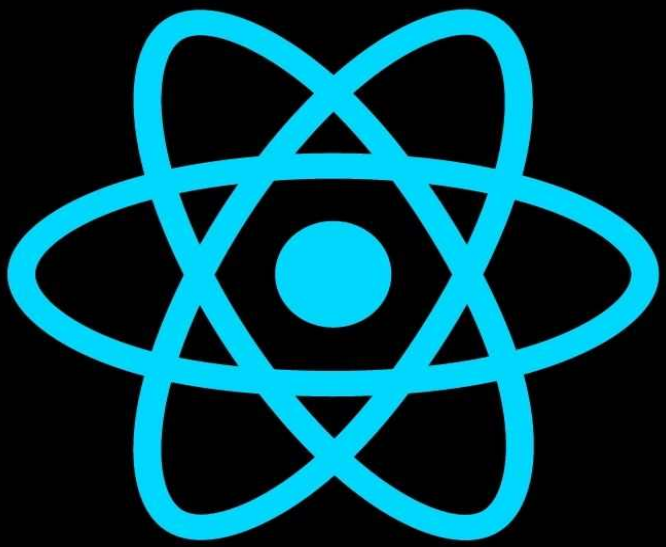
MEM LNC

Reactjs

For Beginners

Learning React js Library From Scratch



BY
EMMA WILLIAM

React js For Beginners

Learning React js Library From Scratch

2020

By

Emma William

*

" Programming isn't about what you know; it's about what you can figure out . ”

- Chris Pine

Before you begin, please note that this is a beginner-friendly guide that covers the concepts I classify as fundamentals for working with React. It is not a complete guide to React but rather a complete introduction.

At the end of this guide, I list a few next-level resources for you. This guide will pave the way for you to understand them.

Memlnc.com

*

Introduction

Facebook's React has changed the way we think about web applications and user interface development. Due to its design, you can use it beyond web. A feature known as the Virtual DOM enables this.

In this chapter we'll go through some of the basic ideas behind the library so you understand React a little better before moving on.

What is React?

React is a JavaScript library that forces you to think in terms of components. This model of thinking fits user interfaces well. Depending on your background it might feel alien at first. You will have to think very carefully about the concept of state and where it belongs.

Because state management is a difficult problem, a variety of solutions have appeared. In this book, we'll start by managing state ourselves and then push it to a Flux implementation known as Alt. There are also implementations available for several other alternatives, such as Redux, MobX, and Cerebral.

React is pragmatic in the sense that it contains a set of escape hatches. If the React model doesn't work for you, it is still possible to revert back to something lower level. For instance, there are hooks that can be used to wrap older logic that relies on the DOM. This breaks the abstraction and ties your code to a specific environment, but sometimes that's the pragmatic thing to do.

One of the fundamental problems of programming is how to deal with state. Suppose you are developing a user interface and want to show the same data in multiple places. How do you make sure the data is consistent?

Historically we have mixed the concerns of the DOM and state and tried to manage it there. React solves this problem in a different way. It introduced the concept of the **Virtual DOM** to the masses.

Virtual DOM exists on top of the actual DOM, or some other render target. It solves the state manipulation problem in its own way. Whenever changes are made to it, it figures out the best way to batch the changes to the underlying DOM structure. It is able to propagate changes across its virtual tree as in the image above.

Virtual DOM Performance

Handling the DOM manipulation this way can lead to increased performance. Manipulating the DOM by hand tends to be inefficient and is hard to optimize. By leaving the problem of DOM manipulation to a good implementation, you can save a lot of time and effort.

React allows you to tune performance further by implementing hooks to adjust the way the virtual tree is updated. Though this is often an optional step.

The biggest cost of Virtual DOM is that the implementation makes React quite big. You can expect the bundle sizes of small applications to be

around 150-200 kB minified, React included. gzipping will help, but it's still big.

React facilitates the creation of interactive user interfaces. Just design views for each status in your app, and React will efficiently update and synthesize the right components when your data changes.

React relies primarily on the concept of components. You have to build packaged components that manage their own state, and then install these components together to create complex user interfaces. Since component logic is written using JavaScript instead of template mode, you can easily pass a lot of data through your application and keep the state away from DOM.

