDKs, NDKs and tools we needed. As a sidenote sometimes Qt has problems checking for all the files that you have installed. In the screenshot up above, you can even see it. This is nothing bad. You can still develop your apps like this. And the problem will disappear given enough time[7].

Also important is the fact that you should always keep the things we just downloaded new and up to date. This will minimise the bugs and problems that might occur when developing.

And with we are done with the installation of all the software as well as setup we needed, now we can start the actual coding so read along. If you are still a little bit confused and or wondering what we are going to use and what everything is supposed to do, do not worry as this will become clearer as we go along.

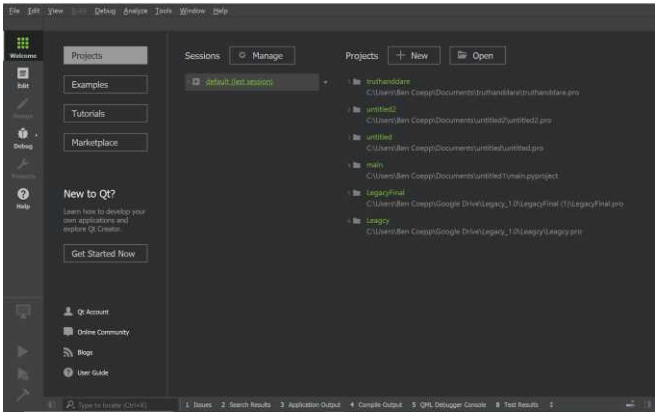
[7] At least that was the case in my experience

2.2 First Baby Steps with Qt

As with any new language and or framework the first thing you will probably do is write a *Hello World* app. And that is the first thing we are going to do. First of we want to verify if our installation and setup are correct and complete and that we did everything as intended. And secondly, I am going to use this as a starting point for the main part of this book.

So, if you have followed the steps until now you are ready to continue, if you still encounter as problem or are not able to do the next steps then go throw the last few chapters or go to my website *bencoepp.io* there you can find a link to my Git Hub where you have the files needed to set everything up. This should allow you to at least follow along.

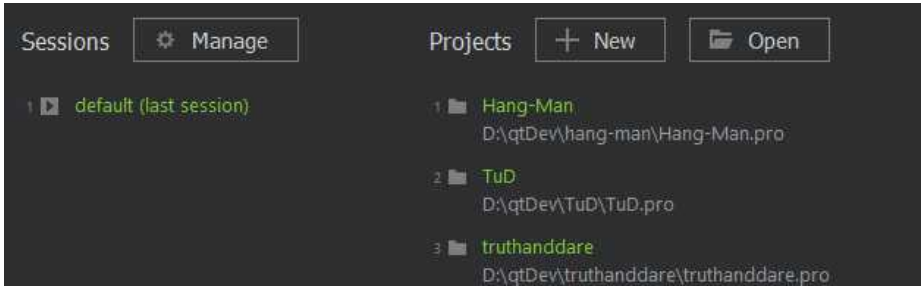
The first thing is opening Qt Creator which we previously installed on our machine. If you are like me and you are itching to jump right into making our first few baby steps. For now, just click on the New Project button that you see or go to files new project and click on it.



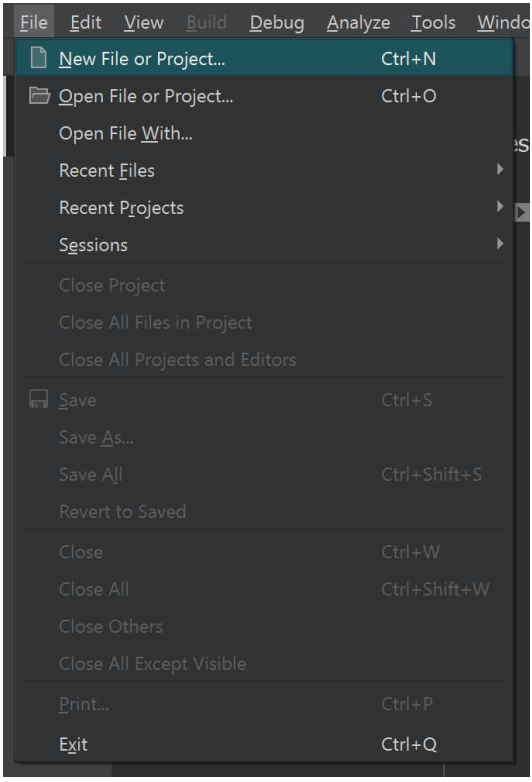
Qt Creator open on Project Tap

You can also see here your recently opened project, as well as a top on the left for your examples and tutorials. Also, the marketplace can be found there.

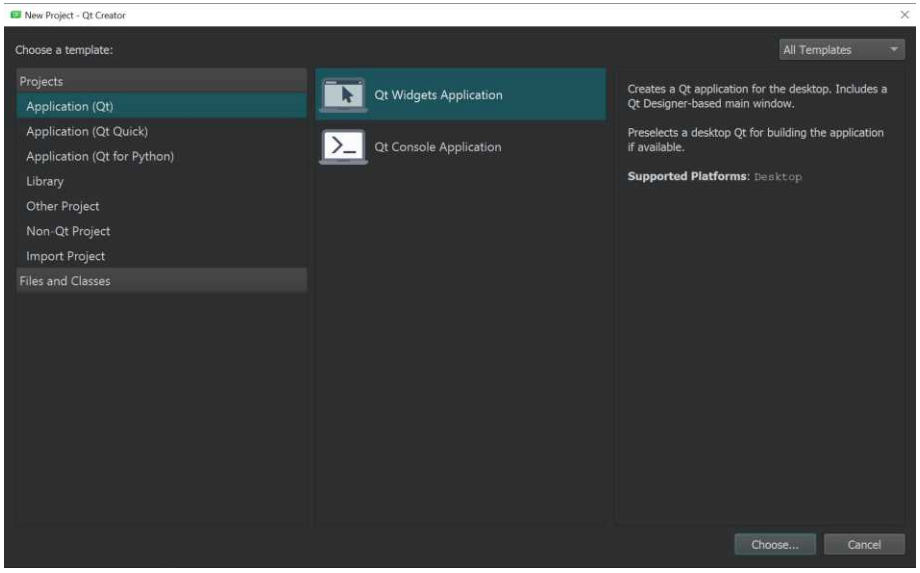
Most interesting in my opinion are the examples and the tutorials these can greatly add to your knowledge in Qt and help you a lot in learning components and features.



Projects New and Open Button

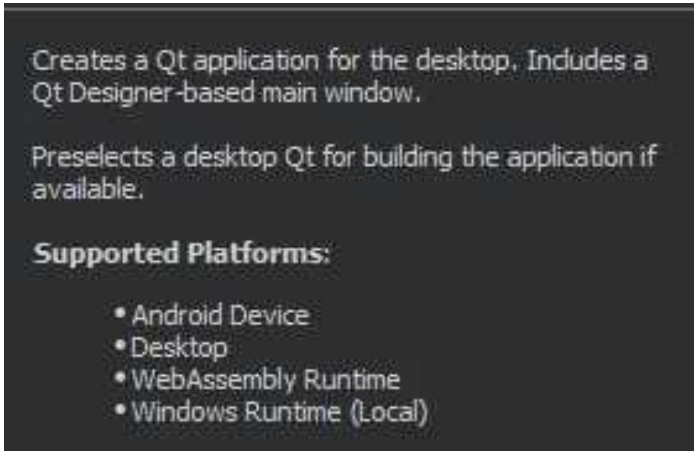


This will open a popup which is like a Wizard for creating our project. The steps are not so difficult, and you probably could go throw them yourself but, for the first time we are going to do this together. This will also be covered in a later chapter, where I go over more crucial steps in setting up a project which right now would only confuse you.



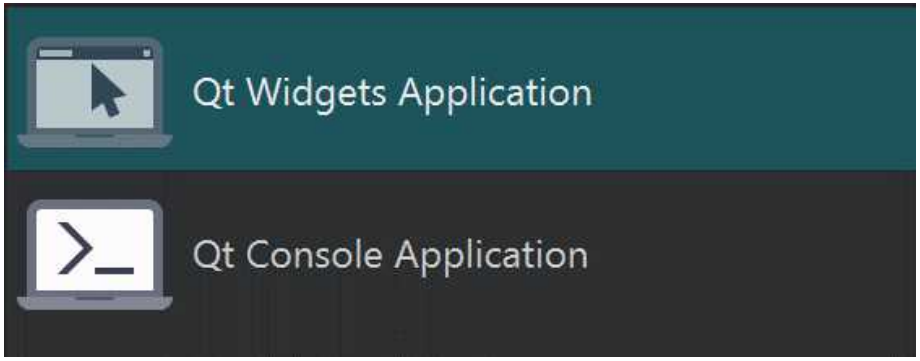
New Project Wizard

The first thing you can have a lock at when the wizard opens is the different project templates that Qt provides and comes with[8]. Some of them are not as useful as others but the ones that are worth remembering I will point out now.



[8] There are a bunch of very specialised templates, you should check them out and if you find something that fits your workflow then you can use it

First of the Qt applications.



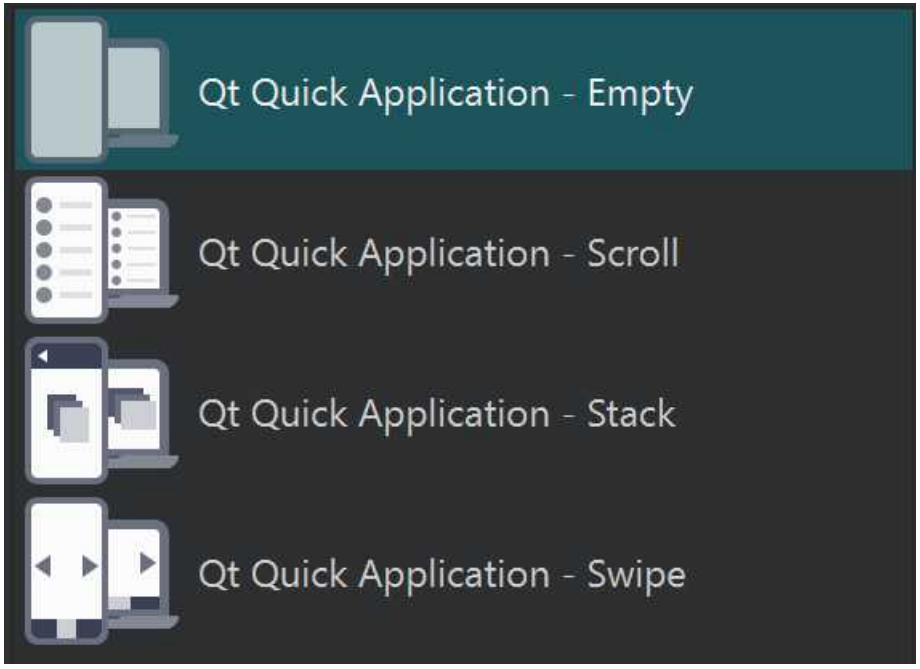
Application (Qt)

These are the standard Qt C++ Applications you can find. The first is a Widget Applications, so simply a basic and native GUI applications that provides a style file and a C++ backend for development. The one below that is a console application, it has all the things included that you would need to build a terminal and or console application. I used the latter of the two multiple times already, trying to build my own Git Terminal and a simple Tetris game in C++ as well as a snake game. But its use cases shine brightest not throw games but throw more developer and workflow related tasks.

We will not be covering this in here in this book. There are a lot of great tutorials as well as learning resources out there how to do this. And if you wanted to go into learning about the C++ functionality Qt has to offer this would be the templates I would choose. As Qt is a C++ framework you can expect there to be a lot of power and functionality under the hood.

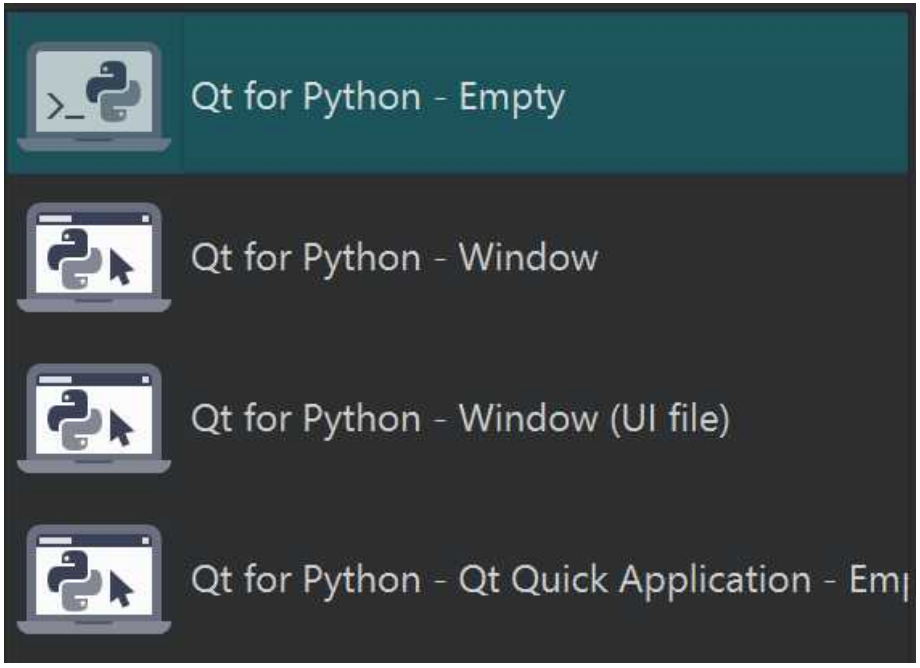
And if you want to truly master and understand Qt and all of its aspects you will also need to learn the C++ side of Qt.

Next up is Qt Quick Applications.



Application (Qt Quick)

Here you have a three wide and especially useful Project Templates, ranging from a Scroll and Swipe Template, which are excellent for trying out these functionalities or adding to them, which you are going to do all the time because they are one of the most essential components Qt provides. The Stack Template is also especially useful, but for the way I use Stack View most of the time not usable. And lastly, we have an Empty Qt Quick Application. It comes with all the elements and files that make up the most basic working Qt Quick Application, and for me is always the starting point for a new project.

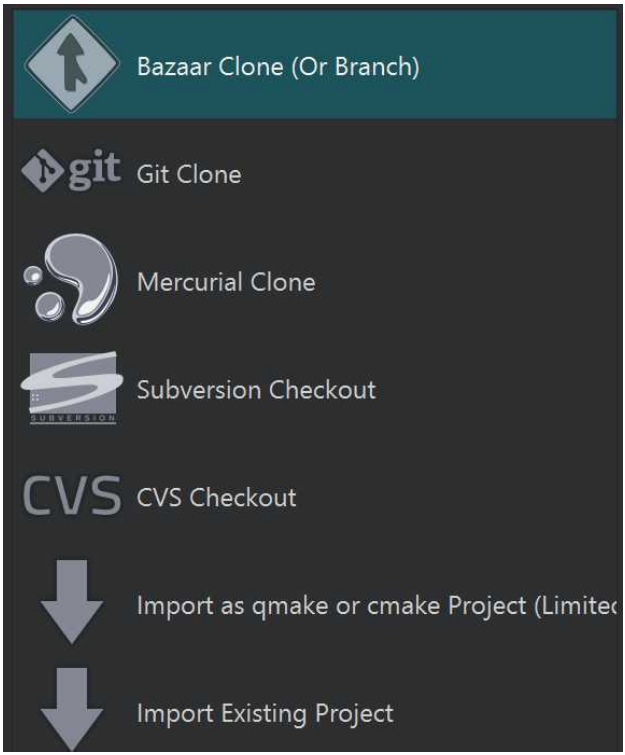


Qt for Python Templates

Next are the Python Templates which I did not have the pleasure of trying out. But they are three provided templates, the first is an empty project that only contains a window component. The next one has the same but in a UI file like the Widgets files you find in the standard Qt Widget Applications.

And lastly, we have the basic Empty Qt Quick Application in the Python version. It is nearly identical to the Qt Quick version but having not a C++ backend but a Python one.

If you are someone that is used to Python you can immediately jump right into it.



VSC Templates

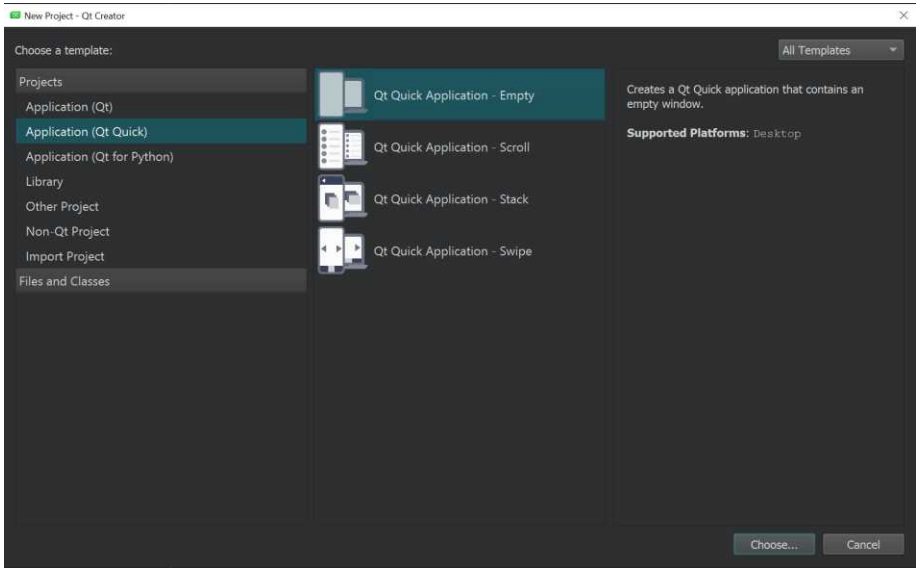
And lastly, we have these quite different but always extremely useful templates. First of you have multiple different options for cloning a project from a Version Control System, like Git. I usually do not use this because of my workflow but I have seen a lot of people that use this.

And it saves you from opening a terminal so there you go. And lastly you have the two options to open an existing project.

For us we want a Qt Quick Application, it is in my opinion the best way of making applications with Qt, and at least in my opinion is the best way of building any application no matter what framework you use. It is the newest, very feature rich and has all the functionality you need to make any application possible. In Qt Quick Applications you mainly write the UI (User interface) using QML a language mainly used in Qt. It allows the creation of highly stylised and animated UIs and applications. And it is my favourite thing to program with, which you will hear quite a bit over the next hundreds of Pages.

We now have the choice between a lot of different templates, which we already talked about. For our HelloWorld Application

the best option is the Empty Template[9], everything else is absolutely overkill. So, select the Empty Template and click Choose.



New Project Wizard QtQuick Templates

Also, templates are not as important as you might think. They basically only provide you with a little bit more boilerplate code right from the get-go, but you can also create this on your own with just a few minutes of work. But if you are a beginner then a quick look into how these templates work, and how they use the components can be a particularly good learning experience so you might want to do it.

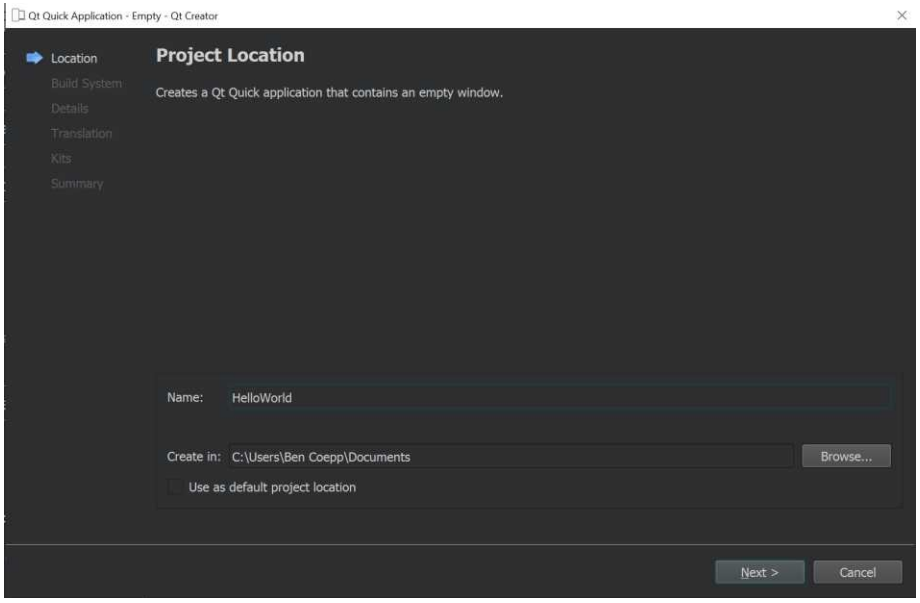
When you continue you can choose the project location? This is an important step of the setup, and maybe not for the reasons you think.

Choosing the name of the project is the easiest part and totally up to you. I would recommend to having any spaces and special

[9] We do not require anything in terms of prebuild Components, but if you want to you can check out what the other Templates have to offer

characters in the name. This also extends to the location you save it in. This is not that important anymore, but a few versions back from Qt 6 there was a problem that you were too able to deploy your application using windeployqt[10]. Therefore, I would always recommend having no spaces or special characters in the name or the location. But if you just want to develop an application or you are not interested in deploying it then feel free to name and place it wherever you want.

When you typed a name for your app and selected the location where you want to save it hit the Next Button and continue.

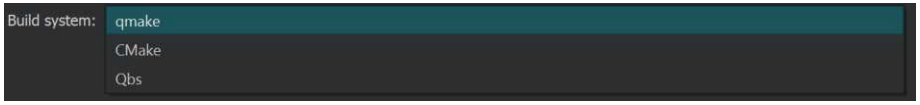


Project Location Page in New Project Wizard

This is a somewhat crucial step in a creating a new application, choosing the Build system. You have the option between,

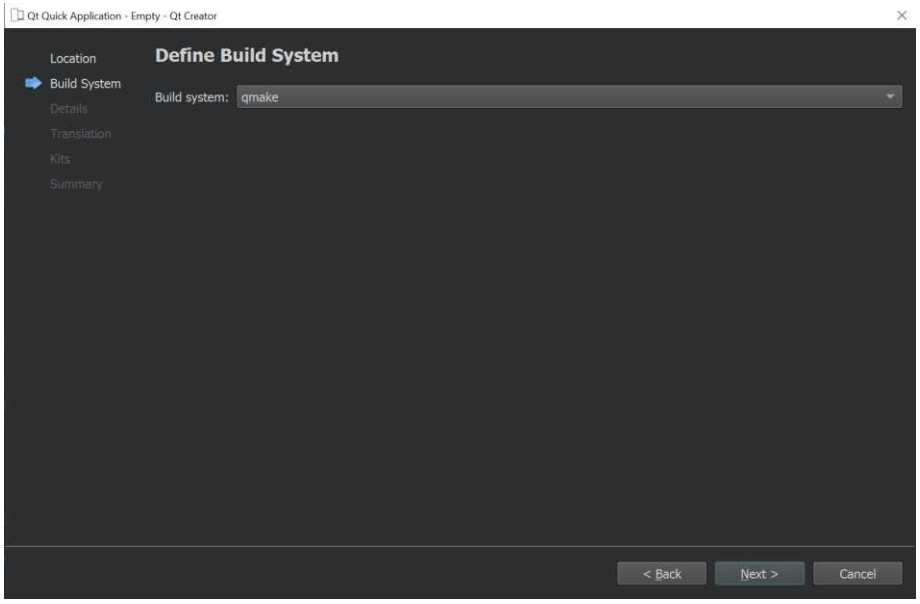
[10] This is a tool created by the Qt Company that helps developers deploy applications to the Windows platform

qmake[11], CMake[12] and Qbs. They all have their separate reasons for use and depending on your type of applications you want to make. And in later chapters we are going to talk about the different benefits for using one over the other.



Different Build System Qt provides

We are going to use qmake here, but all other options are also viable here. Also, CMake is a particularly good choice, as we are built C++ applications and CMake is one of the standard build systems for these.



New Project Wizard Build System

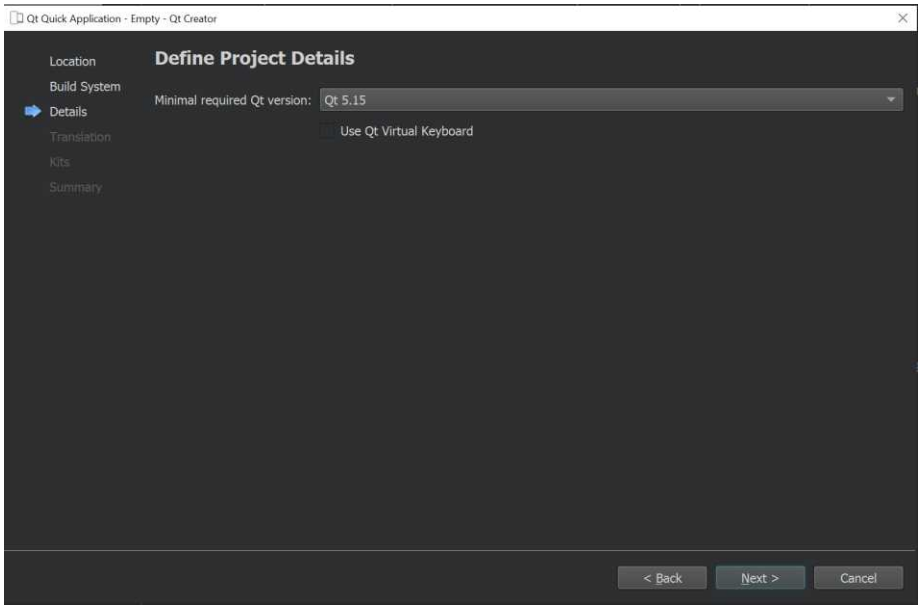
[11] a utility that automates the generation of makefiles tailored to the platform where it is run from

[12] cross-platform free and open-source software for build automation, testing and packaging using a compiler-independent method

When you have chosen your desired build system, for us we are going to use qmake as it is the standard and default for this.

Next up the Qt versions we want to use. For us and in general you should always use the newest version available. For us, this Qt 6[13]. And I would usually keep it as that.

Down below you have the option to use the Qt Virtual Keyboard. right now, it is not needed, so leave it unchecked. And hit Next when you are ready.

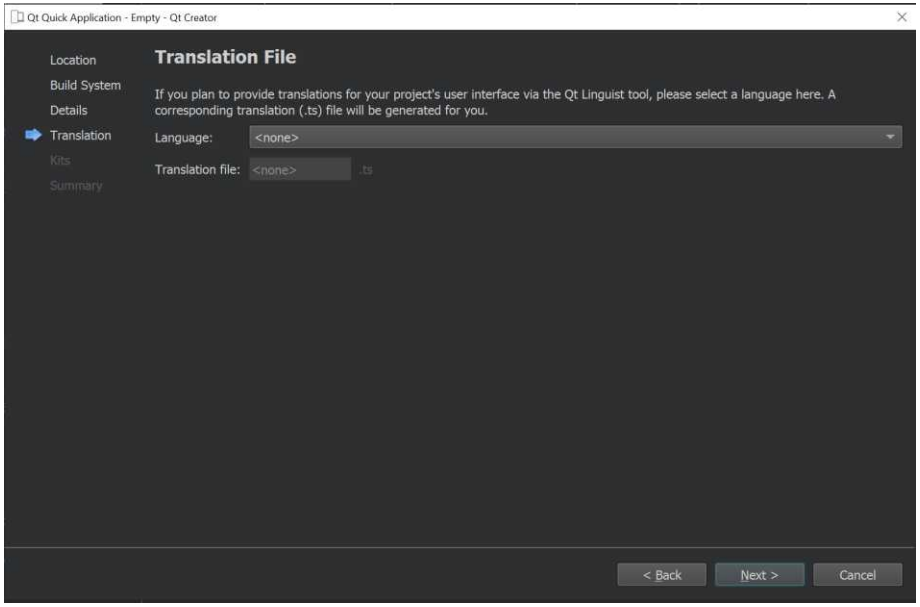


For some reason, the minimal required Qt Version is Qt 5.15, this is also fine, but if you have the option for Qt 6 then choose it

Next up we have the option to add a translation file. This allows us to give the user the choice between different languages. As the users of an application are generally not from the same country. Having multiple languages can be especially useful.

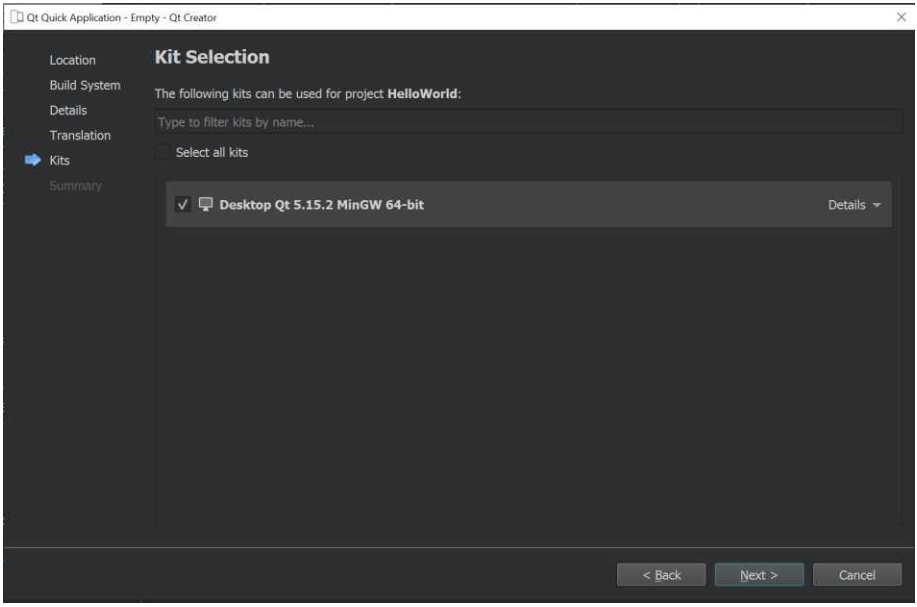
[13] As of time of writing this book it is the newest version available, if you have a newer version then use that, but it is not as important as you might think. The content in this book works no matter the version you are running

But right now, it is not needed. So, we are going to skip this. Hit Next and continue.



Kits are the different platforms you can build your application for. For us know we only want to build it on desktop but there are a lot more platforms out there, like **UWP**, **Android**, **MSVC** and of course Apple/IOS. Later in the book we are going to use a lot of different Kits but right now we are only using MinGW 64-bit/32-bit Kit. It is the basic kit for developing desktop applications. If you have selected it, you can hit Next and continue.

We will stick with this Kit for most of the tutorials and content in this book, if you are on Mac or on Linux, you can also choose another kit if you want to, but for me I will stick to MinGW 64-bit from here on out.



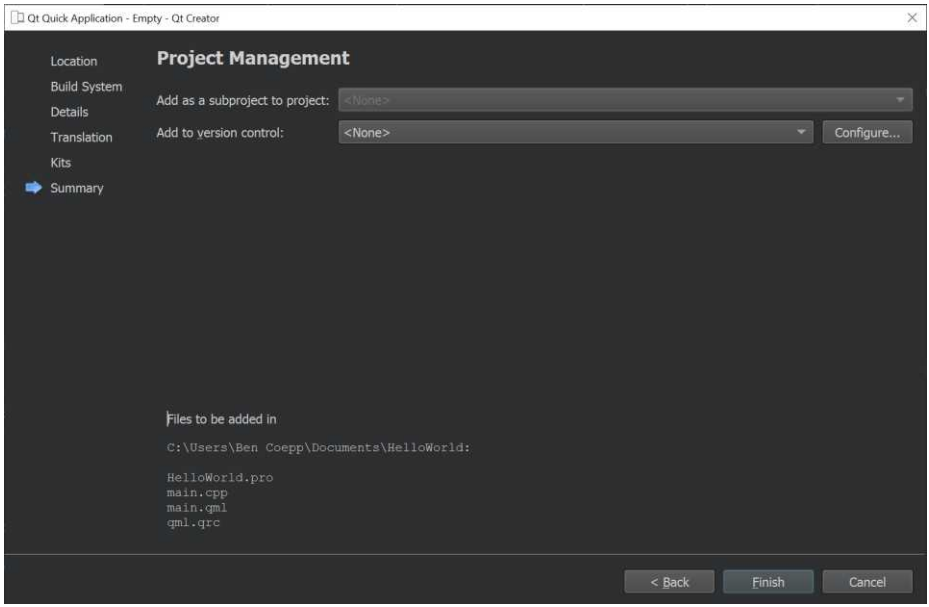
New Project Wizard Kit Selection

Lastly you have the option to add a Version Control System to your project.

This as a side node will be an especially useful thing in the future. If you build anything larger than a calculator that can be built in a few days, you should use one. Reasons for it are a lot but basically you can keep track of all your changes and edits and even if you destroy your entire project you can go back to a functioning version of your application.

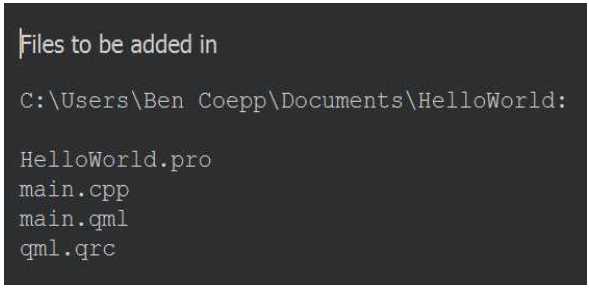
To admit I had completely destroyed some applications I had written, and I did not have VCS on them, so the entire application was more or less completely lost, which was a real shame. So be aware that if you do not use this, you would have a hard time getting your application back if you deleted it or even worse broke it.

So later, I am going to explain how to use it and how to best set it up. But for a tiny Hello World application we do not need Version Control. So hit finish.



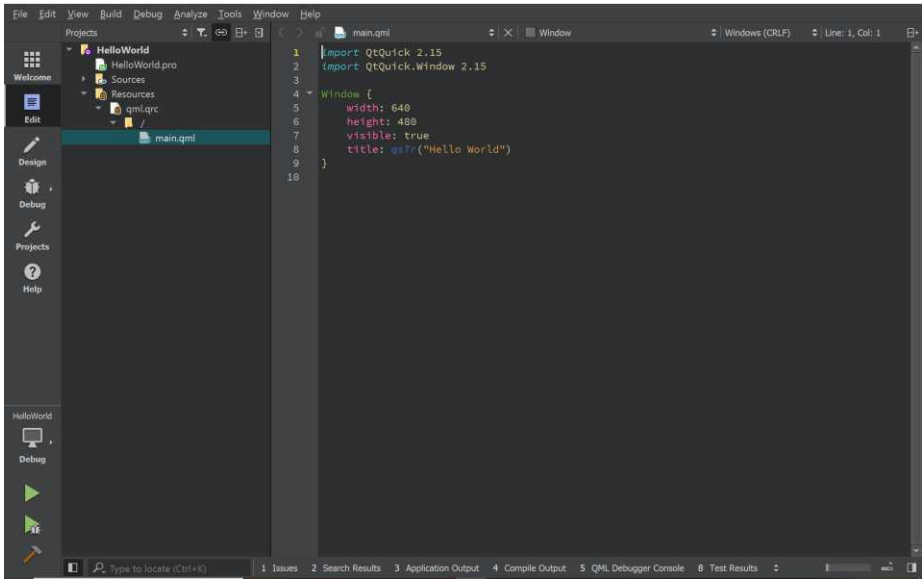
New Project Wizard Project Management

Here you can see the files that are going to be created when you hit Finish. We are going to go over the created files later so stay tuned.



When you hit Finish the project is created and it will be opened. Next up we are going over what was created and how everything works.

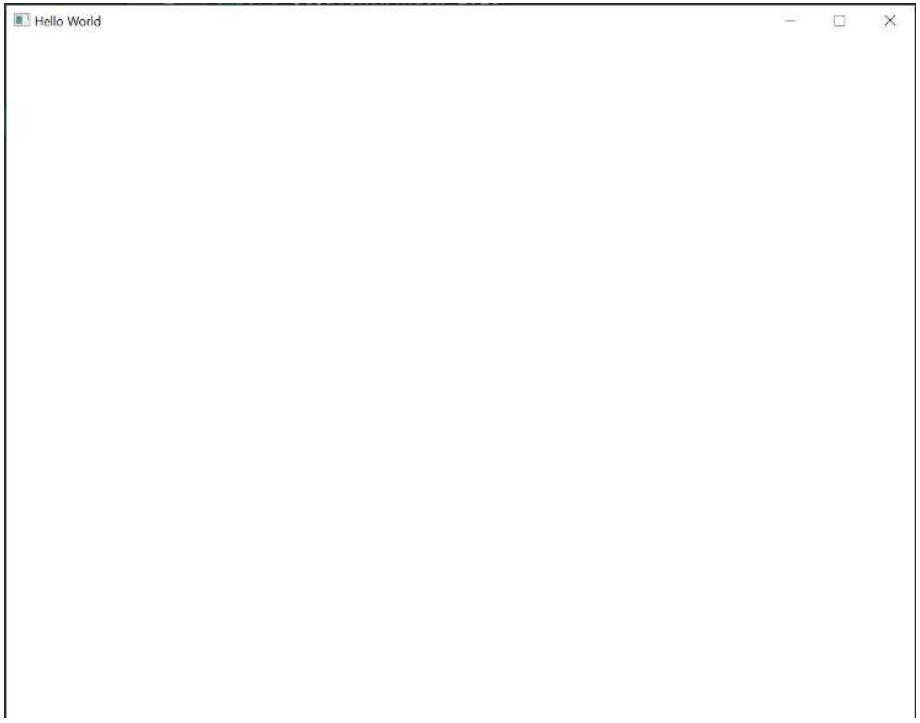
If you have larger project setups and or have multiple Sub-Projects then this can be interesting as you can see here how many files you created, what the name of the file is and what type if file is.



View of an Open Project in editor Tap

Now that we have a project the first thing, we should be doing is running it. There is no sense in programming an application if it will not even run. To test if it runs click on the green Arrow in the bottom left.

If you do not want to debug your application, then click the button without the little Bug beside it. If everything works out fine and all the setup, we did was correct and functional then the application will run, and a window will pop open.



Running HelloWorld application

For now, it is only an empty window with a title that says Hello World. Nothing fancy but the perfect starting point to learn from.

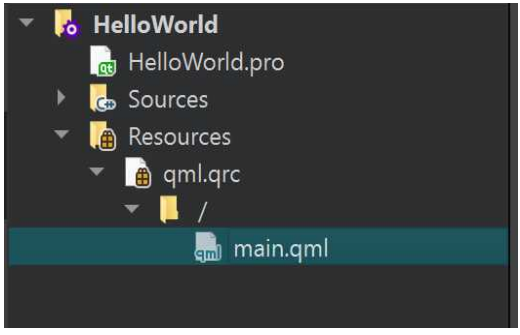
As a side note (you can skip this if you want to), why do you think people often use HelloWorld applications to start off as there first program? Well, I can give you the answer, first of it was one of if not the first program ever run by humans. And secondly it is the simplest program you can more or less create. Basically, the only function of a HelloWorld program is to output or display HelloWorld. It is precise, simple and you can immediately test if your application runs. Also, it will teach you the fundamentals of the programming language you want to learn.

2.3 Explaining the Basics

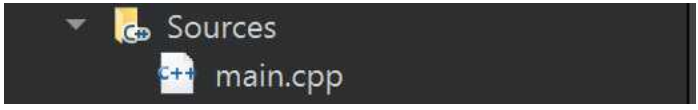
How that we have everything set up and we created our first Hello World application we should have a look at how a Qt application even works and what each part means and works.

2.3.1 Project Structure

Project structure is one of the topics where not everyone will be on the same page, and there are a lot of different ways of during it properly and we are going to go over all the common and recommended ways, as well as some which have their benefits in specific cases.



Generally, Qt apps are structured as follows. At the top you have the .pro file this is like the project settings file and the basic controlling file that handles the building of the project.



Below that you have your sources. Here you will find all your C++ files, scripts, and the like. In an empty project there will only be a main.cpp file in it. This is the heart and soul of a Qt application. This is the link between our qml files and the C++

backend that does the actual displaying of our application.

```cpp
#include <QGuiApplication>
#include <QQmlApplicationEngine>

int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);

    QGuiApplication app(argc, argv);

    QQmlApplicationEngine engine;
    const QUrl url(QStringLiteral("qrc:/main.qml"));
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,
                     &app, [url](QObject *obj, const QUrl &objUrl) {
        if (!obj && url == objUrl)
            QCoreApplication::exit(-1);
    }, Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}
```

Content of main.cpp

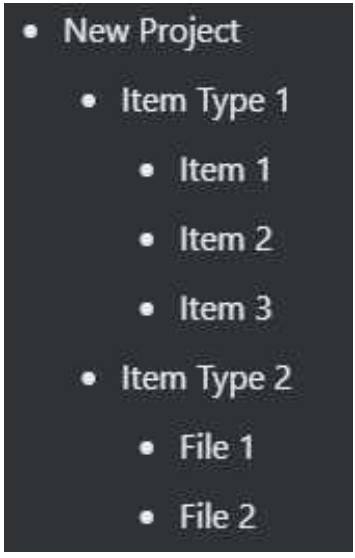
For now, there is only the default setup in it, that lunches the QQmlApplicationEngine and gives it our main.qml File as its starting point. And that is all that you can really see of the Qt backend. The rest is shrouded in mystery. Just kidding we are going to go over that later, but for now you only need to remember that you probably will not need to go in here at all and even if only to add and or change one or two lines. The Rest can be left as it is. Below our sources you will find the Resources.

Empty qml.qrc Folder

This is the place where you will find the actual files of your application. Here you will place your QML files that make up your app as well as Images, Icons and JavaScript files. Overall, you can say that everything that makes up your app goes here.

2.3.2 App Structure

App structure is less about the files and how they are organised and more about how you use them and work with them. As already mentioned, you will work most of your time inside the Resources folder, which houses all your files and the like. This should also be clearly divided between QML files, Images / Icons and JavaScript files. Why you ask? Well, if make anything larger than a calculator you will have most likely over 10 QML files and even more other data. And this can be very overwhelming and confusing if you do not organise and structure them.



There are 2 main thought processes behind doing this. One is the way of making the organisation and structure is throw the file types. This can be helpful if you do not have a very component or Page based application. But it can also lead to you having hundreds of QML files unordered in one place. This sounds absurd and maybe should not be even considered as an option but believe me that there are companies out there that have such a non-existing project structure and just dump all their files with one type into a folder.

The other way is using a more component and element-based approach. Basically, it works by packing everything that is needed for one specific element, page and or section in one folder. This can be extremely helpful, as it is quite easy to

understand which files belong to which element in your app but depending on how large your application is going to get you will have hundreds or thousands of elements and components. A good example for this would be Instagram where you have sometimes up to a thousand posts in one list, and if you do not manage your components right and keep the resource needs in check the user will have quit the bad time. And that can also be very overwhelming. I still prefer this option over anything, as it is easy to get into and if you stick to it will make your life a little easier.

But no matter which structure you chose, we are going to use both in this book, just because there is no inherently better way and if you want to be a professional Qt developer you should probably be familiar with both ways.

2.3.3 How does Qt make an app out of this?

QML files are like a blueprint for Qt Quick Applications. When you type a component into it with all the necessary elements, Qt parses them and presides to display them. Here in our main.qml file you can see a single component.

```
1  import QtQuick 2.15
2  import QtQuick.Window 2.15
3
4  Window {
5      width: 640
6      height: 480
7      visible: true
8      title: qsTr("Hello World")
9  }
10
```

Simple Window component with imports

This Window component is the starting point if you want to make an application that has a window. It has a width and a

height as well as a visibility setting and a title. So, for now standard stuff and easy to understand.

Above the window component you have the imports that are needed for this component to work. You will always use QtQuick in your application. No matter the size or the specific components you will be using QtQuick is always needed[14]. Below that you find QtQuick.Window this is a specific QtQuick package that is needed for our window to work. There are a lot of different windows in Qt, which all have their own QtQuick import you need to use. But they all work the same way. They all have a

- **width**
- **height**
- **visibility**
- **title**

There might be more attributes or properties that are used differently or have a different effect.

This is the simplest application you can make in Qt. It is just a window with a title that is it. As you can see the actual application is only 5 lines big. And these lines are descriptive and easy to understand. This is the best thing in my opinion in QML, it is so easy to understand and read and you do not need to hassle around a lot of exceedingly difficult abstract topics or Words to make a simple window appear. You just type window give it a hight, with and all the other things you need and be done with it.

But if you think this is all you can do with it then you are mistaken. There is always more under the hood.

[14] At least for the common types of application and component it is essential

2.3.4 Structuring Tips and Tricks[15]

Qt is a bit complicated when it comes to organisation and structuring, and I would even go as far as saying there is no correct way of doing it. But there is the suggested way. Which should always be the way that you try to use when building your applications. There are reasons for using the suggested way.

There is a guide written on the Qt Docs that has a lot of tips, tricks, and best practises for Qt. you can just google for it and you will find it. The guide talks a lot about different topics and concepts in Qt and what to be aware of when building applications using Qt. It helped me quite a lot while building larger applications, so check it out.

The only tip or trick I would share with you is, that you first think before you build your application. It might be true that you can change the structure of your application later, but that takes quite a bit of time and has its own share of problems. So, think before you build, and you will not regret it. But that brings me to the second point. Do not be afraid of redoing your structure. Especially If you are working on a large project or something that will be sold, or commercially used. It is better to redo something then torcher yourself and your client the rest of the development throw.

[15] You can find out more under the Qt Docs here: https://doc.qt.io/qt-5/qtquick-bestpractices.html

2.4 First real Projects

Just learning about the theory behind everything only brings you so far. And about now you might be craving a real project. Something to test out Qt and its capabilities.

And for that purpose, I will run you throw a few apps I have built over the years. They show you a broad overview of all the components in Qt and I will explain everything as we go. We will be coding step by step throw everything we need to do. That means you can follow along and code beside me. I want to be going over everything we are doing theoretically, just as much info as we need to build what we want. If you have problems with that, or you miss some information that explains what you need to do, you can go to the Git Hub as already mentioned and get the sources as well as more details about what we are doing in what step along the way.

This also the perfect time for me to mention the videos on my YouTube channel BenCoepp. There you can find a bunch of Qt related videos and topics, how I build and program applications in Qt on a bunch of components that you will use all the time when building applications in Qt, for instance you can find videos on List-, Stack-, Swipe Views. I also might make a few videos in the future about topics this book covers, but this really depends on how good people think this book is.

But enough with self-inserted product placements and let us jump right into our first real application in this book.

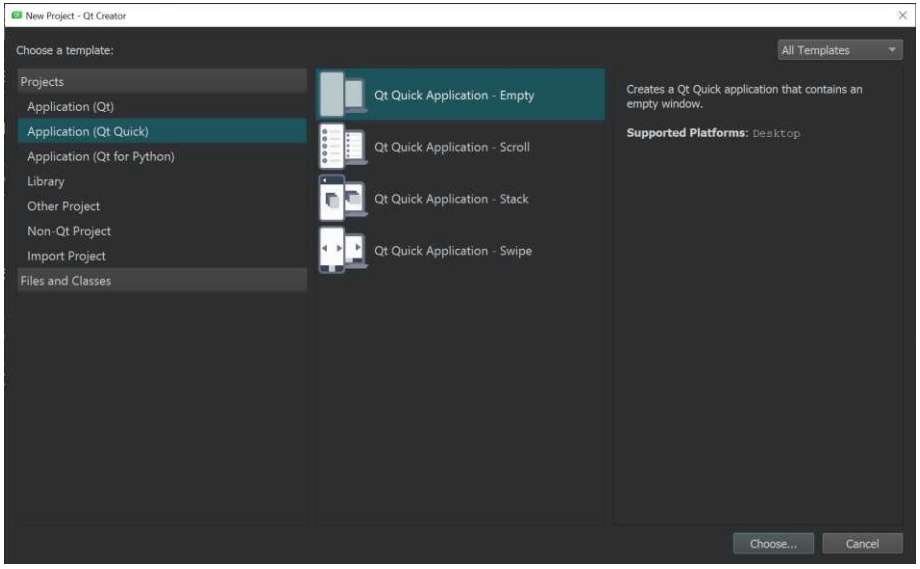
2.4.1 Taskmaster

This in my opinion is a non brainer. Whenever I start development in a new language or framework, I start by building this kind of application. A Task Master, what is that? Well basically it is an application where you can **view, add, delete** tasks to a list. Sounds simple it is simple, but it requires a lot of functionality that you will use daily in any project.

So, we are going to start out with this. So enough with the waiting and let us jump right in.

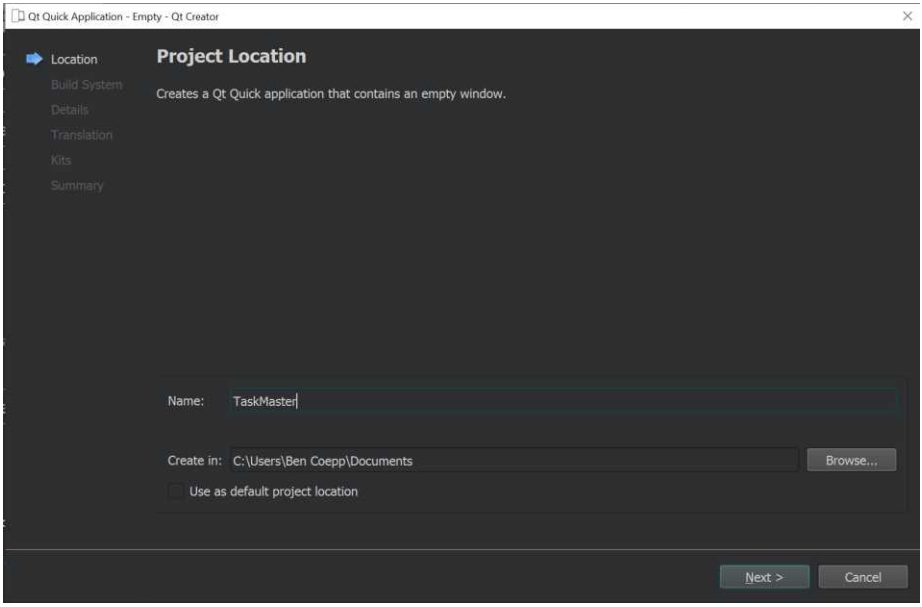
- Project Creation -

This works the same as the last time, but for practise sake we are going to do this again. Open Qt Creator and create a new project.



We are again choosing Qt Quick Application Empty as our template. We could also use the Scroll Template this time as it fits with what we are trying to build, more about that later. But for the ease of use lets stick to the Empty Template.

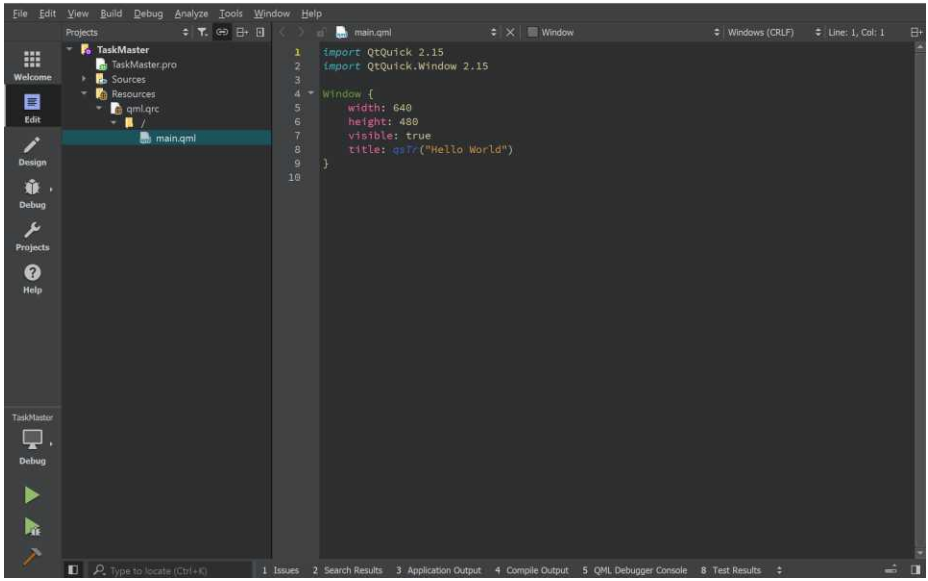
Next up give our project a descriptive name, I will choose TaskMaster. If you typed the name in hit Next and let us jump ahead.



New Project Wizard, pick name and location

The Build System can be left as qmake. As before we are going to choose the newest Qt version available for us. We do not want a Translation file as we are only doing this app for learning and educational purposes. For the kit we are going to choose MinGW 64-bit or 32-bit. We are only developing the app for teaching purposes so there is no need for anything else. We also do not need VSC (Version Control System) in a later project we are going to use this for practise sake but here we

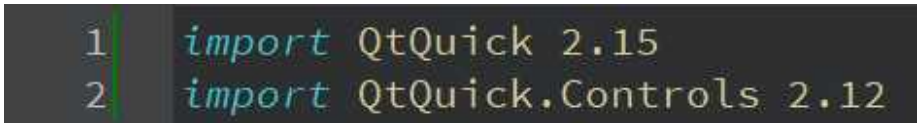
are skipping it. When you are finished hit Finish and the project will be created.



Edit Page after Project creation

As you can see, we are now at the exact same point as we were in our Hello World application.

The first thing I always do when starting a new project is updating the imports in the main.qml file. For whatever reason Qt does not create the QML Files with the newest imports so let us change that.



Qt imports

Currently we have QtQuick 2.15 and QtQuick.Window 2.15 as our imports in our main.qml file. QtQuick 2.15 is the newest version available so we can leave it as it is, but we do not need

QtQuick.Window. What we need is QtQuick.Controls 2.12[16] this is more or less essential for most Qml applications as it has all the basic controls and functionality that you will need to build most applications.

If you changed both you will have this in your file.

```
1  import QtQuick 2.15
2  import QtQuick.Controls 2.12
3
4  Window {
5      width: 640
6      height: 480
7      visible: true
8      title: qsTr("Hello World")
9  }
10
```

You might get an unknown component. (M300) this will typically come if you do not have the current packages imported or the component is not spelled properly. In this case we deleted the QtQuick.Window package from our imports, so window as a component might not work anymore. If you are running the newest version of Qt this should not be a problem anymore but if you get this error then you know why.

```
4  ApplicationWindow {
5      width: 640
6      height: 480
7      visible: true
8      title: qsTr("Hello World")
9  }
10
```

Simple Boiler Plate ApplicationWindow

[16] These are in my opinion essential for QtQuick applications they provide you with the basic functionality you will need. Also, QtQuick.Controls 2.12 is the newest version available to me.

We are going to use ApplicationWindow, its basically just like our window, I just prefer it as it has some nice capabilities under the hood that the window does not have. But in this case, it does not really matter that much what window you are using. Next up we are going to change the title of our application. Currently is says Hello World, but we are not writing a Hello World app so change it to something more fitting.

```
8        title: "Task-Master"
```

Titel property of our ApplicationWindow

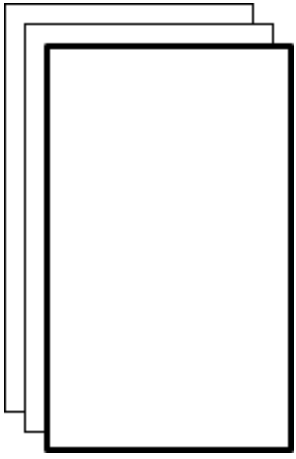
As you can see, I also deleted the qsTr (“”) from line 8. Why might you ask? Well, qsTr is a lovely thing Qt has under the hood, it is an Internationalization tool. You can give this to any title and text attribute that a component might have. And if a user from another country uses your app, you can specify a translation of that text using the translation files Qt provides. This is extremely great when you want to build applications that you are going to publish all over the world. Having a translation is more or less essential then.

Most commonly you will find that there are two or more languages used in translation. You have a primary language that is the main focus in development, most likely it is the native language of the developers, and then you have two other primary languages like English, French or Mandarin for instance, these will make it more or less possible for nearly all people on the planet to understand what you have written.

In our case thou it is not need as we do not want to translate our application to any other language then English.

– Loading the Pages –

Now to the first component we are going to write in our application, a Stack View. With this Stack View we want to change the page when we loaded all our data. What do I mean by that, well the Stack View will have an initial item that is our Loading Page? And in our Loading Page we are going to load all the data we need for our application. If this is a success, then we are going to switch to our Main Content Page. It is the easiest way of making a Loading Page work in Qt.



But what even is a Stack View. Well, the easiest way of imagining it is a bunch of pages stacked behind each other, and you can change which item / page is in front of that stack.

It is one of the most commonly used components for a variety of reasons, in our case for loading another page. But you could also load remote pages, or images throw this.

First of type out Stack View with the corresponding brackets.

```
4  ApplicationWindow {
5      width: 640
6      height: 480
7      visible: true
8      title: "Task-Master"
9
10     StackView{
11
12     }
13 }
14
```

After that add the id to our Stack View, here I chose contentFrame for our id. Mainly because this is the frame where all our content will be brought to, and for the fact that I learned it this way.

But this might not explain what an id even is. The id of a component is like its name. An Id must always be unique. You can call the id of a component from anywhere in the QML File it originates from and if you import this file in another qml file, you can call it there too. This calling of ids gives you the ability to also call the attributes from the component as well as the functions and methods belonging to it. The best comparison I was able to come up with was that of ids in HTML documents. They are also unique names for components and make it possible to interact with the corresponding component throw the id.

With the id written let us add the next attribute we need. Currently our component does not have a with and a hight. We could now just give it a static with and hight, but this leads to

problems when the user resizes the window, so we are going to use anchors.

Anchors are as the name suggests anchors that anchor the component corresponding to the point you specify. For us, the best anchor will be this.

```
12          anchors.fill: parent
```

We can use this to fill our parent, the ApplicationWindow with our Stack View.

This could also be done using this.

```
12          width: parent.width
13          height: parent.height
```

This is also totally acceptable, it is just two lines except for one. Also, when using anchors.fill you also position the element at 0, 0 on the screen, this is not the case with width and height as I show here, it is true that it also positions itself on 0, 0 here but this could also change, with an anchor changing the position is only possible when using a margin intentionally. So, I would in this instance chose the option above so let us do that.

There is a use case throw for using the second option. And that is when you import a page from another qml file. There the anchor option confuses Qt. That does not mean it is not going to work, but you are going to get some warnings about not using anchors in that instance and I would recommend not using them there.

qrc:/Main/Load_Page.qml:4:1: QML Load_Page: StackView has detected conflicting anchors. Transitions may not execute properly.

Warnings like this, but you can more or less ignore them. At least until you get a real problem when using it.

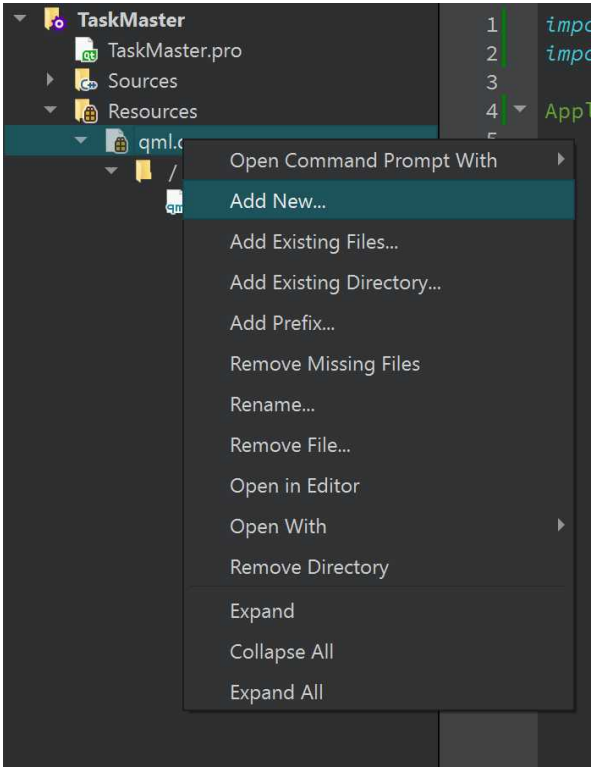
Next, we need to set the initialItem for our Stack View. This is going to be our Load Page.

```
13          initialItem: Qt.resolvedUrl("")
```

This right here is all we need to add. Between the “” we are going to place the URL for our Load Page, but that after we created it.