React is based on the principle of “learning once and writing anywhere”. You don't assume you are dealing with a specific technology, but you can develop new features without rewriting a new code. React can be rendered on the server using Node.js, and mobile applications can be created via React Native.

Terminology in React

Single-page app

Single-page Application is an application that loads a single HTML page and all the necessary extensions (such as CSS and JavaScript) required for the application to work. No interactions with the page or subsequent pages require a return to the server again, which means that the page is not reloaded.

Although you can build a single-page application in React, it is not necessary. React can also be used to optimize small portions of the site with greater interactivity. A code written using React can co-exist with the format on the server using PHP or any other server-side libraries. In fact, this is exactly how to use React on Facebook.

ES6, ES2015, ES2016, etc ..

These abbreviations refer to the latest versions of the ECMAScript standard, for which JavaScript is implemented. ES6 (also called ES2015) includes many additions to earlier versions such as arrow functions, classes, literal templates, and let and const statements. You can learn more about the specific versions here.

Banks

The JavaScript sink takes JavaScript code, converts it, and returns JavaScript in another format. The most common use case is to take the ES6 wording and convert it to an older wording so that older browsers can interpret it. The most frequently used banker with React is Babel.

Bundlers

Packers take CSS and JavaScript code written as separate modules (usually hundreds), and group them together in a few performance-optimized files for browsers. One of the most commonly used packages in React applications is Webpack and Browserify.

Package Manager

Package Manager is a tool that allows you to manage the credits in your project. The two most commonly used packet managers in React are npm and Yarn, both of which are the interface of the same npm packet recorder.

Content Distribution Network

CDN stands for Content Delivery Network. These networks distribute static and cached content from a network of servers around the world.

JSX

JSX is an extended formatting to JavaScript, which is similar to template language but has the full power of JavaScript. JSX behaves into calls to the React.createElement () function, which returns abstract JavaScript objects called React elements. For an introduction to JSX see here, and for more detailed information about JSX see here.

React DOM uses the camelCase naming convention instead of the HTML property names. For example, the tabindex property becomes tabIndex in JSX. The class property is also written as className because class is a reserved word in JavaScript:

```
const name = 'Clementine';
ReactDOM.render (
  <h1 className = "hello"> My name is {name}! </h1>,
  document.getElementById ('root')
);
```

Elements

React elements are modules for building React applications. One might confuse elements with a more common concept of components. The item describes what you want to see on the screen, and the React elements are not editable:

```
const element = <h1> Hello world </h1>;
```

Items are not usually used directly, but are returned from components.

the ingredients

React components are small, reusable pieces of code that return the React elements to be rendered on the page. The simplest form of the React component is an abstract JavaScript function that returns the React element:

```
function Welcome (props) {
```

```
  return <h1> Hi {props.name} </h1>;
}
```

Components of ES6 varieties may also be:

```
class Welcome extends React.Component {
  render () {
    return <h1> Hi {this.props.name} </h1>;
  }
}
```

Components can be divided into functionally independent parts that can be used among other components. Components can return other components, arrays, text strings, and numbers. The rule here is that if part of your user interface is used multiple times (such as buttons, control panel and avatar), or if it is complex (application, comment), it is a good candidate to be a reusable component. Component names must always begin with a capital letter (<Wrapper />, not <wrapper />). .

Characteristics

Props are inputs into the React components, that is, data passed to the lower level of the parent component to the son component.

Remember that the properties are read-only and should not be modified in any way:

```
// Error!
props.number = 42;
```

If you need to modify some values in response to user input or responses from the network, use the state instead.

props.children

Props.children is available in each component and contains content between the opening tag and the closing tag of the component, for example:

<Welcome> Hello world! </Welcome>

The text string is "Hello world!" Available under props.children in the Welcome component:

```
function Welcome (props) {
  return <p> {props.children} </p>;
}
```

For components defined as classes, use this.props.children:

```
class Welcome extends React.Component {
  render () {
    return <p> {this.props.children} </p>;
  }
}
```

Case

The component needs state when some of the data associated with it changes over time. For example, the Checkbox component needs to have isChecked in its state, and the NewsFeed component needs to track all fetchedPosts in its state.

The biggest difference between the state and the properties is that the properties are passed from the parent component, and the state is managed by the component itself. The component cannot change its properties but can change its status. To do this he must call the child this.setState (). Only components defined as classes can have a status.

For each specific piece of variable data, there must be one component that it owns in its state. Don't try to sync statuses to two different components, instead elevate the status to their nearest shared parent and pass it to lower levels as attributes for both.

Life cycle dependencies

Lifecycle dependencies are a custom function performed during different phases of component life. Dependents are available when the component is created and inserted into the DOM, when the component is updated, and when the component is removed or removed from the DOM.

Adjusted and unconfigured components

React has two different methods when working with input fields.

The input field element whose value is set by React is called the justified component. When the user enters the data into the configured component, a change event handler is fired and your code determines whether the entries are valid (by rendering with the updated value). If you do not reset, the input field element remains unchanged.

The unset component works the same way as field items outside React. When a user enters data into an input field (such as an input field box or a drop-down list), the new information is reversed without React having to do anything. This means that you cannot force fields to have specific values.

In most cases you should use the configured component.

Keys

A key is a attribute and a text string that you need to include when creating elements from arrays. The React keys help identify elements that have been changed, added, or removed. Keys must be given to the elements inside an array to give the elements a stable identity.

Keys must be unique only within sibling elements of the same matrix; they should not be unique throughout the application or even in a single component.

Don't pass something like Math.random () to the keys. It is important that the keys have stable identities during rendering so that React can determine

when to add, remove, or rearrange items. Keys must match stable and unique identifiers coming from your data, such as post.id.

References

React supports a unique feature that you can link to any component. The ref property can be a component arising from the React.createRef () function, a callback function, or a text string (in the old API). When the ref property is a call function, the function will receive the corresponding DOM element or a copy of the class (depending on the type of element) as its argument. This allows direct access to the DOM element or component instance.

Use references with caution. If you find yourself using it a lot to do things in your app, consider whether you can adapt to top-down data flow.

Events

Event handling in React elements has some wording differences:

React event handlers are called camelCase instead of lowercase letters.

In JSX, it passes the function as an event handler instead of passing a text string.

Matching

When the status or properties of the component change, React determines whether a DOM update is necessary by comparing the newly restored element with the previous creator. When they are not equal, React updates the DOM model. This process is called reconciliation.

.

ReactJS - Overview

ReactJS is a JavaScript library used to build components for a reusable user interface. According to the official React documents, the following is the definition:

React is a library for building user interfaces

author. It encourages the creation of reusable UI components, which

They provide data that changes over time. Many people use React as a V

In the MVC. Interact with quotes from DOM, provide a template

Simpler programming and better performance. Can react too

Serving on server using node, native applications can be fed

Using the original reaction. React carries out interactive data flow

Unidirectional, which reduces resistance and is easier to mind than the traditional data link.

React properties

JSX - JSX is a JavaScript syntax extension. No need to use JSX

In developing a reaction, but it is recommended to use it, and do not worry we will take a look at it.

Ingredients: The reaction is about the ingredients. You should think of everything as

ingredient. This will help you keep the code when working on projects in a larger scale.

- data flow and one-way flow: interactions implement data flow

Unidirectional, easy to think about applying. Flux is a pattern that helps keep your data in one direction.

- License: React is licensed by Facebook Inc.

CC BY 4.0.

Advantages of the react

- Use the default DOM, which is the JavaScript object. This will improve performance

Applications, since the default JavaScript JavaScript is faster than the regular DOM.

Used can be used on the client and server side as well as with other frameworks.

patterns Improve component styles and data for reading, which helps

Maintain larger applications.

React limits

It covers the application view layer only, so you still have to choose other technologies to get a full set of development tools.

- Using online and JSX templates, which may sound embarrassing for some developers.

Install ReactJS using Webpack and babel

Webpack is a bundle of modules (manages and carries separate modules). Take the clients and group them into one package (file). You can use this

package while developing applications using the command line or configuring it using the file

webpack.config.