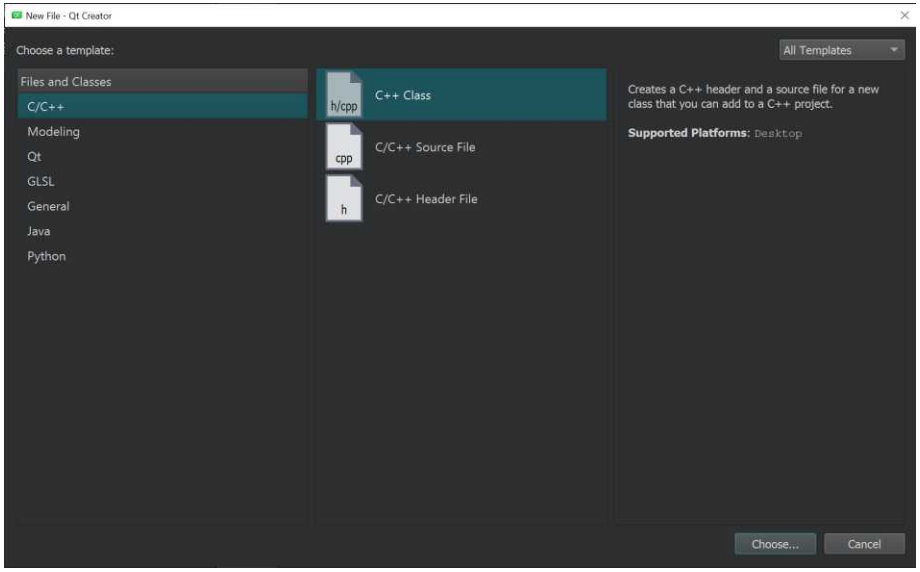
If you added everything so far, we could move onto the next point. Creating the Load Page. Creating the Load Page is not so hard so following along.

By the Way we will tend to create a lot of pages all the time, so refer to this chapter if you need to create a new page and you do not know how to do this again.



Adding new Files

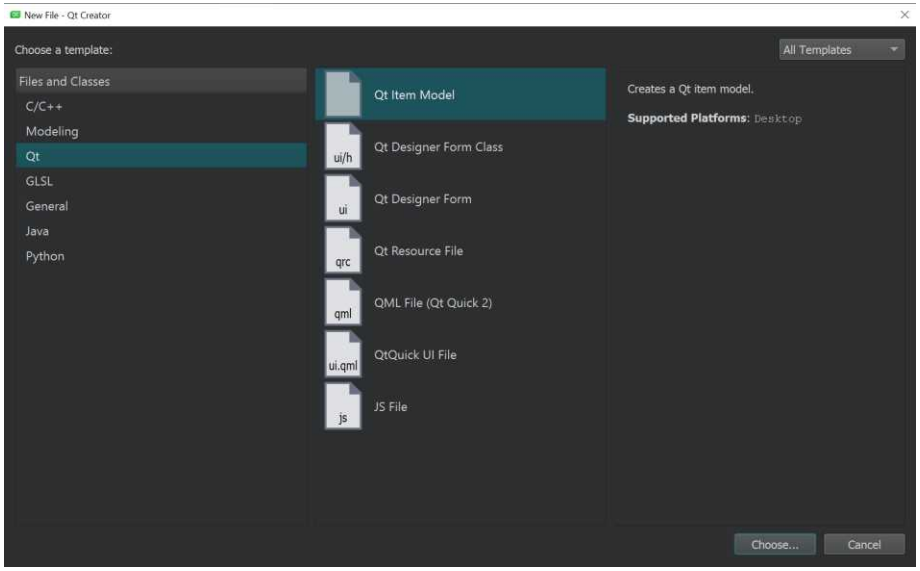
Go over to the left where the project tree can be found. There you need to right click on qml.qrc. This will open a menu where you find Add New… when you click this a wizard will open.



C/C++ File Wizard

This will present you with a similar wizard to creating a new project. First of you need to choose what type of file you want to create. The first time you open this wizard up in each project you will be presented by the C/C++ templates. These will be important but not right now.

We need a QML file. And for that go over to the left and chose Qt.

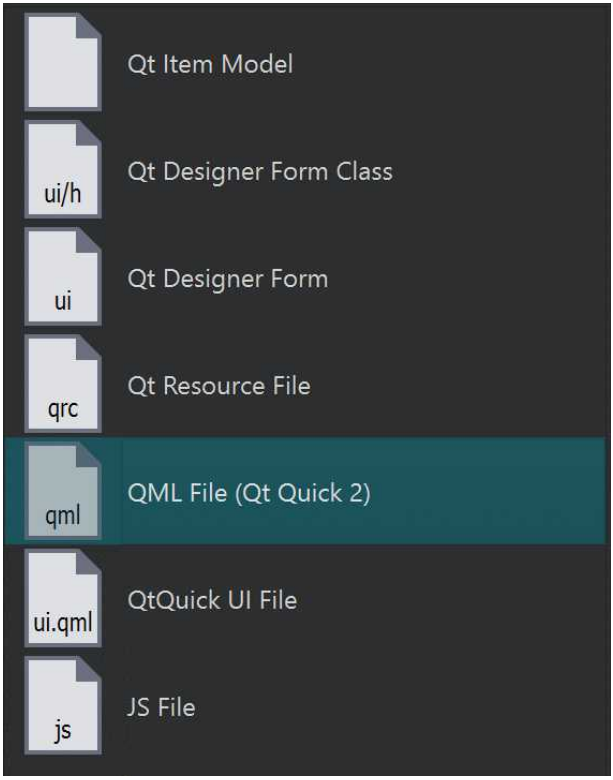


Qt File Wizard

This houses a lot of the files you will be using daily. Like qrc resource files for organising your files and project structure. QML files and QtQuick UI files which serve the same purpose of being the elements that make up the UI of a Qt Quick Application and lastly JS files.

As you can see there are a lot of different files and file types here, and they all have their select use case, some are a little easier to understand like for instance JavaScript, if you need to write a lot of JavaScript functionality or you have a bunch of functions that you do not want to have in your UI, then this is a perfect option. This also extends to qml files as they are what we are using for the UI Elements, but some other files like qrc or ui.qml files are a little harder to understand.

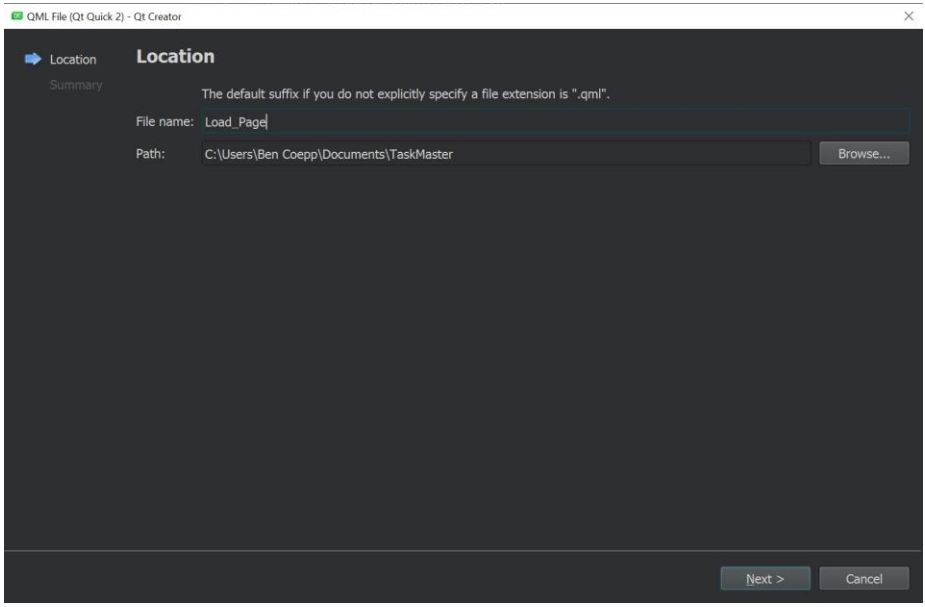
My only advice to you is that you look up their respected use case on the Qt Docs and then see if you need it, I do not use many of these files thou, most of the time just Qml, JavaScript and maybe qrc when I really need to.



We want the QML file, so select it and hit Choose. This will bring you to the next page in the wizard where you need to give the file a name and a location. In this case I will name the file Load_Page. You could also use CamelCase, but I prefer to separate the name of the file and what type of component it is throw a _.

That is just my preference, you can always choose to use another naming scheme but remember that you need to keep that naming scheme for the entire project.

There is nothing more frustrating than trying to understand one's code, but the naming scheme changes mid throw. So, remember to keep one for one project.



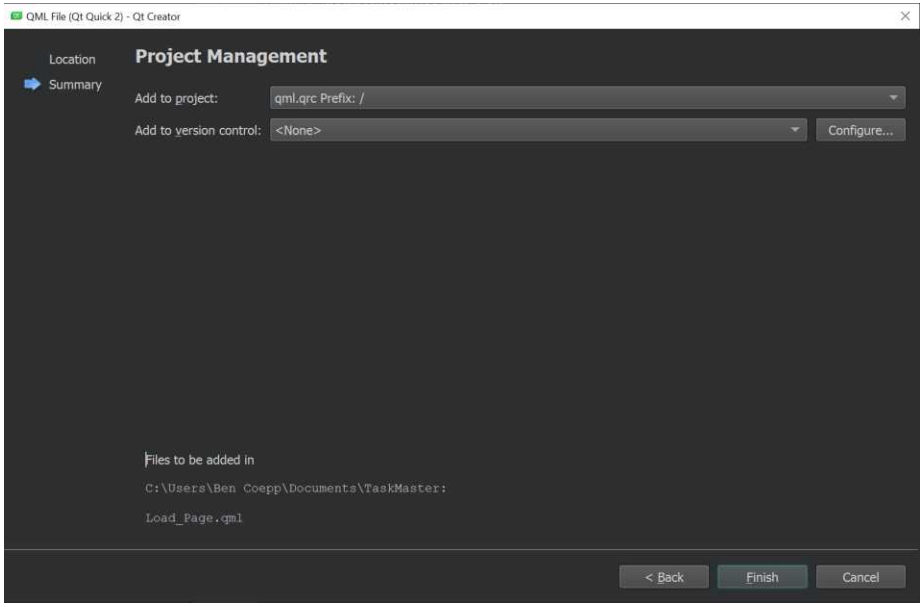
New Project Wizard Project Location

You might also choose the path where the file is saved in, this can be helpful if you have an already existing file structure, or you want to create one. My way of doing it mostly consists out of having all files related to one component in one folder, this is particularly good if you view the project structure, but it takes a few extra steps to set up.

In our app here we do not change the location. Just the default location is sufficient. As a side node the default location will always be the root folder of your application.[17]

When you have decided for the name and the location hit next.

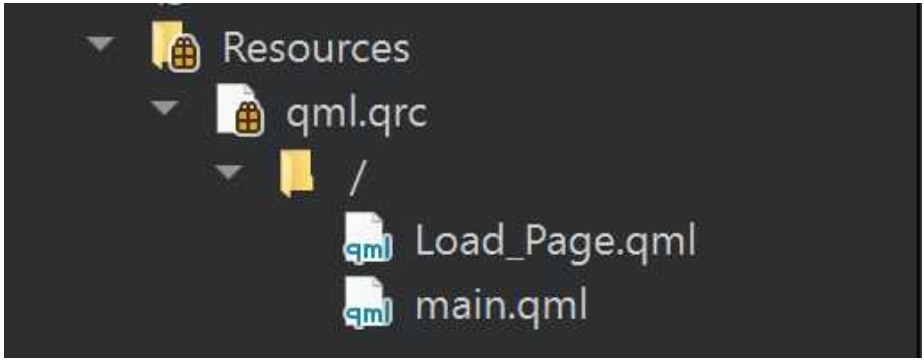
[17] This might not be that great depending on how many Files you have, if you have over 15 you might want to add Folders in which you place your Files to minimise the clutter that the Files create



Add to Project Management Wizard

On the last page in the wizard, you will have the choice to add the file to a Version Control System if you have it active to the project. And you can choose to which prefix you want to add the file.

But mainly this is the summary page where you can see what is going to happen and what files are being created. When you are done hit Finish and this will create the file in our project.



As you can see here the Load_Page.qml was added right above our main.qml file. If you want, you can always choose to move the file later.

Also try to keep the project tree as clean and organised as you can. Depending how big your project is bound to grow, having an unorganized project tree can really hinder your ability to work.

```
1  import QtQuick 2.0
2
3  Item {
4
5  }
6
```

Empty new QML file

If you open our newly created file there is not a whole lot to look at. And that is not helpful. But one of the first thing you might realise is that the QtQuick import is again completely out of date again. So, first of change this to

```
1  import QtQuick 2.15
```

And we are also going to add QtQuick.Controls to our project.

```
2  import QtQuick.Controls 2.12
```

I wish Qt would always use the Qt imports we have in our main.qml as a base for all future files but I can understand that this a little hard to implement and it makes no real difference in most cases. You just need to remember to update and add this to every new file you create.

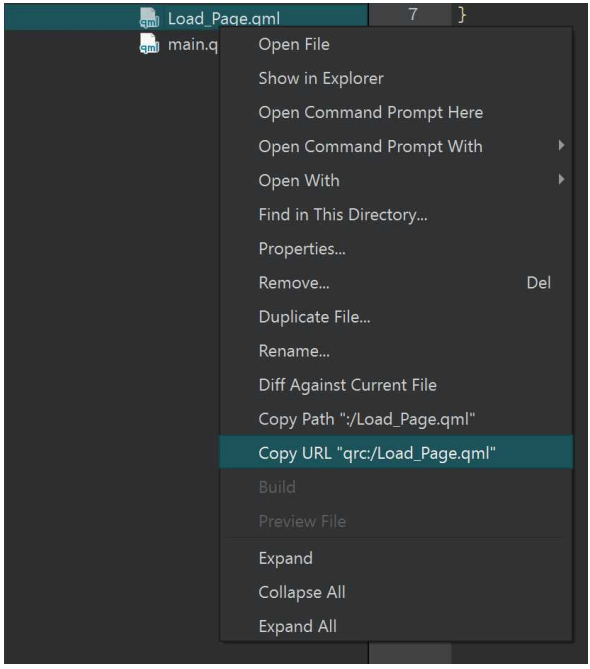
Currently the Item inside our Load_Page.qml has no width as well as no hight. So, lets add that.

```
5        width: parent.width
6        height: parent.height
```

As mentioned already in a few steps ago we will fill the parent of this Item with the Item itself. This is a genuinely nice way of keeping the same aspect ratio and display size in all our components.

With the creation of the page out of the way, let us get the page displayed on screen. For that you need to go back into our main.qml file and change the initial item of our Stack View to the URL of our Load_Page.

You can get the URL by right clicking on the file in our project tree, and selection Copy URL.



Copying URLs

This will save the URL to your clipboard and we can paste it inside of the initial item of our contentFrame.

```
10      StackView{
11          id: contentFrame
12          anchors.fill: parent
13          initialItem: Qt.resolvedUrl("qrc:/Load_Page.qml")
14      }
```

Simple Load- and Main Page Stack View setup

It might be important to mention that you do not need the *qrc:/* in front of the URL here. As our Load_Page.qml is inside of the same directory and Prefix as the main.qml, this means that you could just write Load_Page.qml here. I just end to always keep it in even when it is in the same directory and Prefix as I just copy the URL, also when you have a different project structure with multiple different Prefixes and or even multiple different qrc resource directories, you will need to use the full URL as we did here.

With that we are done with the Stack View in our main.qml. If you were now to start up the application, you will be greeted by nothing. The Load_Page is loaded, but the problem is that there is currently nothing in our Load_Page. So, lets change that.

First of we are going to make a background for our Load_Page. There are a few ways we could do this, one changing the item tag to a page tag and then adding a background attribute. The other and in my opinion preferable option. Adding a rectangle and making it as big as the item. It is a remarkably simple easy to understand and usable option so let us do that.

```
8       Rectangle{
9           id: bgRec
10      }
```

We are going to give this an id; this might not be necessary right here. But it is a good practise to have telling ids for everything. You might never know when you might need to get the id of a component you have.

```
10          anchors.fill: parent
```

For the width and the height, we are going to use anchors again, as with the Stack View. Mainly because I want to save us another line to write, and because here it is the more suitable and elegant solution to the problem.

Now the last thing to do is give the rectangle a colour. I have my colour set so we are going to use the colours I have picked. If you want to use your own feel free to do so. And if you want to get my colour sheet, you can find it in my Git Repository for this project.

```
11          color: "#2C3E50"
```

As I am not a designer, I am not interested in creating a stunning and beautiful App. For me it is all about the functionality.

A simple background will not do the trick for a Load_Page. So, we need to add some sort of Busy Indicator. We could now create our own Busy Indicator that would have a nice animation, our we could just use the one provided by Qt.

To get the Load Indicator working on our Load_Page we could now just type it out, but I would rather use the Designer. As it is our first time using it, it can be somewhat overwhelming so read carefully and you will not get lost.

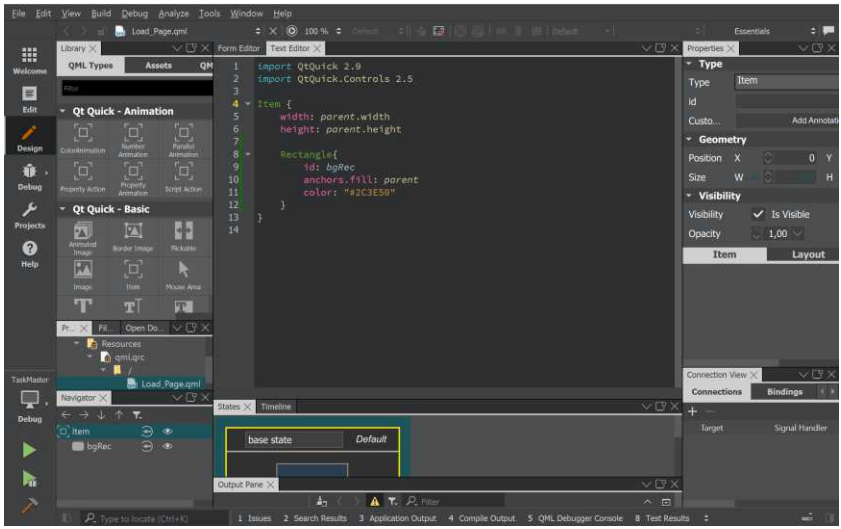
Opening the Designer is not hard, if you have opened our Load_Page you can go the left where the sidebar is located. Here you will find the Design button.

So, click on the Design tap and the Qt Designer will open. Depending on how large the QML file is and how many different components are inside of it this can take a little bit of time.

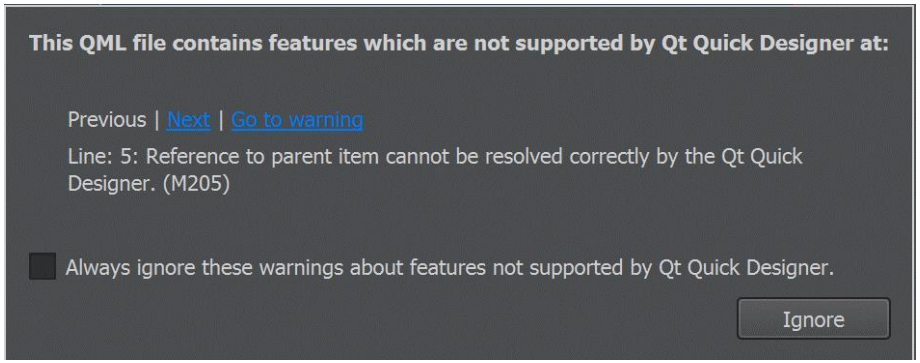
There are also a lot more taps that might interest you. First of the project tap, here you can edit the kits you have as well as the build systems that are available.

Also, the Debug Tap can be extremely helpful as you can find problems, bugs, and other stuff throw the Debug View. And lastly, we have the Help Tap, here you can find information on components and features Qt has. Also, you can open the Qt Docs throw here. So, you might want to check it out when you are lost.

When Qt Designer opened you will be presented with the View you can see in the next screenshot.

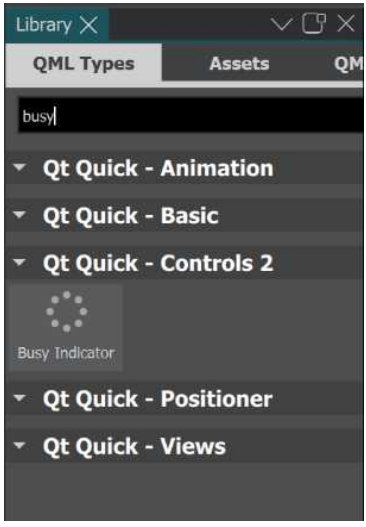


Do not be overwhelmed by all the input thrown in your face, the first thing we are going to is switch to the Form Editor. This can be done right above our Text Editor. There you have the Form Editor Tap. By clicking it the Form Editor will open.

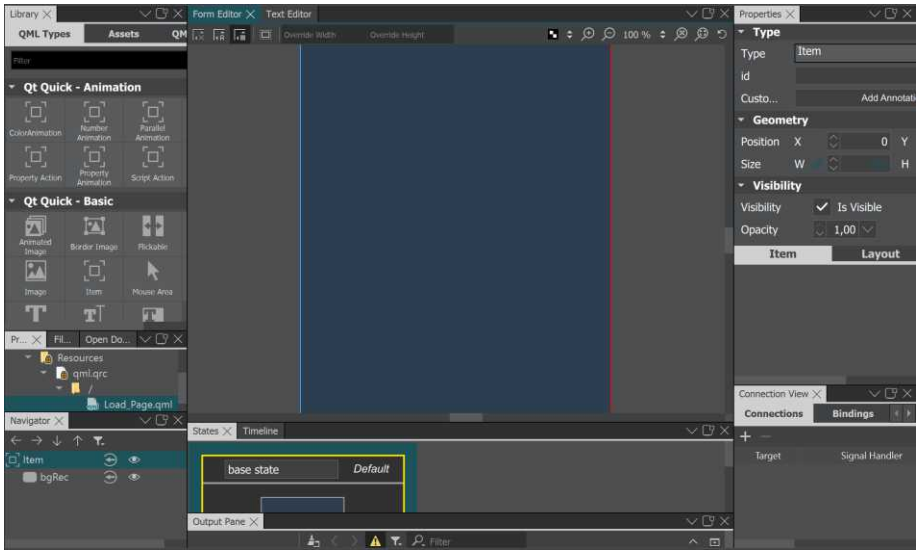


Warning on line 5

Immediately you will be bombarded with a warning. If you were to investigate what this warning stands for, you will find that using patents as a reference to get something works only if there is a parent to the element. We made our items with and height depending on our parent. Currently this item does not have a parent. This is changed when the file it is loaded.



So currently you can ignore this one. You might want to check the box for ignoring this all the time, but I would not recommend it. This warning also pops up on other elements inside of our items if there is a break of the parent structure. And if you were to ignore this warning it might lead to you searching endlessly for the problem.

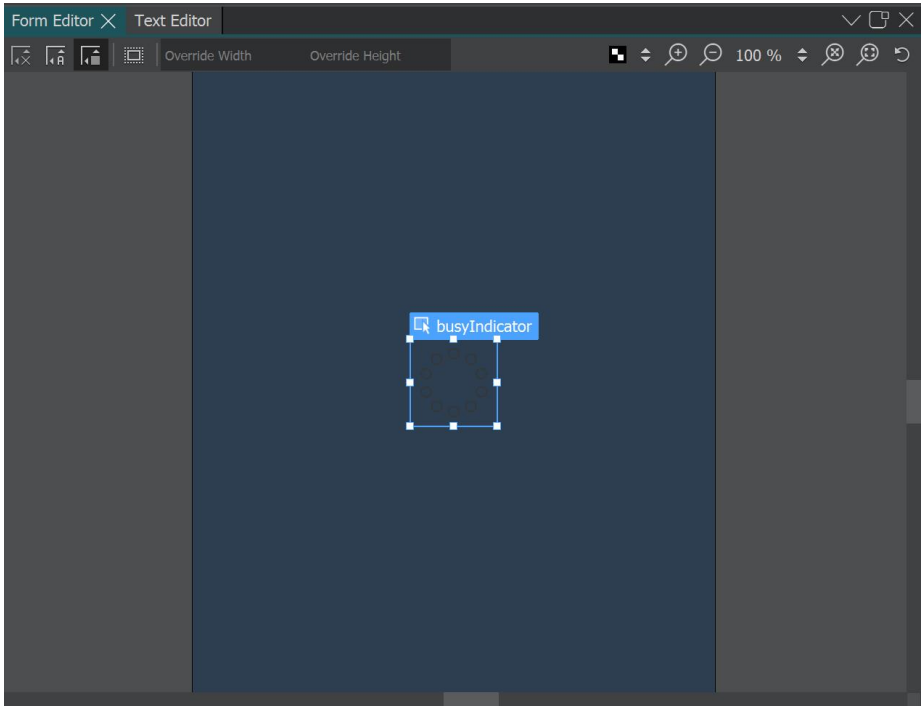


Qt Designer Page

When you clicked ignore for the warning you can now see the Form Editor.

The Form Editor is basically a drag and drop designer. You have all the components you currently imported on the left in the Library Tap. There you can find anything from the basic Animation Components, Images, Labels and Buttons. We are going to go over where to find what and how to search for something in the Form Editor and Designer later. For now, just type in the search bar under QML types and type in busy indicator.

Now just drag and drop the component into the Form Editor and onto our coloured background.



Form Editor close-up

Once you placed it there go back to our Text Editor. You could do this by switching the Tap at the top to Text Editor again. But because we are not going to use the Designer right now let us leave it. Go to the left sidebar and click on Edit this will bring us back to our normal Text Editor.

If you now investigate our Load_Page.qml you will find a new Component was added to our file.

```
13        BusyIndicator {
14            id: busyIndicator
15            x: 150
16            y: 290
17        }
```

Here we have two new attributes that we have not had so far. These are as you might suspect X, Y Positions. And they are very usable in positioning. The problem arises if you have different types of displays and sizes. If you have fixed x or y positions this will lead to a very janky UI and even under some perspective break entire applications.

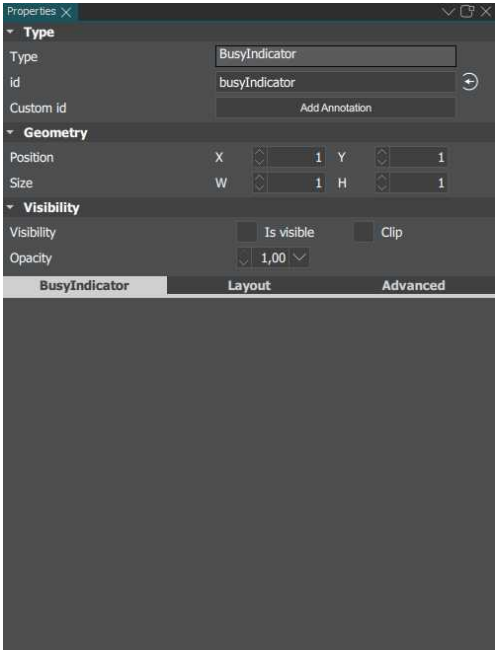
And for that purpose, we will not be using any fixed x or y positioning but anchors. There are many different anchors out there, but the most used and useful ones are:

- **Anchors.Left**
- **Anchors.Right**
- **Anchors.Top**
- **Anchors.Bottom**
- **Anchors.verticelCenter**
- **Anchors.horizontelCenter**
- **Anchors.centerIn**

They are as their names lead on to believe able to align a component to the respected side or centre it. You can combine them in any way you want and the best way of learning how to use them is by just trying them out.

In this case we will use the anchors.centerIn anchor. We want to centre our Busy Indicator on our background in the middle. We could now write this in our normal text editor as we did before, but I want to show you the way you can do this using the Design / Property Editor. Open up the Form Editor again and select the Busy Indicator from the Navigator.

On the right side of our window, you can find a tap called Properties. Here you can as the name suggests edit the properties of all the components you have in our file. This can be very handy when you want to prototype fast, as well as to see what other properties are available.

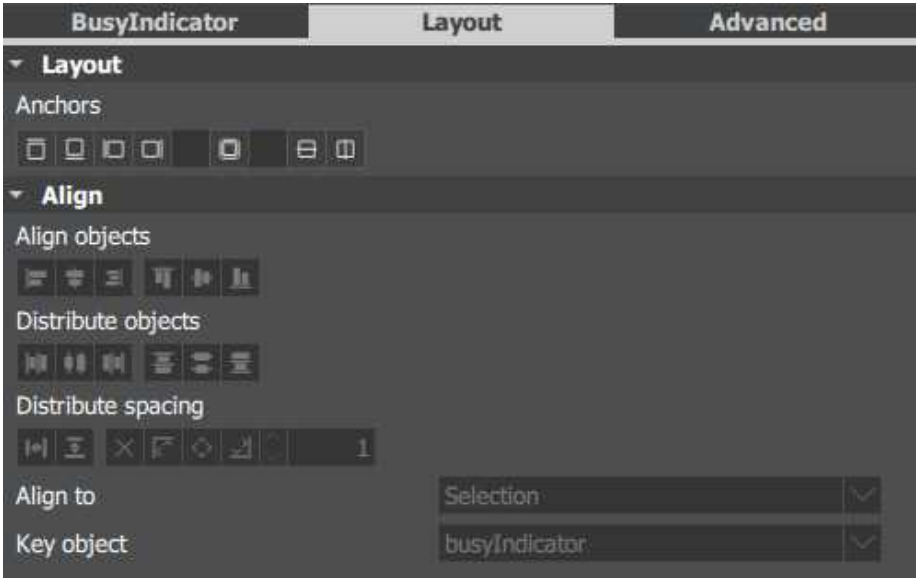


When you have our Busy Indicator selected you can see this inside of our Properties Editor.

You can see the type, id as well as the size and position of our Busy Indicator. You can also manipulate the visibility from here.

But the most important properties for us are still hidden.

To see them you need to switch the tap below from Busy Indicator to Layout. This will give you these new properties to play around with.



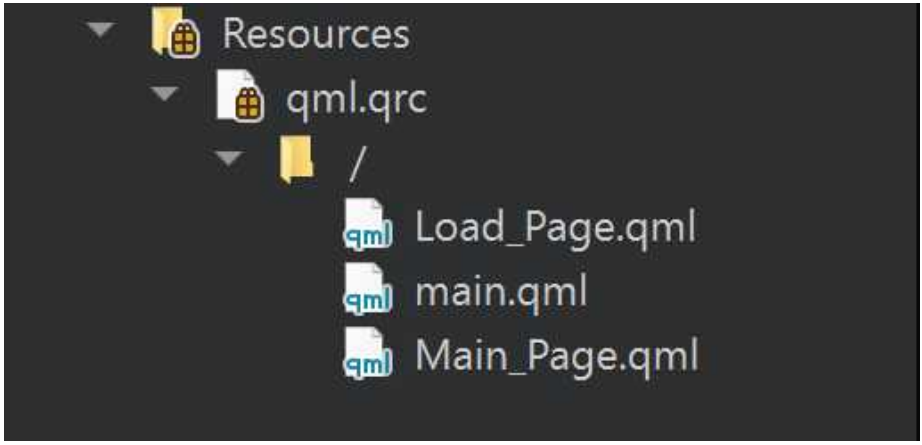
Basically, these are all the properties to anchor, as well as align the Busy Indicator in whatever way and shape you want to.

For us, the most important ones are right at the top, the different anchors. You have all the options from *anchors.right* to the different horizontal and vertical centres. Here you need to just select the *anchors.centerIn: parent*. If you have done that you are left with this.

```
15      BusyIndicator{
16          id: busyIndicator
17          anchors.centerIn: parent
18      }
```

And with that we are done with our Load_Page. Next up is our Main_Page. For that we are first going to create a new Qml file. We already did this two times so you can do it now on your own. If you still need a bit of assistance, then go a few pages back and read up on what to do. The name of the file should be Main_Page and the folder should be the project folder.

When you are done, and the file should be created you are left with this.



A new file in our Project Tree and if you open the file you will find the standard boilerplate QML file.

```
1  import QtQuick 2.0
2
3  Item {
4
5  }
6
```

Empty new QML file

As before we need to change the import to QtQuick to 2.15 and add QtQuick.Controls 2.12. If you have a newer version use that.

```
1  import QtQuick 2.15
2  import QtQuick.Controls 2.12
```

New correct imports

Now we need to add the with and height to our item.

```
5      width: parent.width
6      height: parent.height
```

Next up we are going to give our app a proper background to make it a bit easier on the eyes. As this is our first real application, we are only going to use one page, so we do not do anything fancy.

The rectangle itself is the exact same as the one in our Load_Page. So, copy it over and place it inside our item.

```
8       Rectangle{
9           id: bgRec
10          anchors.fill: parent
11          color: "#2C3E50"
12      }
```

If you are asking yourself why we are using the same id for our rectangle, then I can give you a simple answer. Because we are not importing these pages and we are not using any components in between them, there will not be a problem using the same id. But if you want to be true to form then use a more telling and unique id.

Currently the page would not change if the app has loaded, it would always stay at the Load_Page. This is not helpful so let us add the change of pages to our main.qml.

Inside of our main.qml we need to add this to change the pages after loading.

```
16      Component.onCompleted: {
17          contentFrame.replace("")
18      }
```

This is code snippet does one simple thing. When the main.qml page has loaded and everything is ready to be rendered on screen the contentFrame's item will be changed to that of our Main_Page, so our Stack Views items will be changed. Now we only need to place the URL of our Main_Page into the "" and we are done with the replacing of the page when the loading is completed.

```
17          contentFrame.replace("qrc:/Main_Page.qml")
```

With this done we will next be going over how the app is supposed to work and, then we are going to build it.

- How does the app supposed to work -

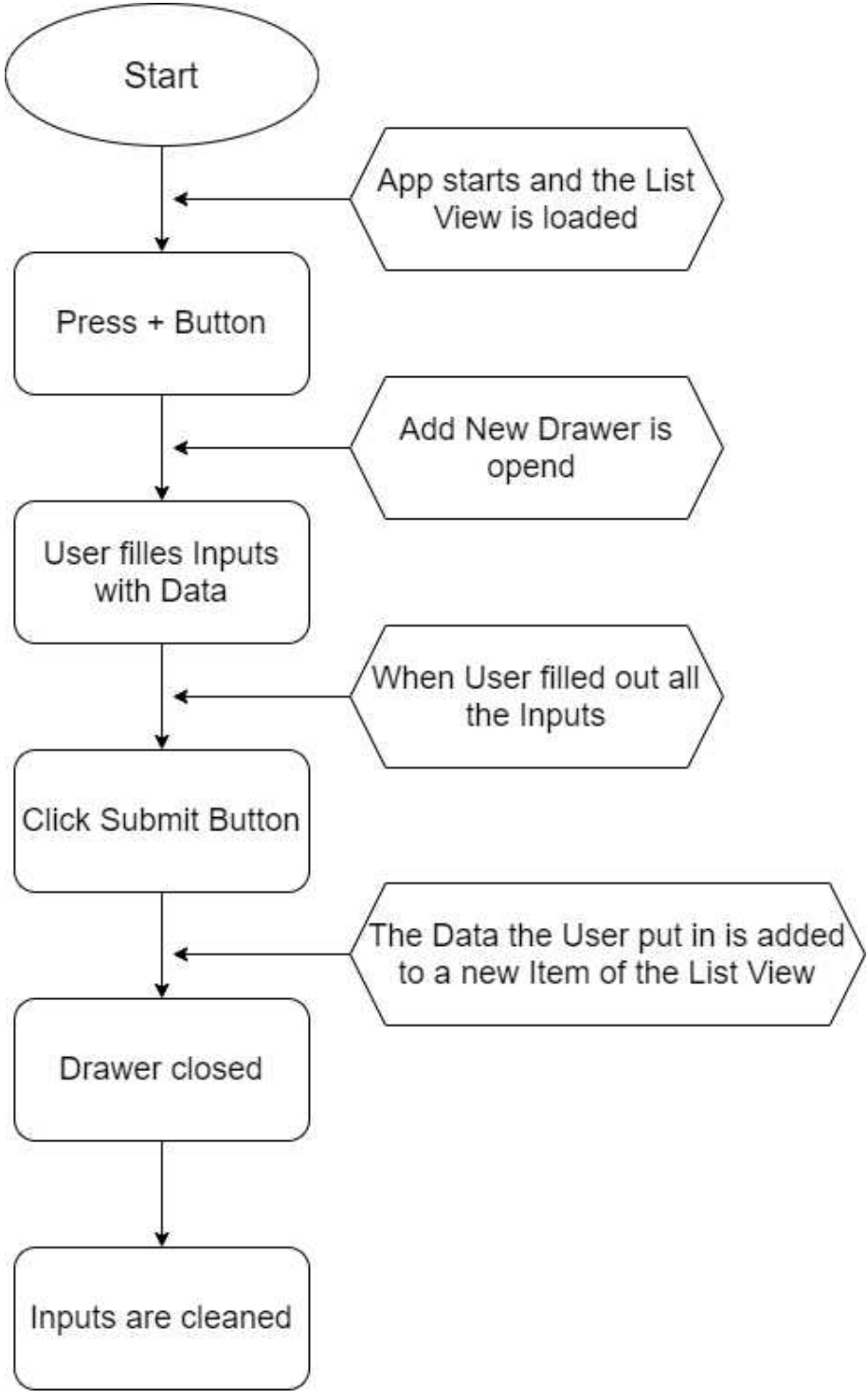
This is supposed to be a Task Master app, we want to create and delete tasks. But what do we need to do this? Well first of we need a List View, this will display all the tasks we have created. Next, we need a button that opens up the inputs for us, and lastly, we need a button with which we can submit the inputs.

This is all the functionality we need, it is not much nor is it that complicated. But to explain a little bit more about what we are doing and how we are going to do this.

On the next page you will find a diagram of how the application should function and what functionality we need. For this application this would not be necessary, we will not create anything really complicated but it is better to have a diagram and a plan so that we are not lost while we create the application.

Basically, the application works like this. The application starts, the app loads the List View, which at this point is still empty. Next the user can hit the + button, this will then open up our drawer where our inputs are located in.

When the user filled out all the inputs, he can click on the submit button and the drawer is closed again.


Start
App starts and the List View is loaded
Press + Button
Add New Drawer is opend
User filles Inputs with Data
When User filled out all the Inputs
Click Submit Button
The Data the User put in is added to a new Item of the List View
Drawer closed
Inputs are cleaned

With the click of the Submit button the application takes the data the user has inputted and created a new List Item from that. With this done, new item is displayed and the inputs are cleaned. Now the user can create a new Item.

– List View and displaying Data –

As it is now clear what we want to achieve, let us start by building the List View we need. The List View is one of the most important elements of Qt and it is used basically everywhere in any applications, I am going to over it in a bit more detail. First add the List View to our Main_Page.

```
14    ListView{
15
16    }
```

Empty List View

Next our List View needs an id, as well as a width and a height.

```
15        id: listview
16        anchors.fill: parent
```

For the id I choose just list view as it is very descriptive, and we will only need one List View. The width and height will again be handled by *anchors.fill: parent*. It will fill the entire screen, and as we will not have anything else to do in the entire application this is perfect.

We also need a header and a footer. The header will be our title for the app, and the footer will house our add button. First of we are going to write our header.

```
17          header: Item {
18              id: headerListView
19              width: parent.width
20              height: 50
21          }
```

The header attribute requires a component as an input. We are going to give it an Item, with an id, a width that spans the entire screen and a height of 50. This will be a bar up top that should always stay at the top of our application.

```
22          headerPositioning: ListView.OverlayHeader
```

If we want to have the header always at the top of our List View, we need to add headerPositioning: ListView.OverlayHeader. There are two other ways the Header Positioning can work, the default way of just scrolling with the content as if the header was a normal item. Or a header that is pushed up and pulled back when the content is scrolled.

Currently there is nothing inside of our header. Now we are going to add the title. For that we are going to use a Label. This Label comes from QtQuick.Controls and is a simple text element with a few more attributes and abilities under the hood.

```
22              Label{
23                  anchors.centerIn: parent
24                  text: "Task-Master"
25                  color: "black"
26              }
```

For most Labels out there, we need a position where the Label should be, a text what the label should display. We also give the text a colour which is not needed in this instance, as Labels have always a default colour of black.

With the title out of the way lets add a Model and a Delegate to our List View. These are some of the attributes as well as a width and a height which are essential for a List View.

But what even is a Model or a Delegate? Models are to put it simply data. Like an Array, List, or an Array List, if you know how they work you will also understand Models. Qt has its own data structure under a Model. But fundamentally they work the same as the already mentioned ones. Now let us implement it into our List View.

```
29          model: ListModel{
30              id: myListModel
31          }
```

For our Model we only need an id, so we have the ability later to interact with it. For the Delegate we will write a bit more.

Delegates are like the housing for our data. The List View takes the data from our Model and like a mask puts our Delegate over it. This is extremely nice and performant way of displaying data that does not take up a lot of work. But let us build our Delegate so that we can see how it actually works.

```
32          delegate: Item {
33              id: myDelegate
34          }
```

The delegate property requires a component to work. So, we are giving it an item with an id. This id will later be used to make the Delegate interactable.

```
34              width: parent.width
35              height: 50
```

We are going to use the same with and height as our Header. I find that a height of 50 is the perfect height for an item. It has a particularly good readability even on larger displays.

Because of the rise of mobile devices this height is also applicable. It is not too large, nor to tiny.

As we want to display text on our Delegate, we also need a Label. So, add the Label and give it an id.

```
37                  Label{
38                      id: titel
39                  }
```

The Label inside of a Delegate can get data from the Model using the id of the type of data we want to display in the Label. As an example, you can see here that our text of our Label should be the title text from the data provided by the Model.

This sound oddly complicated but, it always us to just grab all the data we need for one Item and place the data points.

```
39                      text: titelText
```

Next, we are adding the anchors to our Label. First of the anchors. left, this will place our Label as far left as the parents left. We are also adding the achors.verticelCenter. These two create the standard flow of text as you would suspect.

```
40                  anchors.left: parent.left
41                  anchors.verticalCenter: parent.verticalCenter
```

But we are not done with this. Currently we only have the title of the task, but I want to also have the data, and time of when this task was created. To do that we are going to add another Label to our Delegate. This is going to be our date Label.

```
44                  Label{
45                      id: date
46                  }
```

Give this Label an id, as well as a text property with a reference to the Data Model.

```
44                  Label{
45                      id: date
46                      text: dateText
47                  }
```

Here we are also adding the same anchors as before. If we were now to run this app, and we would add some data to our Model, we would not be able to see the Labels, as they overlap each other. This is not really what we want.

```
47                  anchors.left: parent.left
48                  anchors.verticalCenter: parent.verticalCenter
```

To alleviate the problem, we need to add an anchor.leftMargin. as the name suggests this adds a margin to our left Anchor. Pushing our content to the right. First let us add this to the title Label.

```
42                  anchors.leftMargin: 20
```

We could also add this to our data Label, but a far better solution would be to change the anchor from left to right. This would place our date immediately at the right of our component. But we also should add a margin to the left anchor, so it is not flush to the end of our window.

When we changed the Label, we are left with this.

```
            Label{
                id: date
                text: dateText
                anchors.right: parent.right
                anchors.verticalCenter: parent.verticalCenter
                anchors.rightMargin: 20
            }
```

Date Label with anchors

There is still a little bit more we could do, that would make the Delegate a little bit prettier, but for now this is not essential. If you followed along so far you are left with this.

```
        delegate: Item {
            id: myDelegate
            width: parent.width
            height: 50

            Label{
                id: titel
                text: titelText
                anchors.left: parent.left
                anchors.verticalCenter: parent.verticalCenter
                anchors.leftMargin: 20
            }

            Label{
                id: date
                text: dateText
                anchors.right: parent.right
                anchors.verticalCenter: parent.verticalCenter
                anchors.rightMargin: 20
            }
        }
```

Finished Delegate with Labels

With this we are finish with the raw version of our Delegate. It works in the way that we can use the Model with the Delegate and the data should be rendered when we have something in our Model. Next up we are going to create the Button with which we can input new tasks.

– Adding Data to the List –

This is nearly the most important part of the project, there is no point in having a List View if we cannot add data to it. So that is what we need to do now.

As already mentioned, we want the add Task button to be in the Footer of our ListView. So, let us start doing this.

```
53          footer: Item {
54              id: footerListView
55          }
```

Basic Empty Item Footer

We start by adding the footer attribute and to this footer attribute we add an item with a corresponding id.

We also need to add a width and a height to this. As with the header we are going to set the width to the parents with. And the height we set to 50.

```
55              width: parent.width
56              height: 50
```

But just an Item wont to what we want, so we will add a Round Button with a corresponding id. Round Buttons are basically as the name suggests buttons. We can click them. And that is what we need. Later, we are going to talk about a little bit more about how buttons work and the best way to set them up and work with them.

For now, you can just use what we have typed out here.

```
53          footer: Item {
54              id: footerListView
55              width: parent.width
56              height: 50
57
58              RoundButton{
59                  id: addTaskButton
60              }
61          }
```

The Round Button should also have a width and a height. For now, we are setting these to 40 a little bit shorter as the height of our footer. If this does not fit or we want to change it, we can always do this later.

```
58              RoundButton{
59                  id: addTaskButton
60                  width: 40
61                  height: 40
62              }
```

Currently our button would sit in the top left of our footer and that is not really the way we want it. The best way to get this over to the right is by first of giving it a vertical centre to the parent's vertical centre. Also, we need an anchor to the right.

```
62              anchors.verticalCenter: parent.verticalCenter
63              anchors.right: parent.right
```

Just the right anchor will not do. As with our date Text we need to position it a little bit to the left, so it is not completely flush to the side of our window.

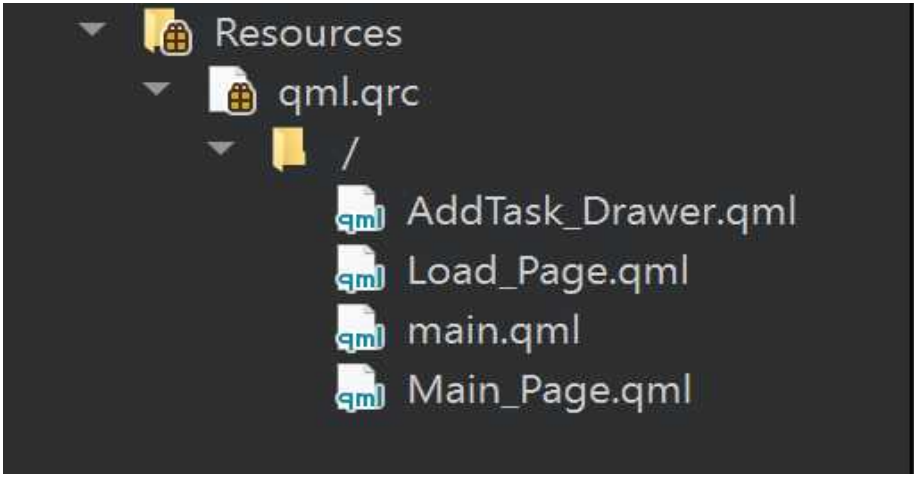
```
64              anchors.rightMargin: 10
```

Now we cannot jump right into the button click and what it needs to do because we currently have no way of inputting any data. For that purpose, we are going to create a drawer that we can use for inputting data.

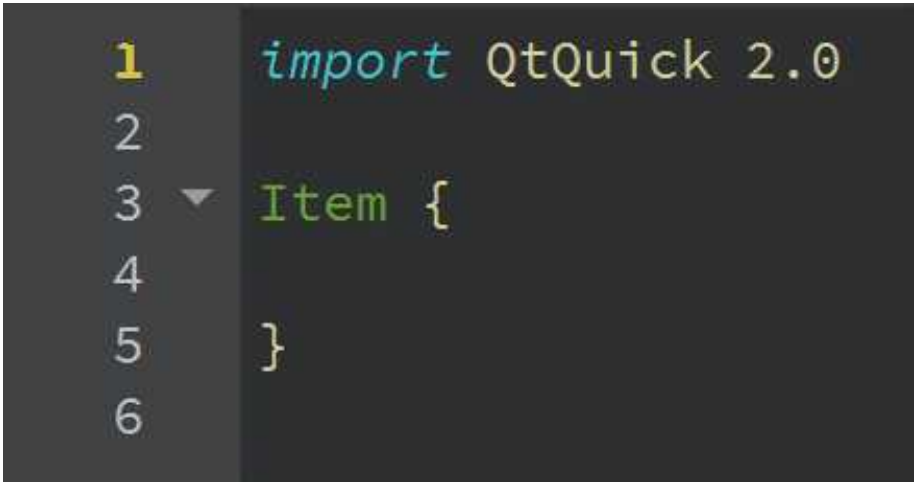
Drawers inherit from Qt PopUps, and PopUps are as you properly now able to popup when they are opened. The background is hidden, and you can interact with the Popup, if you are done then you can click finish and the popup closes, and the data is added to the list. That is the way we are going to use it. I just find drawers are better than PopUps for certain use cases. We are also creating this drawer in a separate file and then using it inside of our Main_Page. This is a use cases that you will find a lot out there. If we were to add this all to our Main_Page, it will be exceptionally long, hard to read and a real problem if we need to search for one specific thing.

So, first of create a new QML file, as this is the third time doing this, you should know how to do this. If you have a problem with it, or you do not remember how to do this, go back to the first time we did this. Or refer to the Git Repository, for more help and information. For the name of the QML file we are going with AddTask_Drawer. We have the functionality in front and the type of thing that the file has in it.

If you have done everything as before you should have created this.



Current Resources at this point



Empty QML file

As before we need to change the import of QtQuick to 2.15 and QtQuick.Controls 2.12, or the newest Version you have available to you. When you done this, we jump right into building our Drawer.

First of Remove the Item Component as well as the Brackets belonging to it. When you have done this add the Drawer Component to our File.

```
1  import QtQuick 2.15
2  import QtQuick.Controls 2.12
3
4  Drawer{
5
6  }
7
```

Our Drawer also needs an id, in this case we should use the name of the File as the id, just put the first letter in lower case, as ids, need to always be lower case for the first letter.

```
4  Drawer {
5      id: addTask_Drawer
6  }
```

The width of our drawer should be the width of our window, so use the parent for reference. The height should also be from the parent, just divided by 2 so it only goas halfway up.

```
6      width: parent.width
7      height: parent.height/2
```

Now that we have a basic drawer, we should make it possible to open it in our Main_Page. This can be done extremely easy in this case. Below our List View component type the name of the file we have our drawer in.

```
69     AddTask_Drawer{
70         id: addTaskDrawer
71     }
```

And with this the drawer is usable in our Main_Page, and we are now going to make it possible to open the drawer with the button we created. The click is handled by the onclicked event of our round button.

```
58              RoundButton{
59                  id: addTaskButton
60                  width: 40
61                  height: 40
62                  anchors.verticalCenter: parent.verticalCenter
63                  anchors.right: parent.right
64                  anchors.rightMargin: 10
65                  onClicked: {
66
67                  }
68              }
```

Basic Round Button with anchors

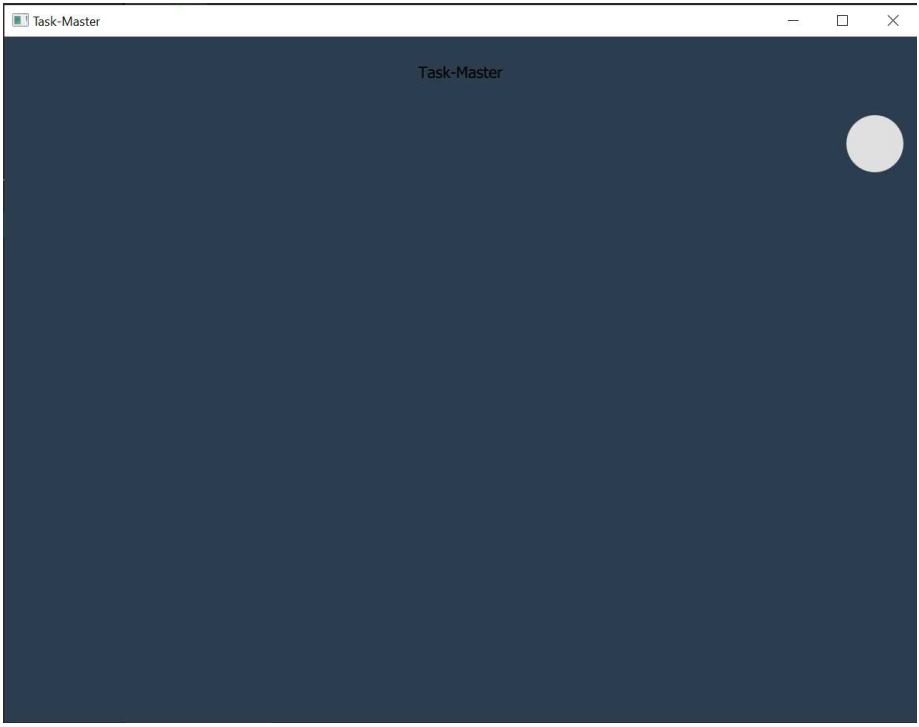
Here we can now use the id of our drawer to open it up.

```
65                  onClicked: {
66                      addTaskDrawer.open()
67                  }
```

Simple onClicked() event

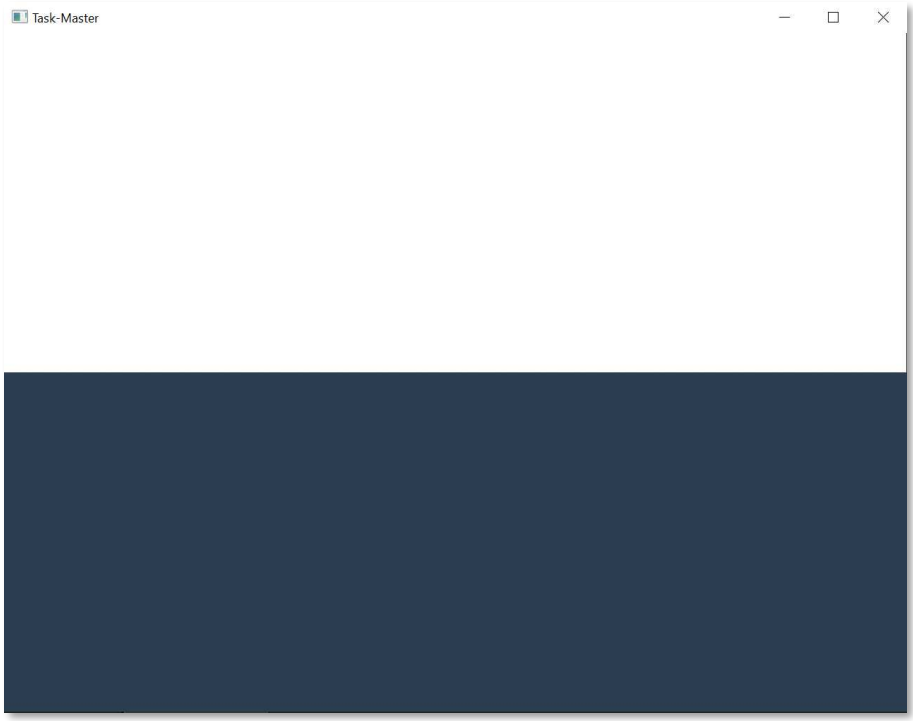
This is the only line we need to make our drawer open. As you can see this is quite easy, and one of the nice things about how ids are used in Qt. You can call any function or method that you want from this id, enabling you to open the drawer up with just one line of code.

If we were to run our application now, we would be created by this window. As you can see our header and footer are rendered, and if we were to click on the button in the footer the drawer will open.



Running Task-Master application

But the drawer opens from the left of the screen. This is not what we want, for the type of footer we want it should open from the bottom. Also, the drawer is white, which maybe does not bug you that much, but I find it really distracting and not good to look at, so we need to change the background too.



First of the position from where the drawer open. This can be done throw the edge attribute. So, go into our AddTask_Drawer file and directly in the drawer add edge. But what edge do we want, well as we want the drawer opens from the bottom, we should add Qt.BottomEdge.

```
8        edge: Qt.BottomEdge
```

With this the drawer will be opened from the bottom, and we can drag the drawer up from the bottom. This is a feature that originates from the mobile development aspects of Qt, but if we want to deploy this app on a mobile device this features comes quit in handy.

Now the background could be done again by using a simple rectangle and placing it inside the drawer, but we are doing it using the background attributes. So, add background as an

attribute to our drawer. We also add a rectangle inside it, as the background attribute requires a component to function.

```
9       background: Rectangle{
10
11      }
```

Inside this rectangle we will use anchors.fill: parent to make this rectangle as big as the drawer itself. The colour should be the same as the background of the Main_Page. This might seem a bit odd but because the drawer has a default shadow to it, it will look nice.

```
9       background: Rectangle{
10          anchors.fill: parent
11          color: "#2C3E50"
12      }
```

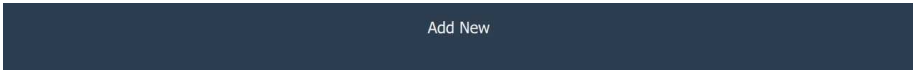
We could now also add some round corners or something, but this is a little bit of tinkering and we are going to do this later.

And with that we are finished with the background of our drawer. Now we only need to make the actual input fields that we use to give the user the ability to type in their content.

The input consists out of a normal text input we use for our title, and a Date/Time input for our date. This is not that hard but doing this the right way is not that easy.

```
14      Label{
15          anchors.horizontalCenter: parent.horizontalCenter
16          anchors.top: parent.top
17          anchors.topMargin: 10
18          text: "Add New"
19          color: "white"
20      }
```

First of we add a Label, this will be the title of the drawer, it just says Add New is cantered horizontally to the parent and at the top of the parent. We also position it a little bit below the top of the parent to have a little bit space between the top and the content. We also gave the Label a white text colour. This is just to make it a little bit more readable.



Just below the Label we are now going to add a text field. This will be our way of putting in the data we want.

```
21  TextField{
22      id: titelInput
23      placeholderText: "Text Field"
24  }
```

Our text field needs an id, so that we can remember and call later, as well as a placeholder text. This placeholder text should always be in your text field, it tells the user what the text field if for, and what should be typed into it. We should also change the placeholder text to something more applicable.

```
21  TextField{
22      id: titelInput
23      placeholderText: "Your Task"
24  }
```

Currently our text field is pinned to the top left. Which is not the place we want it to be. To place it in the right place we need three things. A horizontal centre to the parent, a top anchor and a top Margin which should be so big that the Text Field is below our title we created earlier.

```
24          anchors.horizontalCenter: parent.horizontalPadding
25          anchors.top: parent.top
26          anchors.topMargin: 50
```

For the date and time Input we could now create some fancy option with calendar a different filtering options, but that would need a lot of time and blow this quit simple application out to space. First of we are creating an item which spans is below the first text field and has the same with as it.

```
29      Item {
30          width: titelInput.width
31          height: 50
32          anchors.horizontalCenter: parent.horizontalPadding
33          anchors.top: parent.top
34          anchors.topMargin: 125
35      }
```

Inside this item we will have 2 Text Fields. One for the date and one for time.

```
36          TextField{
37              id: dateInput
38              height: parent.height
39              width: parent.width/3
40              anchors.left: parent.left
41              placeholderText: "0000-00-00"
42          }
```

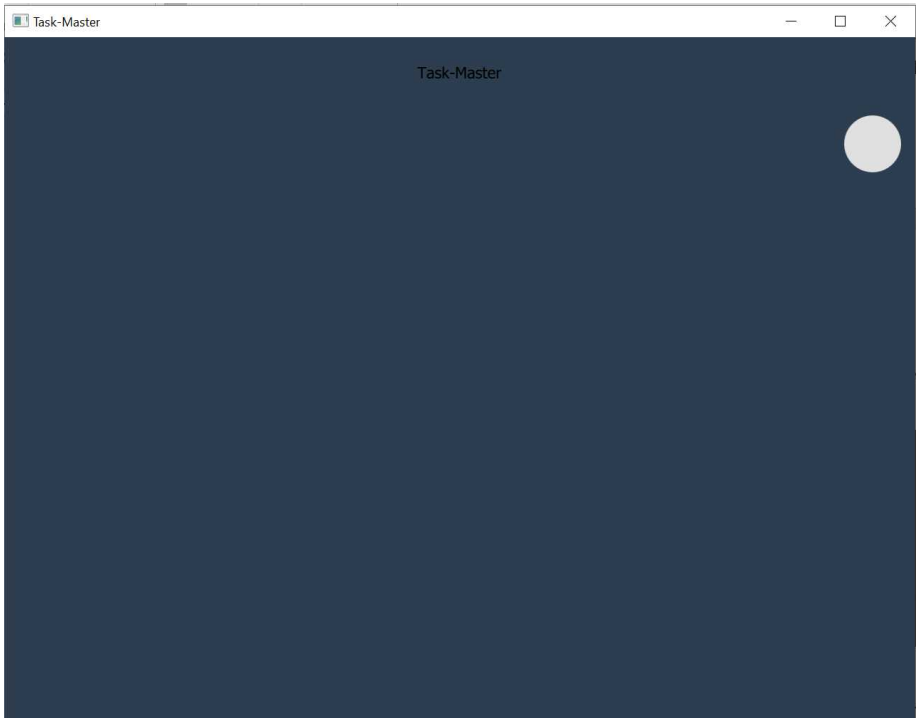
Finished date input with placeholder

Our date input needs an id, as well as a width and a height. The height is the same as the items and the width should be a third of the items. I also added an anchor to the left side, so it stays on the left. A placeholder text was also added. This is in the expected format which we want, to give the user some sort of guide of how to do it. Next is the time input, this is basically the same text field the only differences are the id, anchor, and the placeholder text.

```
43          TextField{
44              id: timeInput
45              height: parent.height
46              width: parent.width/3
47              anchors.right: parent.right
48              placeholderText: "00:00"
49          }
```

Finished time input with placeholder

To see what we did so far let us run the app. For that click, the green arrow in the bottom left, that does not have a bug beside it. If you did everything as I, you will be presented with the app lunching.

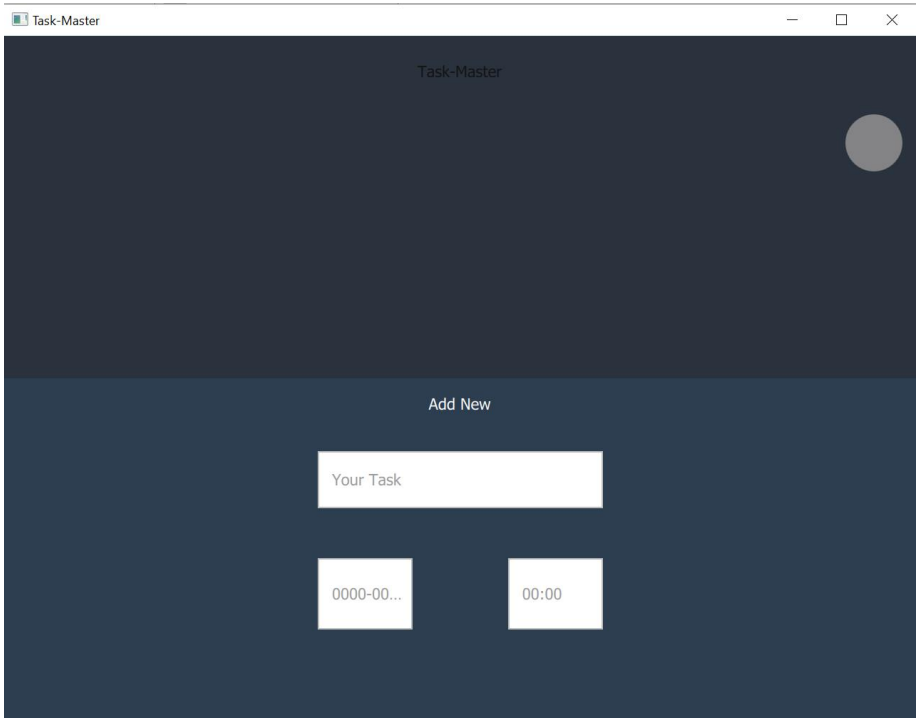


Task-Master Running Main_Page

If we now, click the grey button on the right the drawer will open and we will see our input form so far.

The button is not so currently, and we could do a lot with it, for instance we could change the color, add a text and or make an animation.

We are not going to cover animations in this book, this might be a little bit disappointing, but believe me it would blow this book out of proportions and most importantly they are not essential for a beginner. But I will not leave you hanging either, there in Chapter 3.2.4 Qt Animations, I will give you a brief rundown on the animations you can do with Qt, as well as where and how to learn more about them.



Task-Master Running Add Task Drawer

This looks quit all right so far, the only thing we need to adjust is the width of our date input. It is currently to short which leads to our placeholder text to be cut off. To change this, go to the date input component and set the width to 2.5 / 3 width of the parent.

```
39                  width: parent.width/2.5
```

Now the only thing missing is a button that when clicks checks if everything was filled out and then creates a new item on our List View.

For our button we are going to use a round button again. This is only for stylistic reasons functionally there is no difference between a normal button or a round button. We are going to centre our button horizontally and give fic it to the top of our view. The top margin should be about 200 so it is in the right place.

```
51      RoundButton{
52          id: submitButton
53          anchors.horizontalCenter: parent.horizontalCenter
54          anchors.top: parent.top
55          anchors.topMargin: 200
56          text: "Submit"
57      }
```

The button needs to be added below our item. Also, we are going to give the button a width of 200. This would normally not be necessary, as Qt gives the button a procedural with depending on how long the word is. And 200 is a particularly good with for our button.

```
57                  width: 200
```

Now we come to our click function. This will be the most important part of our application. So, read carefully and refer to the Git Repository when needed.

```
58              onClicked: {
59
60              }
```

First of we need to check the inputs we have in our text fields. Normally you would have a lot more tests, and checks what

inputs are performed but for our use case we only need to check if the inputs are empty or not. If they are not empty, then the program can proceed.

The main part of our application consists out of appending a new item to our List Model. This basically works by using the fieldnames of our data we need and adding the data of our inputs to it. When the button is then clicked the input is written to the model.

For the dateText we are doing something a little bit more difficult. Basically, we are taking the date input and the time input and adding them together, and we are putting " | " in between them as a separator. This basically takes care of our input. This is the easiest way to add stuff to our List Model. This could also be used with a for loop to add multiple items at once, so the possibilities are nearly endless with this.

```
58          onClicked: {
59              if(titelInput.text !== "" &&
60                  dateInput.text !== ""&&
61                  timeInput.text !== ""){
62                  myListModel.append({"titelText": titelInput.text,
63                                      "dateText": dateInput.text + " | " + timeInput.text})
64              }
65          }
```

But if we now would use this to add an item to our List Model, the inputs still retain the data we put in. This is not really what should happen. So, we clean the inputs, so they do not retain any data.

```
62                  myListModel.append({"titelText": titelInput.text,
63                                      "dateText": dateInput.text + " | " + timeInput.text})
64                  titelInput.clear()
65                  dateInput.clear()
66                  timeInput.clear()
```

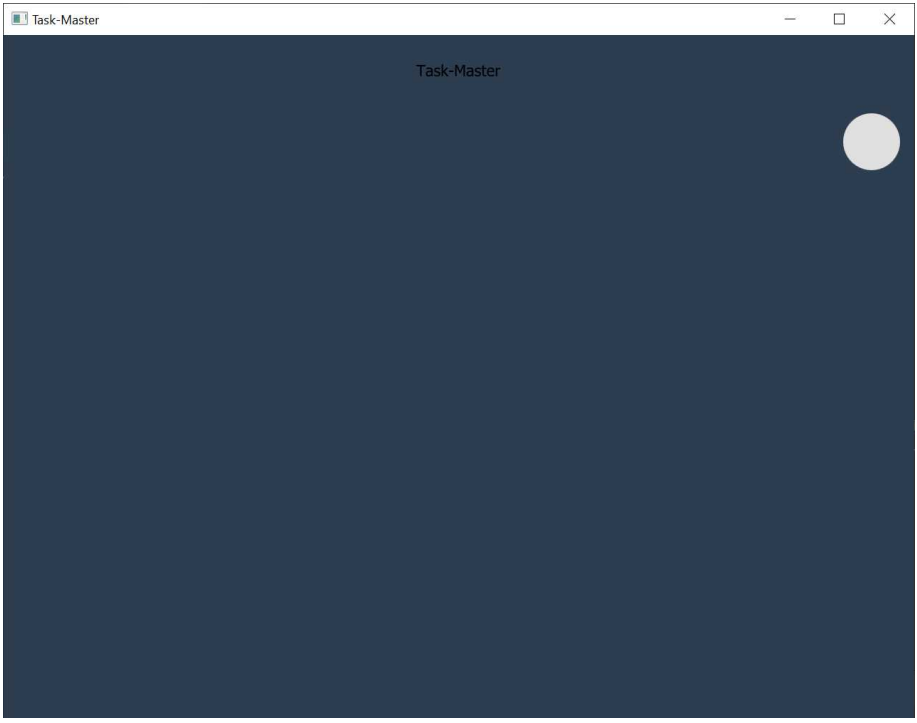
We should also close the drawer because when we put in a new item it should close.

```
67                  addTaskDrawer.close()
```

Now let us see what we have done. Save all the files we created and then click the green button below to build and lunch our application.

This can take a while to compile and then lunch. This mainly depends on your machine and how powerful it is. I currently have a quit dissent machine but depending on the size of the application it takes up to 30 sec to compile and build.

If you are wondering what my machine even is, I have a Ryzen 9 3950X, 32 GB of Ram and a GeForce 1080 from Nvidia. It is nowhere near top of the line anymore, but it is more than sufficient for creating applications. Also, recently I was able to get Qt Creator running on an old Windows 7 Laptop from 2010. So, you do not need a lot of power under the hood.

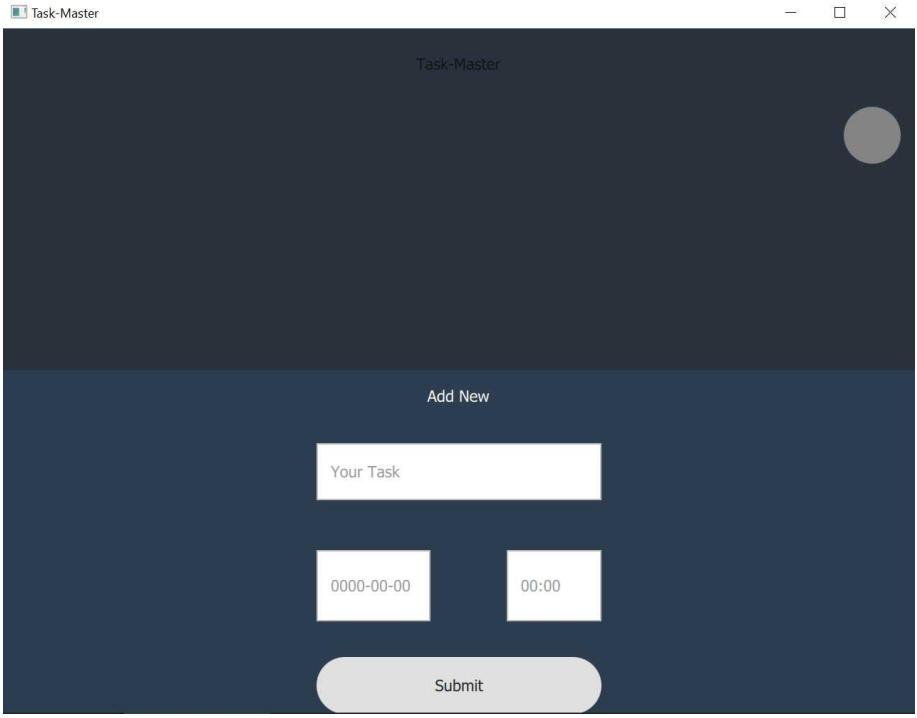


Task-Manager Running Main_Page

When our applications start, we will be greeted by our Load_Page, when our applications have loaded, we are then presented by our Main_Page. Here we have our created List View with our button and our header with our title.

What is shown on screen might not seem that much, but as always it is not important to have the best visuals but the best functionality, as long as that is not your primary goal. We want to learn Qt, so visuals are somewhat important, but the main focus is the functionality.

If we were now to click the button our drawer opens and we can put our data into the inputs. When we do not put anything into our inputs, clicking the button will not do anything.



Task-Manger Running Add Task Drawer not filled out

Here you can see the Inputs with corresponding data, currently we do not have a check function that checks the inputs if they are of the current type and or have the correct meaning.

We could add this, and you would in a real applications standpoint always have your inputs checked so the user is only able to type something in the input you want. But this would be a little bit too complicated for our first application so we can ignore it for now.

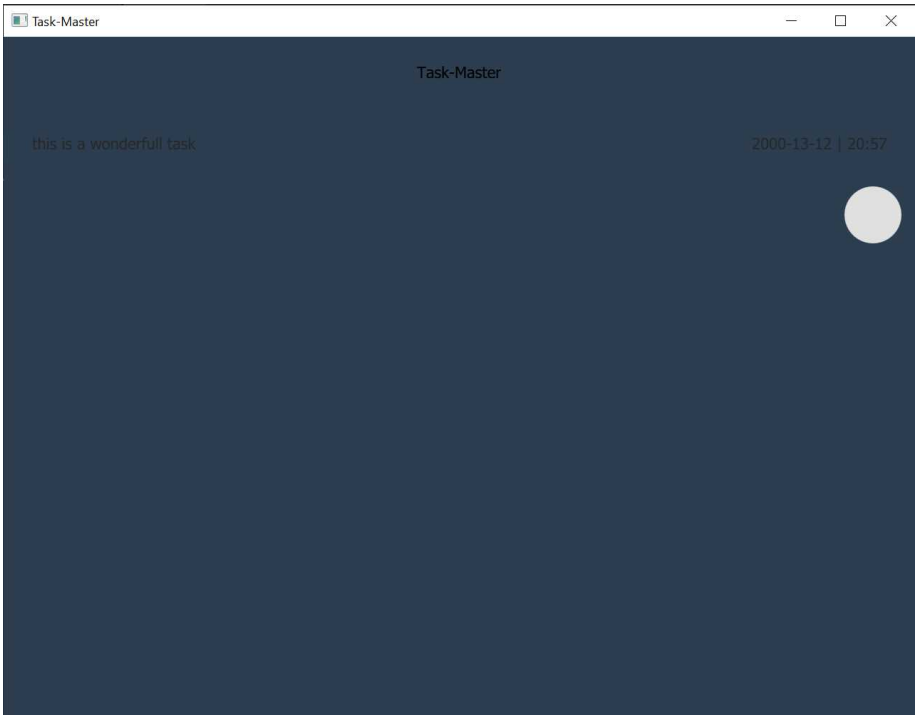
Also, as long as the inputs are not mission critical it is not important if the user typed in the correct stuff. When you have a password or email input it might be especially important to check if the input is correct and you do not want SQL-Injections or wrong inputted data.

Inputs filled with Data

Here you can see our drawer filled out with data. They are only exemplary data, but as you can see, we can put data in, also

because we had the placeholder text, most users will tend to also format there input the same way. And if they do not format it themselves, it is also not bad, as we could format it later on if we wanted to.

But for us we do not need any formatting as we only are displaying the data, and formatting is not that important for us. When we now click the Submit button our drawer is closed and a new item is created in our List Model. As you can see here everything is displayed as it should, and we can see all we need.



Task-Master newly added Item

So technically we are finished with our application. It has the functionality we wanted, and it shows what we wanted. What we are going to do now is making the application a little bit prettier and more usable and fixing up some of the thing we skipped

over. This step is not really needed, but I highly recommend it to you, as it is one of the things you will do all the time if you are a software developer or engineer.

– Deleting Data –

As we stand right now, we can create new entries in our application. And this is all fine and good, but what if you created something which you do not want. Then you should be able to delete it.

This can be achieved most effectively by a Mouse Area and a onPressAndHold signal. A Mouse Area basically is a rectangle that is transparent, and you can then interact with that area. You can click it, interact with a mouse with it, drag it or in this case press and hold it. We can but this inside of our delegate for our ListView and when you press a specific item long enough it will be deleted from the List Model.

```
32          delegate: Item {
33              id: myDelegate
34              width: parent.width
35              height: 50
36
37              Label{
38                  id: titel
39                  text: titelText
40                  anchors.left: parent.left
41                  anchors.verticalCenter: parent.verticalCenter
42                  anchors.leftMargin: 20
43                  color: "white"
44              }
45
46              Label{
47                  id: date
48                  text: dateText
49                  anchors.right: parent.right
50                  anchors.verticalCenter: parent.verticalCenter
51                  anchors.rightMargin: 20
52                  color: "white"
53              }
54          }
```

The fastest way to do this, is by taking our Delegate, and changing our item inside it to a Mouse Area. This will make it possible to interact with the delegate automatically, without needing a tone of editing around.

```
32          delegate: MouseArea {
33              id: myDelegate
34              width: parent.width
35              height: 50
36              onPressAndHold: {
37                  listview.currentIndex = index
38                  myListModel.remove(listview.currentIndex)
39              }
```

We only need the onPressAndHold signal to make this work. The code needed is not that hard to understand, so let me explain. This signal is activated after 800ms, you could change this with an attribute defined in the Qt Docs, but for our purposes and in general I would always stick to 800ms as it is a timeframe which people know by now, so it is not unexpected, and most people have encountered it already.

```
36              onPressAndHold: {
37                  listview.currentIndex = index
38                  myListModel.remove(listview.currentIndex)
39              }
```

The first thing we need to do, is make the current index of our ListView to the index of the Item we are clicking and holding. After that we are taking that index and then removing it from our list model. This is the best way to do this without using C++.

And with that we are now able to delete items we typed in, by simply holding the Item.

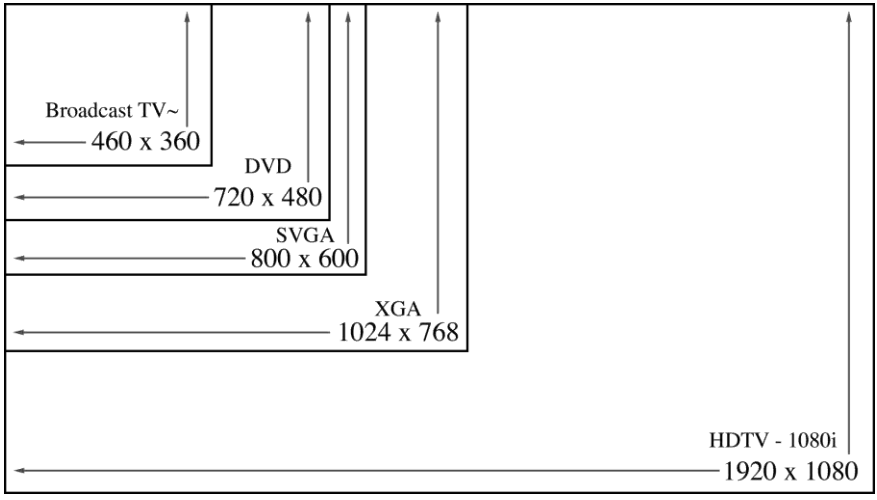
– Cleaning up the Application –

The first thing we can do to make the app a lot prettier is using another width for our application. Having the right size and composition for your app really heightens the usability the user can get.

As we always used anchors for everything, we only need to change the width of our app in one place. What we have in our app is called a responsive layout, so no matter the size of the display the app is usable and works. Changing the width of our application can be done in the main.qml file

```
4  ApplicationWindow {
5      width: 360
6      height: 720
```

We are going to talk about the different types of resolutions that we can use in a later chapter, but just remember, there are some universal resolutions like 1920x1080 or 720x480.



A representation of some common resolutions

This is also the case for mobile applications, just remember that for some smartphones out there the resolution is not 16:9, this means you need to take this into account when you developed for these platforms.

Also, we should change the colour of our text. Having a dark background and black text is not really that readable. So best of we change this to a white text colour.

```
22              Label{
23                  anchors.centerIn: parent
24                  text: "Task-Master"
25                  color: "white"
26              }
```

Titel of our applciation

```
37          Label{
38              id: titel
39              text: titelText
40              anchors.left: parent.left
41              anchors.verticalCenter: parent.verticalCenter
42              anchors.leftMargin: 20
43              color: "white"
44          }
```

Delegate Text for the title of the tasks

```
46          Label{
47              id: date
48              text: dateText
49              anchors.right: parent.right
50              anchors.verticalCenter: parent.verticalCenter
51              anchors.rightMargin: 20
52              color: "white"
53          }
```

Delegate Text for the date of the tasks

You can also use other colours you want. The last thing I really want to change is our button that is sued for opening our drawer to put in new items. Currently it is empty and does not display anything. This is not good. If a user were to see this it would not

be easy to figure out what the button does. So, the fastest way to make this a bit more readable is adding a + as the text of the button.

```
60              RoundButton{
61                  id: addTaskButton
62                  width: 40
63                  height: 40
64                  anchors.verticalCenter: parent.verticalCenter
65                  anchors.right: parent.right
66                  anchors.rightMargin: 10
67                  text: "+"
68                  onClicked: {
69                      addTaskDrawer.open()
70                  }
71              }
```

This really improves the readability of the button and a user can figure out what you mean when they see the button.

And these are all the changes I would do to finish and make our app a little bit prettier and more usable. You also continue to work and refine the application we created, but for us we are going to next topic. Please do not delete the app we created, as we are using it as example for other topics later, and we are going to implement a few features later, which are not essential for the application, but are best explained in this type of application.

– Deploying the Application –

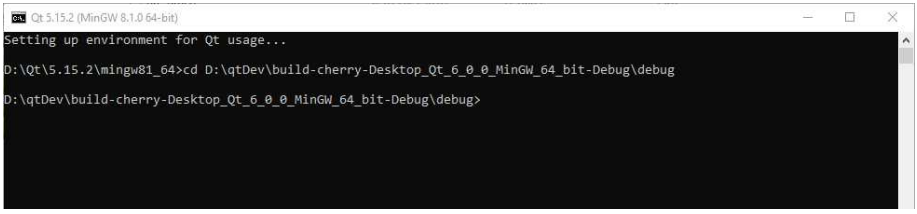
This is the last topic we are going to go over with this application. And that is deploying an application for a platform. In this case we are going to deploy it for windows. If you want to deploy it for another platform, then you can look up the chapter about that specific platform later in the book.

Now how do you need to go about deploying your application to a MinGW platform? Well first of why we do this in MingGW,

there are a lot of other platforms that you can build and deploy on, but MingGW is what I use generally, and it is the one that I run in quit a lot over the years.

First of you need to build your application, this will always create a folder for you in the same directory as your project is, unless you specified another location. When you have done this, you need to use the windows search to find the MinGW 8.1 command line interface. If you have open it up, we can continue.

First of we need to go to the directory of our project. You can do this by using the *cd* command and paste the path to the correct directory behind it.



Version here is Qt 5.15.2

Now to the command that gets all our dependencies so we can actually use the application. Because if you did not know already, you cannot use applications build throw Qt without also copying the dependencies and the Qt files needed over, this is only the case if you did not configure Qt as a static library, but if you followed this book along you will not have done that.

*windeployqt --qmldir **[Path to Project] [Path to the .exe File of our Application]***

This is the command you need to run the deployment of our application. It is not really difficult to understand. When you located the .exe file which represent our application, you need to type windeployqt. This is the standard tool provided by Qt for

deploying applications to the Windows platform. Next, we need to type –qmldir, and behind that we need to specify the path to our project and again the path to the .exe file of our application. Important is that you need to name of the application as well as the .exe at the end to the second path we need for the qmldir.

When you typed all this in the CLI you can then press enter and the build process is going to start. This can take a few minutes to up to half an hour depending on the size of the project and how many different dependencies you take with you.

When everything is done, you will see that there are a few new files added to the directory of the .exe. all of these files need to be taken with the .exe file when you want to run this application on another device. Also, the directory right now is extremely large, usually between 1 and 3 GB.

This is extremely large for a program, and we are not talking about an overly complicated program, even small ones take this much space to deploy. So, keep this in mind.

And with this we are done deploying our application to the windows platform, when you want to learn more about this, go check out the Qt Docs on the matter, and if you are wondering if we are going to do this for android then you are absolutely correctly, but we are going to do this in a later chapter.

– What did we learn –

With this application we learned about List Views in Qt, how to set them up, how to use them and how we can add Items to it. This is not a hard app, but that is not the point. The point was to make you a little bit more comfortable using Qt and figuring out how the structure and a project in Qt even works. So, to give a broad overview let us have a quick list of what we learned.

- **List Views**
- **List Models**
- **How to add and delete data to a Model**
- **Buttons**
- **Text Fields and input types**
- **How to move throw Qt Creator and work with it**

The knowledge you learned in this chapter will be needed in later chapters, so feel free to go back and read up on all the topics you want. Most important will be the ListView, and the Drawer/ Popup component.

These are as I already mentioned some of the most important components in Qt, and you cannot really build an application without them. Well, as some people will probably mention you could try building something like a Popup on your own, and you might be able to do this, but Qt already provides you will a really good solution so you should probably use it.

As a side note, there are also a lot more component Qt has, which could really help you in development and make your development a lot faster and smother, also if there is not a component provided by Qt for a specific instance, then remember that all Qt components can be edited and manipulated throw the attributes and properties until they fit your vison and what you need.

And lastly, I am going to give you now full code snippet of our current code. Why you might ask, well mainly for the purpose that if you have come this far, you can always just refer to these screenshots to see how something is done. And if you were stuck in between steps before you can see how it should be done.

main.qml

```
import QtQuick 2.15
import QtQuick.Controls 2.12

ApplicationWindow {
    width: 360
    height: 720
    visible: true
    title: "Task-Master"

    StackView{
        id: contentFrame
        anchors.fill: parent
        initialItem: Qt.resolvedUrl("qrc:/Load_Page.qml")
    }

    Component.onCompleted: {
        contentFrame.replace("qrc:/Main_Page.qml")
    }
}
```

Load_Page.qml

```
import QtQuick 2.15
import QtQuick.Controls 2.12

Item {
    width: parent.width
    height: parent.height

    Rectangle{
        id: bgRec
        anchors.fill: parent
        color: "#2C3E50"
```

```
        BusyIndicator {
            id: busyIndicator
            anchors.centerIn: parent
        }
    }
}
```

Main_Page.qml

```
import QtQuick 2.15
import QtQuick.Controls 2.12

Item {
    width: parent.width
    height: parent.height

    Rectangle{
        id: bgRec
        anchors.fill: parent
        color: "#2C3E50"
    }

    ListView{
        id: listview
        anchors.fill: parent
        header: Item {
            id: headerListView
            width: parent.width
            height: 50

            Label{
                anchors.centerIn: parent
                text: "Task-Master"
                color: "white"
            }
```

```
}
headerPositioning: ListView.OverlayHeader
model: ListModel{
    id: myListModel
}
delegate: MouseArea {
    id: myDelegate
    width: parent.width
    height: 50
    onPressAndHold: {
        listview.currentIndex = index
        myListModel.remove(listview.currentIndex)
    }

    Label{
        id: titel
        text: titelText
        anchors.left: parent.left
        anchors.verticalCenter: parent.verticalCenter
        anchors.leftMargin: 20
        color: "white"
    }

    Label{
        id: date
        text: dateText
        anchors.right: parent.right
        anchors.verticalCenter: parent.verticalCenter
        anchors.rightMargin: 20
        color: "white"
    }
}
footer: Item {
    id: footerListView
    width: parent.width
```

```
            height: 50

            RoundButton{
                id: addTaskButton
                width: 40
                height: 40
                anchors.verticalCenter: parent.verticalCenter
                anchors.right: parent.right
                anchors.rightMargin: 10
                text: "+"
                onClicked: {
                    addTaskDrawer.open()
                }
            }
        }
    }

    AddTask_Drawer{
        id: addTaskDrawer
    }
}
```

AddTask_Drawer.qml

```
import QtQuick 2.15
import QtQuick.Controls 2.12

Drawer {
    id: addTask_Drawer
    width: parent.width
    height: parent.height/2
    edge: Qt.BottomEdge
    background: Rectangle{
        anchors.fill: parent
        color: "#2C3E50"
```

```
}

Label{
    anchors.horizontalCenter: parent.horizontalCenter
    anchors.top: parent.top
    anchors.topMargin: 10
    text: "Add New"
    color: "white"
}
TextField {
    id: titelInput
    placeholderText: qsTr("Your Task")
    anchors.horizontalCenter: parent.horizontalCenter
    anchors.top: parent.top
    anchors.topMargin: 50
}

Item {
    width: titelInput.width
    height: 50
    anchors.horizontalCenter: parent.horizontalCenter
    anchors.top: parent.top
    anchors.topMargin: 125

    TextField{
        id: dateInput
        height: parent.height
        width: parent.width/2.5
        anchors.left: parent.left
        placeholderText: "0000-00-00"
    }
    TextField{
        id: timeInput
        height: parent.height
        width: parent.width/3
```

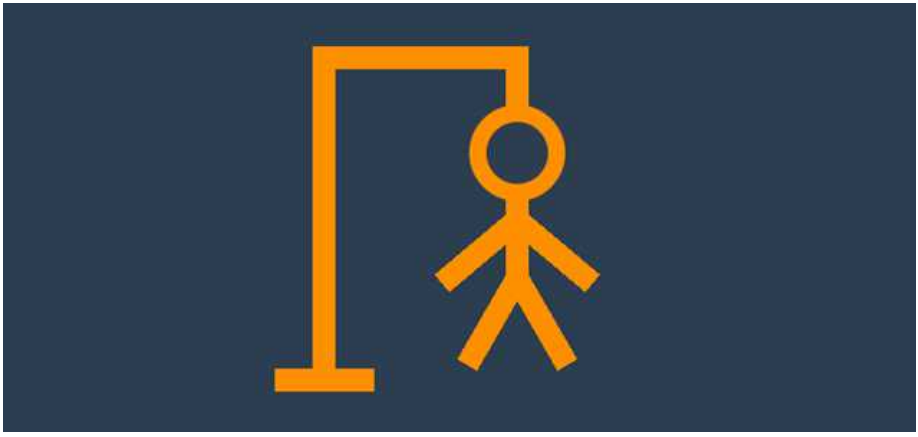
```
            anchors.right: parent.right
            placeholderText: "00:00"
        }
    }
    RoundButton{
        id: submitButton
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.top: parent.top
        anchors.topMargin: 200
        text: "Submit"
        width: 200
        onClicked: {
            if(titelInput.text !== "" &&
                dateInput.text !== ""&&
                timeInput.text !== ""){
                myListModel.append({"titelText": titelInput.text,
                                    "dateText": dateInput.text + " | " +
timeInput.text})
                titelInput.clear()
                dateInput.clear()
                timeInput.clear()
                addTaskDrawer.close()
            }
        }
    }
}
```

This is the only time I will provide you with a code snippet at the end of the project, the other projects we are going to do are a little too long to really print them out as a code snippet. For the other projects you need to go to my Git Hub @BenCoepp. There you will find the project and all the code you are looking for. You can also just search for the project title and Qt or QML behind it with that you should also find my projects. So, let us jump right into the next project.

2.4.2 Hang-Man

This is going to be another extremely easy app we can have a look at. The main principal of how this game works should be familiar with most people in the world. But to those who do not know, you have a word that the player needs to figure out, the player can select letters from the alphabet, and if the letter exists in the word then it is added at the corresponding place, but if the letter chosen by the player was wrong then the Hang-Man, a figurative stick figure is going to start being visible. If the Hang-Man is visible completely then the player has lost. If the player guesses the word correctly then the player wins.

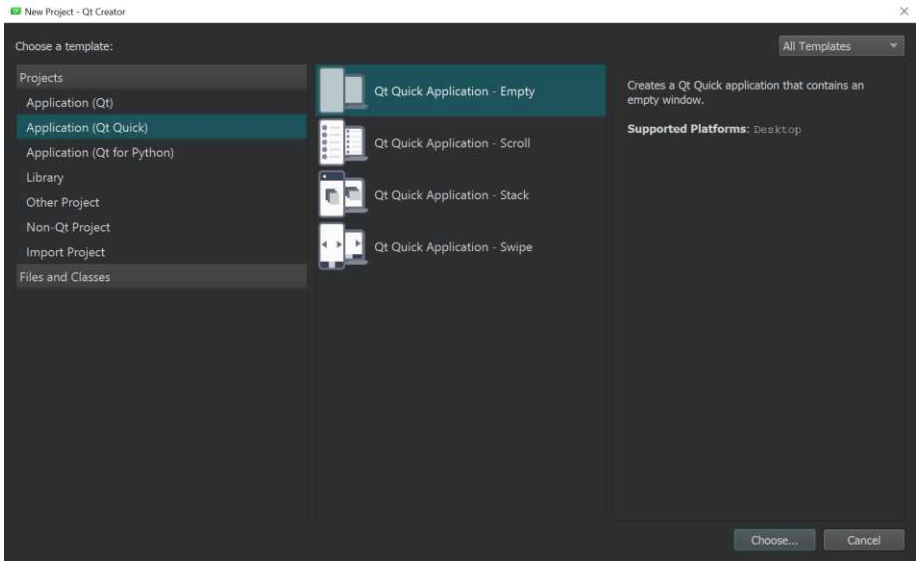
So, in its essence not to complicated, but it allows us to have a look at a few different things we have not looked at so far. First of a little bit more with JavaScript and we can also experiment a little bit more with Qt's visual components.



- Create Project -

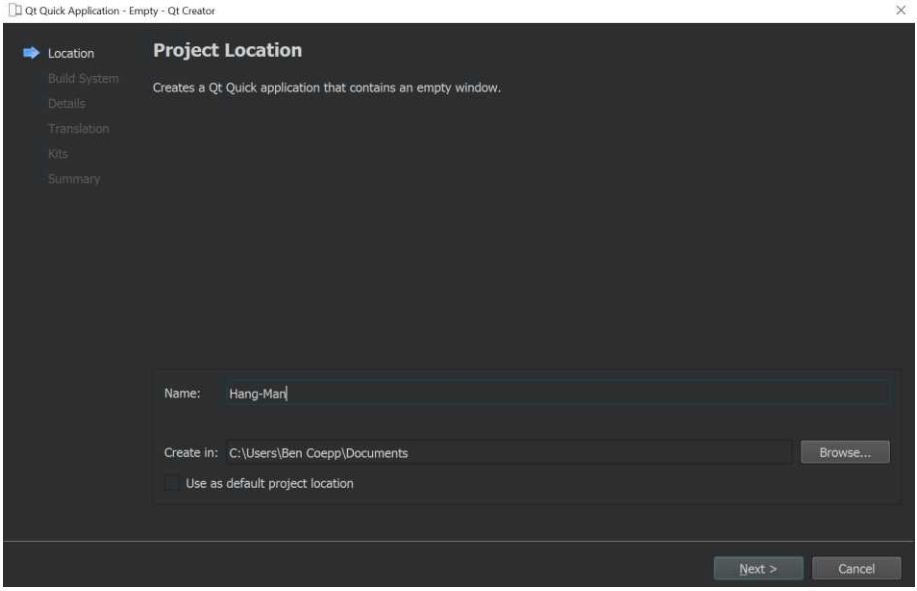
First of let us create our project. This will be a quick round about, mainly because we already did this one and because this would be exceedingly long if we did go throw everything again.

If you are not so confident with creating projects, you can also refer back to the chapter we did this in.



New Project Creation Wizard

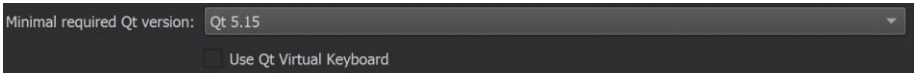
As before we are going to choose Qt Quick Application Empty as our template. We want to make a Qt Quick Application as always, and I do not like having any boilerplate components inside my application which are going to be thrown out no matter what.



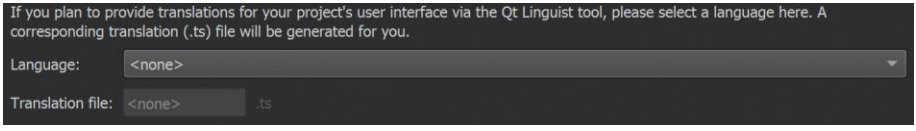
Save location and titel Wizard Page

For the name I chose Hang-Man we want to make a Hang Man application, so this is very fitting in my opinion. For the location you can again choose anything you. But my recommendation is to choose something that fits the application that you want to build and that you can remember.

The Build System we can leave on qmake as it is the one Qt has as the default and for us it will do the trick.

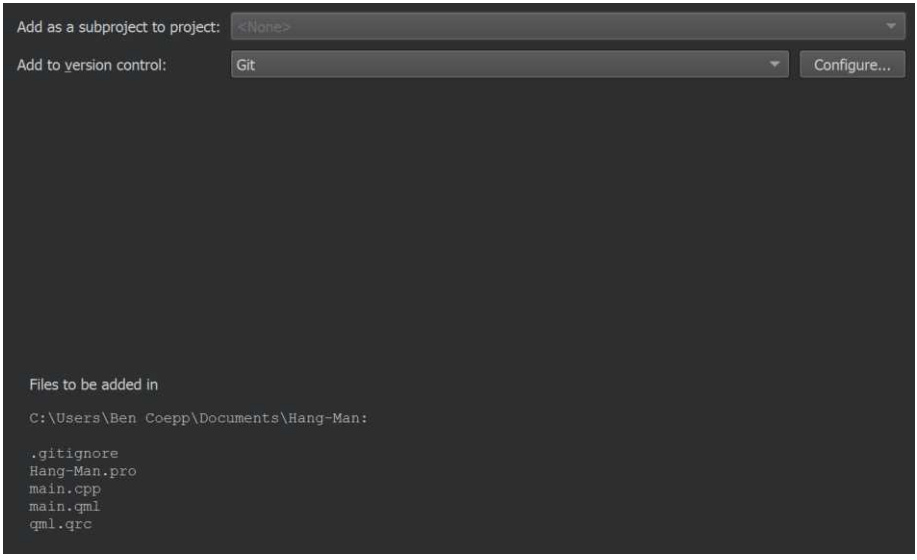


I will use 6 for the development of this application, but there is no difference between Qt 6 or 5.15 in what we are building now, but you should always use the latest available Qt version, as a lot of bugs and problems will be eliminated.



Again, we do not need a Translation file, so leave this as it is and hit next.

For developing the application, we do not need android for now, so choose MinGW and the appropriate but rate for your system and hit next.



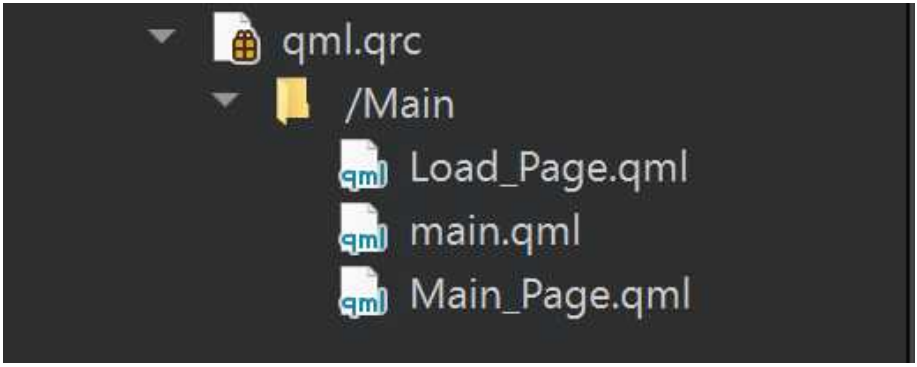
Add to version control Wizard page

Lastly, I choose Git as my Version Control, this step is optional, and I will not go over how I used Git in conjunction with the project, how Qt interacts and works with Git will be covered in a later chapter[18]. But having projects like this in your Git Repository is something great and I would always recommend this to you.

[18] In chapter 3.2.4 Qt in Qt, also in the Rock-Paper-Scissor Game project we will be using Git

- Load and Main Page -

Now as before we want to have a simple Load and Main Page setup, so that when the application loads the user is not left wondering what the hell is going on.



First of we need to change the Prefix in which our main.qml file is located in this can be done by exactly right clicking the Prefix and then selecting change Prefix. Next, we need two new files, one Load_Page.qml and one Main_Page.qml. Both should be located inside our Main Prefix.

Now if we wanted to run our application it would spit out a nasty error telling us that this cannot work, because our main.qml file does not exist anymore.

```
12:46:05: Starting C:\Users\Ben Coepp\Documents\build-Hang-Man-Desktop_Qt_5_15_2_MinGW_64_bit-
Debug\debug\Hang-Man.exe ...
QQmlApplicationEngine failed to load component
qrc:/main.qml: No such file or directory
QML debugging is enabled. Only use this in a safe environment.
12:46:06: C:\Users\Ben Coepp\Documents\build-Hang-Man-Desktop_Qt_5_15_2_MinGW_64_bit-
Debug\debug\Hang-Man.exe exited with code -1
```

This stems from the fact that we changed the Prefix of our main.qml so let us jump right over to our main.cpp and fix the problem.

```
1  #include <QGuiApplication>
2  #include <QQmlApplicationEngine>
3
4  int main(int argc, char *argv[])
5  {
6      QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
7
8      QGuiApplication app(argc, argv);
9
10     QQmlApplicationEngine engine;
11     const QUrl url(QStringLiteral("qrc:/Main/main.qml"));
12     QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,
13                      &app, [url](QObject *obj, const QUrl &objUrl) {
14         if (!obj && url == objUrl)
15             QCoreApplication::exit(-1);
16     }, Qt::QueuedConnection);
17     engine.load(url);
18
19     return app.exec();
20 }
```

And with only the small change in line 11 we are done with the main.cpp file. Now the application starts again, and we want to have the problems again.

```
1  import QtQuick 2.15
2  import QtQuick.Controls 2.12
3
4  Item {
5      width: parent.width
6      height: parent.height
7
8      Rectangle{
9          id: bgRec
10         anchors.fill: parent
11         color: "#2C3E50"
12
13         BusyIndicator {
14             id: busyIndicator
15             anchors.centerIn: parent
16         }
17     }
18 }
```

Now inside of our Load_Page we need this code, it consists out of an item for our root component, a rectangle inside of it for our background and a Busy Indicator so that we have a visual representation of the loading process, this is not always needed, but it is nice to have, as the user wants to know what the app is doing. That is also the hole reason for the Load_Page.

```
import QtQuick 2.15
import QtQuick.Controls 2.12

ApplicationWindow {
    width: 360
    height: 640
    visible: true
    title: "Hang-Man"

    StackView{
        id: contentFrame
        anchors.fill: parent
        initialItem: Qt.resolvedUrl("qrc:/Main/Load_Page.qml")
    }

    Component.onCompleted: {
        contentFrame.replace("qrc:/Main/Main_Page.qml")
    }
}
```

Our main.qml file also needs some changes, first we need to update our imports we use inside our main.qml file. Then we can go ahead and change window component to an ApplicationWindow as before and change the width and height of it so something that resembles more the resolution and aspect ratio of a mobile device. The title we can also change to the title of our application, as this is what we are developing.

Now we can go over the main part of what we need here. Our Stack View. This component allows us to load components and replace them when everything is finished loading. We already went over how this works and how to set this up, and in matter effect both things we changed are the same as what we did in the previous application. So, if you wanted you could also have copied it over, but then you would not learn anything.

With these things changed we are now done with the Load and Main Page Setup; next we are going to go over what the application needs to do and how we want to achieve this.

- Functionality -

We want to make Hang Man, so first of we need to clear out what that is and what we need to make it.

“a game for two in which one player tries to guess the letters of a word, the other player recording failed attempts by drawing gallows and someone hanging on it, line by line”

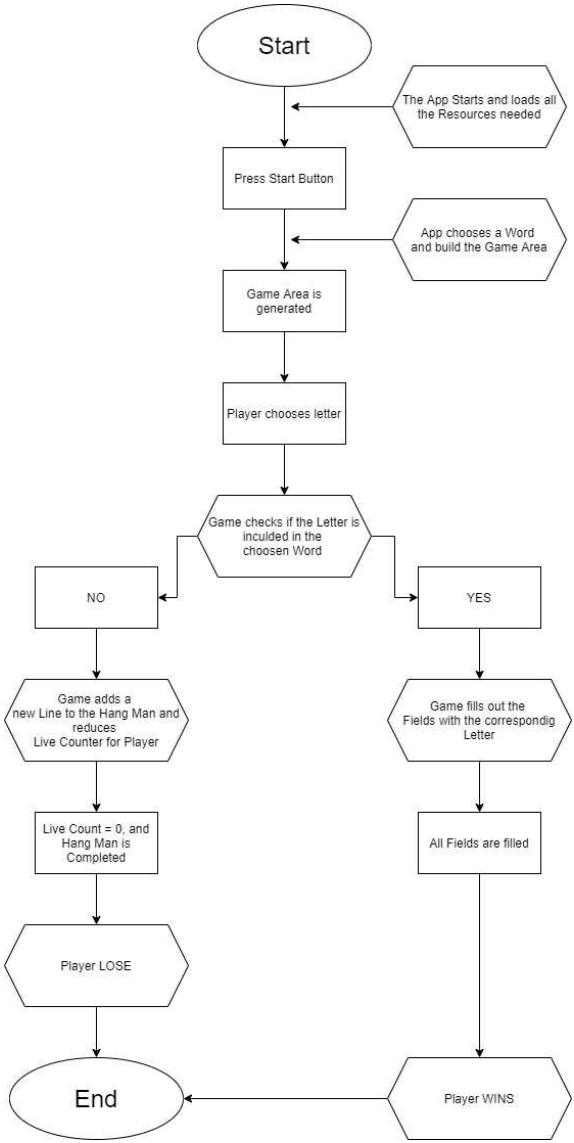
That is the official Wikipedia definition of the game and the one we are using as our guideline for our application. The two players mentioned are for us the computer, and the actual player guesses the letters of the word. But what rules do we need for our application.

- **Only 10 words allowed for the word the computer chooses, we do not want to make it too hard for the player**
- **The player fails when he chooses 10 letters wrong. When this happens, the Hangman is completed**

These are 2 rules our application needs to follow to be an interesting game. It is particularly important that you always have rules for your application. This makes it easier to build the application as you already have a rough outline of how the application should word, feel, and look like.

To get a little bit more technical I will also show you now a diagram of how the application works and the interactions the user will perform. I created this diagram using one of the many different websites out there, where you can create diagrams for free.

I would recommend you try building the diagram on the next page on your own, I know that I probably did not do a really great job with the diagram, but I hope you can read it, and that it is possible to follow along.



This is a quite simple diagram of how the app should function, but it is all we really need. Why are we doing this here and not for the first application? Well, it was not necessary. It takes time

and effort to make these diagrams and they are not needed all the time. Here I think it is a good Idea to do one, first of it allows me to easily explain what we need to do and how to do it, and I have a clear way of going about building each function we need.

So, if you are a bit lost then refer to this chapter and the diagram above, this will help you get back on track.

Overall, these diagrams are a must have for larger and more complex applications and programs and I will go over some of the tricky things you should keep in mind when writing diagrams for Qt applications.

– Building the App –

The first thing we need to build is the game area itself. This in my opinion is best done by using a Swipe View[19]. It makes it easy for the user to understand how the app works, and we can put all the necessary pages on this Swipe View without needing to create four different files.

```
1  import QtQuick 2.15
2  import QtQuick.Controls 2.12
```

Updated imports

As always, the first thing we need to do is change the imports we currently have in our Main_Page.

After that we can start building our Swipe View. This is the basic boilerplate version of a Swipe View, so you can find the code for it also on the Qt Docs for the Swipe View.

[19] Read up on chapter 3.1.3, but Swipe Views are great because they allow you to have a great UI with swipe functionality which for a mobile game is great

```
    SwipeView{
        id: swipeView
        anchors.fill: parent
        interactive: false

        Item{
            id: welcomePage
            width: 360
            height: 640

        }

        Item{
            id: gamePage
            width: 360
            height: 640

        }

        Item{
            id: endPage
            width: 360
            height: 640

        }
    }
```

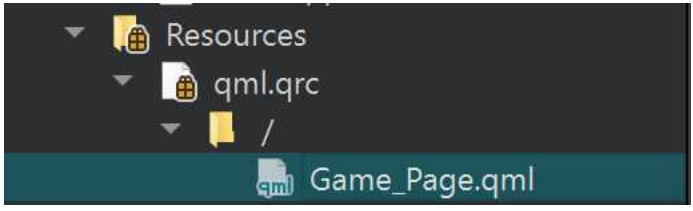
Swipe View in Main_Page.qml

First of the Swipe View needs an id, width height, the id is straight forward and for the width and the height we can use an anchor, so it stays responsive.

We also need the attribute interactive, we do not want the user to swipe and change the view before we want to, so we need to set this to false.

Next up we can create three items inside of our Swipe View, these will be the pages we have in our game. Currently they are the same, only the id is different. You can use a property here to, like anchors.fill which would also perfectly work here. I just used a fixed size here as it is easier to understand and because I do not think that it would make any difference here. But if you were to build this application for production a property would be the better choice.

Next up we can create a new file for our Game Page. Currently it would work just sitting inside the Swipe View, but with the amount of stuff we are going to do inside the Game Page it should really be in its own QML file. So, lets create the Game_Page file.



Create the new file the standard way we did it all the time, and a new empty Prefix will create with the Game_Page.qml file inside of it. We could now use it already, but we want to have a more telling Prefix. Next, we should change the Prefix to something a little bit more reasonable.

With this out of the way, we can change the code inside the Game_Page file to the same we have for our Item in our Swipe

View. We also need to update the imports to the usual. Now we can import our Game_Page into the place where the game page item is located. But before we can do this, we must import the Prefix into there.

```
3   import "qrc:/Game_Page"
```

Import Game_Page.qml

Importing the Prefix like this allows us to not only use the item in our Game_Page.qml file, but also all the components that are inside this Prefix. So, when we would create a custom button for instance, we could also use it somewhere else now.

```
 9   SwipeView{
10       id: swipeView
11       anchors.fill: parent
12       interactive: false
13
14       Item{  id: welcomePage...}
20
21       Game_Page{
22           id: gamePage
23       }
24
25       Item{  id: endPage...}
31   }
```

Swipe View at this point

Lastly, we need to import the Game_Page as a component and give it an id. With this id we can later interact with the component. It might not be necessary but in my opinion, it is always a good practise to always give every component you are using a telling id.

Now we can start building the Welcome page. One of the first things we should do is give the entire app a background.

Currently everything is white, which if you want to go for a clean and Apple inspired look is great, but I do not like white so let us change it.

```
9       Rectangle{
10          anchors.fill: parent
11          color: "#2c3e50"
12      }
```

Just above our Swipe View we can create a new rectangle with anchors. fill parent and then set the color to our preferred background color. If you are wondering where I get all these colors from, then you can simply go to my Git Hub or my website and find a color pallet there. You can use that for all the colors I am going to use, or you can use your own colors however you like.

Not to the actual welcome page, what should even be on there? Well, there are only four things that are important.

- **The title of the app**
- **The button to start the app**
- **Who made the app?**
- **A link to the user agreement and other legal stuff**

Why is the last point important? Well I want to publish this application to the Android App Store and if you want to do that you need some legal documents, they are not hard to get and there are a lot of players who can write them for you but it is very important that you have these when you want to publish a app, if you do not have them you are in a lot of danger for potential law suits and other nasty legal stuff.

Other than that, all the above-mentioned points are fairly straight forward and easy to follow, so let us jump right in.

```
19          Item{
20              id: welcomePage
21              width: 360
22              height: 640
23
24              Label{
25                  id: gameTitel
26                  anchors.horizontalCenter: parent.horizontalCenter
27                  anchors.top: parent.top
28                  anchors.topMargin: 50
29                  text: "Hang-Man"
30                  color: "white"
31                  font.pointSize: 50
32              }
```

Welcome Page with titel Label

We can start of the build of our Welcome page with the title of our application. I will use a Label for that and position it to the horizontal centre and to the top of the parent item. The text is quite self-explanatory, as well as the color. I also choose to give the title a large font size, so it is clearly visible.

As you are probably aware, we are building the welcome page inside of our Item. This is for two reasons one is because this will save us creating another file for it and cluttering up our project tree and because I think that it is not such an important page in our application as that it is essential to have it in its own file.

A little side node the best welcome pages are the ones that are creative and have a unique design. Try to always think of something that best represents the application that you want to build and how this could be best implemented in an application. But do not be afraid to use a standard and safe design when you think this would fit better.

```
34            RoundButton{
35                id: startGameButton
36                anchors.centerIn: parent
37                width: 200
38                height: 200
39                text: "START"
40                font.bold: true
41                font.pointSize: 38
42                background: Rectangle{
43                    anchors.fill: parent
44                    radius: 99
45                    color: "#fe9000"
46                    border.width: 2
47                    border.color: "black"
48                }
49                onClicked: {
50                    //Logik that Pics a Word
51                }
52            }
```

Next is the round button we will use to start the game with. Because we want the button to be round, we can immediately use a round button, we should centre it to the parent, and give it a size of 200 the text should be START as we want to start the application when we click this button. We can also manipulate the text a little by making the font bold and the size 38. Clearly readable but not too large. Now we come to a trickier part. Because we want to have full control over how this button looks and feels we are going to use the background attribute, we need to do this to use a rectangle as our background. This rectangle should have the same size as the button, we can also set the color for our button here. I also choose to have a border around the rectangle, this means that now there is a nice black border that clearly separates the button from the background.

Lastly, we can also implement the onclicked signal to the button, we are going to fill this with the necessary logic later, but for

now you can know that here will be the place where the application will pick a random word and we will switch to the next page.

```
54              Label{
55                  anchors.horizontalCenter: parent.horizontalCenter
56                  anchors.bottom: parent.bottom
57                  anchors.bottomMargin: 50
58                  text: "Made by BEN COEPP"
59                  color: "white"
60                  font.pointSize: 15
61              }
62
63              Label{
64                  anchors.horizontalCenter: parent.horizontalCenter
65                  anchors.bottom: parent.bottom
66                  anchors.bottomMargin: 10
67                  text: "User Agreement other Legal Stuff"
68                  color: "white"
69                  font.pointSize: 8
70
71                  MouseArea{
72                      anchors.fill: parent
73                      onClicked: {
74                          // Link to Legal Documents
75                      }
76                  }
77              }
```

Lastly, we have another two Labels, one which is remarkably like the title Label, just anchored to the bottom and not to the top. The font size is also a little bit thinner than the title Label. The other label is the link to the legal documents, what I usually do is have a label at the bottom of the Page, and inside this Label is a Mouse Area which fills the entire Label. When you click on the Label then you will be bought to another page or to a website where all the legal documents are located.

This is the basic visual stuff for our welcome page, it is quite simple, and we could also improve on this, when we for instance added animations to the clicking of the button. But for the simplicity of this project, we will not do that.

Next, we should think about the functions needed to start up the game. And that means we need a form of model where all the words we want are located inside.

```
14      property var currentWord: ""
15      ListModel{
16          id: wordModel
17          ListElement{
18              word: "Tree"
19          }
20          ListElement{
21              word: "Apple"
22          }
23      }
```

For that the only thing we really need to do is create a List Model, give it an id, and place some list elements inside of it. The List Model can be added inside of our Main_Page file, as the data is needed there. You might think of adding this to the Game_Page but because of inheritance reasons this is not possible, you would not be able to use the id to call on the data from there.

The only thing we need for data inside of our list elements is the word itself. We could provide more data, like letter count. But this requires us to always have this data for every list element. It is far easier to generate this data by looking at the word itself each time.

As you can see above the List Model, I also have a property called currentWord, this will be used to hold the word we choose throw the function we are building now.

```
60          onClicked: {
61              currentWord = wordModel.get(Math.floor(Math.random() * wordModel.count))
62              console.log(currentWord.word)
63          }
```

In our onclicked function for our round button this function should be placed. What does it do you might ask? Well simply put, it generated a random number that is between 0 and the count of the List Model we created.

```
Math.random() * wordModel.count
```

This is what this code does, the function that is around that, rounds that number to a full integer, because we are then using this randomly generated integer to get a specific element from our List Model, we cannot use not round integers.

If you were now to run the application and see what currentWord holds on data, you might rely on that we copied the entire model into there. This might not be necessary, but if we were to add more data to each element in our List Model it would be far easier to get that then, because we already coded everything to work with the entire object rather than only the data.

If you ask yourself now how you get the data from the object, we will have a look at the console.log after our function. Here you can see that you only need to add. word to our currentWord and the object will provide the data for us.

With this we know have a simple function that generates a random word from a list of word we provide. The only thing left to do is switching the program over to the Game_Page in our application.

```
60                  onClicked: {
61                      currentWord = wordModel.get(Mc
62                      console.log(currentWord.word)
63                      swipeView.setCurrentIndex(1)
64                  }
```

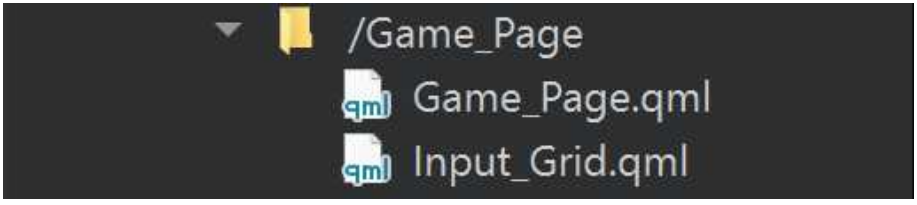
Below our console.log you can add this code snippet to make the switch possible. Basically, you tell the Swipe View we

created earlier to switch to the item that has the index 1, and this is the Game_Page. There are a few more ways you could do this, but I really like this solution as it is remarkably simple and easy to implement.

And with this we are done with the welcome page, next up we are going to create the Game_Page visual components. Mainly we need the following in our app, a ListView that is horizontal that shows all the potential letters that exist and which we filled out. We also need a way to input the letters we want. For that there are a few ways to do it. You could use a Grid View and display all the letters in the alphabet and then the player could click on them, or we could use a normal text Field and let the player type in the letters. Both variants work, but the latter one is not as refined and good as the first, also we would need to implement a check function so that the player only types in letters that are allowed. So, we will be using a Grid View for that.

And lastly, we need a way to display the Hang Man in such a way so that when the count of the players wrongly guessed letters increases is made more and more visible.

But first of let us start by creating the input fields for our application.