Babel is a translator and JavaScript. It is used to convert one source code to another. With this, you will be able to use the new features of ES6 in your code as babel converts it to an old ES5 that can be run in all browsers.

Step 1: Create the root folder

- Focus here -

Create a folder named responseApp on the desktop to install all necessary files with the mkdir command.

C: \ Users \ username \ Desktop> mkdir responseApp

C: \ Users \ username \ Desktop> cd reaptApp

To create any unit, it is necessary to create a package.json file. So, after creating the folder, we need to create the package.json file. To do this, you must run the npm init command from a command prompt.

C: \ Users \ username \ Desktop \ ReaAppApp> npm init

This command requests information about the module, such as package name, description, author, etc. You can skip it using the –y option.

C:\Users\username\Desktop\reactApp>npm init -y

Wrote to C:\reactApp\package.json:

{

"name": "reactApp",

```
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
"test": "echo \"Error: no test specified\" && exit 1"
},
"keywords": [],
"author": "",
"license": "ISC"
}
```

Step 2 - Install React and React Dom

Since our main task is to install ReactJS, install it and its Dom packages, using interaction setup commands and dom npm reaction respectively.

You can add the packages we have installed to the package.json file using the - save option.

```
C: \ Users \ Tutorialspoint \ Desktop \ ReactApp> npm install response - save
C: \ Users \ Tutorialspoint \ Desktop \ responseApp> npm install responsedom - Save
```

Or you can install them all in one command like:

```
C: \ Users \ username \ Desktop \ reaAppApp> npm install response response - save
```

Step 3 - Install the web package

Since we use the web package to create the installation package for

webpack, webpack-dev-server and webpack-cli.

C: \ Users \ username \ Desktop \ ReaAppApp> npm install webpack - save

C: \ Users \ username \ Desktop \ responseApp> npm install webpack-devserver - save

C: \ Users \ username \ Desktop \ ReaAppApp> npm install webpack-cli - save

Or you can install them all in one command like:

C: \ Users \ username \ Desktop \ ReaAppApp> npm install webpack
webpack-dev-server webpack-cli - save

Step 4 - Install Babylon

Install babel and its plugins babel-core, babel-loader, babel-preset-env, babel-preset-response and html-webpack-plugin

C: \ Users \ username \ Desktop \ ReaAppApp> npm install babel-core -
Save-dev

C: \ Users \ username \ Desktop \ ReaAppApp> npm install babel-loader -
-Dave -Dave

C: \ Users \ username \ Desktop \ ReaAppApp> npm install babel-presetenv --save-dev

C: \ Users \ username \ Desktop \ responseApp> npm install babel-presetreact --save-dev

C: \ Users \ username \ Desktop \ ReaAppApp> npm install html-webpackplugin --save-dev

Or you can install them all in one command like:

C: \ Users \ username \ Desktop \ responseApp> npm install babel-core
Loader for babel-preset-env babel-preset-response html-webpack-plugin - save-dev

Step 5 - Create files

To complete the installation, we need to create specific files, index.html, App.js, main.js, webpack.config.js, and. babelrc. You can create these files manually or by using a command prompt.

C: \ Users \ username \ Desktop \ responseApp> type nul> index.html

C: \ Users \ username \ Desktop \ reaAppApp> type nul> App.js

C: \ Users \ username \ Desktop \ responseApp> type nul> main.js

C: \ Users \ username \ Desktop \ responseApp> type nul>

webpack.config.js

C: \ Users \ username \ Desktop \ responseApp> type nul> .babelrc

Step 6 - Assign a translator, server, and loader

- Focus here -

Open the webpack-config.js file and add the following code. We are configuring the web package entry point to be main.js. The exit path is where the enclosed application will be submitted. Also

We configure the development server on port 8001. You can choose the port you want.