First of we need to create a new file under our Game_Page Prefix. For name keeping sake we should call it Input_Grid.

```
4  GridView {
5      id: inputGrid
6      width: parent.width
7      height: parent.height/3
8      anchors.bottom: parent.bottom
```

The Grid View we create is really like the type of List Views we did so far. We first of need an id, as well as a width and height. Both can be derived for us from the parent. We also can immediately set the anchors to the bottom of our parents bottom as we know that that is where we want to locate our Grid View.

```
9      cellHeight: 50
10     cellWidth: 50
```

These are two new attributes we did not yet discuss, basically this is the size of each item inside the Grid, normally this is 100 but that is far too large for our needs. So, we can set this to 50.

```
11     model: ListModel{
12         ListElement{
13             letter: "A"
14         }
15         ListElement{
16             letter: "B"
17         }
```

Know the List Model is a real bummer. We need to create a new list element for each letter in the alphabet, we could also do this programmatically, but I am far too lazy to do this, and for what I know there are only a few ways to do this a little bit easier then typing everything out but that I leave to you.

```
91      delegate: MouseArea{
92          width: 50
93          height: width
94          onClicked: {
95              borderRec.border.width = 1
96              //Send letter to the Test Function
97          }
98
99          Rectangle{
100             id: borderRec
101             anchors.fill: parent
102             color: "transparent"
103             border.width: 0
104             border.color: "white"
105
106             Label{
107                 anchors.centerIn: parent
108                 text: letter
109                 font.pointSize: 10
110                 font.bold: true
111                 color: "white"
112             }
113         }
114     }
```

Now we can come to the delegate of our Grid View, as with the List View we had before this can be seen as a mask for all our data. It consists out of a Mouse Area, that is the same size as the cell size we created earlier. Inside this Mouse Area we have a rectangle, which is transparent color wise, but has a border, which is white but has a width of 0, this will be important later. Inside this rectangle will be a Label, this label has as its text attribute the letter data from our List Model. With this our delegate displays the data.

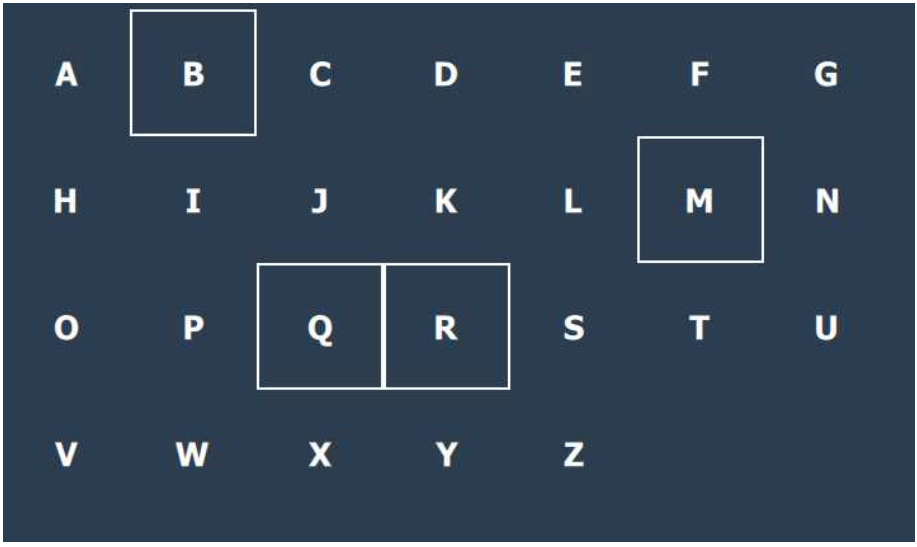
I also added the onclicked event to our Mouse Area, here we will later add the link to the check function later. But for fun and easy of displaying which letter you already clicked I added to

line 96, that when you click a letter, that a white border forms around letter. This means you always know what you clicked.

```
1   import QtQuick 2.9
2   import QtQuick.Controls 2.5
3
4   Item{
5       id: gamePage
6       width: 360
7       height: 640
8
9       Input_Grid{
10          id: inputGrid
11      }
12  }
13
```

Now we can go over to the Game_Page.qml file and add our Input_Grid as a component. We also should give it an id. Know the grid will be displayed inside our application and we can click the letters.

Input Grid View



Input Grid with selected letters

As you can see all is working as intended. You might be wondering what about the empty space in the bottom left. This will be a retry option. I know it makes the game rally easy, but for developing the Game an easy mode is really hand. So, I will implement one.

Also, this button could later be used to start a new round of Hang Man if you are not able to guess the word, or you do not like it. This is a stylistic and function wise unnecessary option I want to implement. But sometimes you need to think a lot out of the box and sometimes you need to make room for these stylistic choices.

Next up we need a way to display the count of letters the word has that the computer randomly provided us. So, lets implement that.

```
14      property var currentWord: ""
15      property var wordCount: 0
16      ListModel{
17          id: listModelCount
18      }
```

First of we need to add another property to our Main_Page.qml file. We should call it wordcount, a better name would be letter count but that will be used later on. The property should be initialised with 0. Below that we need to create a List Model, with an id. This model does not need any elements build in so leave it as it is.

```
 9      ListView{
10          id: listViewLetterCount
11          width: parent.width
12          height: 50
13          orientation: ListView.Horizontal
14          anchors.bottom: parent.bottom
15          anchors.bottomMargin: inputGrid.height+20
16          interactive: false
17          model: listModelCount
18          delegate: MouseArea{
19              width: 50
20              height: parent.width/10
21
22              Rectangle{
23                  anchors.bottom: parent.bottom
24                  anchors.horizontalCenter: parent.horizontalCenter
25                  width: parent.width-10
26                  height: 4
27                  color: "white"
28              }
29          }
30      }
```

As already mentioned, we are going to use a List View as our display of how many letters are needed. The List View in itself is remarkably similar to what we are already used to seeing. The model used is the one we created earlier, the binding throws the id in this case even throw the project structure.

To place the List View above the Input Grid, we just made an anchor.bottom and then a bottom margin, that has as its value

the height of the Input Grid + 20. This means it is right above the inputs but not too far above.

We also made the ListView not interactive, we do not wont the player scrolling around and doing stuff he should not do. And this is the best way to prevent this.

A new attribute we used here is the orientation of the ListView. This basically turns the ListView on its side and displays all the items horizontally. If you want to do something similar it is a good Idea to do it this way.

The delegate we created is really simple. It is a basic Mouse Area without any click functionality and a rectangle at the bottom of this Mouse Area that is a little bit smaller than the width of the Mouse Area. This will be the spaces which tell the player how many letters the word has.

```
64                  onClicked: {
65                      currentWord = wordModel.get(Math.floor(Math.random
66                      wordCount = currentWord.word.length
67                      for(var i = 1; i <= wordCount; i++){
68                          listModelCount.append({"space": "-"})
69                      }
70                      console.log(currentWord.word)
71                      swipeView.setCurrentIndex(1)
72                  }
```

And here is the magic that creates these lines. Basically, we insert into the onclicked event we used some new code. First of we need to get the length of the word we randomly generated. For that we can simply call currentWord, get the word from the object and then get the length from it.

With the length we can then create a for loop that goas from 1 to the length of the generated word and for each adds a new item to the List Model for listModelCount. The data we add is not important, so add whatever you want. We just need the number of items as letters in our word.

And when you start up our application you can see that for the word Apple the correct number of spaces was created. These spaces will always form from left to right or the other way around. They will never generate from the middle. So, everything we do needs to be aligned to this to.

I do not really like this as it does not look as good as I want to have it. But it does the job and the perspective it creates is really good.

Now we need to have a look at the check function that checks if the letter pressed is in the word and if so, places the word at the appropriate index inside a new ListView we need to create later on.

But first of let us start with the check function.

```
94          onClicked: {
95              borderRec.border.width = 1
96              //Check if letter is included in the Word
97              if(currentWord.word.match(letter)){
98                  console.log("Yes Letter:" + letter + " is in the Word")
99              }else{
100                 console.log("NO Letter:" + letter + " is in the Word")
101             }
102         }
```

With this quite simple if statement we can now first of check if the letter is included and we then give a console log that it was included.

If you did everything until now and you were to start up the application you might relies on that for the first letter in the word it works, but for all the other letters it does not. This is because we are not searching for the letter but for the exact matching one. So upper- and lower-case matter. We could now set a

parameter to the match function and disable the problem, but because I want to have all the word in bold and uppercase anyway, we can change them in our word List Model.

```
19      ListModel{
20          id: wordModel
21          ListElement{
22              word: "TREE"
23          }
24          ListElement{
25              word: "APPLE"
26          }
27      }
```

Now you might rely on that we are not done with the functions we need, so let us finish them.

```
32      ListView{
33          id: listViewWord
34          width: parent.width
35          height: 50
36          orientation: ListView.Horizontal
37          anchors.bottom: parent.bottom
38          anchors.bottomMargin: inputGrid.height+40
39          interactive: false
40          model: ListModel{
41              id: wordOutputModel
42          }
43
44          delegate: MouseArea{
45              width: 50
46              height: parent.width/10
47
48              Label{
49                  anchors.centerIn: parent
50                  font.pointSize: 20
51                  font.bold: true
52                  color: "white"
53                  text: letter
54              }
55          }
56      }
```

World List View

First of we need another horizontal ListView just like the one we build for the spaces. The delegate is only different in that it does not have a rectangle inside it, but a Label, that gets the letter from the model.

This is for all intense and purposes a normal List View, if you compared it to the one, we created in our first application. The only difference is the orientation.

```
94          onClicked: {
95              borderRec.border.width = 1
96              //Check if letter is included in the Word
97              if(currentWord.word.match(letter)){
98                  // Found Letter
99                  console.log("Yes Letter:" + letter + " is in the Word")
100                 var index = currentWord.word.indexOf(letter)
101                 console.log(index)
102                 wordOutputModel.insert(index, {"letter": letter})
103             }else{
104                 //Player did not find letter
105                 console.log("NO Letter:" + letter + " is in the Word")
106                 hangManCounter++
107             }
108             winCheck()
109         }
```

The primary function that handles everything is only really changed in that aspect as that we create a var called index and then find the index of the letter that was selected. This is only the case when the letter even matches the current word.

Also, when the word does not match a counter is increased.

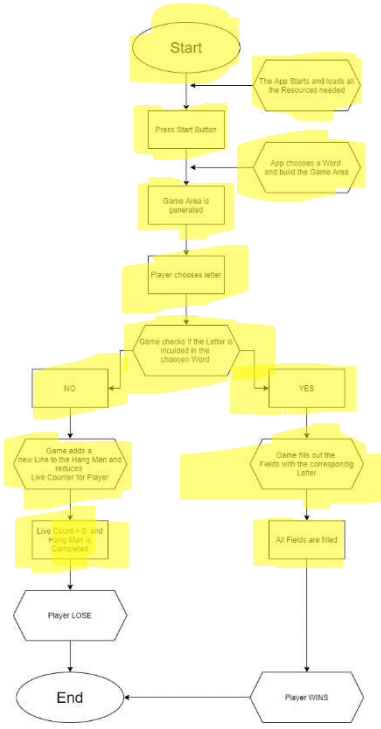
```
14      property var hangManCounter: 0
15      property var currentWord: ""
16      property var wordCount: 0
```

This counter should be added to the Main_Page.qml file above the other properties. This counter is especially important later when we come to the winCheck function.

```
127     function winCheck(){
128         if(wordOutputModel.count == wordCount){
129             //Player has won
130             console.log("Player won")
131         }else if(hangManCounter == 10){
132             //Hang-Man is complete
133             console.log("Player lost")
134         }
135     }
```

winCheck function at this point

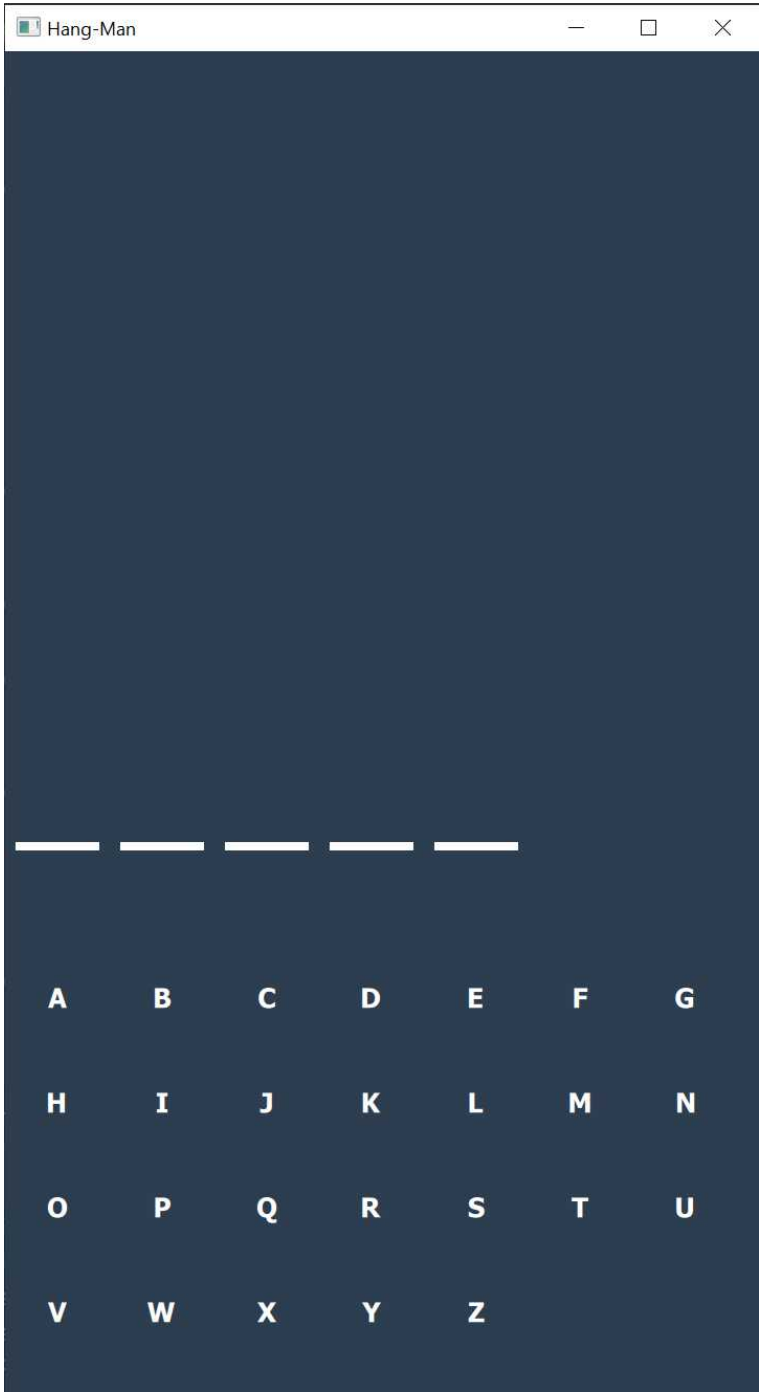
The winCheck function in its simplicity checks first of if the count of added words is equal to the wordcount we generated earlier. This would mean for the program that all letters were found and the player has won. A quite simple check. For the player losing, the check is really not different. Thanks to the counter we implemented earlier we can just check if the counter is equal to 10. If that is the case, then the player has lost.



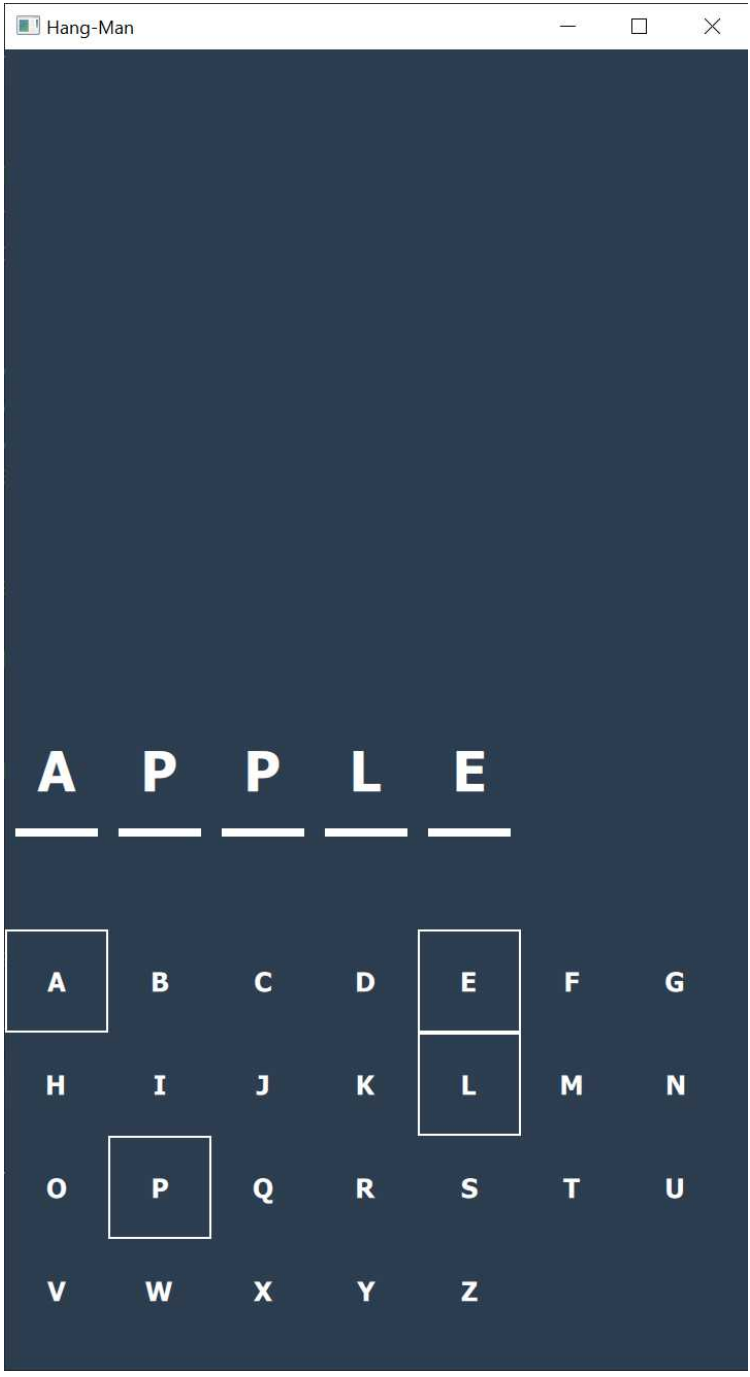
Currently we are at this point in developing our application. As you can see, we already finished quit a lot in our development, but there are still a lot of things we need to finish and finalise.

This will probably take a while, but always refer back to our chapter about the functionality of our application, there you can check how far you have come and what you really achieved.

Now if we were to start the application and we start plying, you can see two things happening.



Running Hang-Man Application



Running Hang-Man Application Word filled out

The game works, but you need to click on the duplicate letters two times so that the appropriate fields are filled. And you must select letters in the way the word is formed. So here the word was Apple. If you did not start with A and worked your way down from there, you will get an error telling you that there is no forth index in the List Model. And this is true.

```
qml: 4
qrc:/Game_Page/Game_Page.qml:40:16: QML ListModel: insert: index 4 out of range
```

Console Output

These two bugs must be fixed so that the application works as it should.

Other than that, you can see that the application works as intended and even better than you might suspect. The win is correctly accepted, and we can go from there.

For clarity sake, I will first finish the game, so that it is playable completely and then we are going over the two bugs. They are not game breaking, and they can be fixed.

First of we are going to build a Hang Man out of rectangles. After that we are going to import the file and build a function that makes the parts visible depending on the counter.

```
103             }else{
104                 //Player did not find letter
105                 console.log("NO Letter:" + letter + " is in the Word")
106                 hangManCounter++
107                 buildHangMan()
108             }
```

Console.log that prints out when word is found

Right below were we count up the counter for the Hang Man, we need to add the name of the function we need to build.

```
138     function buildHangMan(){
139         if(hangManCounter == 1){
140             rec1.visible = true
141         }else if(hangManCounter == 2){
142             rec2.visible = true
143         }else if(hangManCounter == 3){
144             rec3.visible = true
145         }else if(hangManCounter == 4){
146             rec4.visible = true
147         }else if(hangManCounter == 5){
148             rec5.visible = true
149         }else if(hangManCounter == 6){
150             rec6.visible = true
151         }else if(hangManCounter == 7){
152             rec7.visible = true
153         }else if(hangManCounter == 8){
154             rec8.visible = true
155         }else if(hangManCounter == 9){
156             rec9.visible = true
157         }else if(hangManCounter == 10){
158             rec10.visible = true
159         }
160     }
```

if clause for making the Hang-Man visible based on the hangManCounter

The function itself is not that complicated, as you can see it is just a big if statement that filters the current state of the counter out and then tells the corresponding rectangle to turn visible. It is nothing special, and there are a few better ways to do this, one would be to use a switch case, but you cannot really make this any smaller, not unless you want to make it a bit more complicated.

```
MouseArea {
    id: root
    anchors.top: parent.top
    anchors.horizontalCenter: parent.horizontalCenter
    width: 360
    height: 360

    Rectangle {
        id: rec1
        x: 19
        y: 319
        width: 87
        height: 20
        color: recColor
        visible: false
    }

    Rectangle {
        id: rec2
        x: 52
        y: 38
        width: 20
        height: 283
        color: recColor
        visible: false
    }

    Rectangle {
        id: rec3
        x: 52
        y: 38
        width: 189
        height: 20
        color: recColor
        visible: false
```

```
}

Rectangle {
    id: rec4
    x: 221
    y: 39
    width: 20
    height: 54
    color: recColor
    visible: false
}

Rectangle {
    id: rec5
    x: 189
    y: 89
    width: 84
    height: 84
    color: "transparent"
    radius: 99
    border.width: 15
    border.color: recColor
    visible: false
}

Rectangle {
    id: rec6
    x: 221
    y: 168
    width: 20
    height: 91
    color: recColor
    visible: false
}
```

```
Rectangle {
    id: rec7
    x: 200
    y: 231
    width: 20
    height: 91
    color: recColor
    rotation: 210
    visible: false
}

Rectangle {
    id: rec8
    x: 242
    y: 231
    width: 20
    height: 91
    color: recColor
    rotation: 150
    visible: false
}

Rectangle {
    id: rec9
    x: 252
    y: 170
    width: 20
    height: 91
    color: recColor
    rotation: 130
    visible: false
}

Rectangle {
```

```
        id: rec10
        x: 190
        y: 170
        width: 20
        height: 91
        color: recColor
        rotation: 230
        visible: false
    }
}
```

Here I have the Hang Man in its parts as code snippet for you. As I did it here is not recommended. It is not responsive, and it could break very easily. But if you wanted to make this responsive then good luck. This would be a hell of a thing to do and it would take a long time. A far easier solution is by leaving the size of the Mouse Area at a fixed angel, and just making it so that it is anchored to the correct position at the top. This means that if you were to use this app on a larger screen then you would only see the Hang Man in tiny. But that is an unavoidable sacrifice that you need to make from time to time.

```
 9      MouseArea {
10          id: root
11          anchors.top: parent.top
12          anchors.horizontalCenter: parent.horizontalCenter
13          width: 360
14          height: 360
```

Mouse Area fixed Version

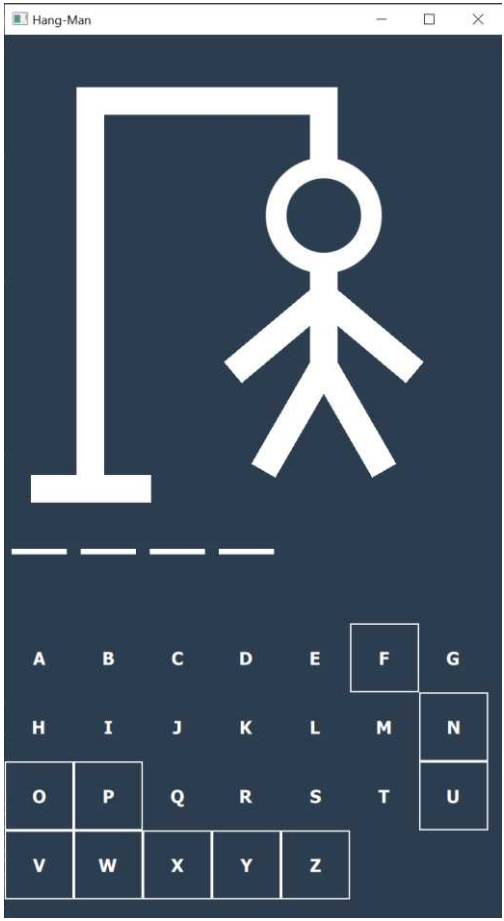
Just by changing the Mouse Area like this you can nearly completely elevate the problem and we now have a working Hang Man.

Now as I already mentioned this is a really bad way of doing the Hang-Man. It is not responsive, it can break very easily when we are not careful. And it is from a software engineering

standpoint very ugly. A far better solution would be to have multiple images and then cycle throw these images using a simple function. It is still not great but a lot prettier as a bunch of Rectangles.

Another solution would be to use multiple SVG Images, and then color them in depending on if they need to be active or not. It would mean that you still need a bunch of these images, but it is also a bit nicer.

But all of these options are very time and resource intensive and for the simple act of learning not necessary. But if you want to you can try to improve this on your own when you are done with the project.



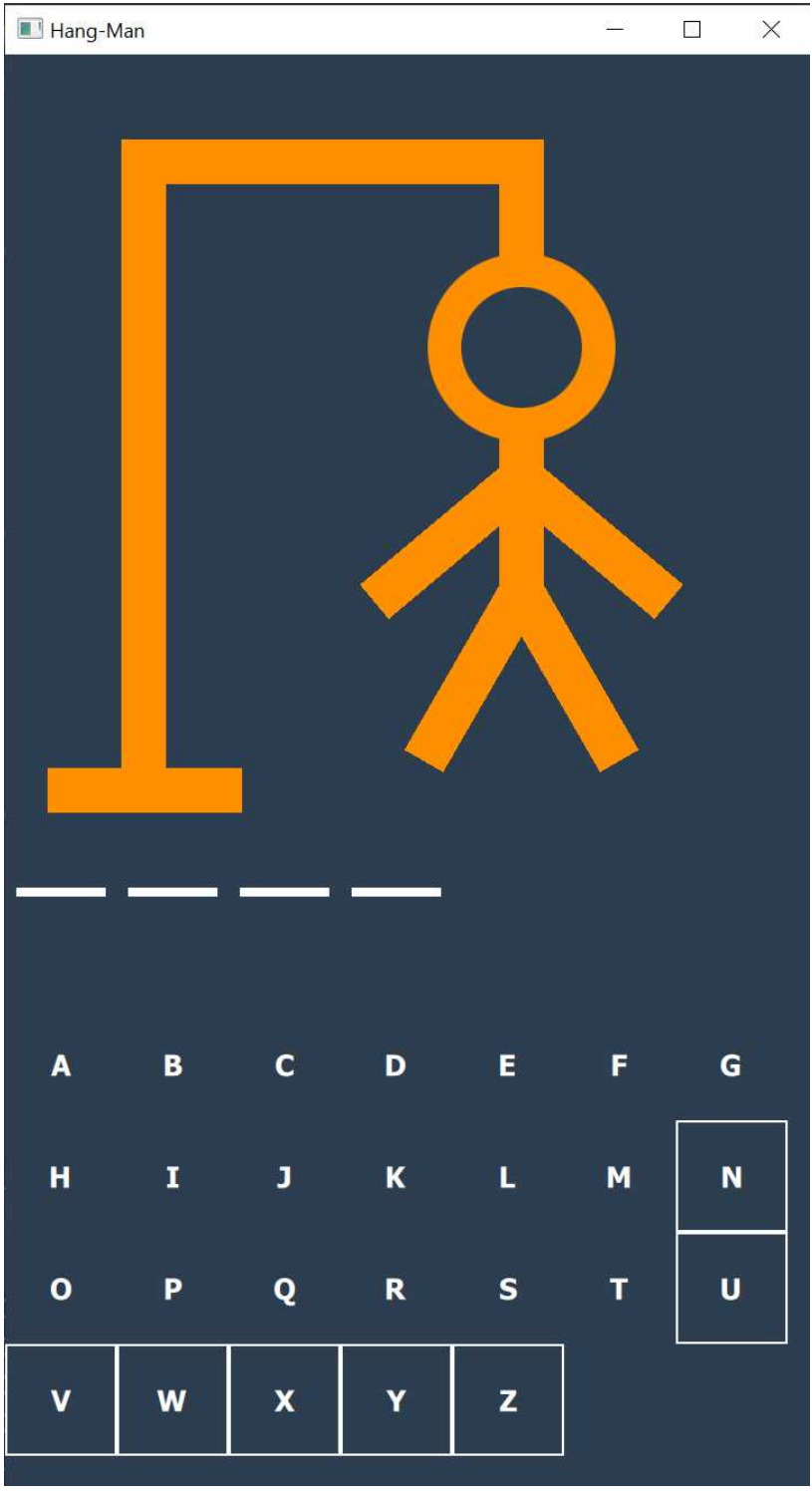
Here you have an example where the Hang Man is completed. Now another thing we can do to improve the overall game is by changing the colors of the rectangles for the Hang Man into something a little bit more colourful. You might now roll your eyes, but the main thing I want to teach you is not repetitive teaching you to change something, but how to make it easier to change something like this later.

Currently all the colors we used to be on every component. Meaning that if you were not to change this color you would need to change this on every rectangle. Right now, this is unavoidable, but for the future we can do something a little bit better.

```
4  Item{
5      id: gamePage
6      width: 360
7      height: 640
8
9      property var recColor: "#fe9000"
10
11     MouseArea {
12         id: root
```

New added property recColor

First of add a property right below our Item, it is not possible to place it inside of our Mouse Area, there it would not work. We can call it recColor as it holds our color. This color can be found in the color pallet for this project, on my website or the Git Repository for this project. Now we can copy and paste the name of the property in place of an actual color. This means that when you want to change the color now you can just change it here and not at every place. This can also be used in larger projects where you have even more places where color is important.


Hang-Man
A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z

Now it looks good in my opinion. Now the only things remaining is the win or loss page.

The way we can do it is by changing the currently index of our Swipe View in our Main_Page to that of the last item in the Swipe View.

```
106         Item{
107             id: endPage
108             width: 360
109             height: 640
110 
111             Label{
112                 anchors.top: parent.top
113                 anchors.topMargin: 50
114                 anchors.horizontalCenter: parent.horizontalCenter
115                 text: "PLAYER"
116                 font.pointSize: 50
117                 font.bold: true
118                 color: "white"
119             }
```

The last page starts simple off with just a Label that says PLAYER. We position the Label at the top of our view and centre it horizontally. We also give it a large size and make it bold. The color of the rectangle should also be white.

```
121             Label{
122                 id: winLossLabel
123                 anchors.top: parent.top
124                 anchors.topMargin: 150
125                 anchors.horizontalCenter: parent.horizontalCenter
126                 text: ""
127                 font.pointSize: 50
128                 font.bold: true
129                 color: "#fe9000"
130             }
131 
132             Label{
133                 anchors.bottom: parent.bottom
134                 anchors.bottomMargin: 150
135                 anchors.horizontalCenter: parent.horizontalCenter
136                 text: "WANT TO"
137                 font.pointSize: 30
138                 font.bold: true
139                 color: "white"
140             }
```

Next up we have another Label. It is similar positioned and styled as the first one, but a bit larger. Also, the color is the orange we used so far. The text attribute is currently empty and is filled with the win and or loss function, we created earlier.

```
128     function winCheck(){
129         if(wordOutputModel.count === wordCount){
130             //Player has won
131             console.log("Player won")
132             winLossLabel.text = "WON"
133         }else if(hangManCounter === 10){
134             //Hang-Man is complete
135             console.log("Player lost")
136             winLossLabel.text = "LOST"
137             winLossLabel.color = "#d23742"
138         }
139         swipeView.setCurrentIndex(2)
140     }
```

As you can see, we were able to use the id, we gave the Label to change how it says and what color it has by calling the id, and then changing the attribute. Depending on if the player won or lost, we want to display that as text, and if the player lost then we want a different color as is normally the case.

The other Label is as the two before amazingly simple, just anchored to the bottom.

```
142             Label{
143                 anchors.bottom: parent.bottom
144                 anchors.bottomMargin: 100
145                 anchors.horizontalCenter: parent.horizontalCenter
146                 text: "TRY"
147                 font.pointSize: 20
148                 font.bold: true
149                 color: "#fe9000"
150             }
```

This one just below is more or less the same, only smaller and with a different color then the first.

```
152             RoundButton{
153                 anchors.bottom: parent.bottom
154                 anchors.bottomMargin: 50
155                 anchors.horizontalCenter: parent.horizontalCenter
156                 width: 200
157                 height: 40
158                 text: "AGAIN"
159                 font.bold: true
160                 font.pointSize: 30
161                 background: Rectangle{
162                     anchors.fill: parent
163                     radius: 99
164                     color: "#fe9000"
165                 }
166                 onClicked: {
167                     swipeView.setCurrentIndex(0)
168                 }
169             }
```

Return Button

And lastly, we have the button, with which you can go back to the first page. A quite simple button quite like the button we created with which you start the game. For the onclicked function, it is a one liner, with which you jump back to the first page.

Now, just jumping back to the first page is not going to work, as when you were to begin a new game, all the data from the game before would still be in the models. So, we need to clear the models of any data. You might think of doing this here were we changed to the first page, but that would not work. The best way to do it, is by doing it immediately in the winCheck function.

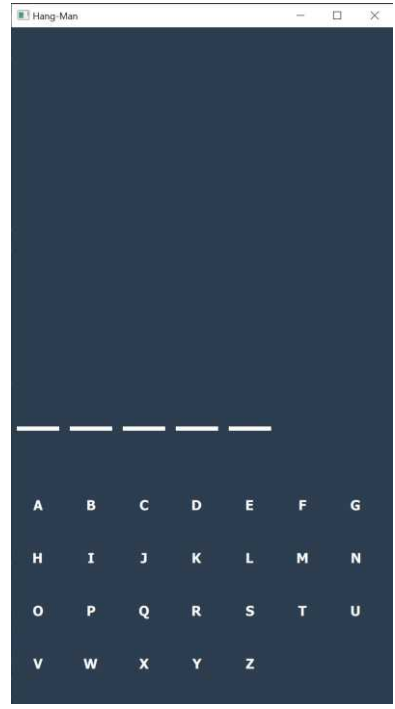
```
128     function winCheck(){
129         if(wordOutputModel.count === wordCount){
130             //Player has won
131             console.log("Player won")
132             winLossLabel.text = "WON"
133         }else if(hangManCounter === 10){
134             //Hang-Man is complete
135             console.log("Player lost")
136             winLossLabel.text = "LOST"
137             winLossLabel.color = "#d23742"
138         }
139         swipeView.setCurrentIndex(2)
140         wordOutputModel.clear()
141         listModelCount.clear()
142     }
```

WinCheck Function

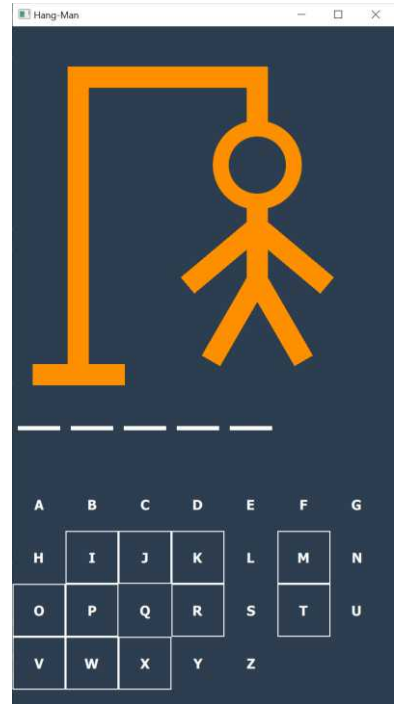
With the two models cleared, the game can begin anew.

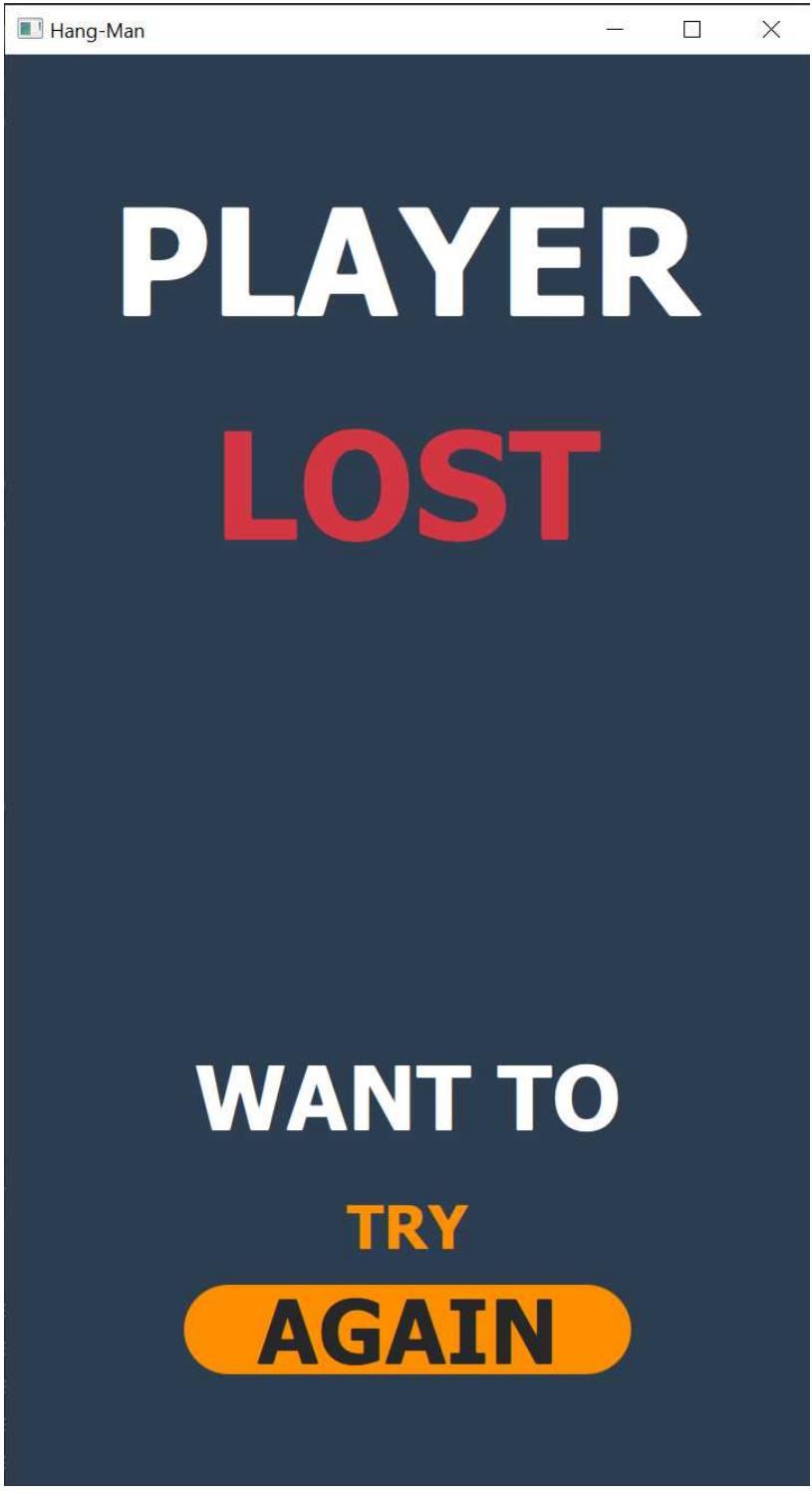
Now with the app build let us have a look at what we build and how everything looks like. This is just a recap of what we did in this project. I will not be able to provide all the images of every component, as this would be far over 100 pages just for that. If you want the code then go to my website bencoepp.io or my Git Hub BenCoepp, there you can find nearly everything you are looking for when it comes to the code for this project.


Hang-Man
Hang-Man
START
Made by BEN COEPP
User Agreement other Legal Stuff


Hang-Man
A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z


Hang-Man
PLAYER
WON
WANT TO
TRY
AGAIN


Hang-Man
A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z


Hang-Man
PLAYER
LOST
WANT TO
TRY
AGAIN

As you can see the application is working as intended and does what we want. The game loop also works. And with this we are done with the application, in this project you learned something about

- **Horizontal List Views**
- **Model structuring**
- **Larger projects**
- **Creating custom components**
- **Interconnecting components**
- **Functioning game loops**

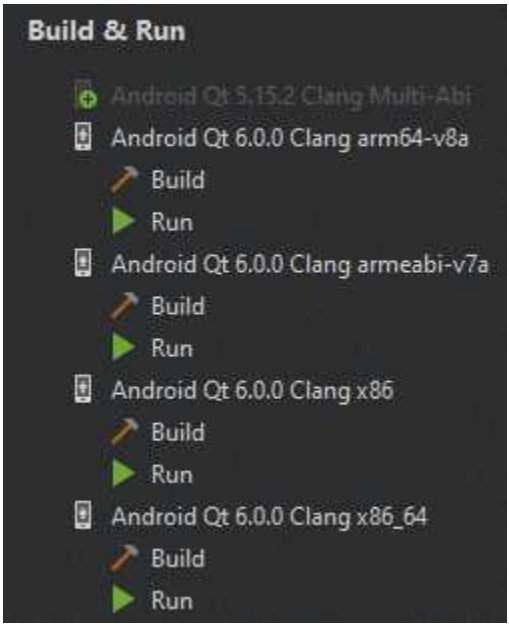
These might not seem that important, and you are correct when you say that this was not a large or complicated project. But that was not the intention when I created this project. It was for the simple learning experience you can gain from creating a small game or application on your own.

And on a side node, repetition is key in my opinion for learning something like a new framework or programming language. You cannot learn something effective when you do not repeat it over a dozen times. So maybe you should reread the book from time to time when you want to practise a little.

– Deploying the App –

Now to the actual deploying of our application. This is a particularly important part of most developments, and projects for that matter. But unfortunately, it is not covered very often.

Because we want to deploy our application for Android you need to have followed the steps in the early chapters, there we set up the Android SDKs and NDKs we need for this.



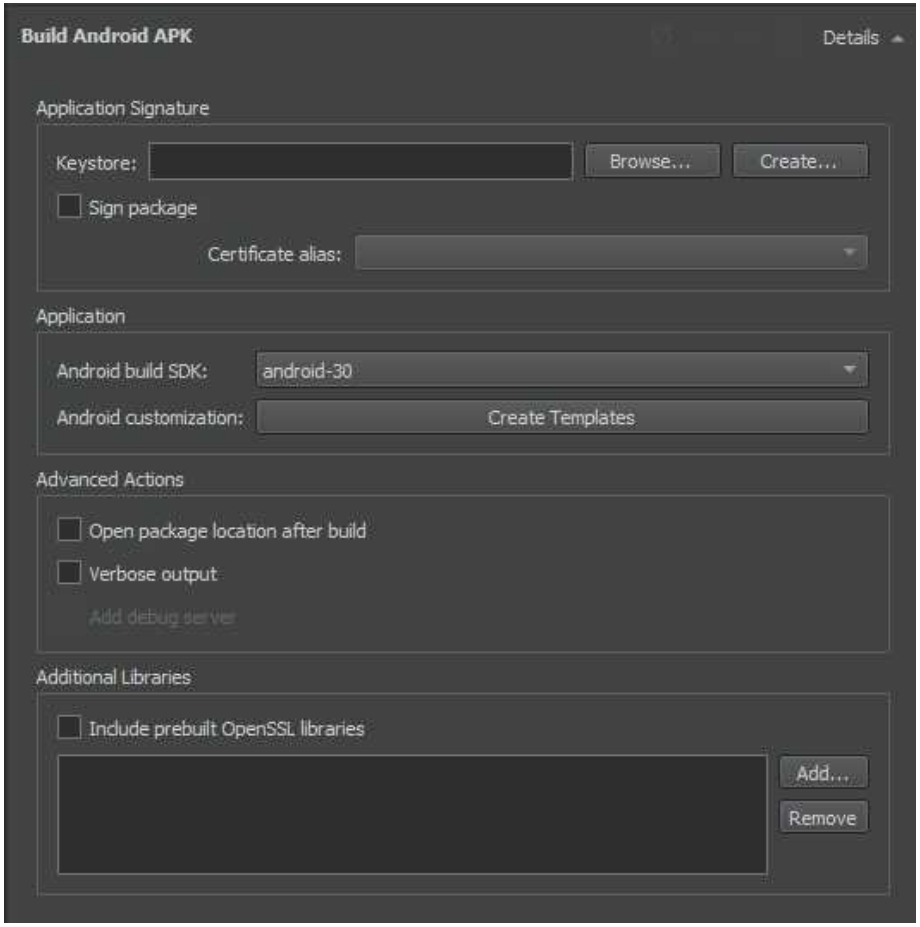
Now to the actual deployment of our application to an Android device. First of you need to go to the Projects Tap at the left side, when you click on it, it will reveal a list of all the available packages you currently have installed, and those that can be used on the project will have a little green plus beside the Icon.

The first thing is adding all the Android Qt Kits, there are four different kind of kits, in some existing Versions they were bundled up into one kit, but here you need to add all four.

The next thing is clicking in on one of these kits, this will open up the Build Settings for the specific kit. Here you can manipulate and edit a lot of the underlying setting of how Qt build Android packages. If you want to learn more about all the

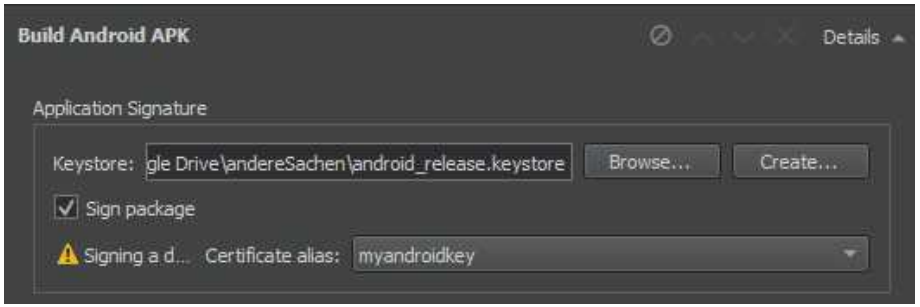
settings that exist here, and which should you manipulate, you can read about that in the Qt Docs.

On a side node, I usually enable all the kits I tend to use right from the get-go. I know that we did not do this in this project, mainly because I did not want to confuse you the reader. But if you are going to create your own project from now on, you can also active and configure all the kits you are going to use.



Build Android APK Drop-Down

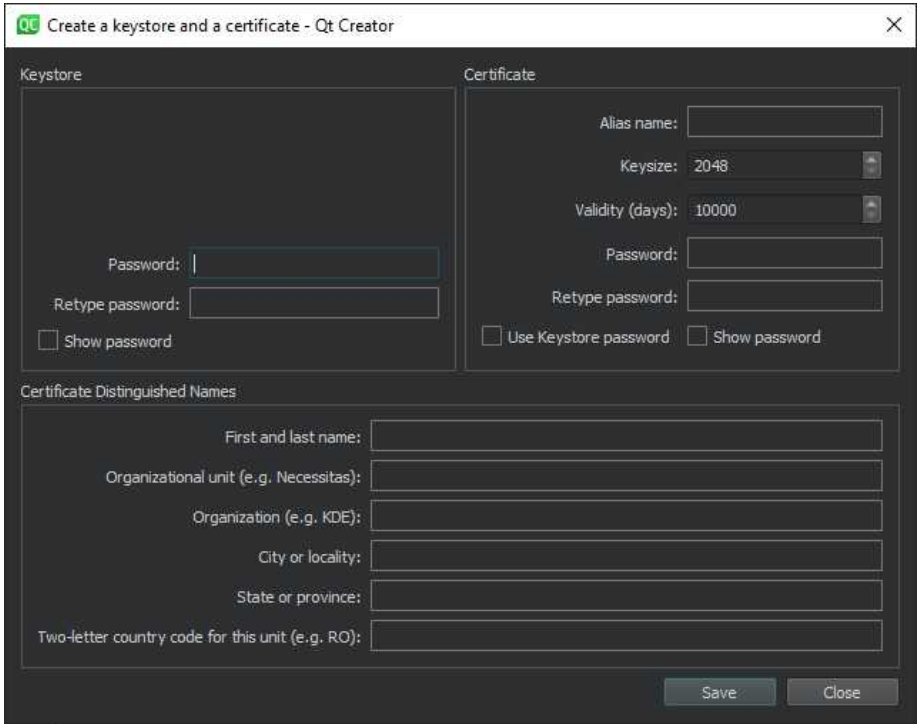
The first thing we need to do here, is open the Build Android APK, when you opened that up you see the content depicted in the screenshot above.



Signed signature added to our Build Android APK settings

The first thing we need to do here is sign our application. If you have never deployed or published an application, you might scratch your head and wonder why we need this and what it is used for. Well, in professional development, especially when you want to publish an application on the App Store, Play Store. For that we need to have a keystore. This authenticates the application to you, so nobody can steal your application and publish it, because for the he would need this keystore.

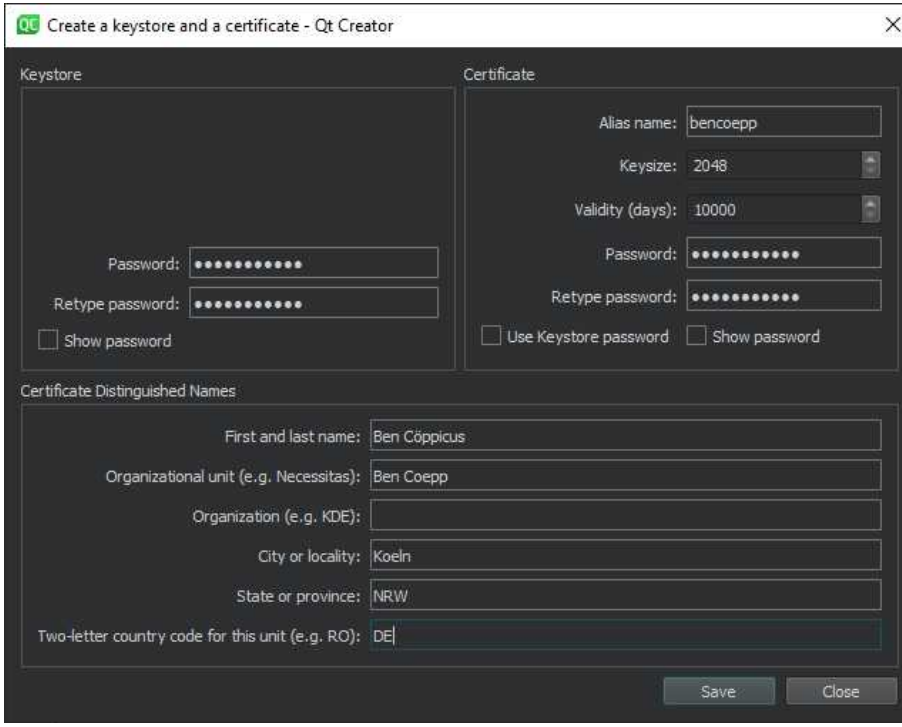
So, if you have one, then click browse and find your keystore, but if you do not have one, then click on create and let us create one.



Create keystore Wizard

This will open up a popup that will ask us for a lot of information. Everything should be self-explanatory, just a few things to remember is that you will need to remember the password. You cannot change the password if you forgot it, so remember it otherwise you will not be able to publish your applications anymore.

Also, the information about your distinguished names is important for Google or Apple, do not lie here or you can get in a lot of trouble.



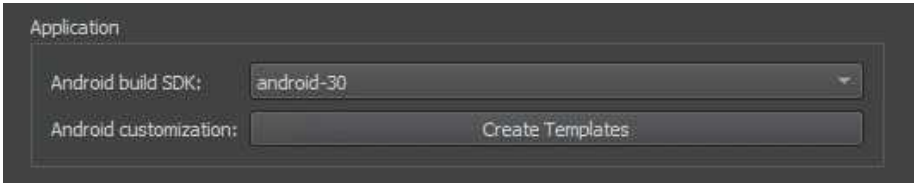
Filled out Create keystore Wizard

When you filled everything out, check the inputs, as there is no editing or reducing this, and then you click save. This will then ask you for a location where the keystore should be saved. Here you should head my words carefully. It is particularly important that this keystore is never deleted. Save it multiple times, on different devices and in the cloud if you can. If you lose the keystore you are not able to publish this application anymore.

Enter keystore Password Popup

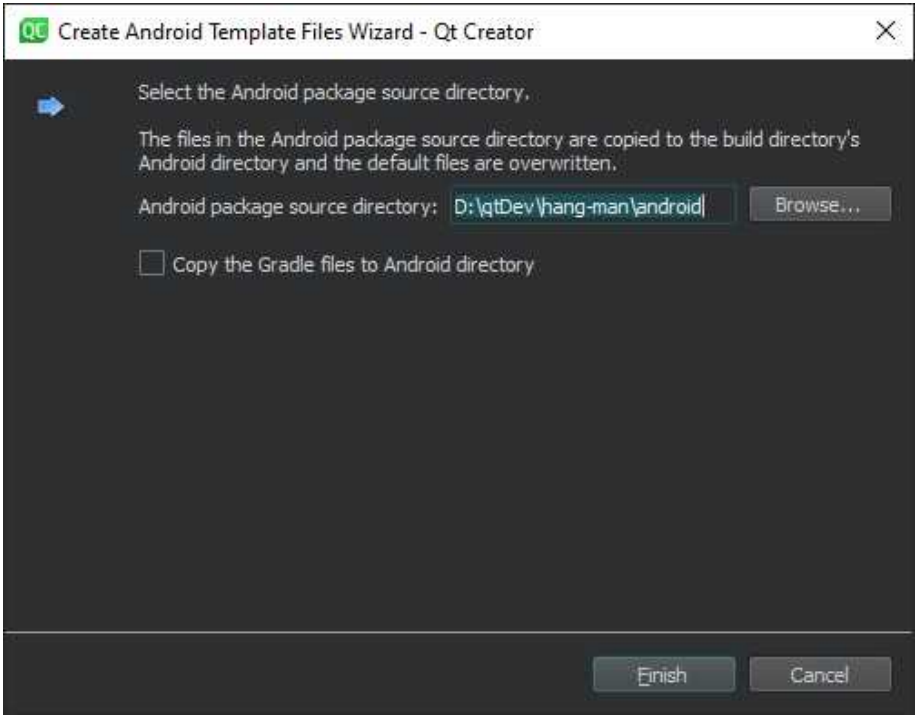
If then clicked Save, it will create the keystore and save it in the destination you selected. Next it will close the first popup and open a new one right away. Here it will ask for the password of the keystore you just created.

If you still remembered what you had as your password type it in and click OK.

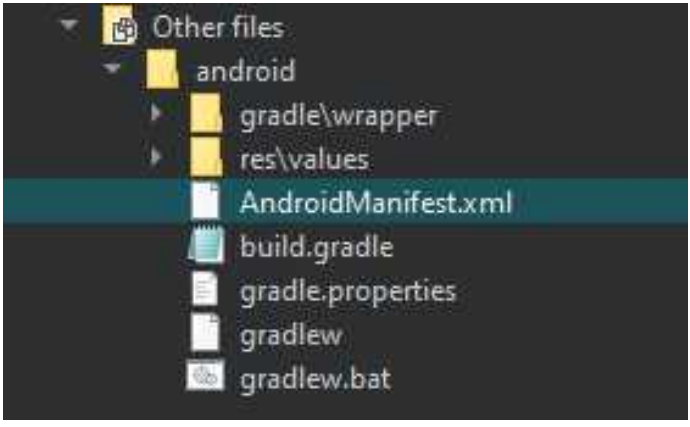


Create Template button

The next thing we need to do is creating a new Android Template. If you never created an Android application this will seem pointless and a little bit confusing, but it is necessary, so create it.



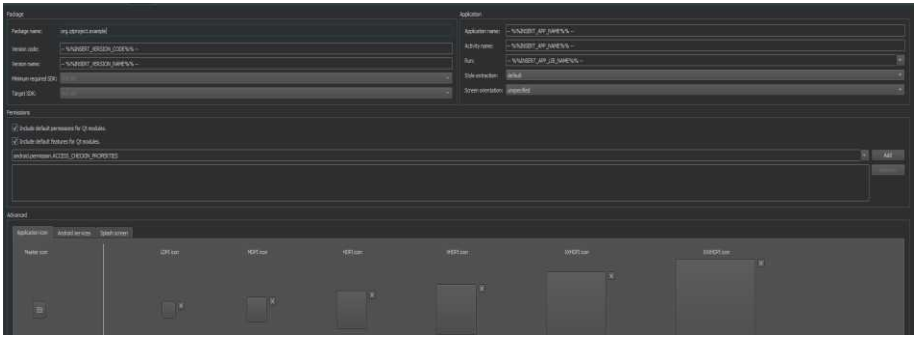
First of it will open a wizard, where it asks you where the Android package source should be. You can leave everything as it is, there is no real point in changing the source of the Android files, as this just creates a lot of problems down the line. Also do not select the checkbox, this is something that does not work for our project here so leave it as it is.



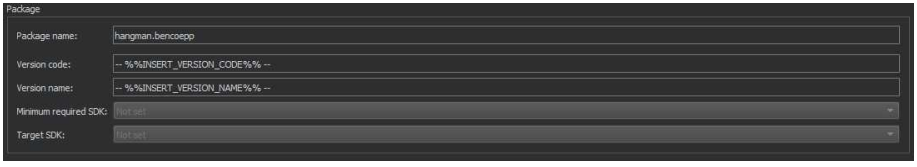
Android Template Files in the Project Structure

When you clicked finish, a few new file scans be found in our project directory. These are the files we created just now by creating a new template.

Some of these files are less important than others, but the most important file is the AndroidManifest.xml. if you are wondering, no it does not have anything to do with politics, rather it is a file that holds all the Information about our application.



As you can see here there is a lot of things, we can fill out here. Typically, you will tend to do this only ones when you create the file the first time, and then only occasionally change a few things in here when you really need to, but other than that you will leave this file as it is.



The first patch of important Information is for the package. Here you can find the package name, the version code and name as well as the minimum and target SDK. For you this is properly empty or filled with some random stuff, so let us fill it with something a little bit more fitting.

The package name should in my opinion always be comprised out of two things. The name of the application and the name of

yourself your company. So, in my case it was hangman.bencoepp. hangman because this is the applications name and bencoepp because that is the aliases I usually use.

Next is the version code and name, for now I just leave it as it is, mainly because I currently do not need it, and you will only change this if you tend to release this software.

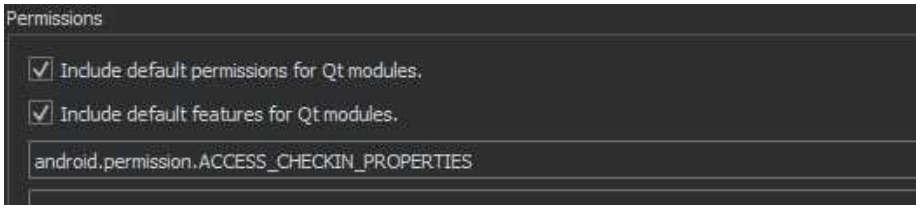
The minimum SDK and target SDK are greyed out, and we are not able to edit them. And in most cases, you probably can leave this as what Qt has as its default. The only reason you probably want to change this is because you have an application that is run by people that do not have the newest device and you require an old SDK version. But other than that, leave it as it is.

Right next to the package information is the information about the application. These settings are even easier to understand the first.

First of the Application Name, Activity Name and Run are for me always configured with the Application Name, for us this is Hang-Man. These three inputs serve the same purpose of being the display name, as well as the name Android shows when you run the app.

Next, we have the style extraction property, this is a new setting, that you can use when you want to differentiate the different rendering and styling options Android has. For instance, if you want to use Androids Native styling that you can find on your Android Device you would select Default. Because we made all or styling on our own, we can choose none here.

Lastly, we have the screen orientation, this setting tells Android how the application is orientated. You have the typical assortment of Portrait, Horizontal, Landscape and a few more. For us Portrait is the most sensible one, so we need to select that.



Android Manifest Permissions

Just below the package and application Information, we have the permissions. Android is difficult about what applications are allowed to do. Because Qt requires a few default permissions, so leave the checkboxes as well as the line below them as they are, you cannot really do anything about the permissions needed, so basically you can nearly always have this as Qt has it as default.

Now we can come to the advanced options. These are interesting options that really help you with a lot of things, first of you can set an application icon for your application. So, if someone downloads your application you have the application icon as your icon of your application. This icon also shows when you run the application.

Next, we have the Splash Screen. This is also a new feature that allows us to set a Splash Screen for the application. So, while the application loads, you have this Image on display.

Before that you had the option of building an extremely complicated loading setup that handles this, or you just did not have a Splash Screen. Because I did not create a Splash Screen, I just set the application icon as the image.

Now that we edited all the settings, we needed to us can first of deploy our application. Before we can do this, we need to do a few things.

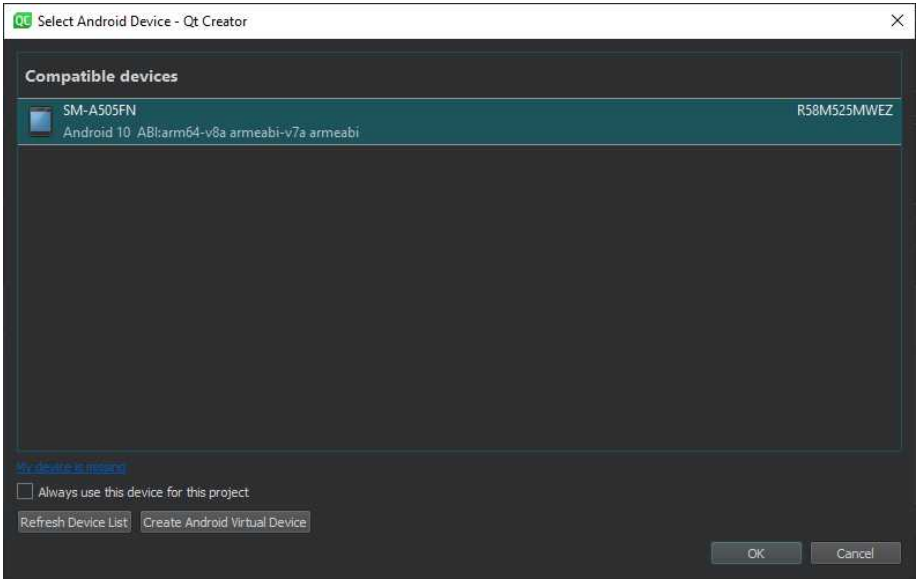
First of you need to plug your Android device into your PC with a USB Cable. If you done that, you can click on the green arrow just as we did before when we wanted to lunch our application.

If you have problems doing the next few thing, you might want to read ahead on **Chapter 2.4.3.11 Deploying Application to Android**. There you can find a lot of information about how to get your application ready for Android, as well as an example how to get the application running on mobile. Also, there are some workarounds explained when you run into some specific problems.

Some thinks that I can share with you right now are first of, that if you click the green button and you are not able to see your device you might need to check a checkbox on your device to confirm that you allow your pc to have access to your phone. If you do not get any dialogue box or conformation dialogue you might need to go to your device settings and enable developer mode. This can be done throw a variety of way, so google how it is supposed to work for your device. After you have done that rerun the application.

If your device does not popup in the compatible device list, you might need to choose a different kit to run your application. This

is because there are a few different types of android devices out there. Choose the one you need and then run the application under that kit. What kit you need can be found right under your device name. On the next screenshot you can see how everything should look like when it works.



Select Android Device for deployment

This will open a new popup that allows us to select the device we want to use. At this point you should see a few permission request on your mobile device. You need to allow all of them. When you did this, you can click ok and the application will start building.

If your mobile device is not showing up here, you need to select a different kit under the Projects Tap. Usually, you can see the kit you require in the Device list. But if this is not the case then just try all out.

BUILD SUCCESSFUL in 17s

26 actionable tasks: 26 executed

Android package built successfully in 22.233 ms.

-- File: D:/qtDev/build-Hang-Man-Android_Qt_6_0_0_Clang_armeabi_v7a-Debug/android-build//build/outputs/apk/debug/android-build-debug.apk

When you set up everything correctly, you can see this in the compile console. With this the application will now deploy on your device and start up immediately.

If that is not the case, then you should have a look at the chapters where we installed Android Studio and installed the NDK and SDK.

Android package built successfully in 22.233 ms.

-- File: D:/qtDev/build-Hang-Man-Android_Qt_6_0_0_Clang_armeabi_v7a-Debug/android-build//build/outputs/apk/debug/android-build-debug.apk

Also, if you want to publish your application on one of the many Play Store / App Stores. You need the APK or APP file for that, this can be found in the build directory of our application. Also, the link to this file can be found in the console, so just grab it from there if you want to.

And with this we are done with deploying our application to a mobile device. We have our app deployed, it works, and we have created an APK file we could now publish to all the different platforms out there, if they match the kit and are Android.

But believe me when I tell you that there is a lot more that you could and maybe need to do when you want to publish for a mobile device. The topic is large and complicated, and some people only work with this. So be prepared to tinker a lot with this when it comes down to deploying your application on your target device.

Also, for the Google Play Store you need a few specific setting to publish for that. First of you need an APP file, this is basically a simple file that combines all the four different APK files you could create into one file that you can then publish to the Play

Store. Also, the permissions you use may require special explanations on the Play Store as Android is not too keen on giving out permissions to any unapproved application.

– What did we learn –

As always, we now should have a look at what we learned in this project. First of we did a few things that are identical to the first project, like the Load and Main Page setup, I am not listening these again as they were just to repeat them again to learn it better.

But what did we learn new? We will let us list what we did.

- **Grid View**

  We already used List Views a few times, and even in this application we used List Views two times to achieve a specific visual component. The Grid View is not so different compared to the List View, but it has a totally different use case, and that is why we used it for the input of the different letters. There are also different ways of displaying and using the letters, but I prefer the Grid View for this.

- **Custom Components**

  We learned about setting up custom components, how to create them and how to best use them. This is one of the things you will tend to do a lot when you create different applications on your own. As this is such a thing you will do all the time it is not too hard to setup or to use, but it is a nice way of training yourself to use them when it is the best time.

- **Deploying to a Mobile Device**

  This is the biggest part in my opinion, learning of how to build an application is one thing, but understanding how to deploy the application to the desired platform is a totally different beast.

  It is not as hard to do, but there are not many tutorials about this topic. This is first of because the deploying of an application is always the last step in any development, and therefor is the least interesting and least covered topic. I really hope you learned how to do this, and if so, you should be able to do this on your own from now on.

As you can see, we learned a few things in this project. Some of these are extremely important for development in general, others only if you are interested in the specific topic. But none the less you need to repeat and use the now learned knowledge to keep it up to date. So, I hope you learned something new and maybe repeat it to solidify the knowledge learned.

2.4.3 Rock-Paper-Scissors Game

We already created two different applications, the Task-Master and the Hang-Man Game. Both taught us how to use different components and elements Qt offers and that you need to use all the time.

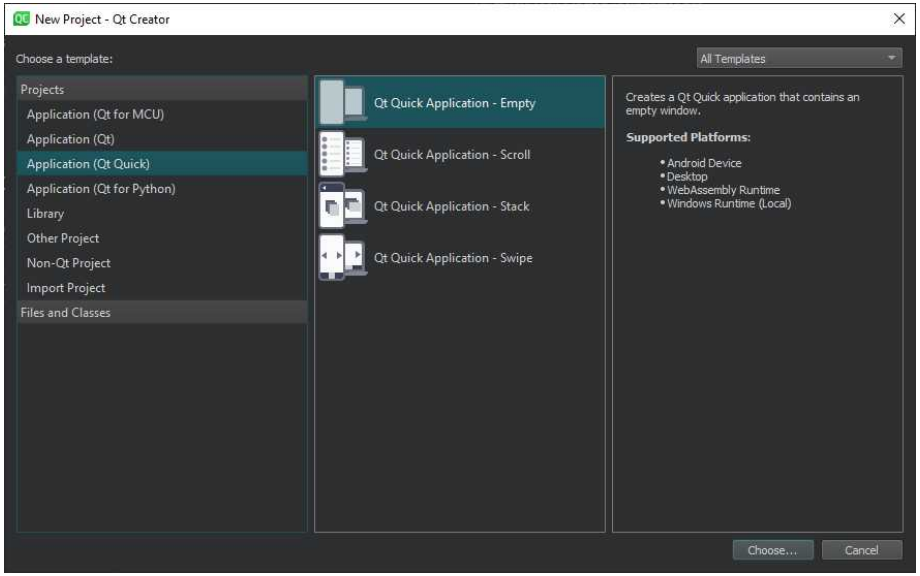
In this Chapter we are going to create a Rock-Paper-Scissors Game, this will not more difficult than the previous, but we are going to focus again on a game loop and on the visual components we are going to create. Again, the main point with this is just teaching you a little bit more about Qt and how to use it.

- Project Creation -

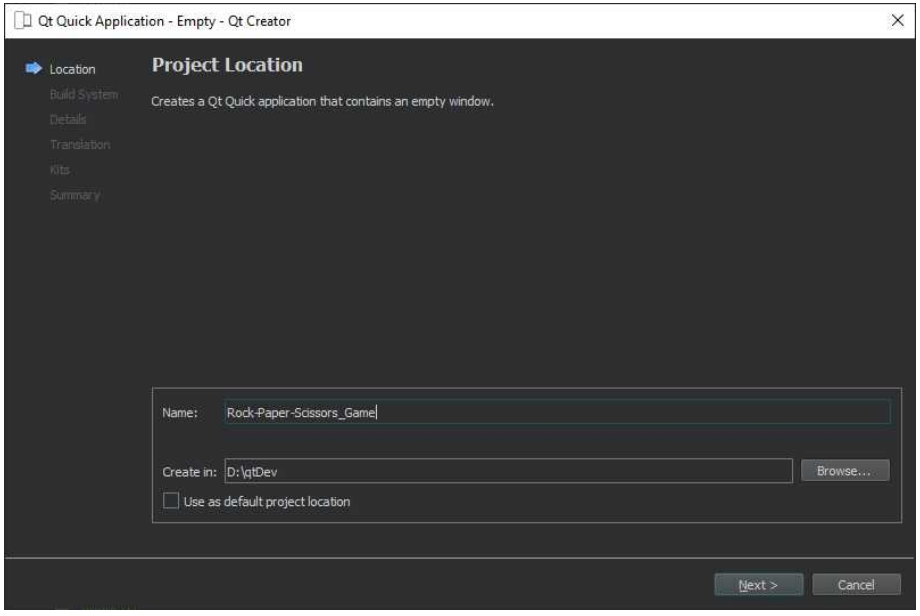
As always, we start by creating our project. This time I am also going to cover the use of Git in this Project. All pats that revolve around the use of Git in our Project will be marked, and you do not need to follow them to understand the project, but I want to keep them in this time, as I believe it can help a few people work with Git and Qt in conjunction.

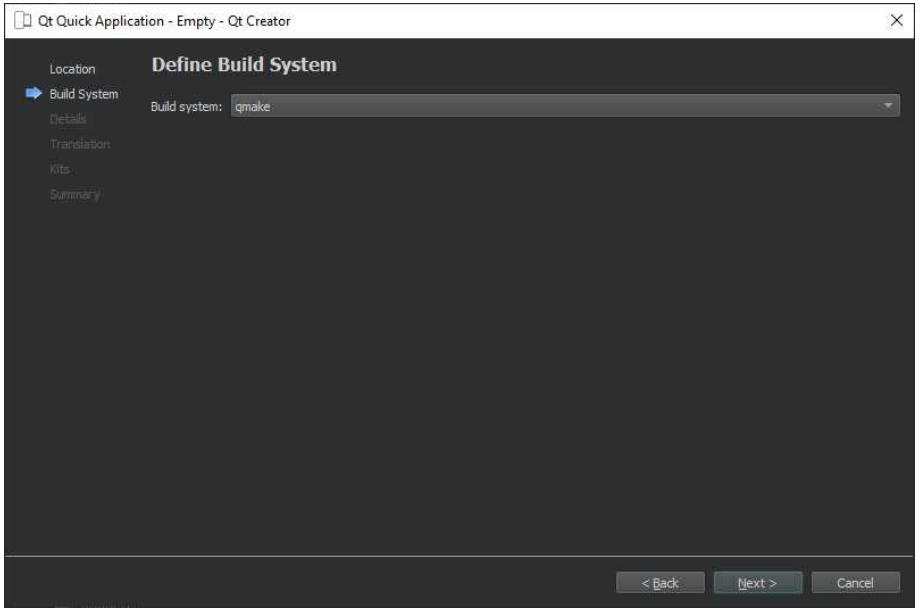
As always, we start with creating a new project, so open up Qt Creator and let us start.

We are again choosing Qt Quick Application - Empty as our development template. First of because this is the template, we are now most familiar with and because we do not need anything new or different in the template to get started. When you clicked on the correct template you can click Choose and continue.

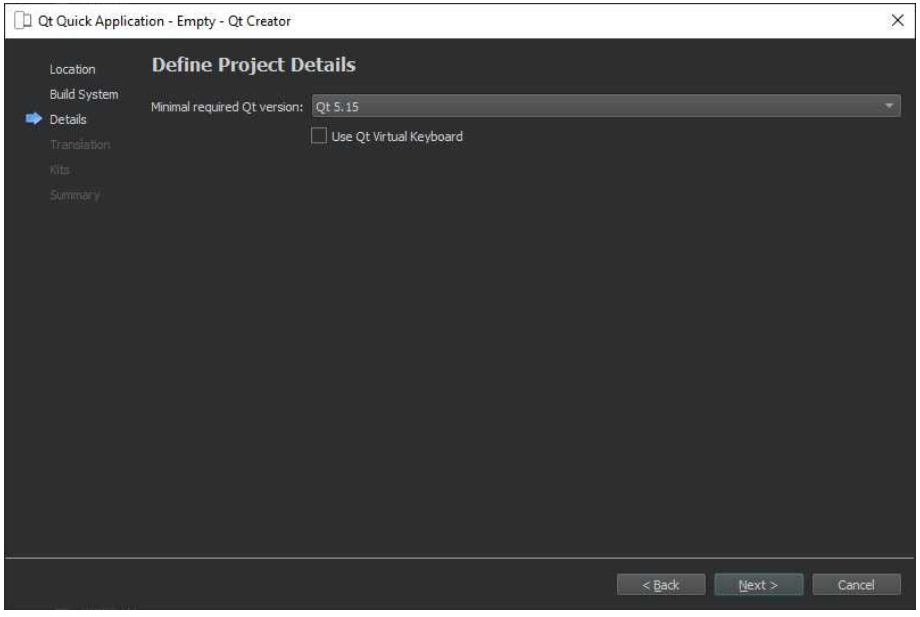


For the name of the application, I have gone with Rock-Paper-Scissors_Game, it fits what we are doing and describes what the application is going to be. Also, for the location the same limitations are valid as always.

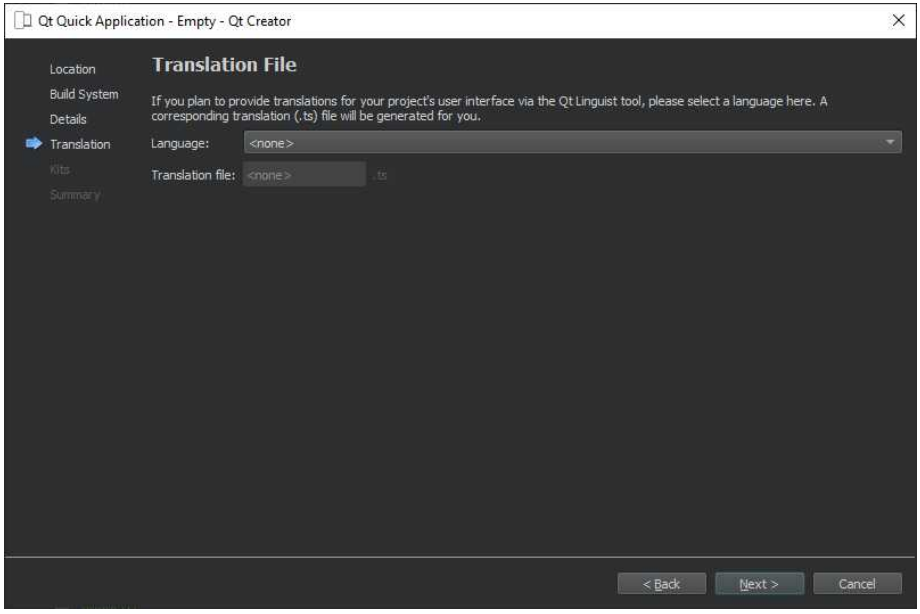




Build system as well as the minimal Qt Version we need stay the same as before. We do not need anything new here, but if you are a little rusty with what they mean, check out the Chapter with the first few steps with Qt.

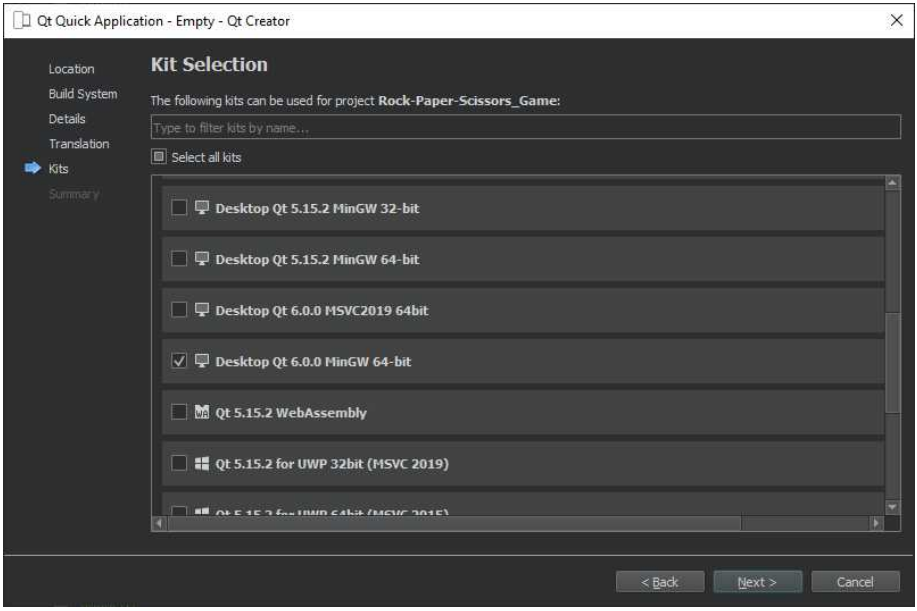


A translation file is again not required but recommended if you were to build an application to a finished point.

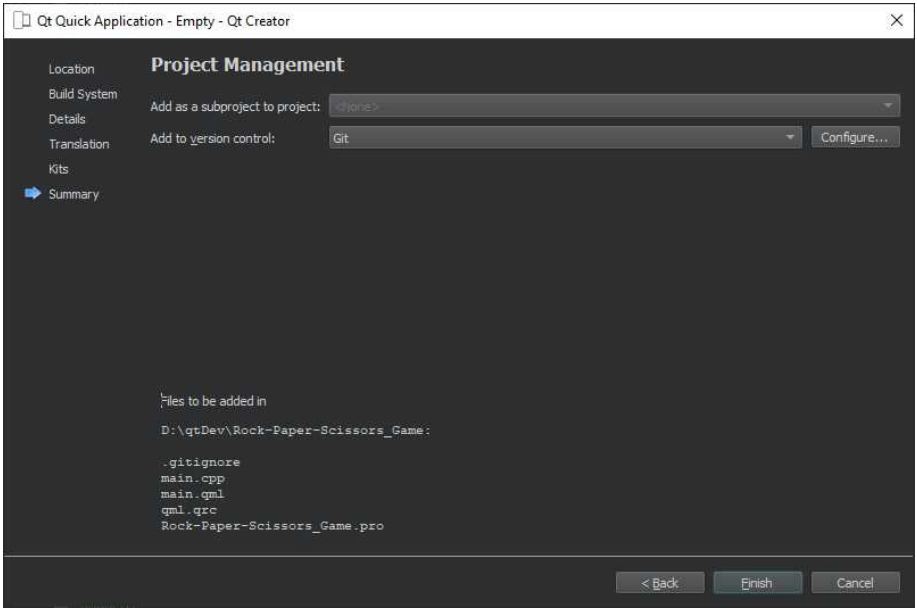


For simplicity, I only choose MinGW 64-bit as my Kit, this is only for the reason that that is the kit I am using to develop the application on.

And as I already mentioned this is the main kit, we used throughout our development in both the previous projects so far. If you did not choose this because you require another kit then choose that instead. The kit is really not that important for us.



But for the final deployment, we are going to use the Android kit we already used in our previous application.



Finally, the most important page in the project creation wizard, and the only page that we have any notable changes on. Here you can see that we changed the version control to Git. Normally we left this empty, but because I want to show you a little how Qt works with Git, we are turning it on now. If you do not want to use Git here, you can also leave it out and skip the parts later on when we actually use it.

```
1  import QtQuick 2.15
2  import QtQuick.Controls 2.12
3
4  ApplicationWindow {
5      width: 360
6      height: 640
7      visible: true
8      title: "Rock-Paper-Scissors"
9  }
```

Simple ApplicationWindow with correct imports

As always, we need to change the imports Qt uses, Qt Quick needs to be changed to 2.15 and we can change the Qt Quick.Window to QtQuick.Controls 2.12. These are the basic things we always need for a normal functional application. We also can immediately change the Window component to an Application Window component. Inside here we can change the title to something a little bit more fitting like ***Rock-Paper-Scissors Game***. This should represent our game a little bit better.

We are going to create a Mobile Application, so we can also change the width and height of the application to fit a mobile aspect ratio a little bit better.

```
5       width: 360
6       height: 640
```

As we are creating a Mobile Application, we can ignore creating a Load- and Main-Page setup as we can use the splash screen android provides us with. Now that we are done with creating the project, we can have a look at how the application should function.

- Functionality -

As we want to create a Game revolving around playing Rock-Paper-Scissors, meaning we have a player, as well as a bot playing against each other. Each round the player gets to choose between rock, paper, and scissors. Then the bot chooses randomly one of the three and they are compared. Against the normal Rock-Paper-Scissor game rules, paper beats rock, rock beats scissors, and scissors beats paper. These are the most basic rules you can find for this kind of game, I know that there are a lot of different rules and other options then rock, paper, and scissor, but I will not use them here as they just make the game more complicated without adding that much complexity.

So, what else do we need:

- **Basic Rock-Paper-Scissors Game Rules**
- **Game-Loop from start to finish**
- **Animations that show how the game is progressing**
- **Win-Loss Point System**
- **Win-Loss Points are saved locally**

But let us have a look at the diagram of our application, this might clear up how this should all work. Also refer back to here whenever you are not sure how we want to do thinks, or you do not understand why I do certain things.

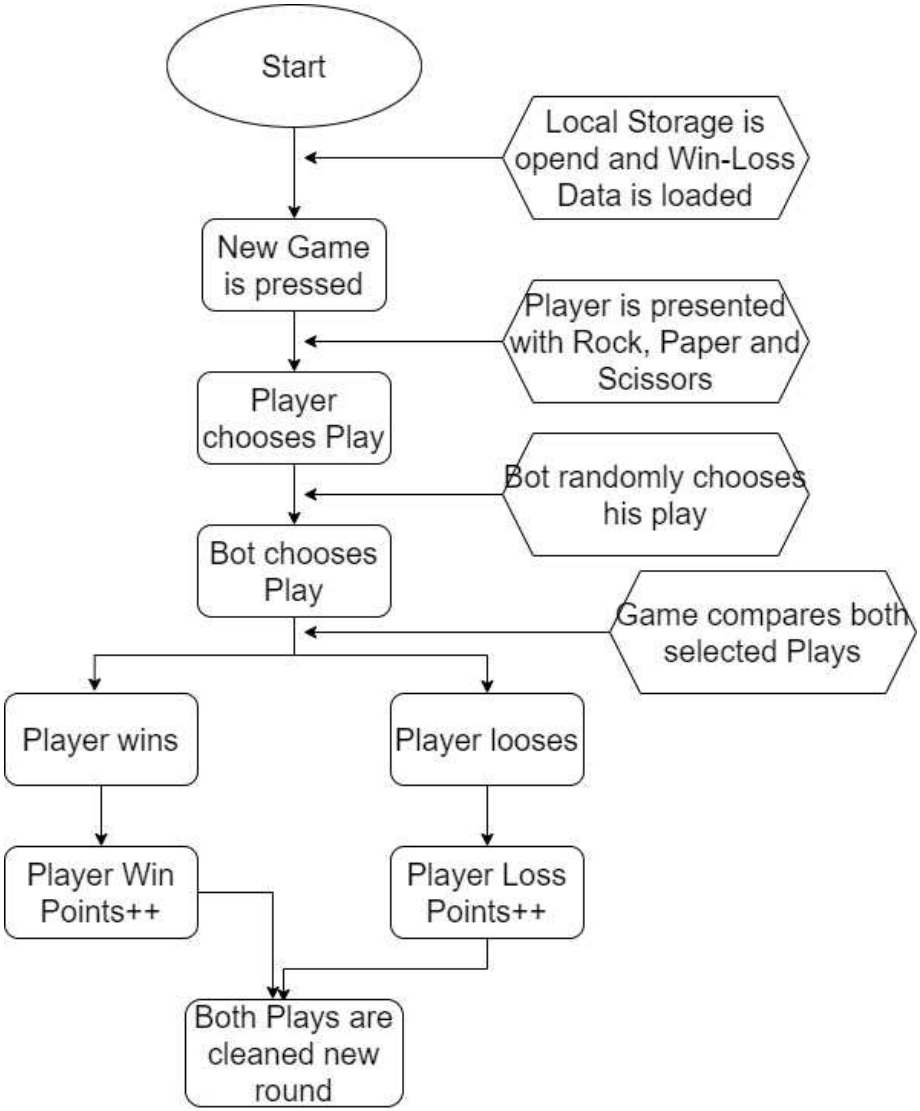


Diagram for the Rock-Paper-Scissors Game

As you can see the structure of the game is not difficult and compared to what we had previously it could also be called

simple, but as I already said this is just training and practising what we have learned and me showing you different ways to create applications.

– Creating the basic Game –

We can start out creating our basic game by creating a Swipe View and putting three Items inside of it. The Swipe View should have an id, a width and a height that can be derived from *anchors.fill: parent*. And a currentIndex, so that while developing we can switch to 0,1, or 2 to be directly transported to the corresponding Item inside of our Swipe View.

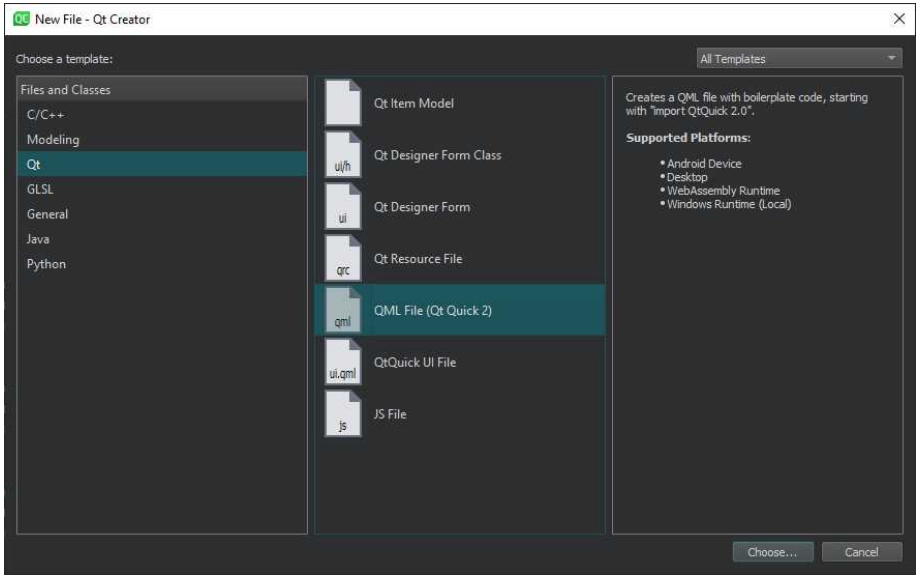
```
10      SwipeView{
11          id: swipeView
12          anchors.fill: parent
13          currentIndex: 0
14          interactive: false
15
16          Item{
17              id: home_page
18          }
19          Item{
20              id: game_page
21          }
22          Item{
23              id: end_page
24          }
25      }
```

Simple Swipe View

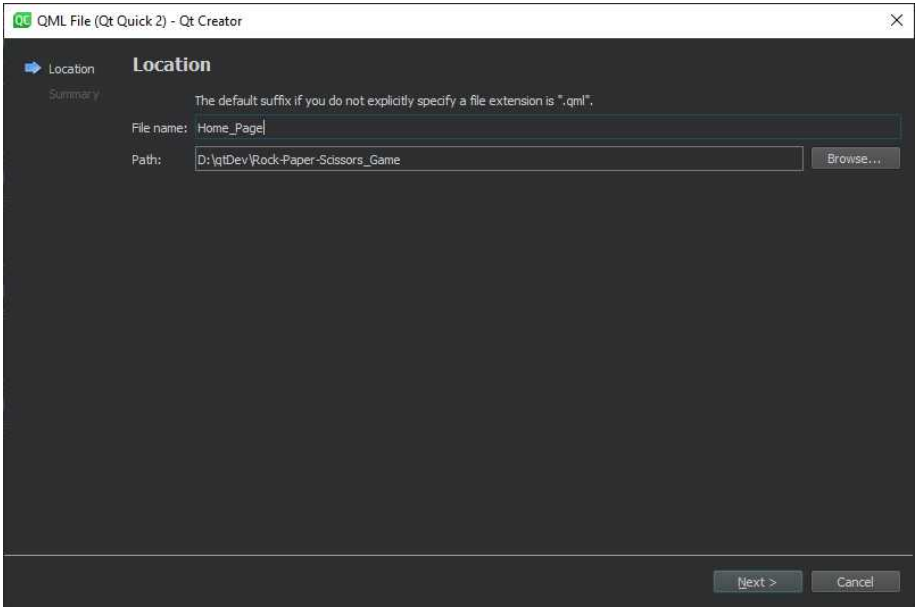
The Items we should have inside of our Swipe View are our Home_Page, that is going to be just a page with a button in the middle throw which you can start the game. Nothing special but a must have, so that your players know what is going on.

As it is right now, we would have all the Items and pages in one file. This is not really that great, first of because readability and secondly because it is extremely hard to keep track of what is going on if you have everything in one file so let us separate our Items.

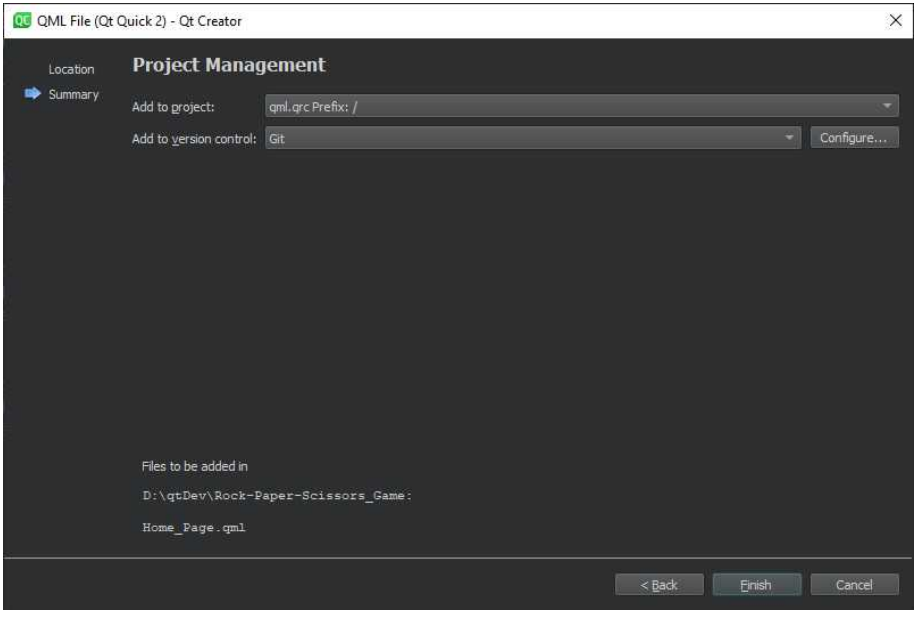
This can be done as we have done a few times before. We are going to create three files, each for a corresponding Page. As before you need to right click on our qml.qrc folder and then select Add New. And this is going to open up our New File Wizard.



As before we are going to select QML File as our new file, as that is what we need. When you have done that you can click Choose…, this will open up the next page in the wizard. Here you can give our files a name. In our case we can use the id of our Items we have in our main.qml.



The path for our file can be the project location. If you have everything as mentioned you can hit Next and we can continue.

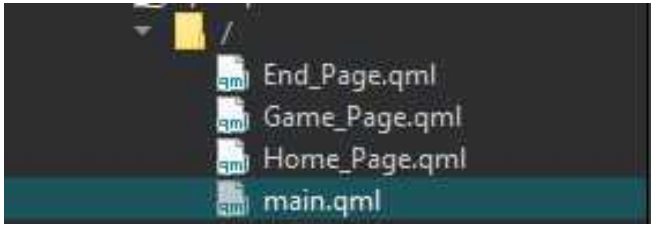


Lastly, we come to the page where Qt asks us where in the project, we want to add this file, but because we do not have any sort of project structure at the moment, we can just leave everything as it is. Important to mention might be that Add to version control should be Git, as we are going to Git later on.

```
1  import QtQuick 2.15
2  import QtQuick.Controls 2.12
3
4  Item{
5      id: home_page
6      width: 360
7      height: 640
```

Now to clean up a bit, we can change the code in our newly created file a bit. We can more or less copy and paste the Item we had in our Swipe View corresponding to the name of the file inside the file. Also, we should update the imports to the same we have currently in our main.qml.

When you have done this, we can now do this for the two remaining files we need. The process is the exact same and should not be too difficult for you, but if it is then you can follow the way we did it for Home_Page.



When you have done everything the same as the first file you should be left with three newly added files in our qml.qrc directory. The Code in all of them should be more or less

exactly the same, with the only difference being that the ids of the Items are different.

```
10      SwipeView{
11          id: swipeView
12          anchors.fill: parent
13          currentIndex: 0
14          interactive: false
15
16          Home_Page{
17              id: home_page
18          }
19          Game_Page{
20              id: game_page
21          }
22          End_Page{
23              id: end_page
24          }
25      }
```

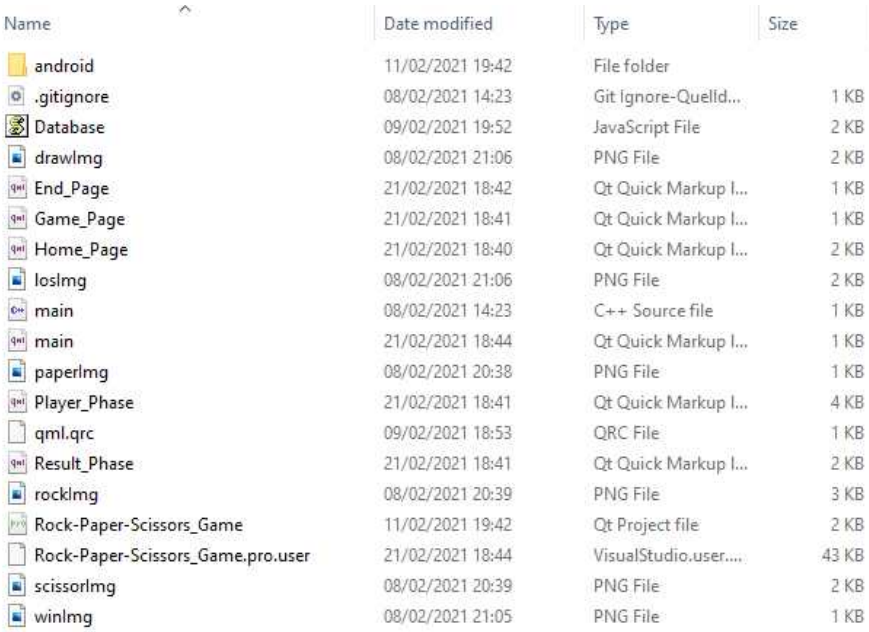
Lastly, we can change our main.qml a little, the items of our Swipe View can be changed to the corresponding name of the Files we just created. This will immediately link them together and we can then use them.

I left the id, as well as with and height of our items as they are and only changed the name of the component. This was first of because the Swipe View requires that the items inside of it have a fixed with and height and secondly it means that we did not need to change so much code.

– Adding the Project to Git –

As I promised we are going to add this project to Git, this first of is a good example how this is done in Qt, and you can refer back to this when you create your own projects later down the line.

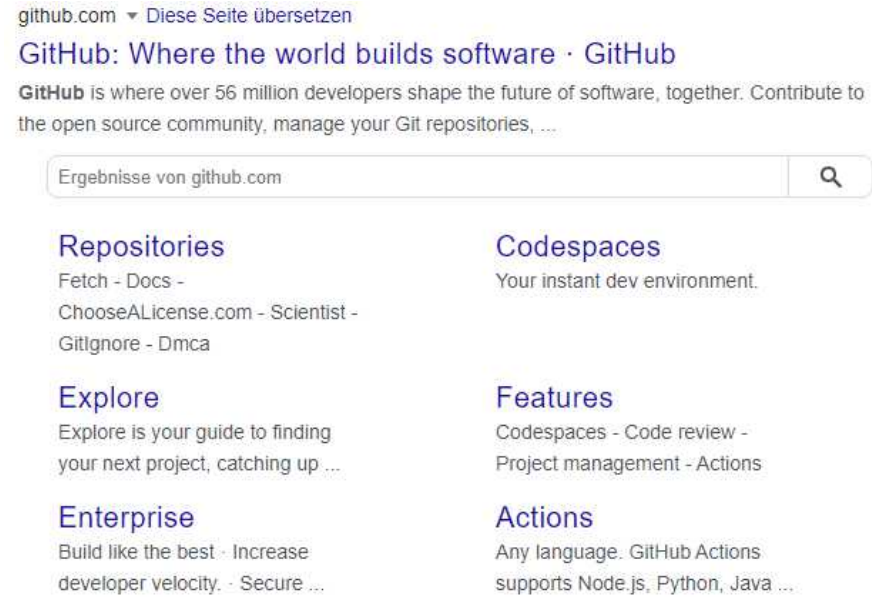
The first thing you need to do is open up the project directory on your drive.



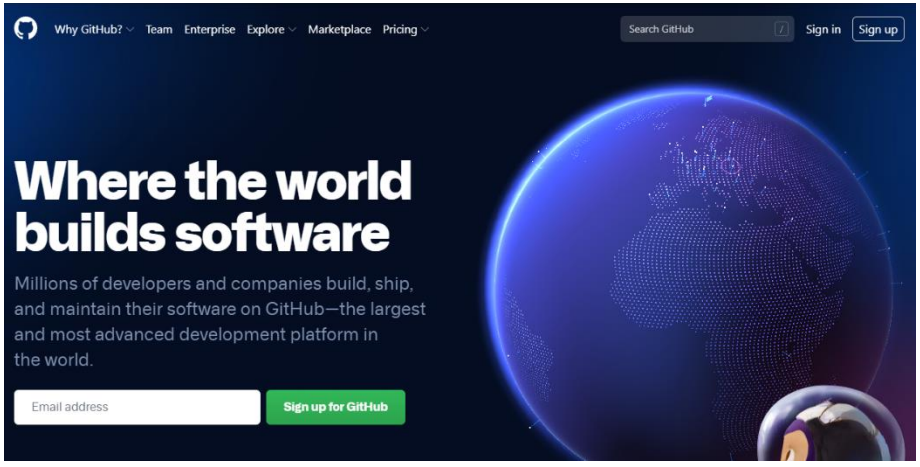
Name	Date modified	Type	Size
android	11/02/2021 19:42	File folder	
.gitignore	08/02/2021 14:23	Git Ignore-Quelld...	1 KB
Database	09/02/2021 19:52	JavaScript File	2 KB
drawImg	08/02/2021 21:06	PNG File	2 KB
End_Page	21/02/2021 18:42	Qt Quick Markup l...	1 KB
Game_Page	21/02/2021 18:41	Qt Quick Markup l...	1 KB
Home_Page	21/02/2021 18:40	Qt Quick Markup l...	2 KB
loslmg	08/02/2021 21:06	PNG File	2 KB
main	08/02/2021 14:23	C++ Source file	1 KB
main	21/02/2021 18:44	Qt Quick Markup l...	1 KB
paperImg	08/02/2021 20:38	PNG File	1 KB
Player_Phase	21/02/2021 18:41	Qt Quick Markup l...	4 KB
qml.qrc	09/02/2021 18:53	QRC File	1 KB
Result_Phase	21/02/2021 18:41	Qt Quick Markup l...	2 KB
rockImg	08/02/2021 20:39	PNG File	3 KB
Rock-Paper-Scissors_Game	11/02/2021 19:42	Qt Project file	2 KB
Rock-Paper-Scissors_Game.pro.user	21/02/2021 18:44	VisualStudio.user....	43 KB
scissorImg	08/02/2021 20:39	PNG File	2 KB
winImg	08/02/2021 21:05	PNG File	1 KB

This should look kind of like this at the moment. As you can see there are all the files we created, as well as all the common Qt files that were created when we created the project.

Now that we are ready, there are a few ways to go about pushing this project to Git. But they all start the same by opening up a Git Account. So, this is what we are going to do first. Google Git Hub and click on the first link that comes up.

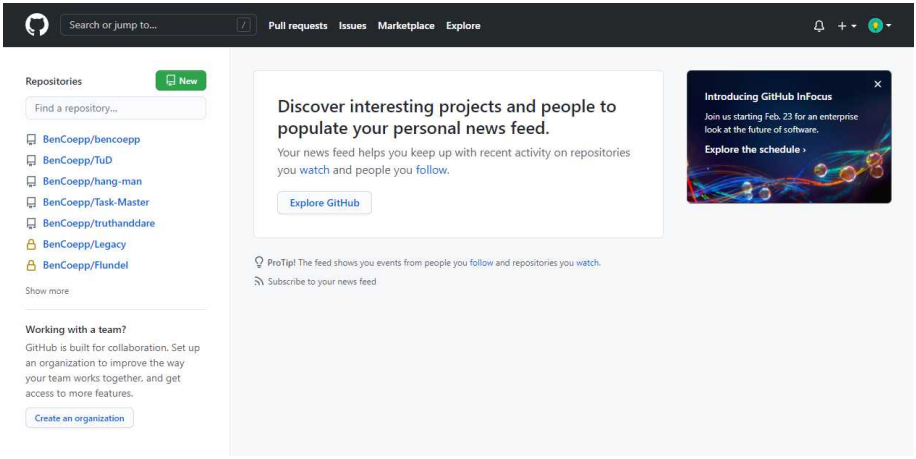


Google search for Git Hub



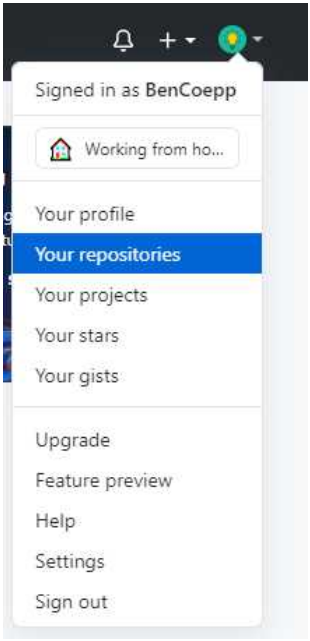
Git Hub homepage

Next you can click the Sign in or Sign-up buttons in the top left. I already have a Git Account, so I will sign in, if you do not already have one you can sign up. The sign-up process is not too difficult, so go ahead and do that.



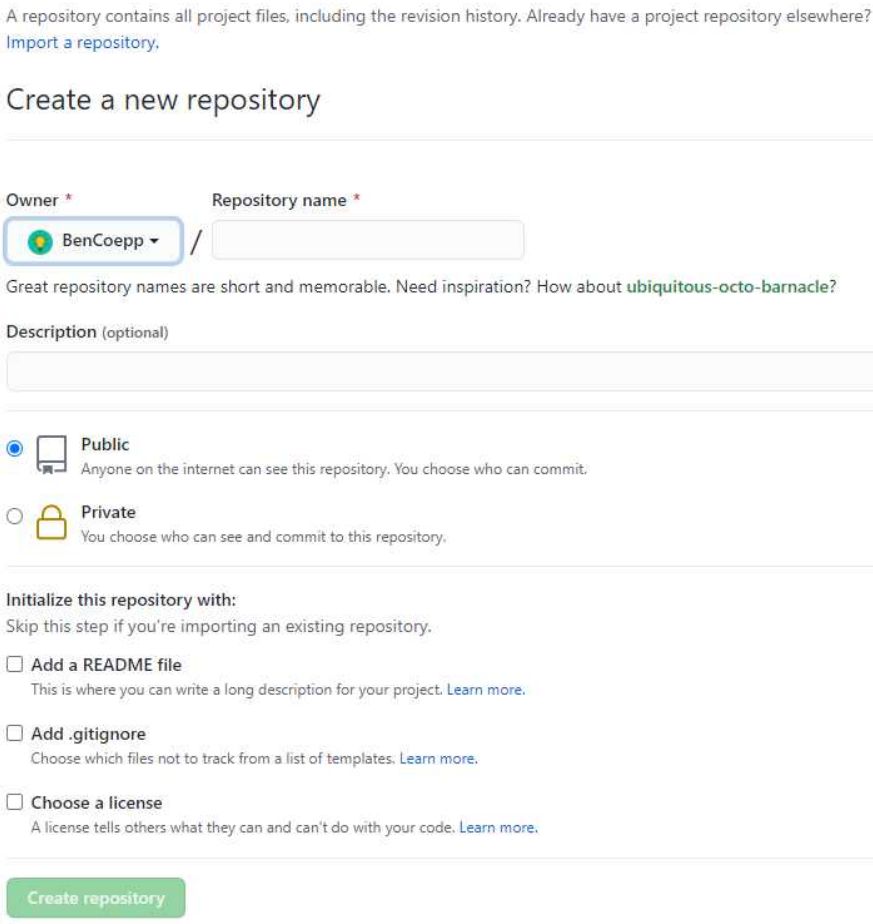
Git Hub logged in page

When you are signed in you should see this page. We now need to create a new repository, there are a few ways to do this, one is by the green button in the top left with the bookmark. The other way would be when you click on your account profile icon, this will open up a drop down.



From there you can go to your repositories and create the new repository from there. But we can just click the green button on the left.

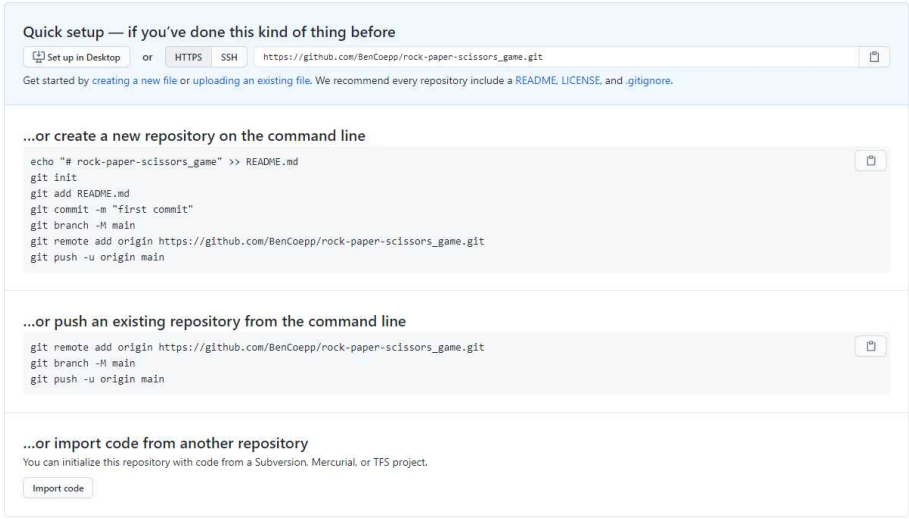
This might also be a good time to tell you to follow me on Git Hub. I tend to update already existing projects from time to time, and I will try to keep the projects covered in this book up to date to the book. Also, there might be some other projects or repositories that might interest you. I have developed quit a lot using Qt and some of the projects are already a few years old. But they might still be interesting for you.



This will transport us to a new page, we need to fill out a few things. First of we need to fill out the name for our repository. For us, a good name could be *rock-paper-scissors_game*.

Other the name we do not need anything else right now. You could change the repository to private if you do not want to make the repository visible for the entire internet. After that you

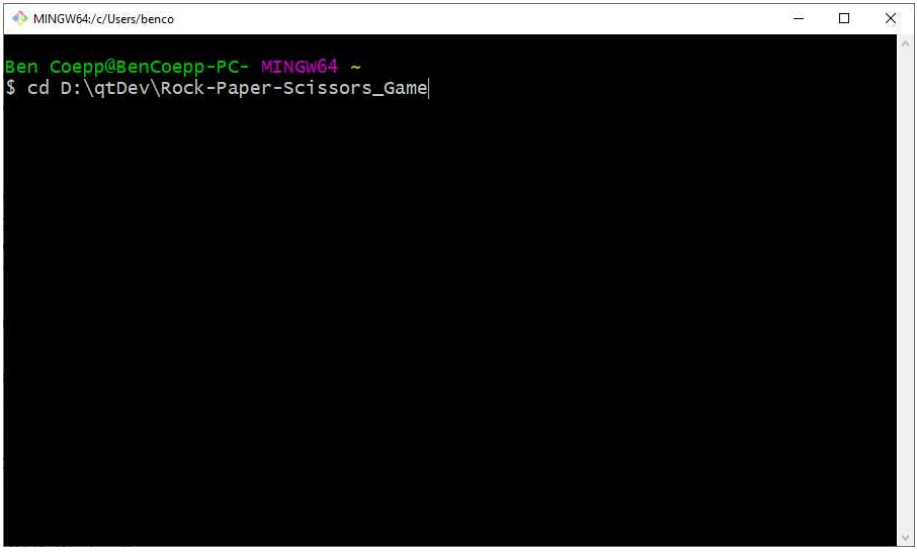
can click the green Button at the bottom that says *Create repository*.



Guide to create a new repository, or adding a new repository

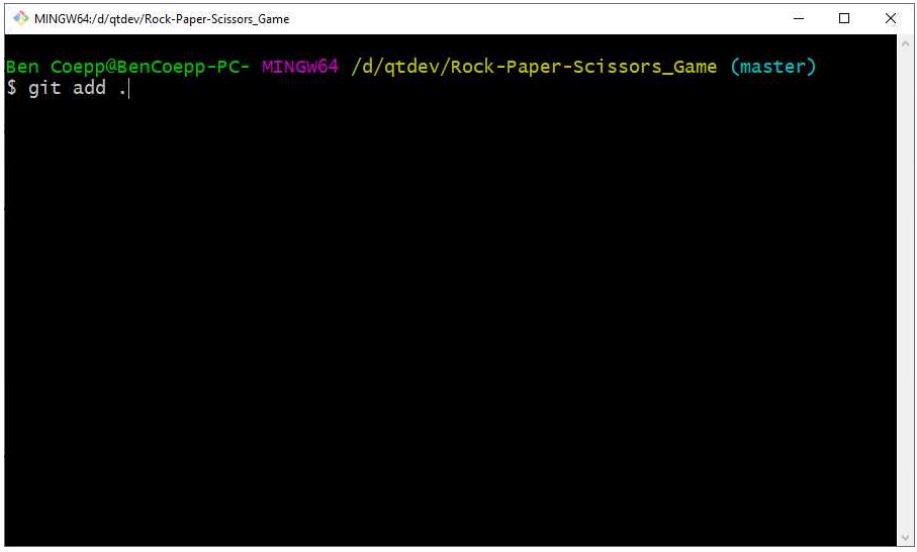
After a while of loading, you will be presented by this new page, here you can see the commands we need so that our project is brought to Git Hub. As we already have a repository on our local machine, as a side note Qt always creates a repository for you when you create a project with version control enabled from the start, we need to use the second set of commands to get the project online. Now getting the project online, can be done throw a few different ways, but the most standard and commonly used one is by Git Bash. If you do not have Git Bash, search for it online and choose the link in the next screenshot.

This brings you to the Git for Windows website. There you can just download and install Git for your machine. When you installed Git Bash, you can open it up in the directory.



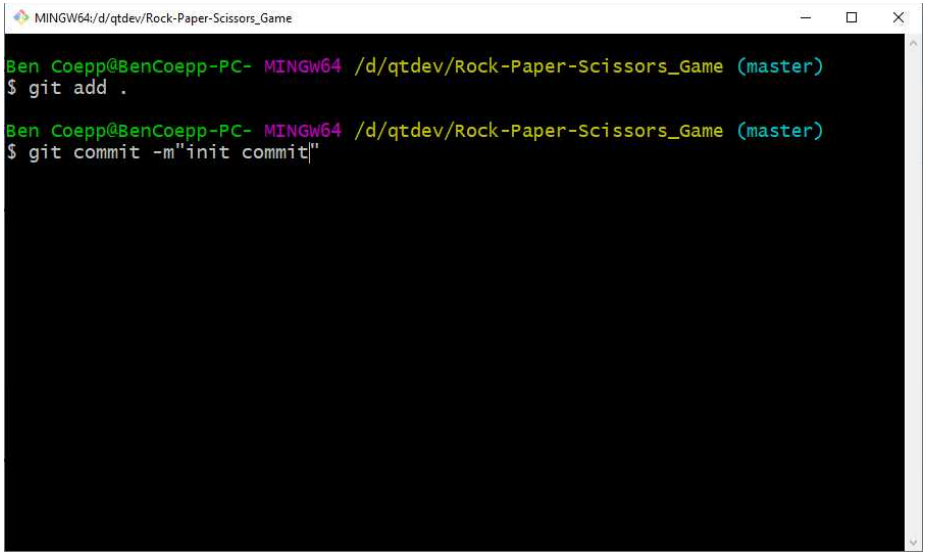
Going to the folder where we saved the project

Here you can see how to move to the correct folder using cd. You can see that you moved into the correct folder when you see a (master) on the right of the path.



Adding all the newly created files

Next, we can use the command *git add .* this will stage all of the files we created with Qt, you can imagine this like adding a lot of files to a list.



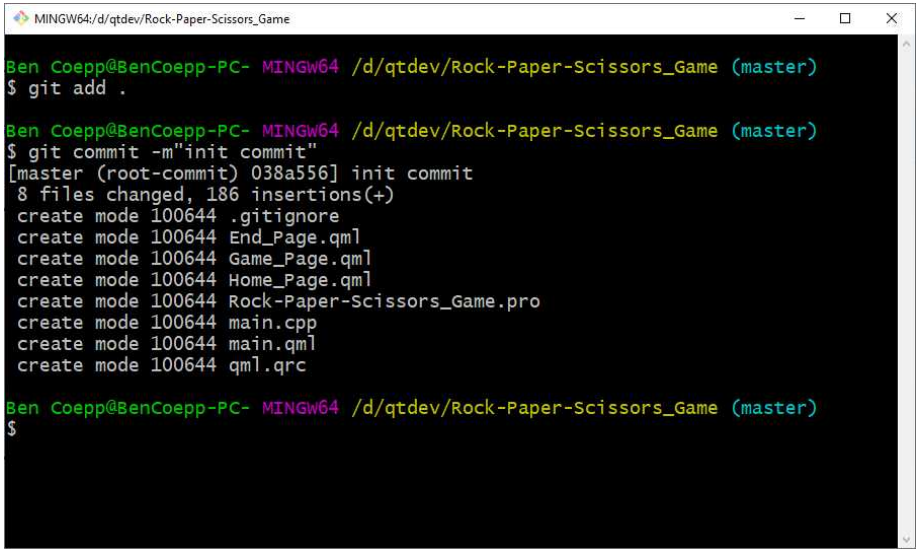
Committing all the files we just added

The next command we need to run is *git commit -m"**commit msg**"* this will take the staged files and added a commit

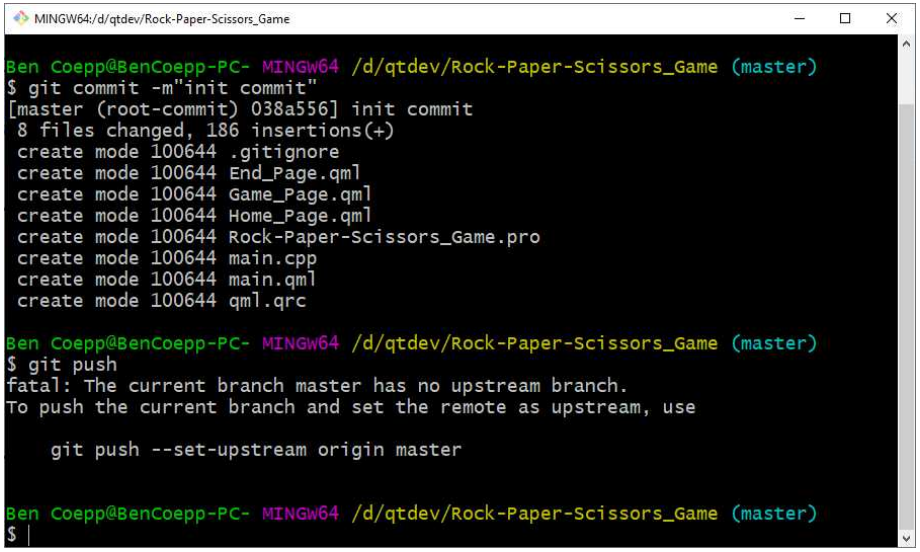
massage. This is now called a commit. This commit needs to be pushed now to Git Hub. In development we call this this *origin remote*.

As a side note, I would recommend that you try to keep the commit messages as professional and precise as you can. There is nothing more embarrassing for a developer when your boss comes to you and tells you that commit, ***I like turtles*** is not that professional.

Also, it is not possible to guess what the changes in the commit do by the name of it, so always say what you are doing in the commit so everyone knows what is going on.