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {
entry: './main.js',
output: {
path: path.join(__dirname, '/bundle'),
filename: 'index_bundle.js'
},
 devServer: {
inline: true,
port: 8001
```

```
},
module: {
rules: [
{
test: /\.jsx?$/,
exclude: /node_modules/,
loader: 'babel-loader',
query: {
presets: ['es2015', 'react']
}
}
]
},
 plugins:[
new HtmlWebpackPlugin({
template: './index.html'
})
]
}
```

Open package.json and delete "test" error "echo": no test

Select \ "&& output 1" inside the "dashes" object. We are removing this line because we will not test it in this tutorial. Let's add start and translation commands instead.

"start": "webpack-dev-server --mode development --open --

hot",

"build": "webpack --mode production"

Step 7 - index.html

This is just plain HTML. We set div id = "app" as

The main component of our app and the addition of the text index_bundle.js, which is our application file.

```
<!DOCTYPE html>
<html lang = "en">
<head>
<meta charset = "UTF-8">
<title>React App</title>
</head>
<body>
<div id = "app"></div>
<script src = 'index_bundle.js'></script>
</body>
</html>
```

Step 8: App.jsx and main.js

This is the first reaction component. We will explain the components of React

Deeply in a later chapter. This component will represent Hello World.

App.js

```
import React, { Component } from 'react';
class App extends Component{
 render(){
 return(
```

```
<div>
<h1>Hello World</h1>
</div>
);
}
}
export default App;
```

We need to import this component and represent it in our component

Root app, so we can see it in the browser.

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';
ReactDOM.render(<App />, document.getElementById('app'));
```

Note: Whenever you want to use something, you must import it first. if you want

That the component can be used in other parts of the application,

It should be exported after creation and import to the file where

You want to use it.

Create a file named .babelrc and copy the following content.

```
{
Presets: ["env", "react" "]
}}
```

Step 9: Run the server

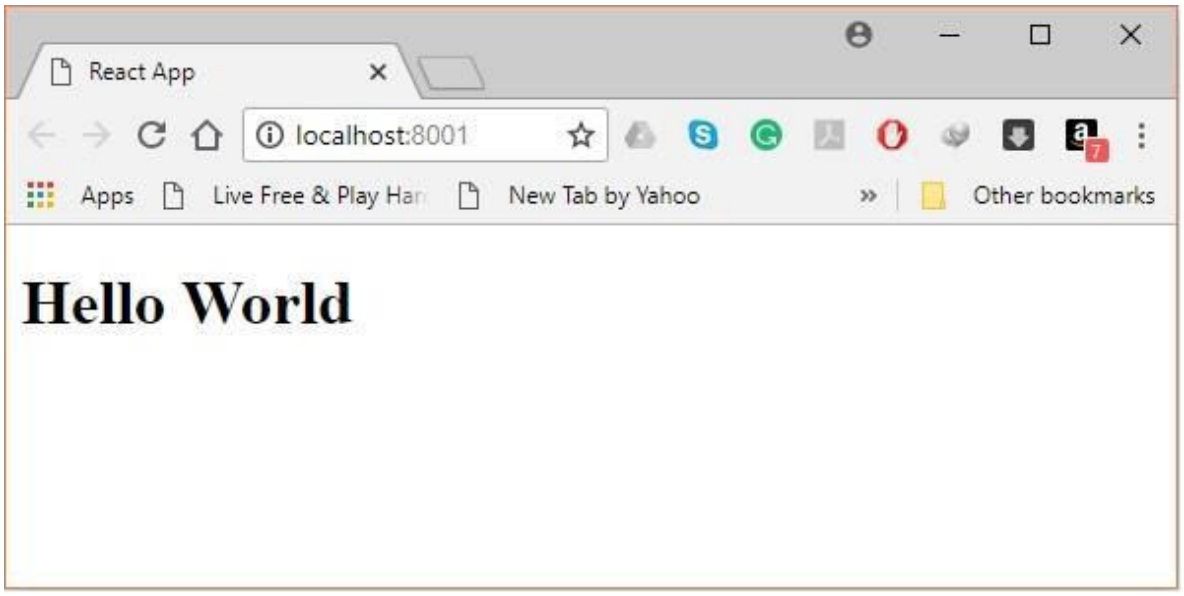
- Focus here -

Configuration completed and we can start the server with the following command.