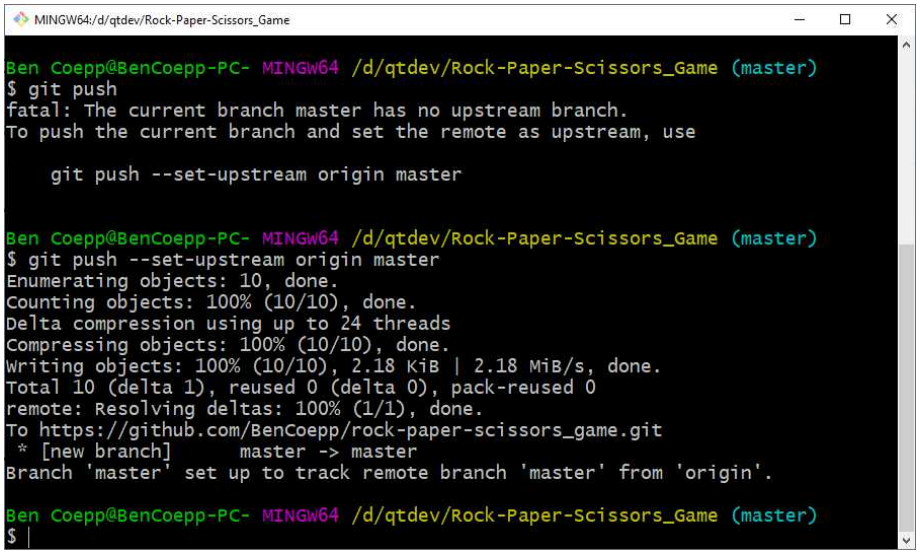


Result of committing the files



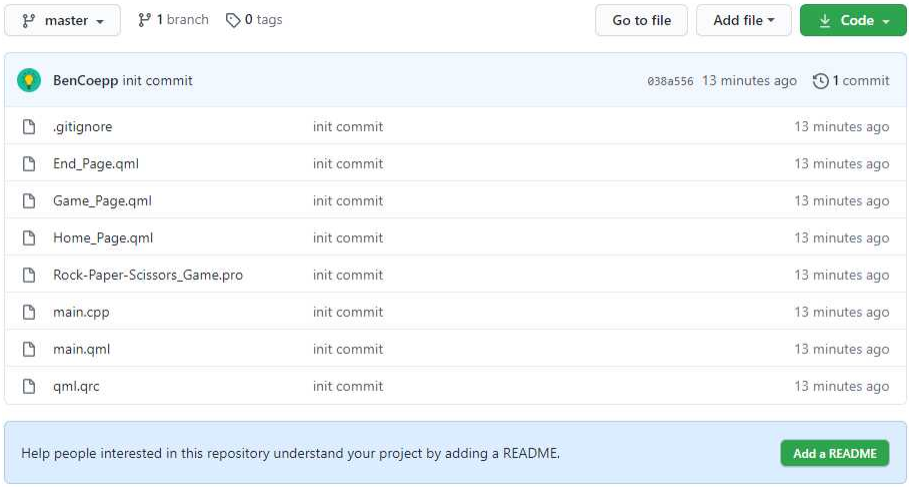
Git push with fatal error

To upload the files now to our remote, we need to run the command *git push*. This will not work right now, because when you push for the first time, you need to clarify an upstream to the origin remote.



Successful git push

You can just copy and paste the command that git tells you to run. It might happen, that a dialog box opens that asks you to log in with your Git Hub credentials. When your log in credentials is correct the push will start and the project will be uploaded to Git Hub. When it tells you that it is done, you can go back to the site of Git we had open, and when you reload it, you can see all the newly added files.



On Git Hub project site

With this, our project is now online on Git Hub, later on when we created a few more files and changed others, we are going to open Git Bash again and then push the new files and changes to Git Hub. But for now, let us continue with the development of the application.

- Creating the Home_Page -

First of go back to our newly created Home_Page.qml file. There we are going to create a rectangle as our background component.

```
1  import QtQuick 2.15
2  import QtQuick.Controls 2.12
3
4  Item{
5      id: home_page
6      width: 360
7      height: 640
8
9      Rectangle{
10         anchors.fill: parent
11         color: "#3e5a79"
12     }
13 }
```

Simple Home_Page.qml

The color we used here is the same that we used throughout all of our applications until this point. Next, we can create the button throw which we can start the game.

```
13      RoundButton{
14          id: startGameBt
15          anchors.centerIn: parent
16          width: 200
17          text: "Start Game"
18          background: Rectangle{
19              anchors.fill: parent
20              radius: 20
21              color: "#fd7e35"
22              border.width: 1
23          }
24          onClicked: {
25              swipeView.setCurrentIndex(1)
26          }
27      }
```

Start game button

The button we have here is remarkably similar to the ones we created so far, we have an id, an anchor to centre the button on the screen and a background component so that we can have a slightly customised button. Lastly, we have our onClicked event, that switches to the next Page in our Application.

```
4   ApplicationWindow {
5       width: 360
6       height: 640
7       visible: true
8       title: "Rock-Paper-Scissors"
9
10      property var winCount: 0
11      property var losCount: 0
```

Properties win and los count

To make our Home_Page a little bit more interesting we can represent the win and los counter on the page. First of we need

to add two properties to our main.qml file, they need to be available to the entire page and the best way to do that is declaring them in our main.qml.

Next, we can create a Label inside of our Home_Page.qml, the text should be of color white to be readable on the relatively dark background, and we should bin the label to the bottom centre of our page, using *anchors.bottom: parent.bottom* and *horizontal.center: parent-horizontalCenter*.

```
28      Label{
29          anchors.bottom: parent.bottom
30          anchors.horizontalCenter: parent.horizontalCenter
31          color: "white"
32          font.bold: true
33          text: "Win:: "+ winCount +" | "+ losCount+" ::Loss"
34      }
```

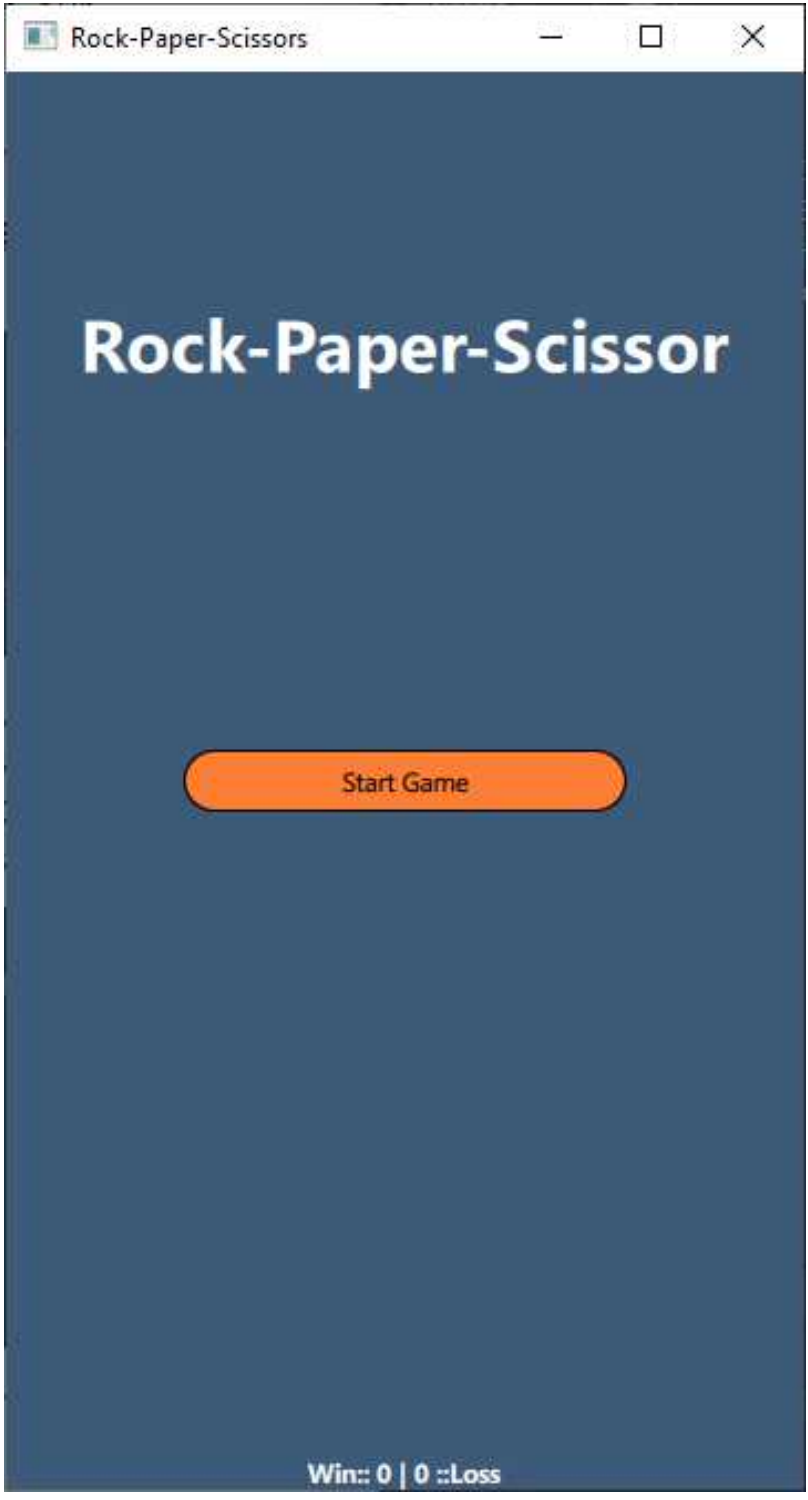
Win and los counter output

The only thing really missing on the Home_Page now is the Title of our Application. The label is not too difficult, we centre it to the top and horizontal centre and then give it a top margin so that it is not right up at the top, after that, it should be also white to be better readable and the font size should be around 25 so it is nice and readable even from a far.

```
14      Label{
15          anchors.horizontalCenter: parent.horizontalCenter
16          anchors.top: parent.top
17          anchors.topMargin: 100
18          text: "Rock-Paper-Scissor"
19          color: "white"
20          font.pointSize: 25
21          font.bold: true
22      }
```

Titel Label

Now if we were to run our application you will see that everything is rendered as it should so far.


Rock-Paper-Scissors
Rock-Paper-Scissor
Start Game
Win:: 0 | 0 ::Loss

The only thing we probably should change is the buttons width and height, as well as that the button should be round. I just find this aesthetically more pleasing, you might want to leave the button as it is, that choice is up to you.

```
24      RoundButton{
25          id: startGameBt
26          anchors.centerIn: parent
27          width: 200
28          height: 200
29          radius: 99
30          text: "Start Game"
31          background: Rectangle{
32              anchors.fill: parent
33              radius: 99
34              color: "#fd7e35"
35              border.width: 1
36          }
37          onClicked: {
38              swipeView.setCurrentIndex(1)
39          }
40      }
```

Start Game button

After the changes, our button should look something like this. And with this we are more or less done with our Home_Page.qml. It is not a really difficult page, and it did not take that much to pull off, but as always there is a lot more you could do, like adding animations, or making everything a little bit prettier, but that is not our main mission here.

- Creating the Game_Page -

The Game_Page will consist out of a Stack View, this Stack View will load the pages throw out the game loop. Also, in this Game_Page the functions for the game will be located in. But enough about the rambling if you want to have a clearer overview of what the functionality should look like, you should have a look at Chapter 2.4.3.2, there we discussed how the game should work and function.

```
1   import QtQuick 2.15
2   import QtQuick.Controls 2.12
3
4   Item{
5       id: game_page
6       width: 360
7       height: 640
8
9       Rectangle{
10          anchors.fill: parent
11          color: "#3e5a79"
12      }
13  }
```

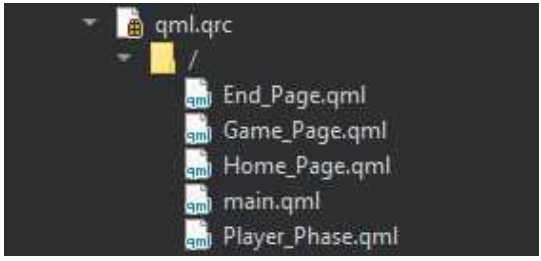
Game_Page.qml

First of we can again update the imports, I know this will probably bore you to death by now, but this really needs to be done every time we have a new file.

```
14      SwipeView{
15          id: gameFrame
16          anchors.fill: parent
17          currentIndex: 0
18
19          Player_Phase{
20              id: playerPhase
21              width: 360
22              height: 640
23          }
24          Result_Phase{
25              id: resultPhase
26              width: 360
27              height: 640
28          }
29      }
```

Swipe View with Phases

Inside of our Game_Page we need to add a Swipe View, that Swipe View needs an id, so that we can interact with it later on, as well as an *anchors.fill: parent*. Inside of here we can now place these custom items inside the Swipe View. As you probably think right now, we do not have these jet, so let us create them.



As we already created multiple new pages throughout this book and even two new ones in this chapter, I assume you will be able to create the Player_Phase.qml without any problems.

```
4  Item {
5      anchors.fill: parent
```

After you created the file, you can go inside and we can start editing the code. As always update the imports if you did not do this already. After that we can give the item an *acnhors.fill: parent* to make it responsive.

```
7      ListView{
8          id: optionListView
9          anchors.bottom: parent.bottom
10         height: 50
11         width: parent.width
12         orientation: ListView.Horizontal
```

Now we come to the real part of the application. We need to create a ListView here, this ListView will be used to display the options the player has to choose from to play. In this case it might only be Rock, Paper, Scissor but non the less they need to be displayed so the player can choose from them.

For the attributes, the ListView should have are, it should be anchored to the bottom of our page, have a width that is identical to its parents. A height of 50 should be enough for the icons. Lastly, I also added the orientation property to our List View. We already used this in the Hang-Man project. Here I do not want our Items to be displayed vertically, but horizontally. This is in my opinion just nicer to look at.

```
13          model: ListModel{
14              id: optionModel
15              ListElement{
16                  img: ""
17                  value: "rock"
18              }
19              ListElement{
20                  img: ""
21                  value: "paper"
22              }
23              ListElement{
24                  img: ""
25                  value: "scissor"
26              }
27          }
```

Next, we can add our List Model to the corresponding model for our ListView. For now, we only have three List Elements inside of our List Model. These are the options the player can choose from. Each List Element has to values inside of it, one is the actual value of the element, and the other is going to be a link to an image so we have a visual representation of the element.

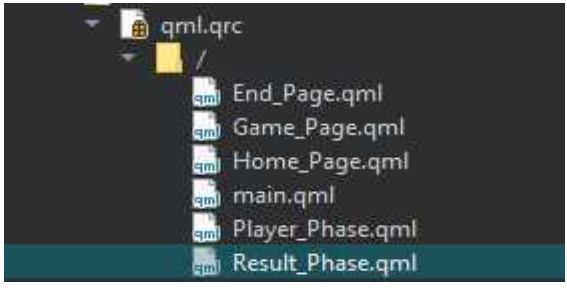
This is not the best way of implementing all the different options we could have in our game. But I like this option, as it allows us to quit easily implement another option, delete one and or give the player the option to implement their own options.

But this is to be expected of the List Model we have here. You probably are already quite familiar with the different functionalities a List Model has to offer.

```
28          delegate: Item {
29              width: parent.width/3
30              height: parent.height
31
32              MouseArea{
33                  anchors.fill: parent
34                  onClicked: {
35                      //function that starts the game
36                      console.log(value)
37                  }
38
39                  Image {
40                      anchors.fill: parent
41                      antialiasing: true
42                      source: img
43                  }
44              }
45          }
```

The delegate also follows the List Models theme of being fairly simple. It consists out of an item, the width of this item, should be a third of the width the parent has, so just the width/3. The height can be the same as the parent's height.

Inside of the item, we are going to place a Mouse Area, this will fill the item and have a onClicked even inside of it. For now, we will only have a simple console log where we print out the value of the item we clicked. Inside of this Mouse Area we can also place an Image that can display the img data. We can just fill its parent with the Image and set antialiasing to true. The source of this Image should be just img, as this will hold the data of the element.



As we are nearly done with the Player_Phase.qml we can create Result_Phase.qml. Like before we did this multiple times already.

So, I suspect you can do this on your own. If that is not the case then go back to the previous chapters were, we did this again.

Next open up the Result_Phase.qml and we can start editing the code. As always update the imports and add an *anchors.fill: parent* to the Item already present in the file. Next, we can add the first real component to the item, a Mouse Area.

The Mouse Area should have a width and height of 50 for now, it should be cantered on the screen. We can already also add the onClicked event for this Mouse Area. In this case we can immediately tell the gameFrame, which to give you a hint is the second Swipe View we created, to go back to the first item or just index 0. This can be best done throw the *.setCurrentIndex()* method.

Inside this Mouse Area we can also add another Image, that is filling the Mouse Area and has for now an empty source property.

```
import QtQuick 2.15
import QtQuick.Controls 2.12

Item {
    anchors.fill: parent

    MouseArea{
        width: 50
        height: 50
        anchors.centerIn: parent
        onClicked: {
            gameFrame.setCurrentIndex(0)
        }
        Image {
            anchors.fill: parent
            antialiasing: true
            source: "file"
        }
    }

    Label{
        anchors.bottom: parent.bottom
        anchors.bottomMargin: 100
        anchors.horizontalCenter: parent.horizontalCenter
        color: "white"
        font.bold: true
        font.pointSize: 25
        text: "Win:: "+ winCount +" | "+ losCount+" ::Loss"
    }
}
```

Below the Mouse Area we can add a Label. This label should display the current Win and Los of the player. If you do not want to type everything out, you can go to the Home_Page and copy the Label from there and only change the properties you need here. Other than that, we position this Label also to the bottom of our page but give it a bottom margin and push it a little higher.

```
32            MouseArea{
33                anchors.fill: parent
34                onClicked: {
35                    //function that starts the game
36                    console.log(value)
37                    gameFrame.setCurrentIndex(1)
38                }
```

Now we can also change the onClicked function in our Player_Phase.qml file a little. For now, we can just add *gameFrame.setCurrentIndex(1)*. We need this to change the page of the Swipe View.

```
4   Item{
5       id: game_page
6       width: 360
7       height: 640
8
9       property var playerOption: ""
10      property var botOption: ""
11
```

playerOption and botOpton added to Game_Page.qml

Inside of our Game_Page.qml we can add two properties. The first one is the playerOption and the botOption. They will hold the option that they for instance the player has made.

```
32  MouseArea{
33      anchors.fill: parent
34      onClicked: {
35          //function that starts the game
36          console.log(value)
37          gameFrame.setCurrentIndex(1)
38          playerOption = value
39          botOption = optionModel.get(Math.floor(Math.random() * optionModel.count))
40      }
```

Now to the real fun of our application, go back to our Player_Phase.qml and let us change the onClicked event a little.

First of we can add that the playerOption should be the value we clicked. That is pretty straight forward and understandable.

To set the botOption we need a little bit more code. Basically, we get a random Item from our optionModel. The function you see in the brackets is basic Java Script. We basically generate a random number and that is then going to be the botOption.

```
function winCheck(){
    if(playerOption === botOption){
        //draw
    }else if(playerOption === "rock" && botOption === "scissor"){
        //player win
    }else if(playerOption === "paper" && botOption === "scissor"){
        //player win
    }else if(playerOption === "paper" && botOption === "rock"){
        //player win
    }else if(playerOption === "scissor" && botOption === "rock"){
        //bot win
    }else if(playerOption === "scissor" && botOption === "paper"){
        //player win
    }else if(playerOption === "rock" && botOption === "paper"){
        //bot win
    }
}
```

Now that we have a player and bot Option, we can write our winCheck function. For now, this can just be an extremely ugly if-else statement that just checks all the possible variants of rock, paper, and scissor.

```
function winCheck(){
    if(playerOption === botOption){
        //draw
    }else if(playerOption === "rock" && botOption === "scissor"){
        //player win
        winCount++
    }else if(playerOption === "paper" && botOption === "scissor"){
        //player win
        winCount++
    }else if(playerOption === "paper" && botOption === "rock"){
        //player win
        winCount++
    }else if(playerOption === "scissor" && botOption === "rock"){
        //bot win
        losCount++
    }else if(playerOption === "scissor" && botOption === "paper"){
        //player win
        losCount++
    }else if(playerOption === "rock" && botOption === "paper"){
        //bot win
        losCount++
    }
    gameFrame.setCurrentIndex(1)
    //upload to local storage
}
```

We also can add right now the basic functionality of increasing the los and win count. This can be easily done by just using the ++ operator and increasing the number by 1.

At the bottom of the long if-else statement we can add the change of the gameFrame index. Basically, when the function finishes the player is transported to the Result_Phase of our Application.

```
property var playerOption: ""
property var botOption: ""
property var winState: 0 //0=draw 1=player Win 2=player los
```

Next, we can add another property to our Game_Page, the winState property. For now, this property should be initialised with 0, on the right of that you can see a comment where I list the other possibilities this property can have.

```
Image {
    anchors.fill: parent
    antialiasing: true
    source: if(winState===0){}else if(winState===1){}else if(winState===2){}
}
```

The Image on our Result_Phase currently does not have anything as its source, but we cannot put anything really in there fixed. We need to write a function as the source, basically when the winState represents a value a specific image will be placed as the image source. It is a really simple function, but not the mist elegant way to build this functionality.

```
onClicked: {
    gameFrame.setCurrentIndex(0)
    winState = 0
    playerOption = ""
    botOption = ""
}
```

Above the image we had our onClicked function that was relatively empty so let us add a few more things to the event. Basically, when the player clicks this Mouse Area the winner was already selected and displayed, an so we can empty all of

the properties we have. This is not really necessary in this case, but I really like to empty properties that get different values next time, I do not know why I do this, but hey this is just my style.

```
36      Timer{
37          id: resultTimer
38          interval: 500
39          repeat: false
40          running: false
41          onTriggered: {
42              gameFrame.setCurrentIndex(0)
43              winState = 0
44              playerOption = ""
45              botOption = ""
46          }
47      }
```

Below all of the components we added so far, we can also add a Timer component. This is a component we did not have so far, so let me explain what it is.

A Timer can be best described as a clock, it ticks down time you set in its interval property and when the time is up you get an onTriggered even and then you can run a function. This is extremely great when you want to trigger certain behaviour in your application in a time-based frame.

Why do I want to use this here? Well, when the player gets to the Result phase, he has the option to click the image and get back to the Player_Phase page, but maybe he does not want to click there, then I do not want the player just sitting there the entire time, so when this timer is up the same functions are going to be run as when the player clicked the Mouse Area.

```
17          Image {
18              anchors.fill: parent
19              antialiasing: true
20              source: if(winState===0){}else if(winState===1){}else if(winState===2){}
21              onSourceChanged: {
22                  resultTimer.start()
23              }
24          }
```

A small problem we have so far is that the timer will not start immediately when we come to this page, the best way in my opinion to start the timer is by listening to the source change event of our Image. This will trigger when the player comes to this page and the winState is not 0 when that is the case there is no change in the source of the image as the source was already there. But if it is anything else then 0 then the source will be changed and the timer can then be started right away. This is a fairly nice way of doing something like this.

Now that we are done so far with the functionality, we can now go and grab some icons for our Rock, Paper, and Scissors. I got mine online, and if you want to get the same then check out the Git Repository, there you will find all the images you will need.

[20]

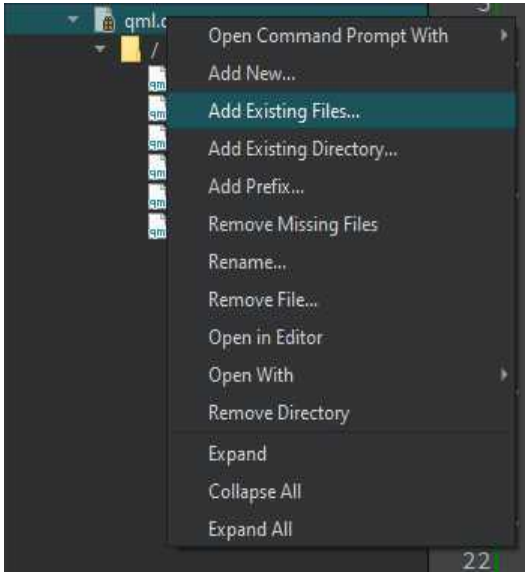
I choose these images above, I like them and they fit the style I am going for, but as always, they are not meant to be so pretty, I only want them to represent the value and that is it.

But a far better topic then what images I choose is how to get images into your project. I will now present you with the fastest

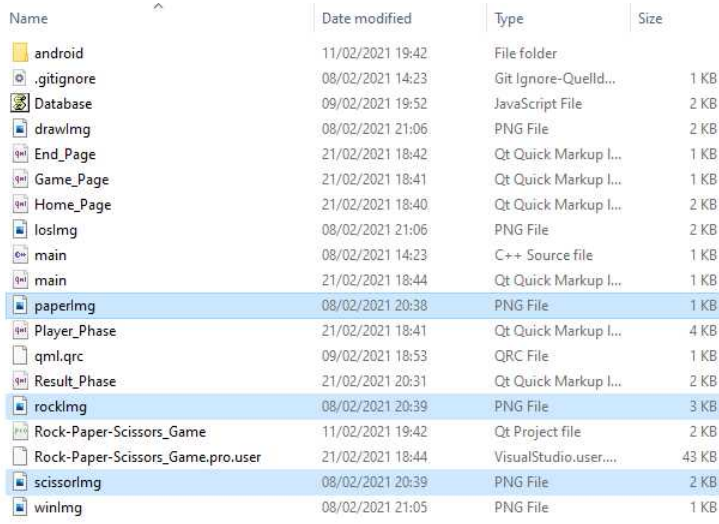
[20] I made these icons myself using inspiration online. They are not really good, and there are a lot of better ones out there but I wanted to do them myself

and easiest way I know to get them in your project, it is not the best, or nicest way but I like it in its simplicity.

Before we can start the way to import the images into our project, you should have downloaded all the images we need into our project. Just dump them into the project folder and that is it.

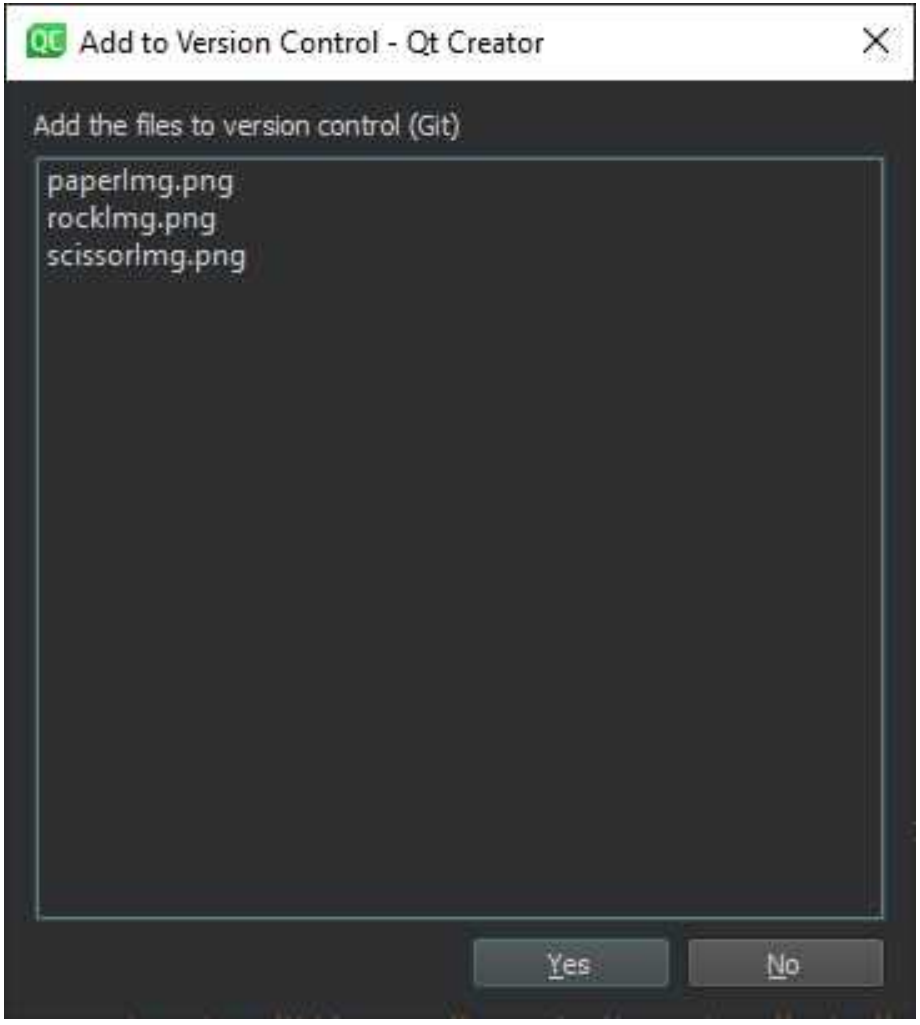


Now that you have all the images in the project directory, let us import them into our project. For that you should right click the qml.qrc directory in our project tree and then select the Add Existing File. This will open up an explorer, it should automatically show the project directory and here you need to select all the images you want.



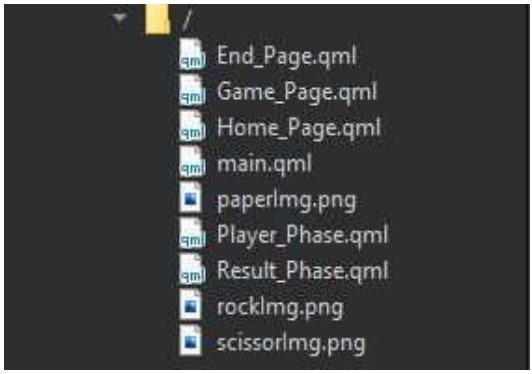
Name	Date modified	Type	Size
android	11/02/2021 19:42	File folder	
.gitignore	08/02/2021 14:23	Git Ignore-Quelld...	1 KB
Database	09/02/2021 19:52	JavaScript File	2 KB
drawImg	08/02/2021 21:06	PNG File	2 KB
End_Page	21/02/2021 18:42	Qt Quick Markup l...	1 KB
Game_Page	21/02/2021 18:41	Qt Quick Markup l...	1 KB
Home_Page	21/02/2021 18:40	Qt Quick Markup l...	2 KB
losImg	08/02/2021 21:06	PNG File	2 KB
main	08/02/2021 14:23	C++ Source file	1 KB
main	21/02/2021 18:44	Qt Quick Markup l...	1 KB
paperImg	08/02/2021 20:38	PNG File	1 KB
Player_Phase	21/02/2021 18:41	Qt Quick Markup l...	4 KB
qml.qrc	09/02/2021 18:53	QRC File	1 KB
Result_Phase	21/02/2021 20:31	Qt Quick Markup l...	2 KB
rockImg	08/02/2021 20:39	PNG File	3 KB
Rock-Paper-Scissors_Game	11/02/2021 19:42	Qt Project file	2 KB
Rock-Paper-Scissors_Game.pro.user	21/02/2021 18:44	VisualStudio.user....	43 KB
scissorImg	08/02/2021 20:39	PNG File	2 KB
winImg	08/02/2021 21:05	PNG File	1 KB

When you have selected them, you can click open. This will lead to the explorer closing.



Images that are going to be added

When you have Git enabled this popup will open up. For all external files you add to your project tree, Qt will ask you if you want to add them to your version control. In our case I will choose yes, as I want these images in my Git Repository. And mostly you will probably do to so hit yes.



When everything was done correctly you will have a few new files in your project directory. Now I probably suspect a lot of people finishing now internally, as this looks extremely unorganised, and you are correct, but we are going to fix this later on.

```
13          model: ListModel{
14              id: optionModel
15              ListElement{
16                  img: "qrc:/rockImg.png"
17                  value: "rock"
18              }
19              ListElement{
20                  img: "qrc:/paperImg.png"
21                  value: "paper"
22              }
23              ListElement{
24                  img: "qrc:/scissorImg.png"
25                  value: "scissor"
26              }
27          }
```

Now that we have all the images in our project, we can add them to the places we need them in. The first place is in our options List Model. There we can add the corresponding images to the corresponding value and place it in the img property.

```
MouseArea{
    anchors.centerIn: parent
    width: 50
    height: 50
    onClicked: {
        //function that starts the game
        console.log(value)
        gameFrame.setCurrentIndex(1)
        playerOption = value
        botOption = optionModel.get(Math.floor(Math.random() * optionModel.count))
        winCheck()
    }
```

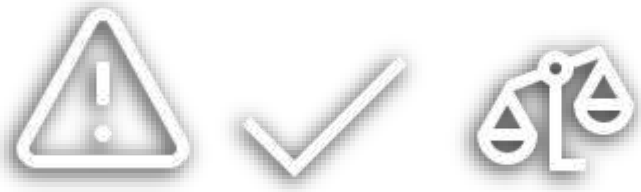
Next, we can add our nearly finished winCheck function to our Mouse Areas onClicked event. So, every time we click the Mouse Area we are moved to the next page and our winCheck function tells us who the winner is.

```
SwipeView{
    id: gameFrame
    anchors.fill: parent
    currentIndex: 0
    interactive: false
```

A small problem that we should also fix right around know is the fact, that if you were now to run our application, you would see that you can still swipe our Swipe View, which completely break our application. To stop this from happening we can just add the *interactive: false* attribute to both our Swipe Views.

```
SwipeView{
    id: swipeView
    anchors.fill: parent
    currentIndex: 0
    interactive: false
```

Next, we can have a look at the currently empty Image on our Result_Phase.qml page. We already build the function that takes the winState and then adjusts the image according to that, but we do not have any images for it right now.

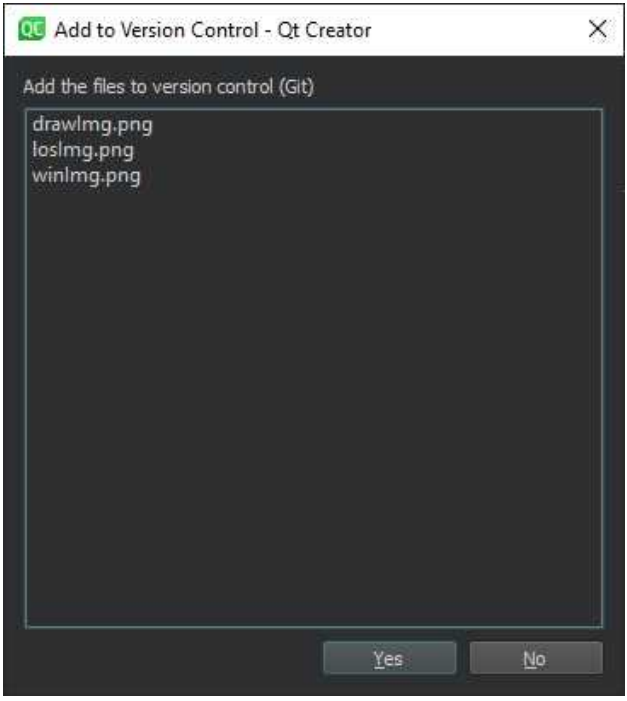


[21]

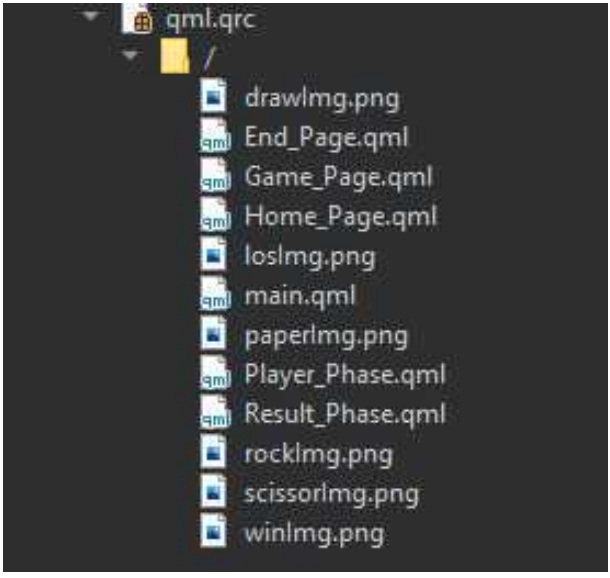
The images you can see above are the ones that I will use for this purpose. Basically, we have a check icon when you won, a warning sign when you lost and a balance when you got a draw.

They are not the best images you can use, and there are a few that probably would fit better on screen, but for the simple testing of images and our purpose of learning more about the topic it should do fine. As before we add these images by adding them as existing files, this will again open up the explorer and you need to select all the images you want to import. When you have done that you can click open and the popup will open up again.

[21] These images were again made by myself, you can find them in the repository on Git Hub



As before we can add all the images to our version control, when you have done that and clicked yes, you will see that the new images were added to our project tree. And now it looks even worse than before.



```
16          Image {
17              anchors.fill: parent
18              antialiasing: true
19              source: if(winState===0){"qrc:/drawImg.png"}
20                      else if(winState===1){"qrc:/winImg.png"}
21                      else if(winState===2){"qrc:/losImg.png"}
22              onSourceChanged: {
23                  resultTimer.start()
24              }
25          }
```

Now that we have imported our images, we can fill out the source function with the corresponding images. If you do not know how to get the URL of the image, then exactly right click the image in your project directory, there will be an option in the menu that opens up where you can do this.

Other than that, fill out the function with the URLs as you see in the screenshot above.

```
6       MouseArea{
7           width: 100
8           height: 100
9           anchors.centerIn: parent
```

A small change I made here is that I changed the size of the Mouse Area on our Result_Phase.qml to something a little bit larger. Other than that, we do not need to change to much here.

```
6       ListView{
7           id: optionListView
8           anchors.bottom: parent.bottom
9           anchors.bottomMargin: 50
10          height: 50
```

Also, I added a bottom margin to our option List View, just because it looks nicer and is a lot easier to read then before.

```
    MouseArea{
        anchors.centerIn: parent
        width: 200
        height: 200
        onClicked: {
            swipeView.setCurrentIndex(2)
        }
        Rectangle{
            anchors.fill: parent
            radius: 99
            color: "#fd7e35"

            Label{
                anchors.centerIn: parent
                font.bold: true
                font.pointSize: 25
                text: "Stop Game"
            }
        }
    }
```

The last thing we really need to do is add a stop game button to our Player_Phase.qml page. Currently there is no way to stop the game, which is not good, and we need to give the player the option to quit the game.

This can be best done by simply having a Mouse Area, with a Rectangle inside of it and a Label inside of that, this label only neds to say that you can stop the game when you press there. This is nothing special but it allows the user to see that with this the game can be closed.

- Creating the End_Page -

```
import QtQuick 2.9
import QtQuick.Controls 2.5

Item{
    id: end_page
    width: 360
    height: 640

    Rectangle{
        anchors.fill: parent
        color: "#3e5a79"
    }

    Label{
        anchors.bottom: parent.bottom
        anchors.bottomMargin: 100
        anchors.horizontalCenter: parent.horizontalCenter
        color: "white"
        font.bold: true
        font.pointSize: 25
        text: "Win:: "+ winCount +" | "+ losCount+" ::Loss"
    }
}
```

First of we need to change the End_Page.qml to look something like this. Basically, we have a Rectangle as our background component, and a label at the bottom of our screen that is more or less the same we used on the Home_Page.qml or Result_Phase.qml, only the position and size are a bit different.

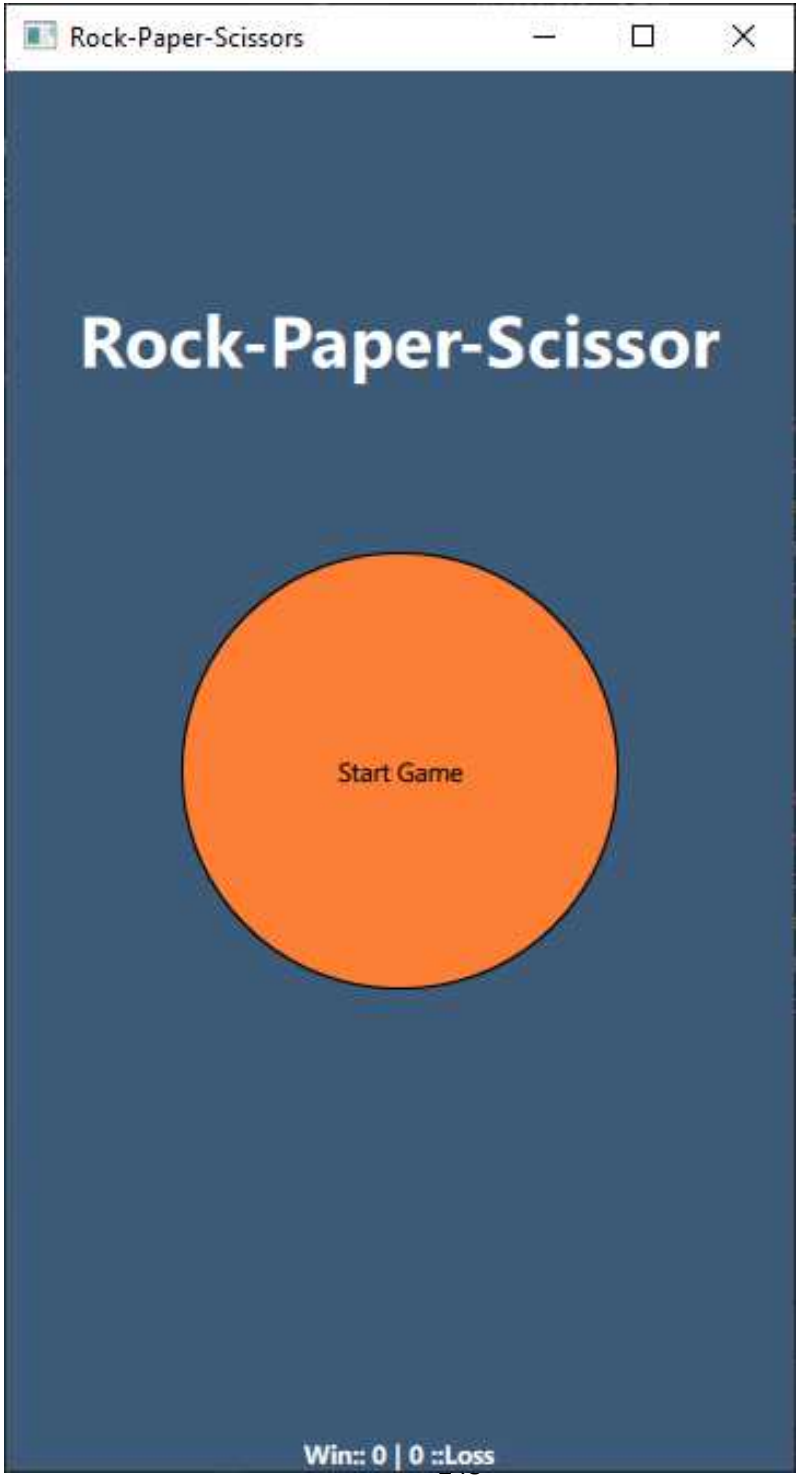
After that we can add another Label above the first one, we created, this will just tell you that you played a good game, not considering that you might lost just now, but hey.

```
    Rectangle{
        anchors.fill: parent
        color: "#3e5a79"
    }

    Label{
        anchors.top: parent.top
        anchors.topMargin: 100
        anchors.horizontalCenter: parent.horizontalCenter
        color: "white"
        font.bold: true
        font.pointSize: 25
        text: "Good Game!!"
    }

    Label{
```

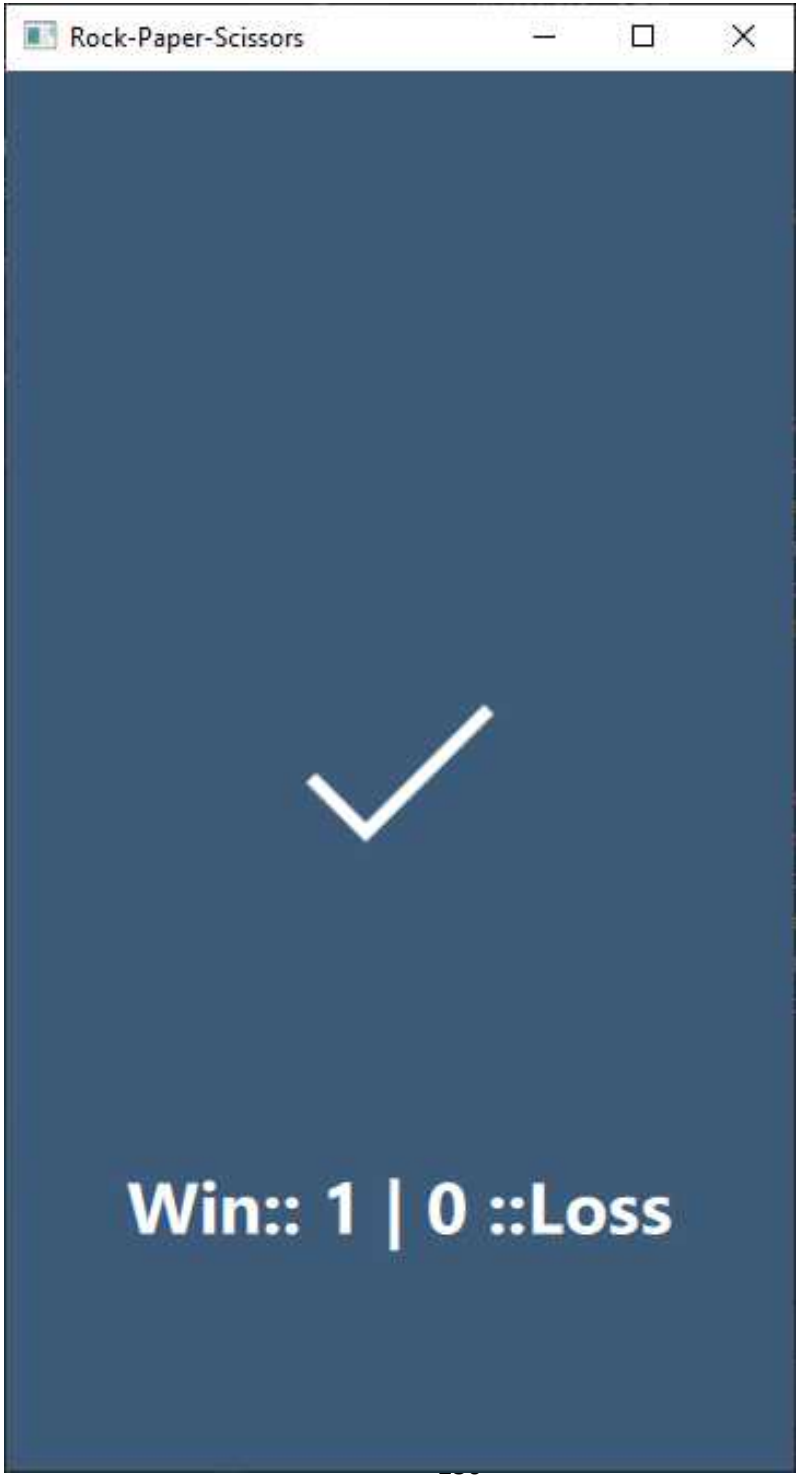
This is a basic end page. In my opinion the best end pages are the ones that only tell you the bare minimum that is needed. Here we only have a Label that tells you that you had a Good Game and below we have something that displays the score of the game and that is it. A remarkably simple and clean end to our game.

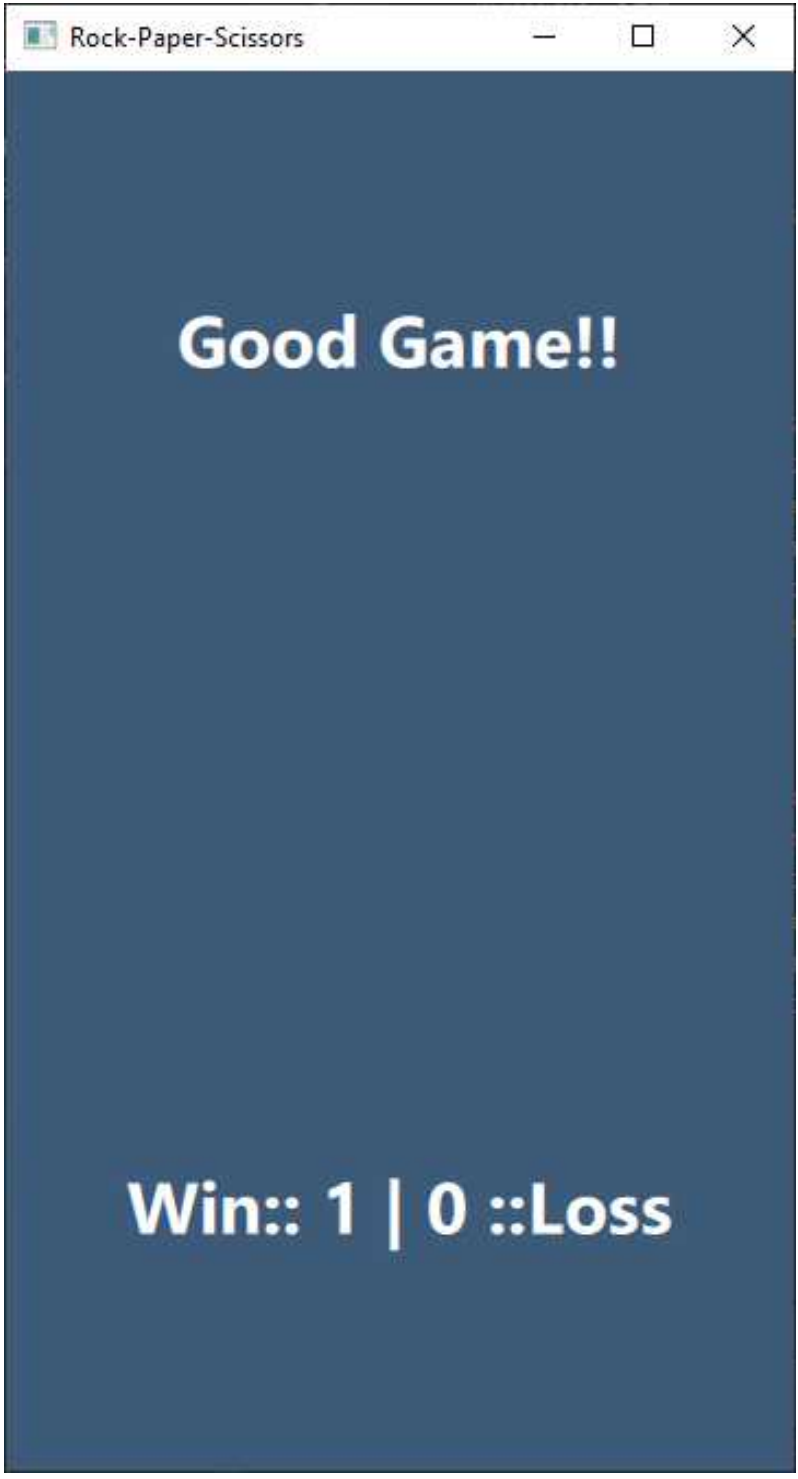
As a lot of people do you might want to make the End_Page a little bit prettier and make it stand out a bit more. But for what I was aiming for this was not necessary. But it would be a good learning experience so feel free to do this after this project if you want to.

And with that we are more or less done with the writing of our Application, so let us have a look at what we did.


Rock-Paper-Scissors
Rock-Paper-Scissor
Start Game
Win:: 0 | 0 ::Loss


Rock-Paper-Scissors
Stop Game

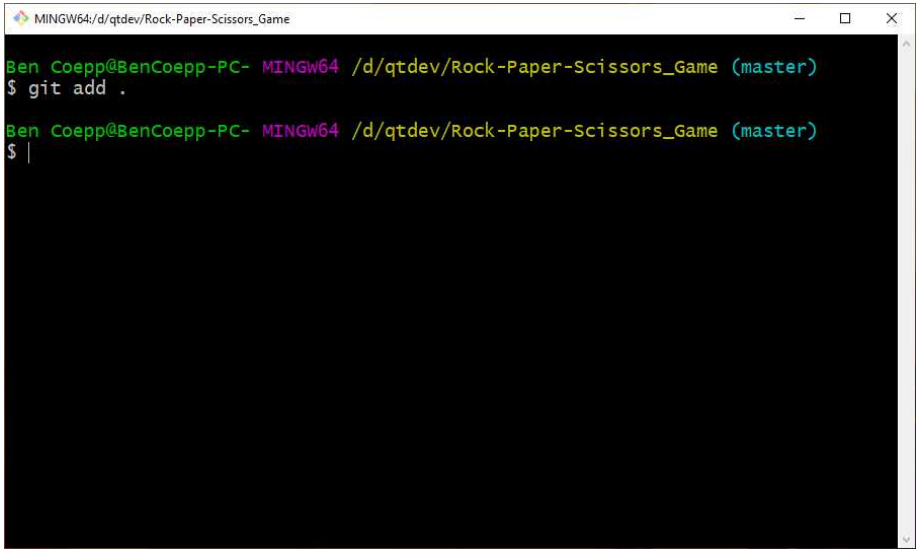

Rock-Paper-Scissors
Win:: 1 | 0 ::Loss


Rock-Paper-Scissors
Good Game!!
Win:: 1 | 0 ::Loss

If you want to run the application, you can just hit the green arrow down in the left bottom. If you have done everything as we did it so far, the application should run just like you saw right now.

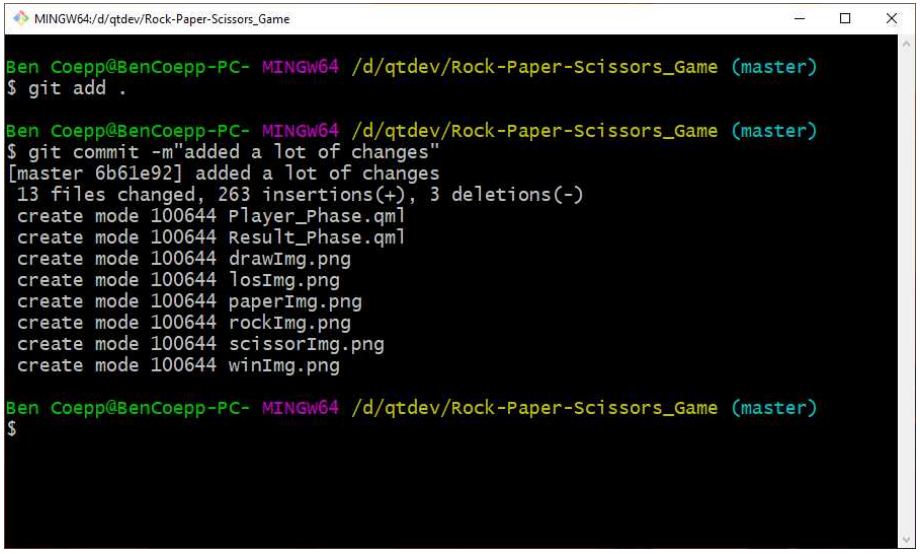
The application in that sense is now finished and running, and we have made a Rock-Paper-Scissor game that could be considered finished.

At this point we can push our changes and new files to git, this means that even when something would happen to your device you could just clone the repository remotely and have the project again.



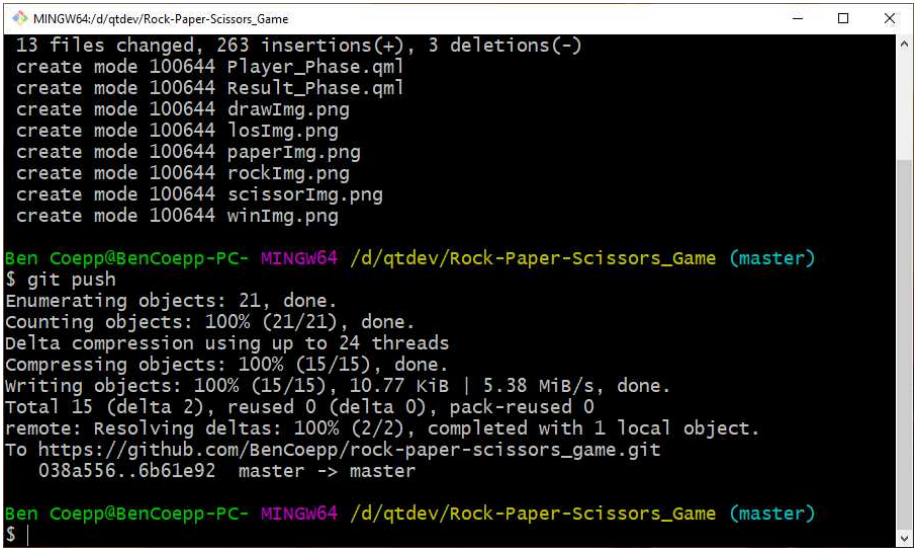
The first command we need to run is *git add .* , this will add up all the files and changes we did so far. If you want a full explanation of what git add . does you should read up on the chapter 3.2.3 there you will find a bit more information about Git, but the best place to learn about Git is the internet. I do not know a lot about Git, just enough to work with it and to not course any unnecessary problems.

There are also a lot of really good book about Git out there, so if you are more the type for books then feel free to get one of those as they are a great learning tool to understand Git a lot better.



The next command we ca run is *git commit -m””*. This command takes all the added files and puts them in a commit with a commit massage to it.

As you can see here, I did not do a great job adding a great commit message to it. And I would even go so far as to call this a terrible commit massage. Generally, you want to say extremely specific what you did in your commit, which files you touched and what changes you made. This makes it easy for people to understand what you did, but this is not the case in this example.



Lastly, we can run the *git push* command. This will the push all our changes to Git Hub. And with that we are done with the basic creating of our application.

We now have a finished and functional app, that looks terrible and has a terrible project structure, but it is functional. You can run it now and see for yourself, but I will not accept mediocrity, and I want something a little bit nicer.

– Fixing the mess –

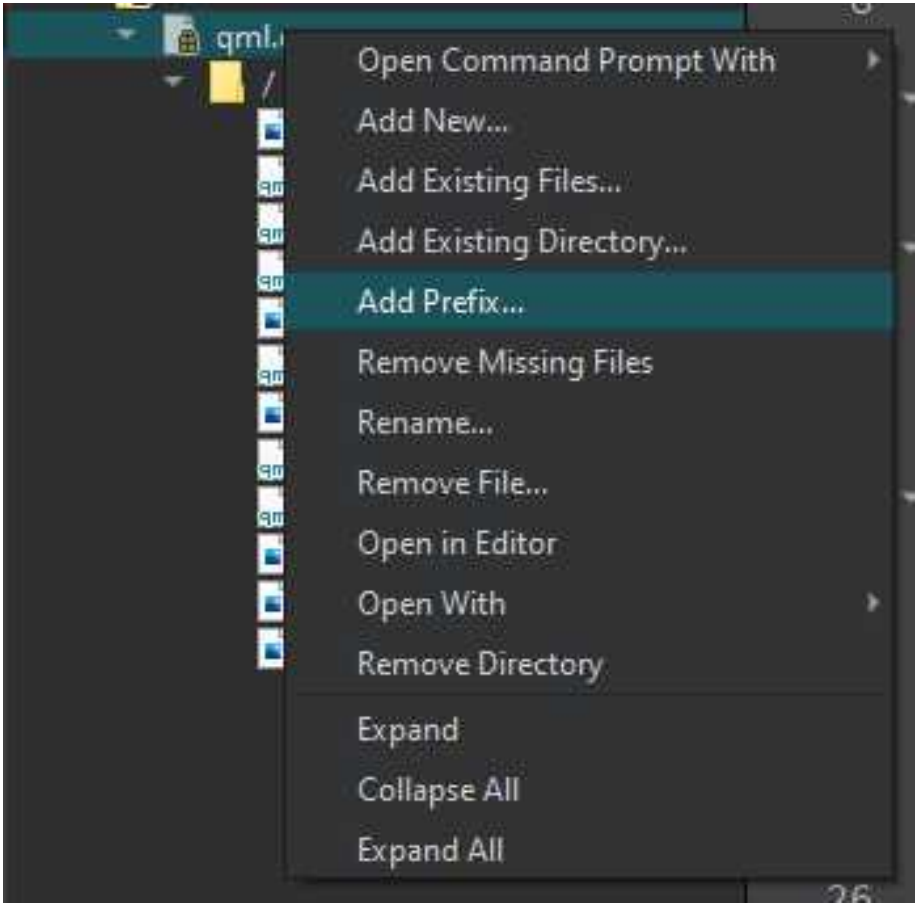
As you probably can see right now, the project has no structure, some functions like the winCheck function are needlessly long, and even worse, some parts of the application look downright horrendous.

And this was intentional, we always try to write and build great applications with great project structure and good written code, but that is not always the case, or our co-workers have done a terrible job with that and so there will come the time where you

need to polish an already existing project so that it actually presentable. And that is what we are going to do next.