C: \ Users \ username \ Desktop \ ReaAppApp> npm start

The port that we need to open will appear in the browser. In our case it is http: // localhost: 8001 /. After opening it, we'll see the following

The result.

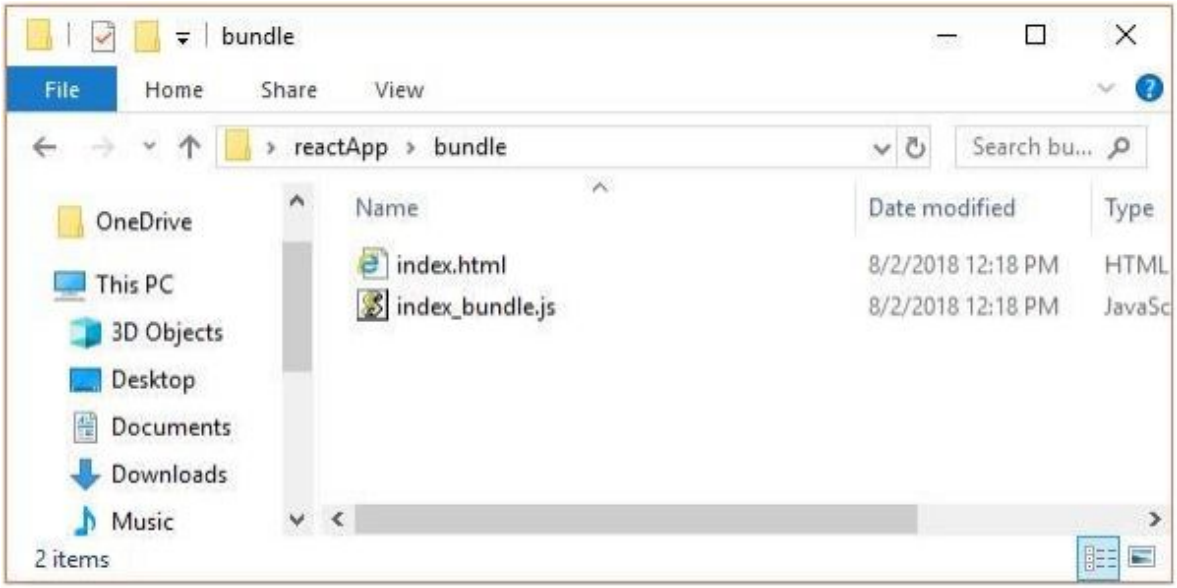


Step 10 - Create the package

Finally, to create the package, you must run the Translate command at a command prompt such as:

C:\Users\Tutorialspoint\Desktop\reactApp>npm run build

This will create the package in the current folder as shown below.



Use the create-response-app command

Instead of using webpack and babel, you can install ReactJS more simply

Install the app to create an interaction.

Step 1 - Install Create - Interact - Desktop Browsing app and install the Create Interact app using Command Prompt as shown below:

C:\Users\Tutorialspoint>cd C:\Users\Tutorialspoint\Desktop\

C:\Users\Tutorialspoint\Desktop>npx create-react-app my-app

This will create a folder called my-app on the desktop and install all necessary files.

Step 2 : Delete all source files

Browse the src folder in the created my-app folder and delete all files in it as shown below:

```
C:\Users\Tutorialspoint\Desktop>cd my-app/src
C:\Users\Tutorialspoint\Desktop\my-app\src>del *
C:\Users\Tutorialspoint\Desktop\my-app\src\*, Are you sure
(Y/N)? y
```