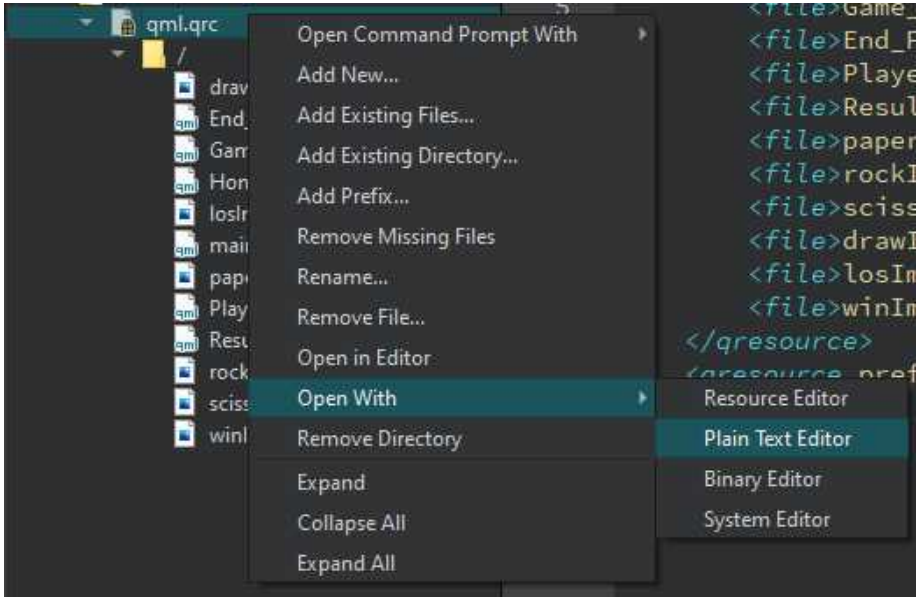
I know I probably caught you of guard with this, and this chapter is somewhat optional, so if you do not want to do this, then you can also skip this to the next chapter where we implement the local storage system.

The first thing we should have a look at is the separation of images into a new prefix. This should be done for two reasons, first of it is far easier to read this then normally, and secondly it gives our project a lot more structure.



The first thing we need to do for that is adding a new Prefix to our Project. You can right click qml.qrc and select the Add Prefix option.

This will open up a wizard that will ask you for the name of the prefix you want to create as well as the language. Or us we only need to fill out the name. I choose Images as the name for the prefix, but you can choose whatever you want. Click ok when you are done.



Opening the Plain Text Editor

When you finished this, you can right click again on the qml.qrc folder and open up the submenu Open With. Here you need to

select Plain Text Editor. This we did not have any look at previously so listen up.

The other options you can see here in the submenu are also ways of opening up and editing the project structure, the one we used previously was the Resource Editor itself. But sometimes the best option is editing the plain text of the file. So, open it up in Plain Text Editor.

```
<RCC>
    <qresource prefix="/">
        <file>main.qml</file>
        <file>Home_Page.qml</file>
        <file>Game_Page.qml</file>
        <file>End_Page.qml</file>
        <file>Player_Phase.qml</file>
        <file>Result_Phase.qml</file>
        <file>paperImg.png</file>
        <file>rockImg.png</file>
        <file>scissorImg.png</file>
        <file>drawImg.png</file>
        <file>losImg.png</file>
        <file>winImg.png</file>
    </qresource>
    <qresource prefix="/Imgages"/>
</RCC>
```

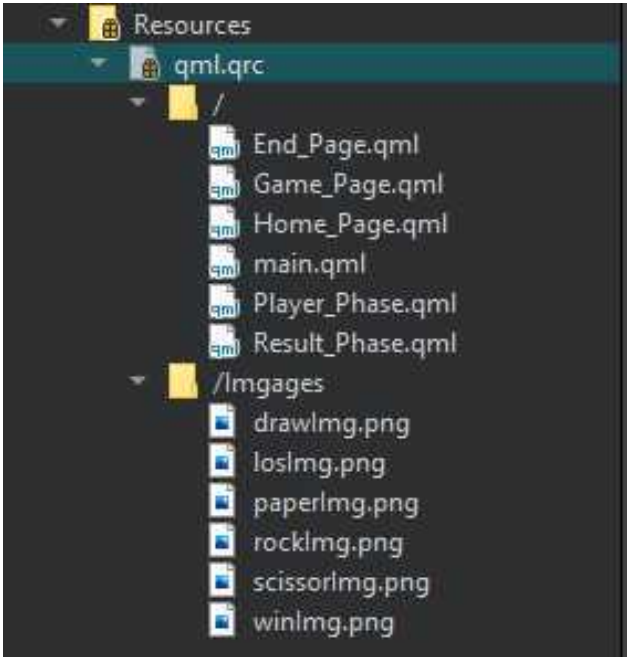
Plain Text Editor content

If you open it up it will look like this. We have all our files and images in the empty prefix up top and our newly created prefix just sitting empty down below.

```
<RCC>
    <qresource prefix="/">
        <file>main.qml</file>
        <file>Home_Page.qml</file>
        <file>Game_Page.qml</file>
        <file>End_Page.qml</file>
        <file>Player_Phase.qml</file>
        <file>Result_Phase.qml</file>
    </qresource>
    <qresource prefix="/Imgages">
        <file>paperImg.png</file>
        <file>rockImg.png</file>
        <file>scissorImg.png</file>
        <file>drawImg.png</file>
        <file>losImg.png</file>
        <file>winImg.png</file>
    </qresource>
</RCC>
```

Update resource in Plain Text Editor

The fastest way to change the project structure in my opinion is just rewriting the file a little like you see in the screenshot above. We moved all the images down into the Images Prefix, and when you hit save you will see this.



As you can see, there is an immediate improvement to our project structure. All the images are nicely located and order away from all other files and it does not look so cluttered all together.

```
34          model: ListModel{
35              id: optionModel
36              ListElement{
37                  img: "qrc:/Imgages/rockImg.png"
38                  value: "rock"
39              }
40              ListElement{
41                  img: "qrc:/Imgages/paperImg.png"
42                  value: "paper"
43              }
44              ListElement{
45                  img: "qrc:/Imgages/scissorImg.png"
46                  value: "scissor"
47              }
48          }
```

But if you were to run the application now, you would get the error that the images are missing or not properly defined. And

this is because we changed the URL of the images. And to fix this we need to add the new URL to the corresponding place.

For the optionModel you can see the correct URLs in the screenshot above, for the Result_Phase ones you can have a look at the screenshots below.

```
16        Image {
17            anchors.fill: parent
18            antialiasing: true
19            source: if(winState===0){"qrc:/Imgages/drawImg.png"}
20                    else if(winState===1){"qrc:/Imgages/winImg.png"}
21                    else if(winState===2){"qrc:/Imgages/losImg.png"}
22            onSourceChanged: {
23                resultTimer.start()
24            }
25        }
```

Now that we are done with this, you can see a great improvement already, but let us not stop here. You might think that we can shrink the winCheck function, but unfortunately this is not really possible, mainly because of the many combinations we already have when we only have three options the player and bot can choose from.

But something we can improve a little and work on, the visual fidelity of our application. And the first thing we can touch is on our Home_Page.qml.

```
30                text: "Start Game"
31                font.bold: true
32                font.pointSize: 25
```

Before we only had the text property, so the text was very plain and not that great. The best thing we can do to immediately improve the product is by simply making the text bold and making it a bit bigger.

This greatly improves readability and makes it a bit more visually stunning. The next thing we can touch on is also in this file.

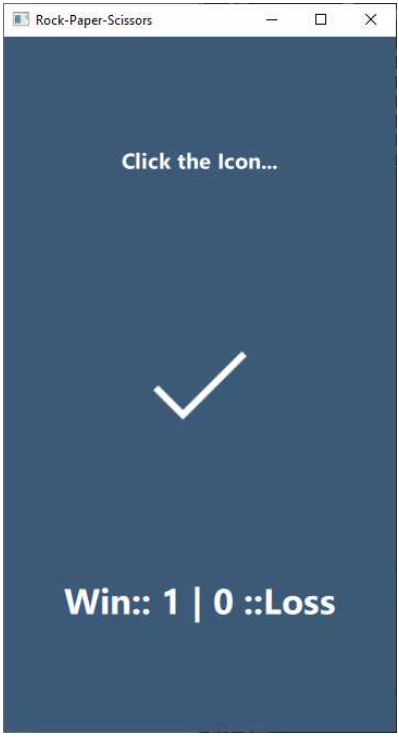
```
43      Label{
44          anchors.bottom: parent.bottom
45          anchors.bottomMargin: 50
```

This is also a fairly quick fix, basically we add a margin to the bottom anchor, this means that the label that displays the win and los of our player is a little bit better presented. A quite simple and quick fix, but it improves the product at show.

The next thing we can add to improve the application is a label above our Mouse Area on our Result_Phase.qml. Currently there is nothing that tells the player that he should click the icon that is displayed here. Only if the player knows what he needs to do, or if he throws luck presses the button, can the player figure it out.

```
4  Item {
5
6      Label{
7          anchors.top: parent.top
8          anchors.topMargin: 100
9          anchors.horizontalCenter: parent.horizontalCenter
10         color: "white"
11         font.bold: true
12         font.pointSize: 15
13         text: "Click the Icon..."
14     }
```

Therefore, we can just add a label above our Mouse Area, centre it to the top of our page and give it a text that tells the player what to do. This is also a fairly quick fix but improves the understanding of the player a lot.



Maybe as a side note, but there are a lot of ways to tell the user what to do, wone is throw the use of text like we just did, this is the easiest and most understandable and the one that users are most familiar with. But you can also use color, or composition of elements to guide the user into doing actions or moving throw the application the way you want. But they all rely on your understanding how the user is going to operate. If you are interested in a subject like this, there are a lot of great topics and papers on the matter.

But this is a topic we are not going to discuss too much in this game, as I believe that this information can also be learned elsewhere.

Now the last thing I want to improve is adding the check image we imported earlier to the End_Page.qml. currently the End_Page.qml does not look that great and a little empty. To change that an image is a great solution.

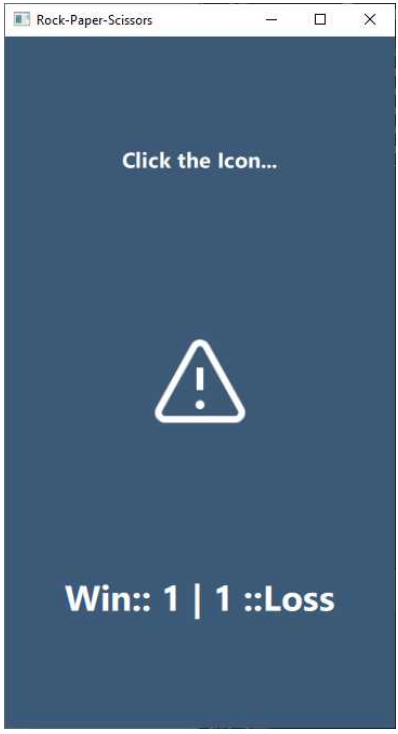
```
24      Image {
25          anchors.centerIn: parent
26          width: 200
27          height: 200
28          antialiasing: true
29          source: "qrc:/Imgages/winImg.png"
30      }
```

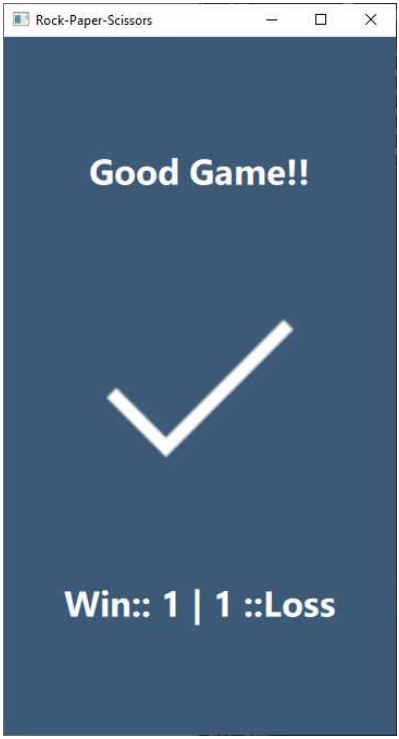
You can just add this Image below our Label we created just a few pages back. For the attributes, the I

mage has, well they are the standard attributes you would suspect, like width and height as well as a source and a position. The antialiasing attribute I just added because I think that sometimes images do not render that great on mobile devices and you need to help them a little.

These were some of the improvements that I think are good and should be implemented into our project. You can also add a lot more, add a bit more flair and visual fidelity to the application but for the purpose of teaching you how to build and make Qt application I think this I enough. And below you can see our finished results.


Rock-Paper-Scissors
Stop Game


Rock-Paper-Scissors
Click the Icon...
Win:: 1 | 1 ::Loss

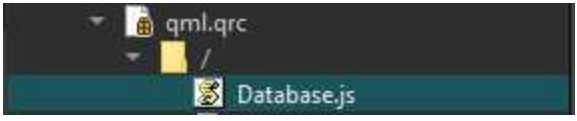

Rock-Paper-Scissors
Good Game!!
Win:: 1 | 1 ::Loss

– Adding Local Storage –

As I already mentioned we want to implement a local storage solution in this application. The applications we build so far were all without storing or saving any data to databases or files. This is fine for most applications, but sometimes you want to save data, or have it in multiple locations at ones. And for that purpose, you would want to use a database.

If you want to know more about databases you can check the Index where to find the chapter later on where I talk about them. But for now, let us build ourselves a local storage solution.

First of you can go ahead and create a JavaScript file. This can be done the exact same way you would create a qml file for instance. You should give the file Database as its name, and when you are done with that you can create the file.



New JavaScript file

As a side note, the feature we are trying to use here is called local storage, it is a feature provided by Qt, that enables us to create an SQLite Database locally on the device we are running our application on.

I love using this, you have your own local database, therefore it is extremely fast. And even more importantly you can create all of your functionality without a remote databases making development a lot easier.

```
function dbInit()
{
    var db = LocalStorage.openDatabaseSync("Database", "", "Data Table", 1000000)
    try {
        db.transaction(function (tx) {
            tx.executeSql('CREATE TABLE IF NOT EXISTS data_table (data_text, data_value)')
        })
    } catch (err) {
        console.log("Error creating table in database: " + err)
    };
}
```

dbInit function

The first function we can create in our Database.js is the dbInit function. As we want to work with a database throw out our application, we need to initialise it somewhere, and this will be done in this function.

First of we create a variable called db, then we initialise this db variable with opening a database sync throw LocalStorage. The values you see inside the brackets are necessary so fill them out. The only ones really important are the name in the front, and the length at the back. For most applications, the length is not really important as you will probably not create so many thousands of items.

Below the declaration of the db variable we have a try-catch block. This checks if the database exists, and if it does not then it creates the databases with the correct data fields. If it is not able to create the database or initialise the database connection it will give us an error.

```
function dbGetHandle()
{
    try {
        var db = LocalStorage.openDatabaseSync("Database", "",
                                               "Data Table", 1000000)
    } catch (err) {
        console.log("Error opening database: " + err)
    }
    return db
}
```

dbGetHandle function

Next function we need is our dbGetHandel. This function only exists so we can save a little bit of code in other functions we

will create later. In dbGetHandel we only have a try-catch block that tries to open a database connection. This is the same openDatabaseSync function we had above so you can copy it down. The catch again prints out an error when it occurs.

Now we can look at the dbSet function, this function creates a new Item in our table. The function starts by us giving it two variables, data_text and data_value. Whenever we want to call dbSet we need to give it these to values.

```
24  function dbSet(data_text, data_value){
25      var db = dbGetHandle()
26      var rowid = 0;
27      db.transaction(function (tx) {
28          tx.executeSql('INSERT INTO data_table VALUES(?, ?)',
29                        [data_text, data_value])
30      })
31  }
```

dbSet function

Next, we call the dbGetHandel function, this opens up our database connection if the database exists. Next, we call db.transaction, this will open a transaction, in this transaction we can now execute our sql.

The sql we have here is considerably basic. Basically, we have a prepared statement that inserts a new item into data_table our table with the values data_text and data_value. It is a quite simple sql statement but it does its job and if you know anything about sql you probably wrote this a million times already.

The dbGet function is the next in line, as the name suggests it is used to get an items value. As before our function starts with initialising db with our database connection using the dbGetHandle. Also, we need to give our dbGet function data_text as a property so we can interact with it later.

```
function dbGet(data_text){
    var db = dbGetHandle()
    var rowid = 0;
    db.transaction(function (tx) {
        var result = tx.executeSql('SELECT data_value FROM data_table WHERE data_text="'+data_text+'"')
        rowid = result.rows.item(0).data_value
        console.log(result.rows.item(0).data_value)
    })
    return rowid;
}
```

dbGet function

Next, we create a new variable called rowid, this will alter be used to return out value we selected.

The actual sql we need to run is again fairly simple, it starts by us opening another transaction and then executing our select inside of that transaction. The sql we execute is a simple SELECT that grabs the data_value from data_table where the data_text is equal to what we provide the function with. As you can see here, we do execute the sql in the standard way, but putting it into a variable this is commonly known as a result set. Now this result set holds an object of our data. This is not any good as with just an object there is not that much to do, so we need to get the data_value out of our object. So, we call our variable rowid and set it to result.rows.item(0).data_value, this will give you the data_value.

If you ask me how I found out what I needed to call to get the data out of the object, well then, I can tell you that I just used the standard local storage example as the code template. And I took a lot of the functionality from there, and I would advise you to do this too. The local storage example from Qt is extremely well structured and great to read throw. So, give it a look.

Next in the function I printed out the result in a console.log, this is not necessary, and I only did this for testing purposes so you could leave it out if you want to. Lastly you need to write *return rowid* after the function. This is a basic SELECT using sql on a table, but it does the job for us so it is completely fine.

```
function dbUpdate(data_text, data_value)
{
    var db = dbGetHandle()
    db.transaction(function (tx) {
        tx.executeSql(
                    'update data_table set data_text=?, data_value=?', [data_text, data_value])
    })
}
```

dbUpdate

The dbUpdate function is next. The only reason we are going to write this, is because I want to show it to you and because it is also a great way to write the functionality we want.

Basically, it is the same function as the dbSet, just that we do not use INSERT as our sql statement but UPDATE. This is again a prepared statement as before, and here it works by looking if it can find the data_text we want to update, but if it does not find it will just create it, and if it finds it will update it to the new value.

Now with the dbUpdate function writing we have all the functions we need for our application so let us implement them.

```
import QtQuick 2.15
import QtQuick.Controls 2.12
import QtQuick.LocalStorage 2.0
import "Database.js" as Database
```

New added imports

The first thing we need to do is add two new imports into main.qml. The first new import is QtQuick.LocalStorage. This is needed because our JavaScript file requires this for the modules and functions that you get from it. The second import is just our Database. To better use our Database inside our main.qml, we need to set an alias to it. Here I have gone with Database, as that is what we are working with. But you can choose another alias if that suits you.

```
37      Component.onCompleted: {
38          Database.dbInit()
39      }
```

The next thing we need to add on main.qml is the dbInit(). We need this to be able to interact with the local database, if you are not sure what I mean by this go back a few pages, there you can find a more thorough explanation.

```
12      property var winCount: Database.dbGet("playerWin")
13      property var losCount: Database.dbGet("playerLos")
```

And lastly, we need to change the winCount and the losCount, before they were just integers, which was fine if we did not want to connect to a database, but here we now need to call our database with its alias and call the function dbGet() with the correct data_text. When the application now starts up the database will open up and the data for these to property is being loaded.

This is in my opinion the best way to work with databases in Qt and it works. There are other more elegant solutions, like wrapping this Database.js file in a component and only having the component interact with the application, and that would be the correct way of doing it when you want to make it professional. But for our purposes this is not necessary and we achieved our goal of getting data from the local database.

The only problem is that currently there is no data attached to these data_texts. This is because we did not create the data. To do this, we first of need to go to our Player_Phase.qml file and change a few lines.

The first thing we need to change is again the imports, there are here the exact same as the ones in our main.qml. so, you might just want to copy them over. If you think that this is not necessary to add these imports at this point, because you think

we already did this in the main.qml and that has all the other files linked to it, then you are mistaken.

The current structure we have does not allow us to simply have the imports and components in one file and use them from there, for that we would need a better project structure. None the less, we need to add the imports and then we can continue.

```
1   import QtQuick 2.15
2   import QtQuick.Controls 2.12
3   import QtQuick.LocalStorage 2.0
4   import "Database.js" as Database
```

After that we can also add the Database.dbInit to our Component.onCompleted event. We already used this, but to give you a refresher what it does, whenever the Item component at the top is being loaded and rendered the function inside of this even is being called. It is a wonderful option to call functions right at beginning of the application.

```
112     Component.onCompleted: {
113         Database.dbInit()
114     }
```

Lastly, we can add the set and update function to our onClicked event of our application. I used both the updated and set function here, you can only use one, or both but basically here they do the same, they take our win- or losCount and add them to our data_text and then add them to the database.

As you can see with two lines of code you can add data to our database. And this is exactly the power behind the local storage component of Qt. it makes it extremely simple to write a basic database and use it in your application, you can interact with it extremely easy and change all the things you want.

```
105         console.log(winState)
106         gameFrame.setCurrentIndex(1)
107         //upload to local storage
108         Database.dbUpdate("playerWin", winCount)
109         Database.dbSet("playerLos", losCount)
```

And with that we are done with writing and creating our local storage solution. If you were to run our application now, you will see that when you start the application, the win- and losCount are null. This will immediately change when you get a los or a win throw the application.

And if you were to know close the application and start it back up again, you will see that the data is loaded and the correct value displayed. And with that we are done creating our local storage.

Yes, some of you will now jump up and say that this is not near enough of what you need to learn to work with databases, and they are correct. You will need a lot more experience and actual practise with working on databases. What I provided, was an overly simplistic and broken-down version of how to get started, and now you are able to start learning more about databases, how to interact and manage them and how to best implement them in your applications.

Next, we are going to deploy our application to a mobile device so that it works on there too.

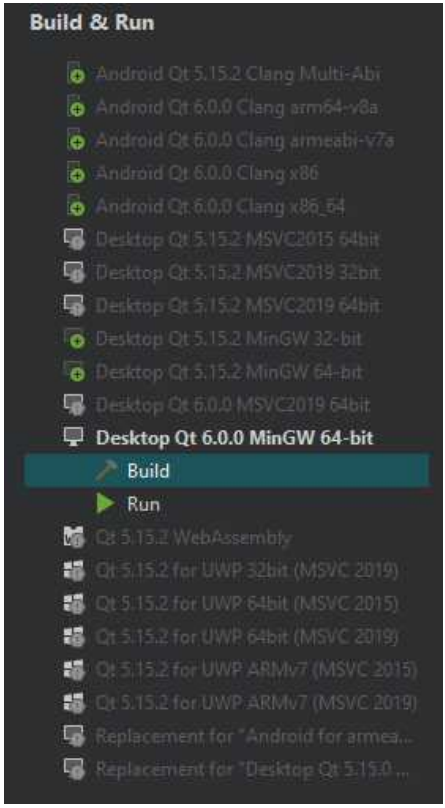
- Deploy Application to Android -

We already did deploy an application to a mobile device in the last project we did, and this is not going to be so different. But why are we doing it then anyway? Well because I want to teach you a little bit.

more about how to do this, and because I want to show you that the local storage also works on mobile. But we are not going to go over all the thing we need to do to make the deployment to work. What my main focus will be this time around on how you would best set this up if it were a production release we were doing.

If you want a little bit more context and explanation what we are doing next you can go to the end of the last project we did, the Hang-Man project and have a look at the chapter where we deploy the application to android.

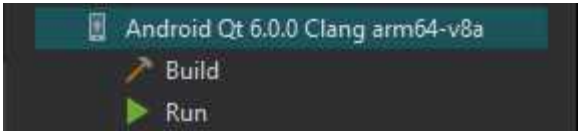
We first of start by opening up the project tap at the left of the screen.



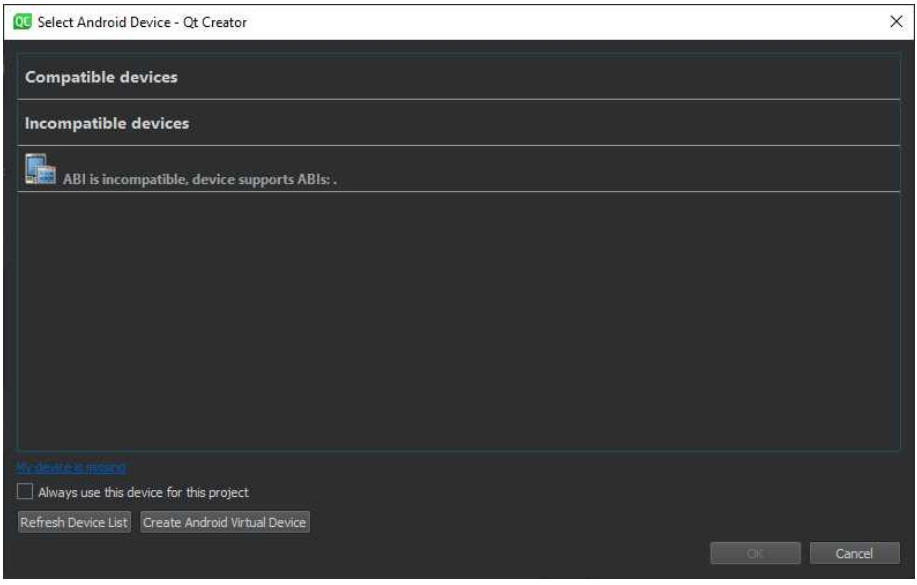
Here you can see the lit of different build and run kits Qt offers us. If you only selected MinGW 64-bit as I did, all other kits are currently disabled. The only kits that are important to us you can see at the top of the list. These are the different android kits you can build for. Why are there so many different kits, you might ask. Well, the Android Operating-System open-source. That means anyone can use and build their own version of android and that is exactly what people did. You could even do this yourself if you have the time and dedication for it.

And combined with the simple fact that there are different chipsets and producers of android phones you are left with a lot of different kits we need to account for.

But what does that leave us with, well we do not need to develop for all kits out there. In development you should only focus on your device you want to run the application on for development. Later on, you can focus on all other, which will then be not too hard as the different android version are not so different and most applications do not really require any changes or special configuration to run properly on the different versions of android. But enough about android let us continue with the deployment.



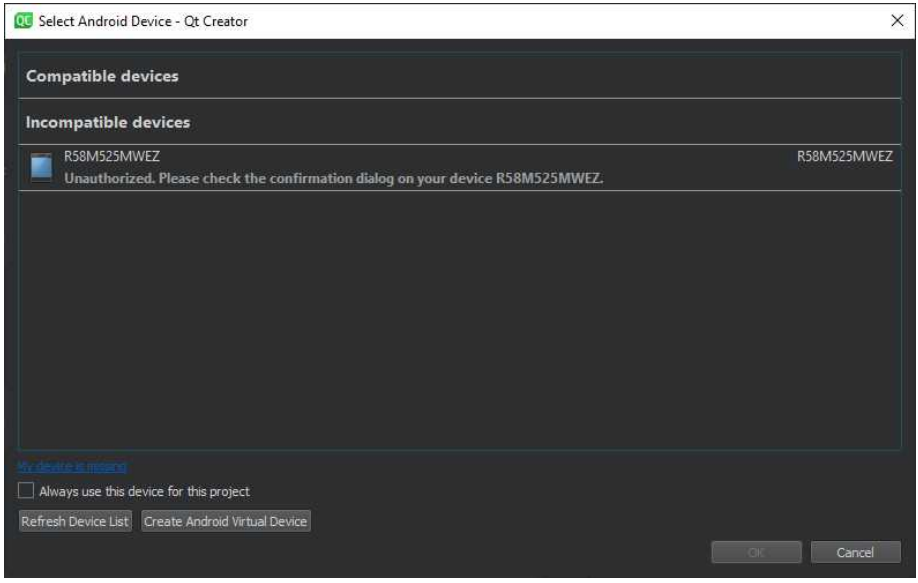
I select the arm64-v8a as my kit, you should choose the kit that corresponds to your own device. If you do not know what kit you need to use, the best way is to run the application with any of the different kits enabled.



This will open up this wizard, this is used to select the android device to run our application on. Currently we do not have any device connected to my machine so nothing is shown, so let us connect a device to our machine. This should be done with a USB-Cable as that is the best way I know of.

If your device is now connected to your Machine you can refresh the list with the button on the left.

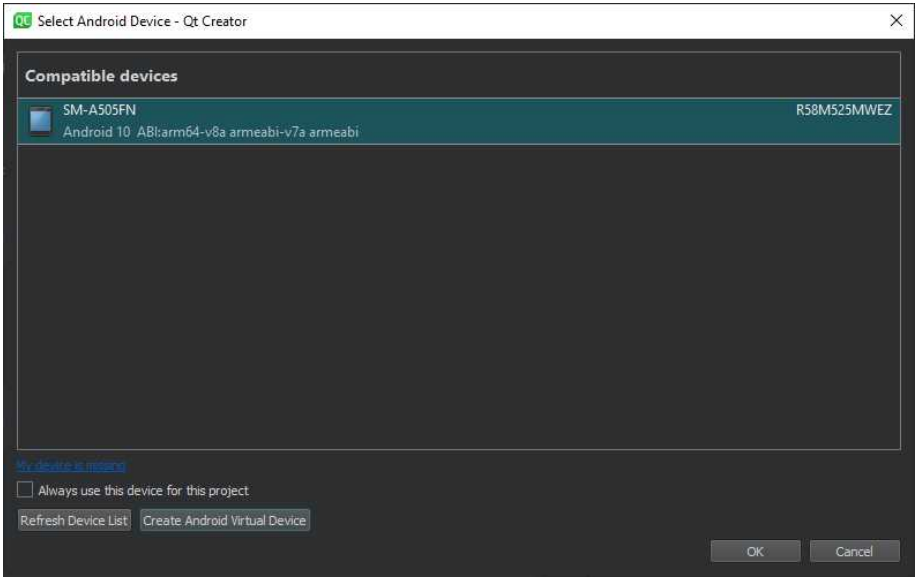
Your device should now be visible on screen. It might happen that the device is shown as being incompatible, but this probably just means that you need to check a dialog box on your device to allows access to it.



If you do not get this message or the dialog you need to figure out how to get your device into development mode. This is different for all device, generally you need to go to the Device Setting and click the version number a bunch of times, and then you can enable developer mode.

If you did all this you can refresh the device list again and the device should now be compatible. If that is still not the case, then you need to start one of the other kits Qt has to offer and see if that one works, but normally Qt will tell you wish kit to use next to the device. So, you only need to run it with that kit enabled. For me this is as I already said arm64-v8a.

You can also see the current android version as well as other kits that would work with it here. When you have selected the correct kit, you will see this wizard down below.



When you can see the page above you can click ok. This will start building your application and deploying it to your device when finished.

This can take upwards of a few minutes depending on your machine. If everything is working as expected you will see your application popping open on your device and you can now start playing with it. You can also look at the Compile Output, which sometimes can be quite interesting. To look at it, you need to go the bar at the bottom of our Qt Creator window and click on the Compile Output tap.

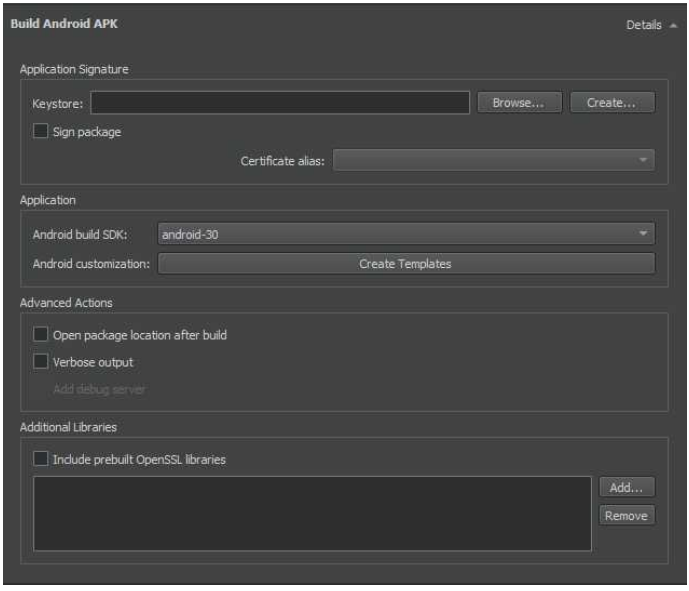
```
Android package built successfully in 35.052 ms.
  -- It can now be run from the selected device/emulator.
  -- File: D:/qtDev/build-Rock-Paper-Scissors_Game-Android_Qt_6_0_0_Clang_arm64_v8a-Debug/andr
Warning: QML import could not be resolved in any of the import paths: QtQuick.Controls.Windows
Warning: QML import could not be resolved in any of the import paths: QtQuick.Controls.macOS
```

When you see this, you know that the application was build successful and everything worked as it should. This is also the case when the application lunches on your device.

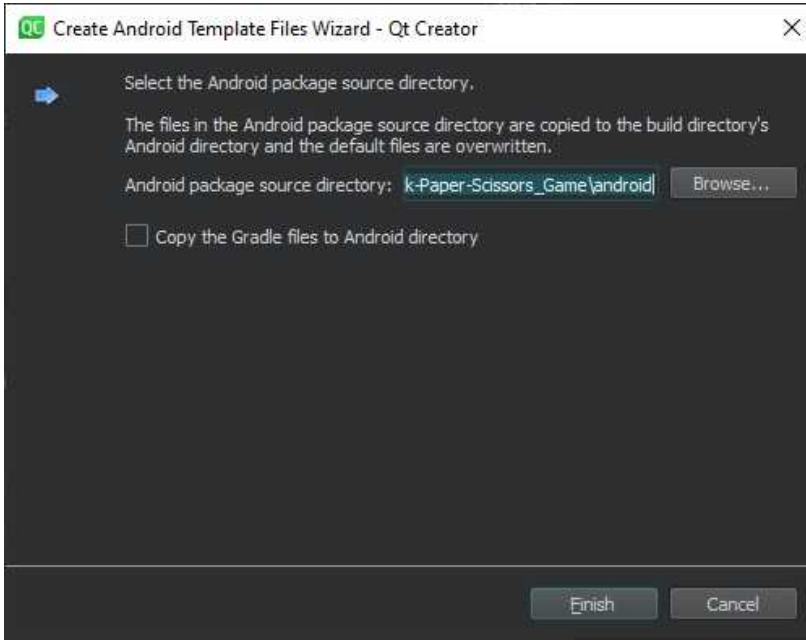
Now you can say that we are done with deploying our application.

But I want to show and talk with you about a few key features in the template you can create for your android builds. We already created one the last time we build an application for android so first of let us do that again.

First of we need to go back to our Projects tap on the left of Qt Creator and there you need to open up the build setting for your kit. Next you need to open up the Build Android APK drop down.



This will be presented to us, we do not need to sign our application this time as we did this the last time and it would not be too different. But we are again clicking on the Create Template button in the middle.



The wizard that opens up can be more or less ignored again, it does not hold anything we really need at the moment. So, click finish and let us continue.

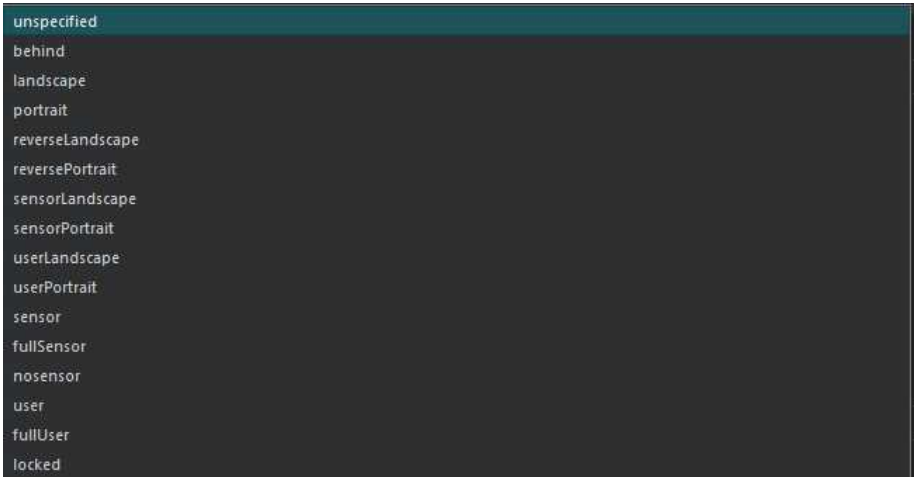
This can take a moment but Qt will now add the Android Manifest and other necessary files to your project and then open the Android Manifesto for you. If that happens, we can have a look at it. I need to preface this here, but we are not going to fill everything out again. We already did this last time and it is not really necessary for us. I just want to talk about two features that are important to me.

The first one is the Style extraction and Screen orientation.

I already talked about them in length the first time we had a look at the android manifesto, but I want to add a few things and remind you of a few more.

The style extractions and screen orientation are as far as I am aware of a new addition to the way Qt handles the Android Manifesto. And this is a good thing, but it might confuse you a little of what all the features do and you can use them for. And I recommend to you know that you should leave your hands of them for the most part.

First of the style extractions. They are good and all and they have their use case, but you should leave it most of times at the default. Next is the screen orientation. And this is something I can see people use quit a lot.



Here you can see a list of all the currently available screen orientation. This can be very overwhelming for a new user, but all it does is tell android what orientation the screen should be. This means for instance that when you select portrait here, the application will not rotate or turn when you tilt your phone into landscape mode.

This can be extremely helpful and great depending on your application you want to build and is in my opinion the best way to get this behaviour. So, it is a really nice addition to the functionality of building android applications with Qt.

And lastly, we have the splash screen. This is also a new addition to Qt for android as far as I am aware of and is an extremely helpful new feature. Before we needed to build complicated loading and Load- and Main-Page setups to build applications we want. And I showed you how to this, because on desktop you will need it so you can display a loading screen.

But for android we have this now. And I am really grateful for it. It makes building applications with android so much easier and faster and know all your applications will look far more professional and well realised as before. Just be sure to use a good high-quality image as the splash screen and you are good to go.

These were all the features I wanted to talk about with you of the Android Manifesto. I hope you found this short little deep dive into my thoughts about these features interesting if so, I appreciate it, but if not, I hope I did not give you too much information right out of the gate.

And with that we are more or less done with deploying our application to android. If you want to learn more about this, go check out the Qt Docs on the matter and maybe read the project Hang-Man again to figure out how to do this in a little bit more detail.

– What did we learn –

Well, we certainly learned a lot of different thing in this chapter, and I hope you enjoyed it. I know we only skimped over some stuff, but none the less I think that we all were able to learn something new and important that helps us understand how Qt work and how we can best use it.

So, what did we learn in this project, let us have a look?

- **Qt Local Storage**

  First off, we learned about Qt Local Storage. This is a local SQLite database that Qt has under its hood. It works like any SQLite database, you can execute your standard array of sql statements and interact with the database and that nicely from the comfort of JavaScript and Qml. As it is local, it is extremely fast, reliable and works extremely well with any model or component that you want.
  In my opinion it is the perfect way to build and manage local data and your databases needs, and I used it extensively over the years. I only gave you a brief interdiction of how this works and how you can interact with it, but I still hope that this can bring you in the right way to learning about databases in Qt.

  Also, as side note, if you want to work with remote databases, you will need to use C++. It is not too hard to set up and works remarkably similar to other languages and frameworks out there, unfortunately I will not have the time here to explain how to do it, and it would not fit the focus of this book as a beginners guide to Qt.

- **SQL in Qt**

This is in the same category as the first thing we learned, we only briefly and very faintly touched on the subject of SQL in Qt and I only explained so much that you understand what we are doing. It would not make sense to explain more than this here. But if you want to learn more about how to work with SQL, the different statements, or things you can do with it you should read another book or watch a video that is especially about the topic.
But what we learned was, how to write an INSERT, SELECT, and UPDATE and how to write each in a prepared statement. If you know a bit more about SQL then you understand that these are the most fundamental command you have in SQL, but for all others this is the perfect starting point with which you can do a whole lot and that should be quite enough for your first few applications.

- **JavaScript functions in external files**

  This is something that seems a little trivial but is especially important if you were to build larger applications. Always writing JavaScript functions inside of your QML files will not be always possible. First of it is extremely hard to read and understand if you have a fairly large code base, and it leads to misunderstanding and confusion when you are not careful.
  I used them only here, as they are not always necessary to begin with, but here they were perfect to minimise the code we would need to write double or triple and they show how you would decrease your code footprint substantially.
  I hope you learned quit a lot in this chapter and that the lessons and tools I showed you made it possible for you

to work now on your own applications that require databases or storing data locally. You can always refer back to this chapter if you need something of an example, and if you want to learn more about local storage, check out the local storage example and documentation provided by Qt. They are an extremely good read for anyone that want to know more about Qt and how it works, and they probably will help you understand the concept better and implement it in your applications.

3 Components, Features and Things to remember

Because we are done with three applications we wanted to build, there is not a lot we need to do now. The only few things left for me is one teaching you about the basic and most used components in Qt, some features which you will need to know about all the time, and lastly a few things which are essential for building applications now adays.

This includes Databases, JSON Git and other topics. These were all not essential for us before, but they are heavily used in development in nearly every project. And knowing at least a little bit about them will help you a lot along the way.

I will not teach you all the ins and outs of these topics and the components in the next few chapters. I will only scratch the surface of what all these subjects and components can do and or mean for a developer. The main point is to give you a little bit information about everything so you have an easier time starting to learn about them.

3.1 Components

A word before we start with the components, I do not intend to explain the components, throw explanations can be found in the Qt Docs and the internet in general. What I will do is first of explain the component in a quick fashion, and then give an example followed by me use case and a few tips and tricks of how-to best use said component.

So, this is only a quick rundown of the fundamentals of the component and what belongs to it. Still, this is a really great place to read up on components you want to use and or if you have problems setting up the component the way you want it.

3.1.1 ListView

List Views are backbone of really any application you wish to make. No matter the size, the experience, and the scope of the project, you will use List Views. They are essential in providing and displaying data. And there is no real way around them.

The two main things a List Views consists out of is a delegate and a model. Both components will be discussed later. The List View itself is the link between them, it takes the data that the model holds and pushes it throw the delegate, outcomes an item that displays the data you wanted.

List Views are extremely versatile, and you will always find a use case and something to do with them. For instance, you will use them creating lists, as you would expect, like tasks or time tracking. They can be aligned horizontally and vertical, you can change the direction and the transitions of all the items inside

the List View. As you can see there are a lot of things, but what are the main ones?

- **A lot of customisation throws a lot of attributes**

If you have a look at the documentation of the Qt List View on Qt Docs, you will find that it has a lot more attributes, signals, and methods then other components. And that with good reason. As a lot of people use the List View in all kinds of ways it is important that there are a lot of option to customise and control how the List View works. This mean that there are less building custom components to supplement a List View, and more a tinkering with the attributes to achieve certain visual components.

Also, if you followed along to the projects, we did so far that there were a lot of different attributes we used in conjunction with the List View.

- **Interconnectivity**

Data and visualising this data are the main point of a List View. And the nice thing about how much and what types of data you can use. As we will discuss later with the model, Qt provides a variety of data types, and the List View works with all of them easy and fast.

There are also a lot of custom build models, like the JSONModel or even custom List Views, that provide a totally different way of getting data and displaying it.

- **Fast build time**

The standard List View is built in a matter of seconds. And I do not mean that as an understatement, the basic elements you need to build a List View are a width and height, and model and delegate. These are all the elements you need. There might be a lot more options you have and even more

attributes you can tinker around with to achieve a perfect List View for your type of application, but generally you do not need much to build a functional List View.

Overall, the main purpose of the List View is providing an easy, reliable, and fast way to build, manage and most importantly display data. It is one of the if not the most essential component in QML that you are bound to use all the time. I best recommend you trying out the projects we did, there we are using List Views in a variety of different ways.

Also important is the fact that there are two more views in Qt, the Grid View, and the Path View. They are fundamentally identical to the List View, the only difference is that the Grid View is structured like a grid, so there is no difference otherwise, just that the elements are in a grid fashion, the Path View is the same thing, as there the elements are displayed on a path. You should read up on all versions if you have the time for it, as it might be better sometimes to use a specific version over the others, but because they are functionally the same you will not need to learn completely new components workings, and how to use it.

```
4  ListView {
5      width: 300
6      height: 300
7      model: ListModel {
8          ListElement {
9              name: "Bill Smith"
10             number: "555 3264"
11         }
12         ListElement {
13             name: "John Brown"
14             number: "555 8426"
15         }
16         ListElement {
17             name: "Sam Wise"
18             number: "555 0473"
19         }
20     }
21     delegate: Text {
22         text: name + ": " + number
23     }
24 }
```

Simple List View with Model and Data

This is the most basic and standard List View you can think of. For a List View to function you need only a few things. A width and a height or a property, a model, and a delegate.

```
7     model: ListModel {
8         ListElement {
9             name: "Bill Smith"
10            number: "555 3264"
11        }
12        ListElement {
13            name: "John Brown"
14            number: "555 8426"
15        }
16        ListElement {
17            name: "Sam Wise"
18            number: "555 0473"
19        }
20    }
21    delegate: Text {
22        text: name + ": " + number
23    }
```

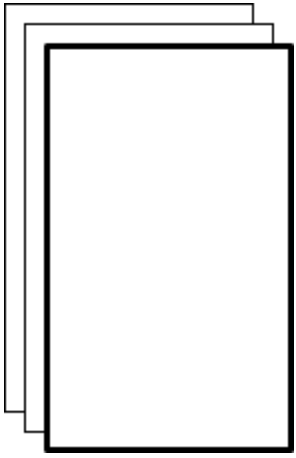
Closeup of model and delegate

The model and the delegate property are the most important parts of the List View. Without them there would not really be a List View. Basically, they act as the data for the List View and the way the data is going to be displayed.

We are going to talk about how the Model and the Delegate work later on, but in general this is how I would define them. You should probably read up on the chapter about Qml List View on the Qt Docs, there are a lot of great examples as well as insights in the documentation so check it out.

3.1.2 Stack View

Stack Views are the essential loading, navigation and displaying pages component in Qt. basically they function like a door, you have a single Component / Page displayed a single given time, and then you can line up all the other components you want to display next, and when you give the command the current displayed Item will be changed to the next one.



The best representation you will find on the Qt Docs, as you can see here it perfectly shows that you can only have one item currently on display and that behind that are a lot more items, which you can switch to when needed.

This is a real performant and reliable way of displaying and retrieving pages or components. It does not take too much space and memory from the user, and even more important it does not require a lot of code to build. The only few things you need, is a width and height, an id by which you can call your Stack View later to change the item displayed and lastly an initial item which will be the first to display. This simplicity allows for an exceedingly high speed to integrate and test this type of

loading pages and providing them for use, also when you display a page or component throw this Stack View you are also able to call onto the functions and ids that are inside said component or page, meaning you can basically interconnect and bind all your pages and components together.

The most common way I use the Stack View is for the Load and Main Page setup and navigation in my application, and for navigation throw a lot of different pages. Both are extremely widespread use cases for the Stack View. But there are also a lot more interesting use cases and unique things you can do with it. The best way to learn about the Stack View is by following the projects and tutorials in this book and trying it out on your own.

We do not really need an example here as we already build plenty of different Stack Views over the last few projects. But if you want a refresher about them, I suggest you check out the beginnings of the projects where we build the load and main page setup.

3.1.3 Swipe View

Left, Right, Up and Down these are all directions you would expect to be able to swipe in a mobile application. And that is exactly what this component allows you to do. In general, you can say that swiping is one of the best ways to interact on a mobile device.

You can switch views in and out of pages and drag navigations from the top or the bottom. They all allow you to make your entire application more interactable and easy to use.

The Swipe View is one of the components that work best in conjunction with other components. If you want to make a menu, that allows your users to switch between different views, then you might want to use a Swipe View. While it does not allow for complete control

over the swipe actions, but if you investigate what you can find on the Qt Docs you will see a lot of attributes then can allow you to create tight and interactive controls for your applications.

They are one of the most used components in Qt Quick, and if you want to build applications for the mobile device you will probably use a Swipe View. The number of things you can do with it are good, and we even build a few applications on our own with the Swipe View, for that see the Hang Man project.

But to give you a simple and precise example let me show you how a simple Swipe View looks like.

```
4  SwipeView{
5      id: swipeView
6      width: 720
7      height: 640
8
9      Item{
10         id: item1
11     }
12     Item{
13         id: item2
14     }
15     Item{
16         id: item3
17     }
18 }
```

Simple Swipe View

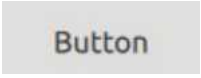
This is the most basic Swipe View I can think of. A basic Swipe View consists out of a width and height and an id to be precise. Next, we

have our Items inside of our Swipe View. And that is it, Swipe Views are a quite easy and simple way of adding multiple pages and page navigation to an application. Especially on mobile devices you want to have the ability to swipe on your device and for instance switch views or pages.

If you want to learn more about Swipe View, I suggest you should go to the Qt Docs and read up on them and go to the last few projects and check out how we used the Swipe View.

3.1.4 Buttons

What framework would be completed without a way to interact with your UI, and the best way and most widely expected way is using buttons.



There are a lot of ways you can interact with a button in Qt, and I will list some of the ways I use to interact with them below.

- **Clicked**

  Well, this is something that most people probably expected, you can click a button. And this will be most likely the most used way you interact with a button. It is easy to use quite easy to understand and it works.

- **Double clicked**

  This is also easy to grasp, when you click a button you can also double click it. The nice thing is that this functionality is already prebuild as a signal in Qt. otherwise you would need to manually build this, it is possible but not so easy to do.

- **Pressed and Released**

  Also, fairly easy to understand. You can press a button and release it. The main reason why you would want to have this is that you can differentiate between the press and the release of a button. So, if what you want to build is reliant on precise timing this is great.

- **Press and hold**

  As with the clicked signal one of the ones you will use so often. Basically, it tracks the time you press the button and when this time is bigger than a specific fresh hold a signal is triggered. Really great for a delete button or function.

- **Toggled**

  A little bit different then you night suspect but if you have a checkable button, this signal is emitted when the button is enabled or disabled. Really great for options.

  You can also style buttons in any way you want, from borders, to the background, you can add icons or anything you want to it. This allows for near limitless possibilities for creating your own styled buttons. There is also another button type, that you might want to use, the Round Button, it is in functionality the exact same as the normal button, but it has rounded corners. You could build your own Round Button by applying a rounded rectangle as the background component of the Round Button, but Qt provides you with a simpler and more elegant alternative.

The best way to experience what you can do with a button is by using it, so make yourself a project and try out whatever you want. I have built a small library of custom buttons with all types of styling and functionality. This is also a great thing on your portfolio if you want to make the same.

But I will not let you hang just there, so here are two examples for some simple buttons that you can build yourself. These are not really fancy or anything special but they are a good example what buttons can do.

```
9   Button{
10      id: button
11      text: "Simpel Button"
12      onClicked: {
13          console.log("You clicked a Button")
14      }
15  }
```

Simple Button with onClicked event

This is the simplest button you can have. You have a text property, this represents the text that is going to be displayed on the Button. And lastly you have the onClicked event. Without this a Button is pretty senseless so probably you will always have some sort of press, click, or hold event that triggers some functionality. And this is all you need for a simple button. It is nothing fancy or special but you will tend to use this quite a lot.

```
9   Button{
10      id: button
11      text: "Simpel Button"
12      background: Rectangle{
13          anchors.fill: parent
14          color: "red"
15      }
16      onClicked: {
17          console.log("You clicked a Button")
18      }
19  }
```

Simple Button with onClicked event and background element

Also, something important is the background property available for all Buttons. In most cases when you want to build something really unique or design heavy you will tend to run into limitations

with the normal Buttons. Mainly they are not flexible on their own for these kind of purposes.

But what you can do is use the background property and give your button a complete and custom background. This is really helpful and powerful and can make it a lot easier to build and design your application.

There is also another type of Button out there, the Round Button. But this button does not differ to much from the normal button, the only real difference is that it has rounded corners.

3.1.5 Mouse Area

The Mouse Area is to but it simply and honestly a button. It has the same functionality as the button and you can even build a button out from it, but it also has some functionality that is larger than that of a normal button. So, let us have a lock at what you can do with it. The Mouse Area can visually be explained as a rectangle that has all the functionality as a button and more on it, but this rectangle is transparent.

As with the normal button you have all the standards click, press, and hold signals but also some new ones. Mostly they revolve around the hover functionality you have in when you use a mouse. So, if you want to build anything very custom with hover functionality you will most likely use this as your starting point and start building whatever you want.

Another thing that you can do with this is drag and drop. Mouse Areas have built in drag functionality, so if you want to build anything with drag functions then you can do this right here.

And this brings me down to the main point what this Mouse Area even is, it is the perfect starting point for building your

custom controls and interactable content. Because it does not have any prebuild elements inside of it you can have complete freedom over how and what you want to build so try it out.

But to give you a quick and dirty example for a Mouse Area see below.

```
9     MouseArea{
10        id: mouseArea
11        width: 200
12        height: 200
13        onClicked: {
14            console.log("You clicked a Mouse Area ")
15        }
16    }
```

Simple Mouse Area

As I already said, the best way I would describe a Mouse Area is a transparent Button without any text. You can place everything inside the Mouse Area, like Images or more complicated design elements.

3.1.6 Text Field

Text Fields are one of, if not the most used component in Qt. you will use them all throughout development from the tiniest input to the largest application, they are essential for doing any kind of importing from normal data like strings integers dates and you cannot really go anyway other than that.

The component text field is not really that difficult to understand. generally, you can say the text fields attributes or properties can be divided into two categories one which is the visual attributes like for instance with height font size colour stuff like this, and the other are the more programmatic attributes like for instance you have the length of the input methods input masks different eco modes displaying text options. all these properties in

conjunction leave us with a good option of imputing different types of different forms of text. I would even go as far as to say that the text field is the one a most useful option and component for in putting any kind of data into your application. the visual properties you can set out of the kind like with hide colour you can set font types generally you can say that there are no real limits to what you can do with a text field I would always recommend you sticking through the more common ways yes you can do outlandish news designs you can also create custom text fields that have their own styles their own way of interacting with the input data. but this is not easy and in my opinion nothing for people that have not much experience with using text fields in the application. also, for the properties concerning the data itself they all revolve around the fact that you can manipulate the data that is being served from the user into the application so you have echo mode you have input checking stuff like this and then you have stuff that the results around the fact that you can display the text provided by the user in a specific format or in a specific mask.

Now you might ask why is this even important well if you have for instance a password input in your application you generally do not want to have the password when you type it in displayed in a raw text format that would mean that if somebody was looking over the shoulder of your user somebody could actually steal the password that's not really a good option and for that we have something called the input mask what does this well Simply put it takes the inputted streaming and converted in such a way that only the specific characters we wants to display are shown, and or are converted to specific characters that reflect the type of text we want to display. for passport for instance this would be the * sign.

```
9       TextField{
10          id: mouseArea
11          width: 200
12          height: 40
13          placeholderText: "Password"
14          echoMode: TextInput.Password
15      }
```

Simple TextField with echoMode

This is a good example of what I mean with the password functionality. You can use the echoMode to manipulate the display text into something different. In this case every letter would be changed to *. A better solution in my opinion would be done throw another echoMode.

```
14          echoMode: TextInput.PasswordEchoOnEdit
```

This echoMode displays the newly added letter a short period of time and then changes it to * after.

There are also other options of inputting data you will have number inputs Cara cells you have different text areas for instance with which you can put more text in with just a text field they are also really great and work wonderful for the type of application you want to build but if you just want to put in some numbers a little bit of text for instance the title or name you will most likely use a text field there's no other way really.

Also, there is no real going about using anything else if you follow the common tutorials and instructions for using input fields in Qt, if you want to put any kind of data into your application you will come across the text field in a lot of tutorials a lot of guides basically anywhere you want to learn how to put data into your application. and that was good reason there are as already mentioned other ways to put data in your application and then work with the data but the number of properties and methods and signals you have on command with the text field

are outstanding and, in my opinion, so flexible that there is no real problem creating very highly integrated inputs that you can first of all work with and Secondly manipulate in such a way that there are no real limits to what you can build.

We already used the text field multiple times in how applications are, the best example is the input for our data in our ListView. this can be found in our first real project. in my opinion the best way for learning how to use the text field is BIA first checking out the cutie dogs and trying to understand how the ListView works and then figuring out different ways you can create inputs. I usually do this by just searching online for designs of specific inputs or just using applications many applications have wonderful types of inputs with complex design and structural elements and there is a lot you can learn from just checking the inputs out and trying to implement them on your own.

But to make it easier for you to work with Text Fields, I will give you a small example down below.

```
9       TextField{
10          id: mouseArea
11          width: 200
12          height: 40
13          placeholderText: "Placeholder Text"
14      }
```

As you can see above, a simple Text Field consists out of a with and a height. This is the simplest version you can build. I always add a placeholder text property to the Text Field as I think that without it the user would most likely not be able to use the input effectively as the user does not know what he needs to input there.

But as I already said, the best way to learn about Text Fields is to build them yourself and try to build some resembling ones that you can find online.

3.1.7 Rectangle

Well, I do not think I need to explain what this component really is, it is a rectangle. It is rendered with a solid color, gradient and or a border if you want.

They are the most useful tools for building your UI, and their versatility is in my opinion great than anything else. To illustrate my point, I am going to list a few attributes that enable the rectangle to be so versatile.

- **Color**

You want a color as the background of your application? Then here you go, it might be a no brainer that a rectangle has this option but here it is

- **Gradients**

You can create any gradient you want. This replaces the color attribute, you can gradient between two or three colors depending on your options.

- **Rotation**

Well, this is also no surprise, you can turn and rotate the rectangle in any direction you want, enabling you to create a multitude of different shapes and perspectives

- **Radius**

This is the best option available if you want to round the corners of the rectangle. Unfortunately, you can only round all corners at the same time. So, if you only want to round specific corners you will need to use a remarkably interesting trick. You have a primary rectangle, where you round all corners, and then you have one or two rectangles that are only on the side where you

do not wont round corners and then you can overlap the rounded corners with that rectangle. And finally, the rectangle should have the same color or gradient as the primary rectangle. And with that you have a rectangle that visually as only specific corners rounded.

With all these attributes you can basically create any number of forms and shapes you want. There are no limits to what you can build using this. I wished that there were some more examples of advanced manipulation of rectangles on the Qt Docs but unfortunately the only real option you have right now is figuring out the things on your own or using this book as example to build what you want.

Now to give you a quick and easy example of how a Rectangle looks like in code.

```
9       Rectangle{
10          id: rec
11          width: 200
12          height: 200
13          color: "red"
14      }
```

Simple Rectangle with color

As you can see, we only need a few properties. Most of the time only a width and a height, and a color. You can also use the standard positioning properties to place the Rectangle wherever you want. They are in my opinion the perfect way to build background element in your project.

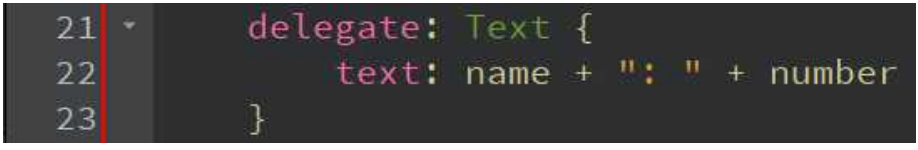
3.1.8 Delegates

Delegates are in their simplicity data aware masks. There use case arises by using them in conjunction with a View and a Model. The View takes the data from the Model and pushes the data into the Delegate. This enables an extremely fast and easy development process of building and visualising data.

There is not much to say further, you can use any form of component you want, and you can also build them data aware and easer then you might thing, so try them out.

We also used them a few times before when we build a ListView, so if you want to refresh your knowledge then you might want to reread that part of the book.

If you want an example of a working Delegate, you should have a look at the Chapter 3.1.1 List View, there is a functional example how you would do it. But if you want to see a simple example of a Delegate see the screenshot below.



This is a quite simple example of how a Delegate functions. Basically, you will always have a component that displays the data from the Model. That is all the Delegate is for. In the example above we have a Model that has the data endpoints name and number. These we can then add together as a string and then print out.

Normally you will tend to only use the Delegate in conjunction with a Model and a List View.

3.1.9 Models

As with any programming language at one point or another you will need to work with data. There are a lot of ways to do this, you can use Arrays, Lists and in Qt you will use Models. Basically, they are ArrayLists where you can put in Items that houses your data. There are a lot of different ways to do this, you have different data Models provided by Qt like the List Model, it provides the basic functionality you want from a list data structure. If you are familiar with List Arrays in Java you will quickly get the hang of the List Model.

Some things before that you need to know about Models in Qt

- All data inside the Model is provides as named data roles
- You can easily bind Qt Models to Qt Views

This means that it is quite easy to get the data from the Model into a view, this works by using the named bindings and pulling the data for each item out of them. This is a wonderful feature that allows a quite easy way to create data that can be visualised. Meaning that no matter what you build and how complicated the model gets, if you have the correct binding you can always pull the data you want.

There are a lot of different Model types out there some you might see a lot are

- **List Model (as already mentioned)**
- **XML Model**

XML is a wonderful definition language that allows the creation of highly integrated and nested data structures, it lends itself well to Qt if you receive your data is provided that way

- **Object Model**

If you come from an object-oriented programming background you will be familiar with the objects inside lists and this is no different, it greatly enhances the way you can build applications as you are not limited to only bindings

- **Integer Model**

This is a little bit sketchy as I myself have not used it, but basically you use the integer as a Model that houses several types

- **Object Instance Model**

If you only have a single object type you can use the object instance as a Model

- **C++ Data Model**

And this is the real maker of Qt, you basically have full fixability over all the different ways you could model your data. You can use Qt's own model structures and different data types or you can use the common ones like Arrays Lists and the like.

As you can see there are a lot of ways you can use Models inside your application. And there are a lot more ways. I just briefly touched over some of the different ways available. But you should also brush on them if you need them later. The best way to do this is using the Qt Docs, there you can find examples as well as explanations about all the different Models and how you can best set them up and use them.

In this book we also used a Model, and that is the List Model, for displaying simple data and displaying it, it is the most widely used option Qt provides. If you need to retouch that knowledge you can also reread parts of this book.

3.1.10 Custom Components

This is one of the things that makes building applications with Qt later so easy and great. You can build your own components and reuse them anywhere in your application. To most people this is nothing new or so interesting, as a lot of other frameworks also provide this as a core concept.

The way that it works really boils down to the core concept of

- Having the Attributes for the custom component the user can interact with
- All the functionality for the component is handled inside of the component and no matter where you use the component its functions stay the same
- You can reuse the component in multiple instances and throughout a project

These are the fundamental things you need to know about custom components in Qt. They are remarkably like the ones you would find in a lot of other frameworks too, so let us have a look at how to create a custom component shell we.

We are going to create a custom button, that when clicked opens another button above it and when you click that button it disappears, and you get a console log. Remarkably simple and not hard to understand, but it lends itself well to understanding the topic at hand.

```
RoundButton{
    id: root
    width: 40
    height: 40
    background: Rectangle{
        anchors.fill: parent
        radius: 99
    }
    property var isOpen: false
    onClicked: {
        if(isOpen == false){
            addButton.visible = true
            isOpen = true
        }else{
            addButton.visible = false
            isOpen = false
        }
    }
    RoundButton{
        id: addButton
        visible: false
        width: parent.width
        height: parent.height
        anchors.bottom: parent.top
        anchors.bottomMargin: 10
        onClicked: {
            console.log("Hidden Button was clicked")
            addButton.visible = false
            isOpen = false
        }
    }
}
```

And here we have a perfect example of how a custom button should look like. We have our first-round button, with a width height and id to match, as well as a background component. Most importantly we have a property, which is exposed so we can use it inside of our button, but also outside if we wanted to.

Inside this button is then our other round button, we interact with both buttons using the onclicked event, and as you can see, they open and close themselves when you click them. And when you click the inner addButton we will get a console log.

This right here is the power of a custom button. You can hide a lot of logic that under normal circumstances would be recreated on many different pages a lot of times, and here you can just build it ones with all the functionality that is need and then use it whenever you need to.

3.1.11 Qt Charts

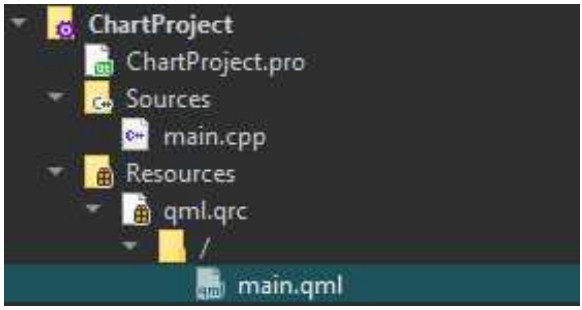
This is one of the components and topics that a lot of people might be interested in. First off, Qt Charts are an amazing tool to visualise and display data in a variety of ways.

From simple Pie Charts and Line Charts to even some really complicated and advanced Charts like Candle Charts. Qt offers nearly all of the Charts your heard can desire, or that are commonly used in data visualisation.

Now, this Chapter will be split into 3 parts. The first were we set up Qt Charts, because it does not work right from the get-go. Next, we are going to create one chart from the Qt Docs, they are more or less boilerplate code, but with them I want to show you the different ways a chart can be displayed and how they work, and how they are built. Lastly, we are going to create a custom Candle Chart, that is styled and displays some stock data.

All of this is just the tip of the iceberg, but we do not have the time to create everything on our own, so we are going to skimp over some parts. When you want more information or you are missing some parts, then you might want to check out the Qt Docs for more information, there you can find anything from the properties you can use to some boilerplate you might want to try out.

Now first of let us edit a simple Hello World project in such a way that we can use Qt Charts. First of you need to create a simple Qt Quick Hello World Application.



We do not need any new files, or anything special. Next, we can edit our main.qml file. We need to update the imports to the newest version, delete the QtQuick.Window and add the QtQuick.Controls to our main.qml. We also should change the window to a ApplicationWindow. We already did this a few times, so you should be remarkably familiar with it, none the less down below you find a screenshot of how your main.qml should look like.

```
1  import QtQuick 2.15
2  import QtQuick.Controls 2.12
3
4  ApplicationWindow {
5      width: 640
6      height: 480
7      visible: true
8      titel: qsTr("Hello World")
9  }
```

Next, we need to import QtCharts. The newest version that of the time of writing this is 2.3, if you have a newer version that please use that.

QtChart import

Now if you done everything as I instructed you to do, then you will get an error that QtCharts is an unknown component. This has 2 reasons, one is that we need to change a few things in the main.cpp and .pro file of our application to make. The second reason is only important when you are using the Qt 6.0, in this version QtCharts is not included. If you want to use Qt Charts you would need to drop down to Qt 5.12 or later. But this is only the case right now at time of writing this book, Qt is bound to be updated and Qt Charts is going to be included. So please follow the next steps before switching the Qt version.

```
1  QT += quick
2
3  CONFIG += c++11
4
5  # You can make your code fail to compile if it uses deprecated APIs.
6  # In order to do so, uncomment the following line.
7  #DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the APIs deprecated before Qt 6.0.0
8
9  SOURCES += \
10         main.cpp
11
12 RESOURCES += qml.qrc
13
14 # Additional import path used to resolve QML modules in Qt Creator's code model
15 QML_IMPORT_PATH =
16
17 # Additional import path used to resolve QML modules just for Qt Quick Designer
18 QML_DESIGNER_IMPORT_PATH =
19
20 # Default rules for deployment.
21 qnx: target.path = /tmp/$${TARGET}/bin
22 else: unix:!android: target.path = /opt/$${TARGET}/bin
23 !isEmpty(target.path): INSTALLS += target
```

.pro unchanged

First of a comparison of the standard .pro file you will find know adays in your project and how it should look.

Really important is that you add in line 1 *gui core widget* behind quick. This will enable Qt to run the qui widget components that are normally unavailable in Qml Quick Applications. I also updated the C++ version to a newer one, this is optional put I would recommend that you always use a newer version of C++.

```
QT += quick gui core widgets

CONFIG += c++17

# The following define makes your compiler emit warnings if you use
# any Qt feature that has been marked deprecated (the exact warnings
# depend on your compiler). Refer to the documentation for the
# deprecated API to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS

# You can also make your code fail to compile if it uses deprecated APIs.
# In order to do so, uncomment the following line.
# You can also select to disable deprecated APIs only up to a certain version of Qt.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the APIs deprecated before Qt 6.0.0

SOURCES += \
        main.cpp

RESOURCES += qml.qrc

# Additional import path used to resolve QML modules in Qt Creator's code model
QML_IMPORT_PATH =

# Additional import path used to resolve QML modules just for Qt Quick Designer
QML_DESIGNER_IMPORT_PATH =

# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

.pro changed

Now to the main.cpp file, we need to edit it also. The basic thing we need to change here is the type if QApplciation Qt uses under the hood.

Normally Qt uses the QGuiApplication as the application type to render the basic Qt Quick Applications. This will not do for us, we need the QApplication for our type.

```cpp
#include <QGuiApplication>
#include <QQmlApplicationEngine>

int main(int argc, char *argv[])
{
    if (qEnvironmentVariableIsEmpty("QTGLESSTREAM_DISPLAY")) {
        qputenv("QT_QPA_EGLFS_PHYSICAL_WIDTH", QByteArray("213"));
        qputenv("QT_QPA_EGLFS_PHYSICAL_HEIGHT", QByteArray("120"));

#if QT_VERSION < QT_VERSION_CHECK(6, 0, 0)
        QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
#endif
    }

    QGuiApplication app(argc, argv);

    QQmlApplicationEngine engine;
    const QUrl url(QStringLiteral("qrc:/main.qml"));
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,
                     &app, [url](QObject *obj, const QUrl &objUrl) {
        if (!obj && url == objUrl)
            QCoreApplication::exit(-1);
    }, Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}
```

Unchanged main,cpp

There are no other changes we need. Yes, there is a quite different structure when you compare the two different versions but they are functional the same. And when you create an application you will probably never relies on the difference.

But we need these changes, so change your main.cpp to the version you can see in the next screenshot. Also remember Qt updates the main.cpp quit a few times so it might happen that there is a newer version throw which you can achieve the same result.

```cpp
#include <QApplication>
#include <QQmlApplicationEngine>

int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);

    QApplication app(argc, argv);

    QQmlApplicationEngine engine;
    const QUrl url(QStringLiteral("qrc:/main.qml"));
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,
                     &app, [url](QObject *obj, const QUrl &objUrl) {
        if (!obj && url == objUrl)
            QCoreApplication::exit(-1);
    }, Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}
```

Changed main.cpp

Now when you have changed the main.cpp and the .pro file you can go and rebuild the project. This can be done by right clicking on the Qt project and then selecting rebuild. When you have done with you can see that the error about QtCharts not being found as a module has vanished and we can finally start building our first Chart.

```
ApplicationWindow {
    width: 640
    height: 480
    visible: true
    title: qsTr("Hello World")

    ChartView {
        anchors.fill: parent
        antialiasing: true

        PieSeries {
            id: pieSeries
            PieSlice { label: "eaten"; value: 94.9 }
            PieSlice { label: "not yet eaten"; value: 5.1 }
        }
    }
}
```

Simple Pie Chart

This is the simplest and easiest to understand example of how a Chart works in Qt. this is the basic Pie Chart, its data needs to

always consist out of two things, one is the value as you would expect and the other is the label.

All Charts need to be located in a Chart View, this is basically a container for the actual Chart. It has a height and a width, you can also set a few more property's like anchors, antialiasing, theming, and animations to it. The Chart View cane be best compared to something like a Scroll View.

Inside a Chart View you will find some sort of Chart Series. These are always named after the type of Chart they represent, like Pie Series in this instance. These Series are to put it simply only there to tell the Chart View what type of Chart it is and how the data should be displayed.

Lastly there the actual data elements, here they are named Pie Slices as they represent the Slice of the Pie. Each Series has their own different type of data. They fundamentally work the same, they consist out of a value, label and a color for representing, and maybe they also have a few other attributes.

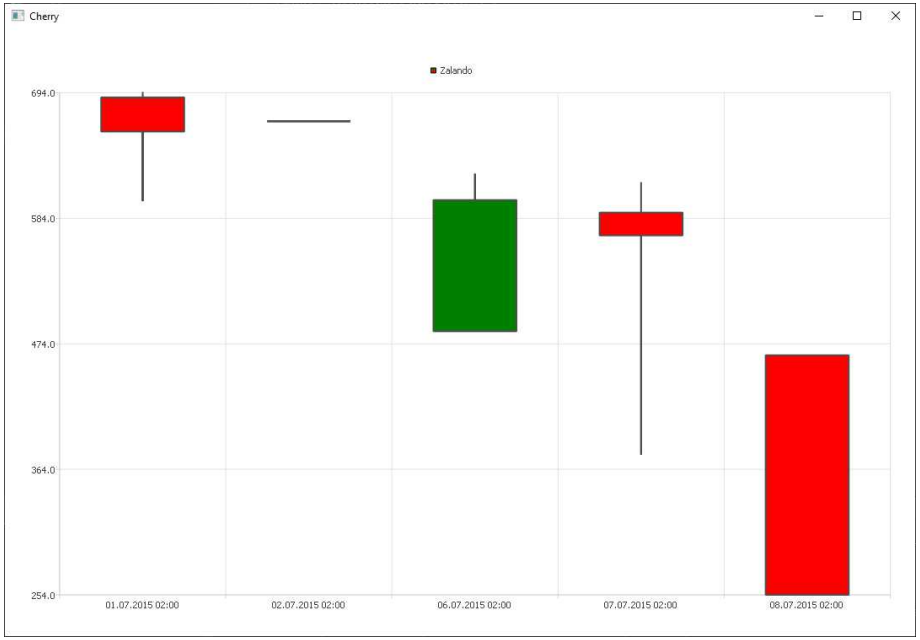
As you already know from the List View, or List Model you can add, delete, and modify existing Pie Slice or any data of a Chart Series. This allows you first of all to make interactive Charts, and secondly you can be assured that you have complete control over the elements.

Now we can create our own Candle Chart, which we are going to fill with our own data. This is going to be not too hard, so follow along and see for yourself.

```
    ChartView {
        anchors.fill: parent
        antialiasing: true

        CandlestickSeries {
            name: "Zalando"
            increasingColor: "green"
            decreasingColor: "red"

            CandlestickSet { timestamp: 1435708800000; open: 690; high: 694; low: 599; close: 660 }
            CandlestickSet { timestamp: 1435795200000; open: 669; high: 669; low: 669; close: 669 }
            CandlestickSet { timestamp: 1436140800000; open: 485; high: 623; low: 485; close: 600 }
            CandlestickSet { timestamp: 1436227200000; open: 589; high: 615; low: 377; close: 569 }
            CandlestickSet { timestamp: 1436313600000; open: 464; high: 464; low: 254; close: 254 }
        }
    }
```

Simple Candlestick Chart

Now as you can see it is remarkably similar to the other Charts. It consists out of a Chart View, a Candlestick Series and lastly Candlestick Sets. The only difference is that the data is a little bit different than before. We do not only have one value, but four different. You have the open, close, high, and low values, all of them combined in this chart leave you with a candle stick that represent the data we have put in. Now how can we stile this Chart, currently it does not look that exiting.



Styling charts is not really difficult, first of you have the option of changing the increasing color and the decreasing color as you can see in this screenshot. I choose the colors green and red for that. They are commonly used as these colors and clearly readable.

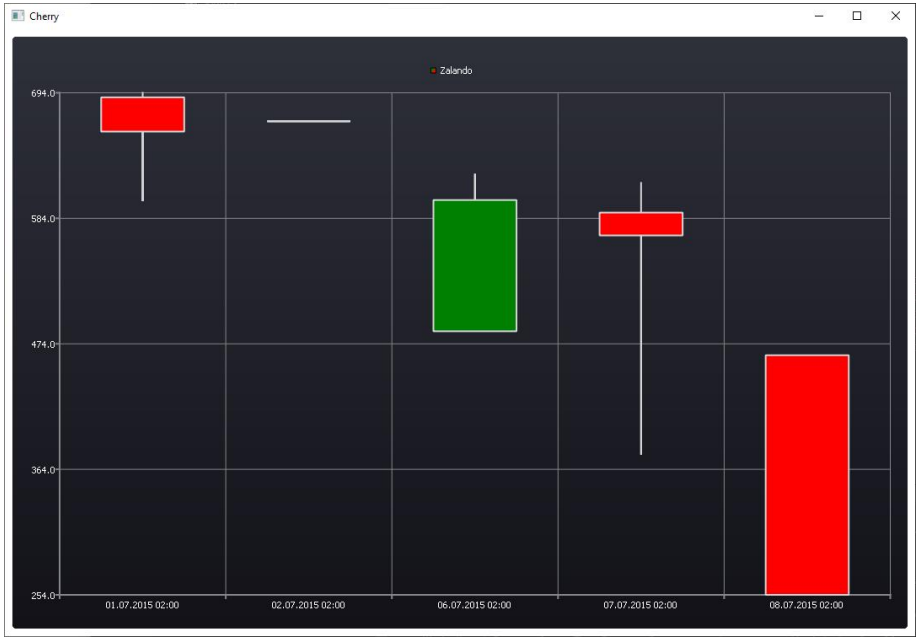
The second option of styling Charts is by using the themes property.

Constant	Description
ChartView.ChartThemeLight	The light theme, which is the default theme.
ChartView.ChartThemeBlueCerulean	The cerulean blue theme.
ChartView.ChartThemeDark	The dark theme.
ChartView.ChartThemeBrownSand	The sand brown theme.
ChartView.ChartThemeBlueNcs	The natural color system (NCS) blue theme.
ChartView.ChartThemeHighContrast	The high contrast theme.
ChartView.ChartThemeBlueIcy	The icy blue theme.
ChartView.ChartThemeQt	The Qt theme.

Different Chart Themes

Here you can see some of the different Chart themes that Qt provides. And some of them are so good, that I personally only tend to use them. It is true that you can do custom styling to every aspect that the Chart has to offer, but it is not really necessary.

I chose the Chart Theme Dark, as it is particularly good in my opinion. I would recommend you to check out other themes and find what you like, and when nothing fits what you want to create them maybe try building your own theme, or custom styled Chart.



Here is a screenshot of how this theme looks like.

Now that you have a liny overview of what Qt Charts has to offer, the best way of learning how to use it and what would best fit your own project, try building your own and displaying some data in it. Qt Charts are extremely versatile and easy to use and they provide you with a very great way of displaying data in a variety of ways.

3.1.12 JSON in Qt

If you want to build any form of application now adays you need to work with data. This data can be in a variety of forms, you can use the already discussed models, or databases. But there is also another way using JSON as your data model.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Here is an example of how JSON looks like. Basically, JSON allows you to group data together using data types, you have for instance

- **Strings**
- **Integers**
- **Arrays**
- **Dates**
- **Boolean**

As you can see you can also use more complex data types like Arrays. JSON is very versatile and you can basically create any form of data structure you want. It is perfect for lightweight data structures that enables you to quickly store and retrieve data.

Now how can you use JSON in your application. There are two ways of using JSON in your application. First of you can use the QJSONModel, these are C++ functions that you can use to convert a JSON Model to a C++ Model with which you can then interact normally like with any other model or class.

The other way is using an Open-Source component, the JSONListModel. This can be found here https://wiki.qt.io/JSONListModel. Basically, you can import the files you need from the link on the wiki, and then you have access to a QML component, that is nearly identical to the normal that the List Model QML comes with.

These two methods provide you with a perfect way of building applications and interacting with JSON as your data. In my opinion you can perfectly work with it in Qt and when you need to you can create even your own JSON data.

So, I hope you are able to work with JSON now, if you want more examples with JSON go visit the Qt Docs there you can find a few examples of how to work with the C++ Models and the QML Components.

3.2 Features

Here are some of the features I want to talk about outside of the context of the tutorials. Mainly two features, one is the C++ Integration. Depending on how deep you want to go into learning Qt, and how ambitious and large the application you want to build is, it will be essential to use C++ as your backend. And secondly Translation Files, a fewer known features Qt provides, which is used in a lot of Qt applications, but not a lot of tutorials shine a light on how to best use it.

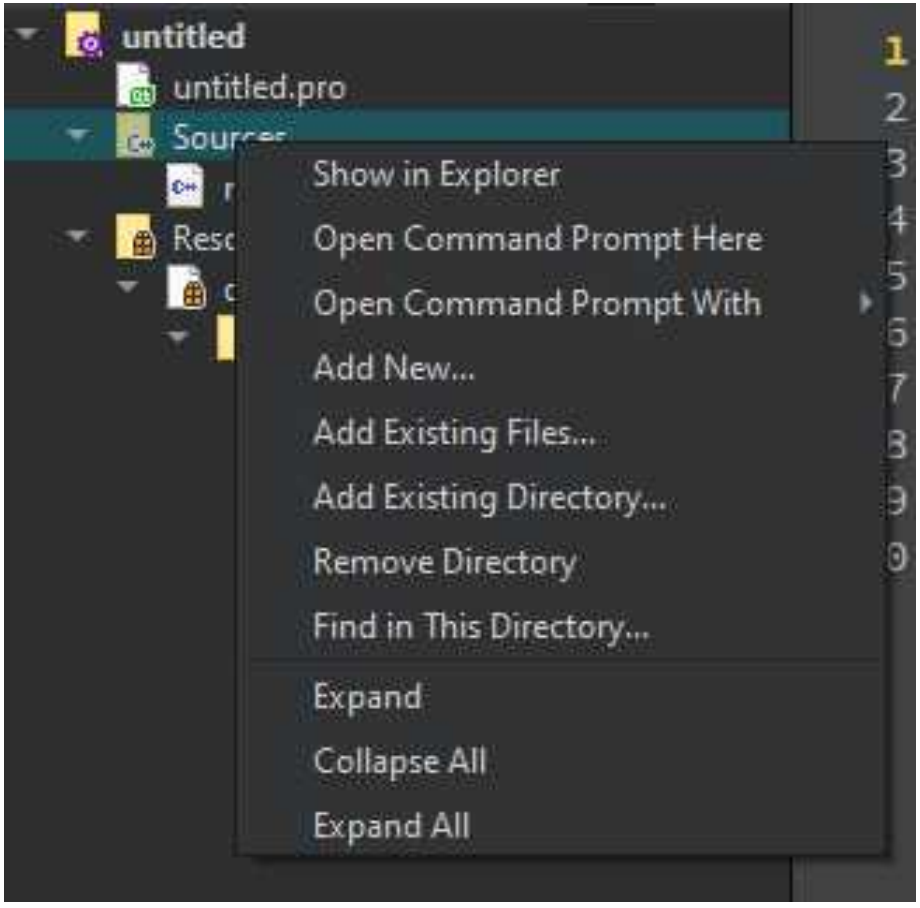
3.2.1 C++ Integration

C++ Integration is an awfully specific and difficult topic, that is awfully hard and difficult to understand in the context of Qt. There are a lot of resources and tutorials about how to set up see integration and I will not do the best job of showcasing the flexibility and the power behind using C++ as our application backend. This is also not the main focus of what I want to achieve in this chapter, my goal is to show you how to set up the integration and firing up some small functions. The real power behind C++ integration comes from knowing how to work with C++ and using the power of C + + for building applications. So, if you know C++ you can immediately start developing complex interactive and highly performant functionality in your application.

But for the people that do not know that much about C++ and how to work with it, you will be learning how to set it up and if the need arises for you to use C++ in your application or you do

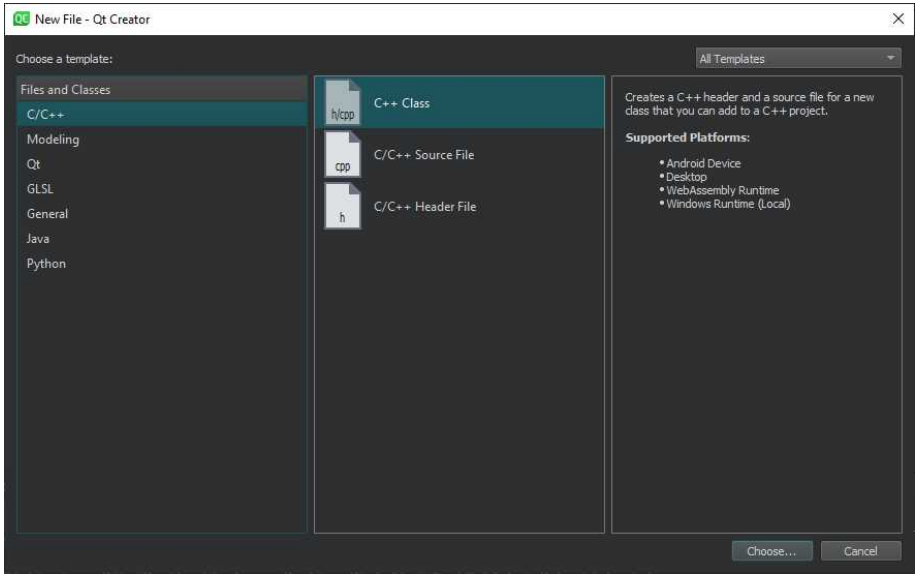
not have an actual choice because believe me there will come times where there are specific functionalities that do not work with the native components and functionality provided. it might be true that most of the components will ever need are already implemented in QML, but to give a specific example if you want to work with files like JSON files for instance you will need to do this using the C++ functions provided by Qt. and these can only be accessed through the C++ back end.

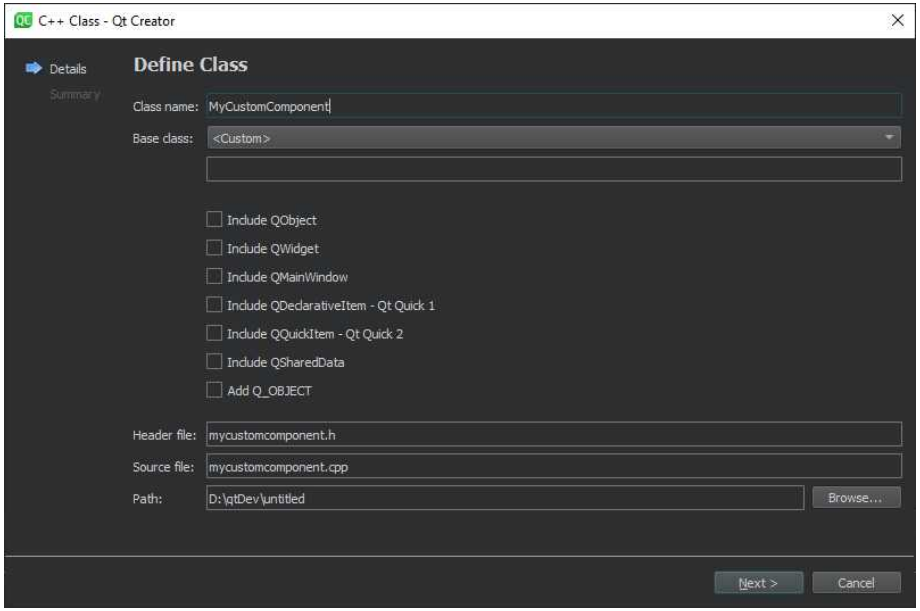
Now how can we best set this Connection up? We will let us find out. First of we need to create a new C++ Class I our Qt Quick Application. For info we are going to do this from a basic Qt Quick Empty template.



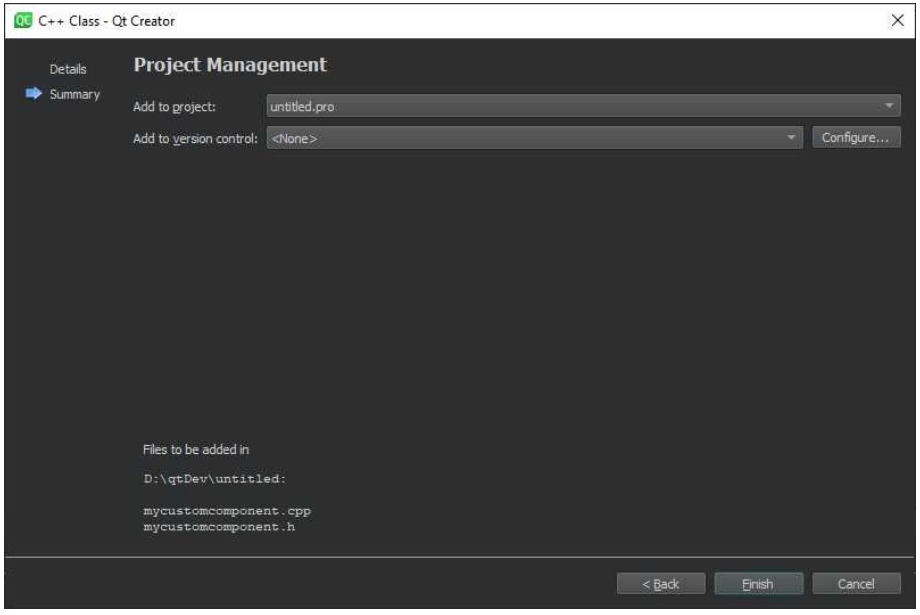
The best way to create the needed C++ Class is by creating it in our Sources. So right click on our sources and select Add New…

This will open up a remarkably similar wizard as the one we commonly used for creating our QML files. Here we need to go under C/C++ and select C++ Class. This could also be automatically selected, but if not then go to it and hit choose.

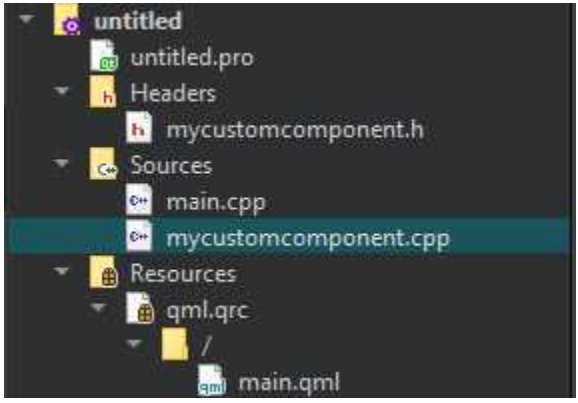




On the next page of the wizard, we need to give our class a name, I choose MyCustomComponent, as that is what we are creating here. We are creating our own C++ component that we then can use to do stuff.



Lastly, we could add it to our VCS, but we do not need that right here and click create.



If you have done everything as I instructed you, two new files should be created. The first on Is the .cpp file of our custom component, this will be below our main.cpp, the other one is the header file for our custom component.

```
1  #define MYCUSTOMCOMPONENT_H
2
3  #include <QObject>
4
5  class MyCustomComponent : public QObject
6  {
7  Q_OBJECT
8  public:
9  explicit MyCustomComponent(QObject *parent = 0);
10
11 Q_INVOKABLE void doSomething(QString input);
12 signals:
13
14 public slots:
15
16 };
```

You should fill the header of our component with the code above. I will not go into too much detail later on what we created here, but just know, that line 11 represents the method we are going to write next, and that will be invokable in QML later.

```cpp
#include "mycustomcomponent.h"
#include <QDebug>
using namespace std;
MyCustomComponent::MyCustomComponent(QObject *parent) :
QObject(parent)
{
}
void MyCustomComponent::doSomething(QString input)
{
    qDebug() << input;
}
```

The code for our mycustomcomponent.cpp looks like this. In line 8 to 11 you can see the method we created. Basically, is a simple function that takes a string input and writes it to the console.

A quite simple function, but it works and it is a good representation of how the component can also take inputs and work with them.

```cpp
    QQmlApplicationEngine engine;
    qmlRegisterType<MyCustomComponent>("MyCustomComponent", 1, 0, "MyCustomComponent");
    const QUrl url(QStringLiteral("qrc:/main.qml"));
```

The line 18 needs to be added to our main.cpp. Currently our component is finished, but you could not do anything with it. For that we need to register the component as a type. This makes it possible to use it in QML.

When you added this line to your main.cpp you can then import our component into the main.qml File we have.

```
import QtQuick 2.15
import QtQuick.Window 2.15
import MyCustomComponent 1.0
```

Next you can add the component into the content for our main.qml and give it an id. As we want to access the functions behind the component, we can use the Component.onCompleted to execute the function when the application finishes building.

Lastly, we only need to call the function and give some sort of text with the function so that it can be displayed.

```
11      MyCustomComponent{
12          id: myCustomComponent
13      }
14
15      Component.onCompleted: {
16          myCustomComponent.doSomething("Hello from C++")
17      }
```

If you have done everything until now, you should get a console output with the string you put in.

```
14:41:48: Starting D:\qtDev\build-untitled-Desktop_Qt_6_0_0_MinGW_64_bit-Debug
"Hello from C++"
QML debugging is enabled. Only use this in a safe environment.
14:41:50: D:\qtDev\build-untitled-Desktop_Qt_6_0_0_MinGW_64_bit-Debug\debug\un
```

As you can see this is a really good way to build up functions and tools, for which you need more functionality then the common QML components. It is not hard to set up or use and wones you get the hang of it you will probably tend to use it from time to time, as it is simpler to do things in C++ sometimes then hustling around with difficult QML models and custom components.

My recommendation is that you try out to use C++ from time to time in your project, as you can build extremely great functionality throw this. Also, you have complete control over all the C++ models and classes Qt has for your, so no matter what you want to build you are able to do so.

3.2.2 Translation Files

This is something funny, and what I see all the time in forums. “Does anyone actually use the Translation Files”. And the answer is yes, there are a lot of use cases for them, and there are a lot of people using it.

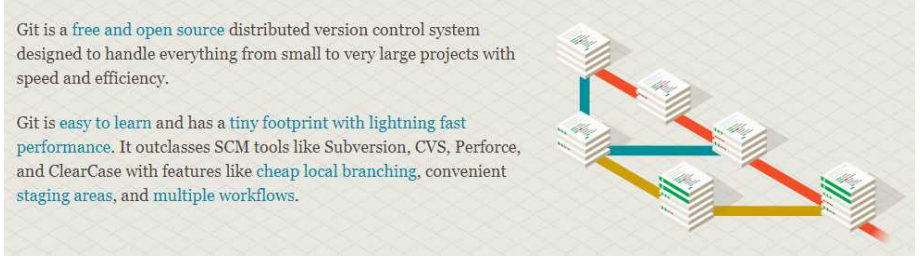
First of one of the things that a lot of people say way this is even a question. In tutorials online, or in books you will not find them. First of in development you do not need them. Development usually is only done in one language. So normally you would not think about adding other languages until the very end of development. This also leads to the second point, why would anyone talk about using it. Generally, all tutorial series, books or videos tend to focus on the first few steps in learning about a language, framework and even the better ones tend to skip over some parts near the end. So, it should be no surprise that there is no mention in any tutorial out there how to best use it.

But using it is not as hard as you might think. First of using it heavily relies on the fact that all the text attributes you want to be changed by your translation file need to be in the correct format. And need to have qstr before them. This is the best way to use the translation file. After that you need to link the corresponding word with the translated version.

Generally, I can say that the best use for a translation files is at the end of your development when the actual publishing and deploying of your application takes place. So, remember that this exists and learn about it later if you really need it.

3.2.3 Git in Qt

Git is one of the technologies that you will find everywhere in development. It is the basic tool most developers use to work together on projects.



What is Git, from https://git-scm.com/

But what Git is should not be that difficult for most people to get and understand. But how can it be used in Qt, how easy is it to set up and should you use Git?

Well for the first question how Git can be used and set up. When you create a new Qt project, you get asked if you want to use a VCS, this stands for Version Control System. There are a lot of different Version Control Systems out there, and one of them is Git. When you set the project up with this option

enabled, Qt will create a .gitignore file. This file simply tells Git that some files cannot and should not be added to the repository.

Also, Qt Creator made the project a Git Repository, meaning that you could immediately push it to a remote repository or share it with the world. This pretty much the same as you would do it with any other project or application you might have.

Now is Git the best option for Qt? Well, that depends on what you typically use. When you are familiar with Git then you will find that this is the same as with any other Git project. Different Version Control System tend to work like Git, for instance AWS with their Code Commit.

Lastly, I recommend that you always use some sort of Version Control. I know this is Developer 101 thing to say, and I know that most people probably already use some sort of Version Control already. But when you have anything larger than a calculator please use Git or another Version Control System. It is so much easier to build and develop applications when you use a VSC, and especially with Git, everyone uses it, meaning that when you are searching for a new job.

Recruiters tend to look for these skills as they are essential for companies. You will always develop with other people and because sending files between each other is not that performant or user friendly VSCs are the perfect solution.

We used Git only in one project in this book, the Rock-Paper-Scissors Game. There we set up a new repository and uploaded our project to Git Hub. It was not really special but a good example of how you probably need to do it when you need to use Git in the project.

3.2.4 Qt Animation

Animations are a big subject when you want to create applications for mobile and or desktop devices. And nowadays nearly every framework has some sort of animation framework. And Qt is no different in that regard. Animations were available for a few years now and are currently at a particularly good point when it comes to features available.

But what types of animations das Qt have and how are they different from each other.

- **Property Animation**

As you should know by now components in Qt have properties attached to them, like width, height, and anchors. These can be manipulated throw Property Animations.

```
PropertyAnimation {id: animateColor; target: flashingblob; properties: "color"; to: "green"; duration: 100}

NumberAnimation {
    id: animateOpacity
    target: flashingblob
    properties: "opacity"
    from: 0.99
    to: 1.0
    loops: Animation.Infinite
    easing {type: Easing.OutBack; overshoot: 500}
}
```

Example from the Qt Docs

At the top of this screenshot, you can see a rather easy Property Animation. This perfectly represents how an animation is in most instances structured. You always need an id, a target to which the animation is applied, the property you want to change, to what the property should change and lastly the duration of the animation.

These are the fundamental property's you need for an animation to play. How you then activate and play the animation is up to you.

- **Number Animation**

This is remarkably similar to the property animation, and you can see an example in the first screenshot. Basically, you can manipulate the numbers behind the property's, like with, height, x, y and in the example opacity. You could also do this throw the Property Animation.

So, in general you can say that the Number Animation is a specialized type of animation that best works with number properties.

- **Transitions**

When you want to switch between two pages of a Swipe View, or add, delete an item from a List View the state of the component is changed. You could do this now instantly so the change takes place the second you changed the component, or you could apply a transition to the state change.

What a transition basically does is link to different states of a component together, and when the state changes the transitions starts playing. Now a transition is fundamentally an animation, so if you are familiar with the property and number animation you are able to build your own animation.

```
states: [
    State {
        name: "PRESSED"
        PropertyChanges { target: button; color: "lightblue"}
    },
    State {
        name: "RELEASED"
        PropertyChanges { target: button; color: "lightsteelblue"}
    }
]

transitions: [
    Transition {
        from: "PRESSED"
        to: "RELEASED"
        ColorAnimation { target: button; duration: 100}
    },
    Transition {
        from: "RELEASED"
        to: "PRESSED"
        ColorAnimation { target: button; duration: 100}
    }
]
```

Example from the Qt Docs

Here is a fairly simple example of how a transition could look like. Basically, you have the different states of your component and then you apply the transition to the state, and when the state is triggered it goas from one state that is declared to another.

A side node be confident that you have the correct state for the from and to in your transition, when you have more than two states, it could break the UI really easy when the wrong transition is triggered.

- **Parallel and Sequential Animations**

Now that you know the basic types of animation, how they work and how to set them more or less up, but sometimes you want

to run two or more animations in parallel or sequential, you could trigger them differently, or at the same time but this is not really that great or performant. But for that exact purpose the Parallel and Sequential Animation exist.

```
SequentialAnimation {
    id: playbanner
    running: false
    NumberAnimation { target: code; property: "opacity"; to: 1.0; duration: 200}
    NumberAnimation { target: create; property: "opacity"; to: 1.0; duration: 200}
    NumberAnimation { target: deploy; property: "opacity"; to: 1.0; duration: 200}
}
```

As you can see here, both Sequential Animations and Parallel Animations both group animations together, may they be Number or Property Animations.

This enables you to lunch multiple animations at the same time, or one after the other. This allows you to save on code or complicated animation structures.

Now you have a general overview over the different types of animation in Qt, how to use them, what an animation is made out of and how to set them up. We also covered the Parallel and Sequential Animation, and what they can be used for.

In my opinion the best way to learn about animations in Qt and how to best use them, you need to try them out. They fundamentally work the same, and even more importantly the normal animation principals also work on the animations in Qt. So, you can learn from UX and general animation principals to set up complicated and highly usable animations that enhance the user experience.

3.2.5 Databases in Qt

When you want to create an application that uses data often, and this data needs to be present and saved all the time. For that most developers would tend to use something like a database.

To those unfamiliar a database simply is a bunch of tables you can interact with throw the database. You can add, delete, and change table cells and columns. The best way to explain this is by excel. It has tables and cells. I know this is not the best way of explaining what a database is, but if you want to know what that is, you should search on Google or YouTube.

For us, the only really important part is that most developers and projects tend to use databases for their high flexibility and the fact that it is so widely used that you will find tutorials anywhere you look.

Qt has two ways I want to mention throw which you can use databases. The first is quite easy to explain as with a lot of other programming languages you have a class that can handle database connections, the results that come from this and so on. It is an option that most developers are already familiar with throw other languages. Also, this is the option that I would recommend to you. It is widely supported throw the Qt C++ Backend and it is fairly easy to understand ones you get the hang of it. But remember, these functionalities are only assessable throw C++. You can create your own component, that you can then interact with throw QML, but the logic needs to be written and executed in C++. If you are not familiar with C++, then this can be somewhat complicated but throw a little tinkering you will get the hang of it.

The second option is the local storage functionality Qt Quick provides. Basically, this is an SQLite Database that you can interact throw QML and Java Script with.

It has the same functionality as a normal database, with the exception that it is local.

Now below I have screenshots of how we set up the Local Storage system in our Rock-Paper-Scissor project. These are the basic functions you will need to best set up your local database.

```
function dbInit()
{
    var db = LocalStorage.openDatabaseSync("Database", "", "Data Table", 1000000)
    try {
        db.transaction(function (tx) {
            tx.executeSql('CREATE TABLE IF NOT EXISTS data_table (data_text, data_value)')
        })
    } catch (err) {
        console.log("Error creating table in database: " + err)
    };
}
```

First of the dbInit function. This will open or create the database when you call the function. It is essential for using databases in your project and you cannot go without it.

```
function dbGetHandle()
{
    try {
        var db = LocalStorage.openDatabaseSync("Database", "",
                                               "Data Table", 1000000)
    } catch (err) {
        console.log("Error opening database: " + err)
    }
    return db
}
```

Next is the dbGetHandel function, we do not want to open up our databases every time when we call a set or get function, so most of the times, you will tend to write a function which sole purpose is to handle the database connection for our other functions.

If you want a better explanation you can read **Chapter 2.4.3.9**.

```
function dbSet(data_text, data_value){
    var db = dbGetHandle()
    var rowid = 0;
    db.transaction(function (tx) {
        tx.executeSql('INSERT INTO data_table VALUES(?, ?)',
                      [data_text, data_value])
    })
}
```

The next two functions are the real core of every local storage and database setup. You will always need to get and set data from your database. And that is exactly what both of these functions do. The function above shows you how to set data in your database. The basic way to do this is just by using a SQL INSERT and then giving the statement the values, you want to set.

```
function dbGet(data_text){
    var db = dbGetHandle()
    var rowid = 0;
    db.transaction(function (tx) {
        var result = tx.executeSql('SELECT data_value FROM data_table WHERE data_text="'+data_text+'"')
        rowid = result.rows.item(0).data_value
        console.log(result.rows.item(0).data_value)
    })
    return rowid;
}
```

The get function works similar, but we run a SQL SELECT on our database. And then we return the value of the item we get.

As you can see the basic functions needed for a Local Storage system. You can also connect to remote databases, but for that you will need C++.

I would recommend you try figuring out how to best work with databases, what project need them, how to set them up. After that you need to learn how to use them, this can be best done throw learning about databases in general. All the knowledge you can learn can also be translated into Qt with databases.

3.3 Things to remember

Now a few things you should remember, first of Qt is an excessively big and complicated topic with a lot of high level and overly complicated subjects that are not covered in this book. I will try to provide resources later to where you can start learning more about Qt its features and the thing you need to know but be aware that the road ahead after reading this book will consists out of trying out things.

Qt is a little bit odd in one subject and that is the case of where to you it. It can do everything, and I do not mean this in the general sense, but in actual effect it can do everything. Some things it can do well, like making Server Applications, Console Applications and highly integrated and reliant on speed Applications. In these cases, Qt is unbeatable in my opinion. It simply outperforms anything else, but you get this with one significant drawback. And that is the amount of time, manpower and lines of code you need to get those things down.

A good example is building a Task Manager. You would never do this in Qt. It is a good example to learn Qt with and with Qt Quick and QML it can be done fast and easy, but nowhere near as easy and fast as with other frameworks, especially Web-Frameworks. They simply outperform Qt and its building process with cheer ease of building. This does not mean that they are better per say, but they have their pros in some aspects.

So, Qt has its downsides, but the moment you want to build anything that requires speed, performance or shear amount of data throughput C++ and Qt for that matter are the right choice. But always remember that depending on the application you want to build you should first of search for the correct way of

doing it with Qt and then build a plan for yourself. A Plan, Structure and maybe even a Diagram. If you have all this and you know that the application is possible, then you can start building it.

On the part of building the application. Failure is inevitable, no matter the preparation you put into it or the amount of determination and will power you will fail and abandon the project. This is nothing bad and it happens all the time. Even the best of us will abandon a project at some point. And even good developers tend to overestimate the amount they can do. So, take small steps, do not overestimate yourself and especially do not thing bad of yourself because you abended a project. If the project does not bring you any money in at that moment, and you are doing it for fun, then the only thing you can always take from any project even those which you abandoned is the learning process and teachings you can get from it. There is always something you can learn and something new to be had.

And with that I leave you be. Do not hesitate to learn new things, start experimenting with Qt and the Features, and Components it provides. There are so many great things you can build with it, and the only thing holding you back is your own imagination and work effort.

3.3.1 Writing Diagrams for Qt

At a certain level and size of a project, you will need to create diagrams. This can be done throw several different ways, and you can use whatever tool, or way you want. But there are a few things that you should always keep in mind while creating diagrams.

Think about the project structure, when you have large and complicated projects that have a lot of QML files and visual complexity, you should always follow that as a guideline throw the diagram. This makes it easier to follow along throw the development process.

The other thing to keep in mind is the naming of files and prefixes. I always have the problem, that I do not follow the naming scheme I have in the diagram in the actual application. This results in a lot of unnecessary problems. If you keep the naming scheme correct you will always be able to find the correct components and refer to the diagram while you create your application.

And lastly, I would recommend keeping a color scheme for the components and the functionality. This makes it quite easy to find the functionality to the component just based on the color of the diagram elements.

Diagrams are in general a good thing to have while you develop, but there are not needed for everything. If you are just designing and programming a calculator, I would not write a diagram. But when you want to build something larger, more complex, and maybe you work with multiple people on the project it might be immensely helpful to have a diagram.

There are a lot of programs out there to write and design diagrams, and they work as you would suspect. But there are not needed to write diagrams, I usually write mine by hand. This is the fastest and simplest way of creating them.

But overall, in general diagrams for Qt are not really that different for diagrams you would use in any other programming project. And the tips I gave you are more of the kind that can help you if you have a similar workflow and development process as I have, otherwise you can also always find your own preferred way.

3.4 Advanced Topics in Qt

Now we come to some topics that are not as necessary as others, but they still should be mentioned and talked about. If you read until now, you should also read these following chapters, they are going to tell you a little bit more about specific topics.

3.4.1 Mobile Applications

As we are in the day and age where mobile applications are always in demands and really everybody wants to build their own mobile application Qt comes with the option of building mobile applications. But this is not some new topic in the framework and the main features were already implemented quite a long time ago for instance the entire cutie quick line-up of components way to build stuff was immediately from the get-go built with mobile devices in mind and this is a good thing as you will probably no if you look for instance in the web development spectrum, their mobile devices are always first priority.

And that was good reason allot of people only use mobile devices nowadays for consuming media using applications and for a business nowadays or company that provides any kind of service or product having in mobile application is not only important but sometimes even essential. and with that we want to build mobile applications.

We already covered off how to set up Android Studio how to implement it is in Qt. So, this is nothing we need to cover right here. I will just give you some general advice on the way when

you want to build mobile applications using Qt. as we already discussed Qt has its difficulties. The best way of solving them is the standard way you would also do it on a desktop but mobile development comes with their own share of problems and you will need to learn a lot about how to best make applications for mobile devices how the best design practises are what you need to do on what you actually need to have a focus on these things don't really translate from a desktop environment to a mobile device environment and you will need to actually really learn some topics and stuff you will take for granted when developing desktop applications. This might not sound really exaggerated and cried untrue but believe me the work you need to do to make a mobile application usable and good looking with cutie requires little bit of rethinking of how to do things and you will run into problems that are little bit uncommon and, in my opinion, take some time to adjust.

Fortunately, we already built a mobile application using Qt just look where they are hanging man project this was created completely from the ground up with mobile devices in mind and therefore, we could immediately alleviate a lot of problems they would come from developing an application that was not from the get-go designed to be run on a mobile device.

We also need to be very fair too Qt as it has been refined and polished over the years regarding developing mobile applications. when I started out developing using Qt I always run into difficult box problems that nobody has ever had anywhere online and I sometimes was the first person on stack overflow to asks questions about specific problems or specific bugs and errors showing up in my code this was really difficult and sometimes even brought me while nearly to the point of switching frameworks, but I prevailed and nowadays this is not really a problem anymore most features you will find on iOS or Android are completely usable inside of cutie and there are so

more less tightly integrated that only for specific few features I'm completely usable out of the get go.

But what does this even meta to you well if you want to build mobile applications using Qt when you will have a lot of options and functionality so that were only a few years back completely alien and unusable in Qt. For instance, Splash-Screens in the Android manifesto with all the different configurations you can do with it java integration the possibility of using different types of local and push notifications really complicated signals and sensory data that you can use or get from the device is own sensors.

So, if you follow this book until now and you are interested in building Android or iOS applications then welcome you are now able to do this, they are not very much dissimilar to desktop applications you could build, and we also discussed how to build them how to set them up and how to deploy them and now the only thing really left is building something so get to building it.

3.4.2 Interactive and Real Time Data

This is a topic a lot of people come up with quite a lot of times while developing applications as it is one of the things that we will tend to quite a lot while developing applications as require connexion to a database on hand large amounts of data. The main way of working with interactive forms or databases in Qt is one by making the fields and inputs aware of the data they need to represent or they are responsible of this is best by using C++ model or anything really where you can specify what type of data you actually want to use an what is going from the inputs to your database or you can use Jason for somethings because there are some external Jason tools and components which you

can use that really enable you to make interactive and aware forms and list use that don't require a lot of work to set up.

A good read would be on the Qt Docs right here: https://doc.qt.io/qt-5/sql-forms.html .

Generally, this is a topic that is extremely difficult to get a hang of, and you will need to take your time with it. Also, you need to keep in mind at the start of the Development what Data you have so you are able to test your inputs. Without that you will run in the problem that the inputs tend not to work because you have the wrong Data for the Job.

Also, something that Users want to have is a representation if what they are doing is correct. This first of reassures the user that what he is doing is correct and good, and even more importantly it minimises the risk of having wrong data inputted into the Application. You can also search online for this kind of topic, there are a lot of great Tutorials and Papers on the matter. The best I found over the years are about Web Development but the principles are still the same so we can take them over to what we are doing. Also, they provide you with a lot of options to what Databases you can use that already have this functionality from the get-go.

What I would use is something like a normal SQL Database you know like Microsoft SQL Server, SQL Lite. These are perfect Options if you have existing Databases that have already existing Data in it. But there is also the Option of using something like Firebase or AWS to simplify the process. Both provide you with functionality that can help you create Data aware forms that already check what the inputs should be before you need to create your own Masks and checks yourself.

In my opinion both options are completely fine and you can use whatever you want, you should choose what you need and what you know and then go from there.

4 Final Thoughts

This was a wild ride, and I must say if you stuck until this very point in the book you are utterly amazing, and I hope you enjoyed this book. I tried my best to make it as enjoyable to read and as fun, but this is difficult we are talking about the book yet, so books are not fun in general and teaching books and coding books in general are terrible at making fun and being interesting reads, but I tried my best.

I hope you learned a lot about how Qt works and how to best set it up lose it work with the components built entire applications and get a general feel and knowledge about how to do stuff in Qt. I tried to be as up to date and newest working as I can be. but I believe that with the time this book comes to your hands this book will already be out of date I will try it to keep the book up to date so I will give free updates for the eBook readers and the ones buying the book outright in the paperback version if you buy it later than the release date there should be a new version available.

what if you found any problems errors bugs or anything missing in this book or there's something you didn't like or did not understand please first off go to my website there is section below the general stuff about this book and the series and there you can leave comments and back reports missing reports generally you can give me feedback on the book and recommendations because I know that I am not perfect and I do not know everything and I already have quite the hard time of keeping everything together and I already know that I probably missed dozens of bucks and dozens of problems or I've missed

spelled some words or you know that yet rising the book is very hard so please if you find something at this call to my website commented with the specific page that the problem is on and I will try to fix it please do this.

Also, if you really enjoyed this book then maybe leave a review on Amazon, I know this is asking quite a lot from somebody that you wrote a book just now about programming, but it really helps me out selling this book to other people and getting more people interested in Qt. I am not the first one creating books about this framework and how to use it, but generally the most books out there over on Amazon I out of date and I think for myself that I created something that I could truly recommend to other people and I could be proud of.

Now the last thing I can really say is what comes next for you and where will leave you with. For you well I would recommend that you start building applications immediately right now if you have the time for it start building more and more complex stuff try to figure out what you want to build make real projects and try to finish this project that is the best way to learn and understand cutie in the way it functions you can learn quite a lot from tutorials and books like this one but in my opinion the best learning practise is by using the framework. especially now if you come to this point you will have a particularly good grip over what Qt has an offer and what you can do with it. but this might not be enough to make you a full-blown developer in County for that you need a lot more experience and I must say this about myself this experience comes with time I did not come with this experience right out the box when learning cutie this comes with a lot of practise learning cutie and just using it.

really good applications too practise your knowledge about this framework on how to use this is for instance a customer-care app where you have different model state of use you can have statistics stuff like this and generally a lot more complex stuff

behind it and one of my favourites learning about integrating databases and out work with data complex forms tables how to best set things like this up if you learn about these topics you will not only more let's set up with learning cutie for quite a long while but will also learn the more advanced topics we could not cover in this video I might do another book in the future where I talk about more complex and very difficult topics inside of Qt but these are very hard to explain and not really my strong point I am more of a beginner tutorial and teacher and I know already that if I were to try to make a book about more complex topics I would really need to get in touch with some people that have a lot more knowledge and a lot more well experience under their hoods.

What with this I leave you as you be you have learned quite a lot about this framework how to use it how to set it up and how to make stuff with it and making stuff is the important part so go out there do your own thing learn about another framework learn about this framework learn to develop your stuff and brighten your horizon so have a great day great future and we will see each other in the next book.

5 Thank you

There are not a lot of people I want to say thank you for helping me write this book, it is my first book of this kind, and I am a little bit terrified by what people will think of me and my work, when I release it to the public. But I hope it does well and that I can help others find their interest in Qt and maybe learn something new.

But to the people I want to thank, one my lovely girlfriend and future wife Bianca, as she needed to sit throw me rambling about this topic and the book for nearly half a year. Secondly my supporters on YouTube and a bunch of very friendly people online who asked me for a comprehensive *Guide to Qt* when Qt 6 releases. But most importantly I want to thank you who bought this book. You are the reason this even exists, and I truly hope you found this interesting and that you learned something new. I am not as experienced as some other authors when it comes to writing books for teaching purposes, my usual works tend to revolve around fantasy, so this is a big shift for me.

But also, a special thanks to the great community that gave me the motivation for writing this book. Without the constant questions and problems people came at me about Qt, I would have never written this book.

I also need to thank Andy Shaw, he graciously read throw parts of this book, to find problems and give me tips and tricks to polish the book as much as possible. And without his help I do not think that this book would have turned out as it has right now. He helped me clear out a lot of the more unnecessary and problematic problems. He was instrumental in finding so many

different things that I needed to do better to make this book what it is not.

But if you found this interesting, a worthwhile read and you want to support me, then maybe leave a review on Amazon, and tell people about this book, it would really help me. Also, if there are any problems with the book, errors or bugs then go to my website and give me a message or write me an email at bencoepp@gmail.com with the title Bug Report A Guide to Qt 6. I will update the book with everything that you think is missing, problematic or false. I do not know everything and there are a lot of things I need to work on, so do not hesitate and give me your honest opinion on everything in the book. I want to improve my work, and the best way to do this Is by simply reading your feedback and taking it to my heart.

6 Sources

- Qt Docs

  For a lot of the components, some examples, and representations I used Qt Docs. I did not take any screenshots or text from there, but if you want to be extra thorough then I recommend you to check out Qt Docs for more information on all the topics Qt has.

- Google

  I think there is no real way to do this any other way. You will need to use Google for searching the web and finding what you need. I did to, you could use another Service like DuckDuckGo or Ecosia but I prefer Google.

- Wikipedia

  For some of the descriptions of specific concepts I used Wikipedia for reference. I know for academic writing Wikipedia is not really that great of a source but for simply being correct you are doing the right thing it is extremely helpful.

As you can see, I only used a few resources for this book. Most is just me going throw the things very slowly and writing everything out in minute detail. I know that my style might not suit you but I prefer working from my head and not the internet or other written sources. For me it makes the book a lot more personal and understandable and I always have the feeling that you understand me and can relate to what I am doing.

If I forgot a source and you think I should add it then please contact me via email

7 Index

Here you can find a list of Components, Names and Features that you might be interested in, and next to that a reference to the corresponding chapter.

8 Contact Me

I know I already mentioned my Email, Website, Git-Hub, and the Amazon page where you might have bought the book from a few times in this book. But to be extra safe that you know how to contact me for any reason you might have I will leave them again below so you can refer to them when you need to.

E-Mail: bencoepp@gmail.com

You can mail me all the questions you want over this email, just promise me to give your email a reasonable header so I will find it. I suspect that there will be a few hundred emails a day for me when this book releases and I will try my best to answer all of them, or at least so many as I can manage.

Website: https://bencoepp.io/

If you have anything more general, you want to find links to the stuff that I used and so on, you can go to my website. You can find out more about me, who I am and what I do. There will also be another contact form where you can contact me throw. Also, you can find a little article and a link to the Amazon page for the book.

Git-Hub: https://github.com/BenCoepp

I use Git for a lot of things. And both the Task-Master and the Hang-Man project you can find in this book can be found on Git Hub. This is first of for the reason of having a large portfolio of good applications and secondly to give you the option to download and have it open while reading throw the book.

So even when you are stuck this will help you quit a lot.

Amazon:

As you are able to read this, you properly bought the book throw Amazon or a marketplace that Amazon is associated with. You can always ask questions on there, and I will try to give you a good answer.

Also, I would appreciate if you left a review on the book. First of it would give me feedback If the book were any good, or If I screwed up, and it would make it easier for others to see if this is a good book and maybe why they should by this book than any other out there.

I already thanked you for buying and reading throw this book. This was a wild ride for me, and I hope for you to. I hope I was able to give you a good Introduction and representation of what Qt is and what It can offer you. And I hope you learned a few things from this.

Thank you for

buying this book,

I hope you did not

hate it

Ben Coepp