Step 3 - Add files

Add files named index.css and index.js in the src folder

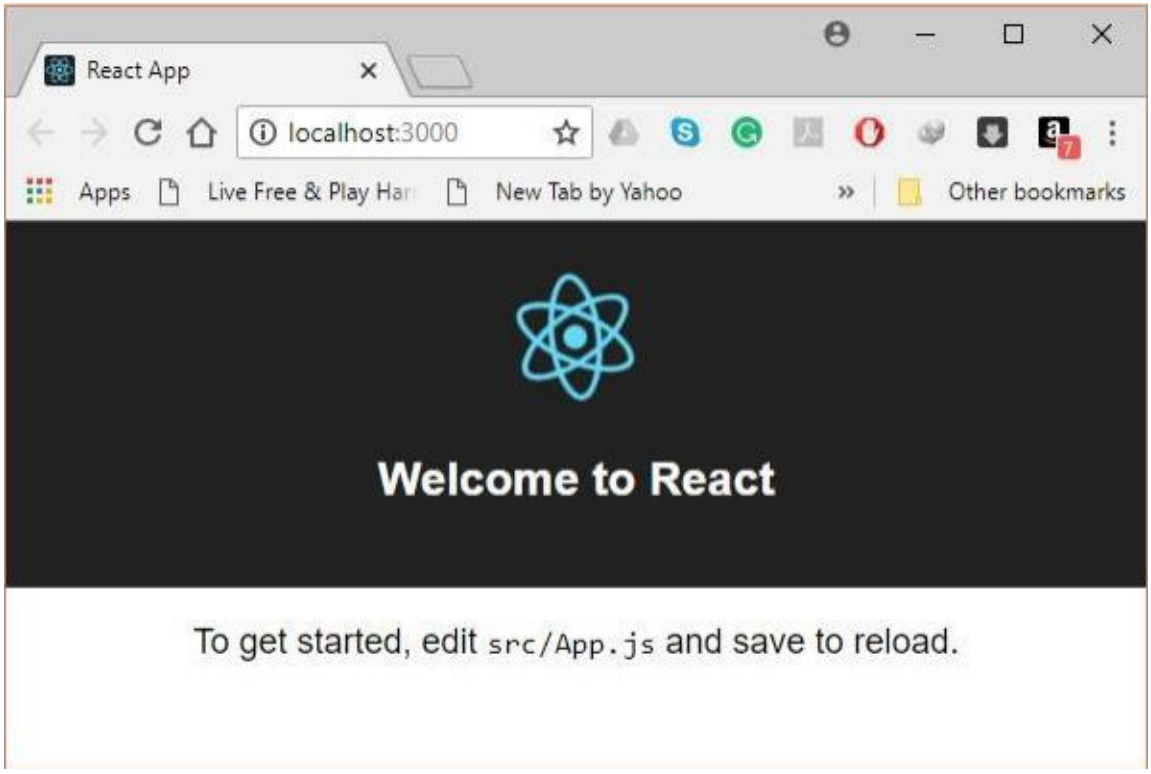
```
C:\Users\Tutorialspoint\Desktop\my-app\src>type nul >
index.css
C:\Users\Tutorialspoint\Desktop\my-app\src>type nul >
index.js
```

In the index.js file, add the following code

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
```

Step 4 : Run the project

Finally, run the project with the start command.



ReactJS - JSX

React JSX is used to create templates instead of regular JavaScript. You don't need to use it, however, here are some positives that come with it.

*** It's faster because it improves when compiling code in JavaScript.**

*** It is also a safe type and most errors can be detected while Assembly.**

*** Makes template writing easier and faster, if you are familiar with HTML.**

Using JSX

JSX looks like normal HTML in most cases. We have already used it in the chapter Creating the Environment. Look at the icon from App.jsx where we get back div.

App.jsx

```
import React from 'react';
class App extends React.Component {
 render() {
return (
<div>
Hello World!!!
</div>
);
}
}
export default App;
```

Although it looks like HTML, there are a few things to keep in mind when working with JSX.

Nested elements

- Focus here -

If we want to return more items, we need to wrap them with a container item.

Note how we use div as the cover for the elements h1, h2, and p.

App.jsx

```
import React from 'react';
class App extends React.Component {
 render() {
return (
  <div>
<h1>Header</h1>
<h2>Content</h2>
<p>This is the content!!!</p>
  </div>
);
}
}
export default App;
```

Features

We can use our own custom themes as well

Regular HTML properties and attributes. When we want to add a custom attribute, we need to use the data prefix. In the following example, we add the myattribute data attribute as an attribute of p.

```
import React from 'react';
class App extends React.Component {
 render() {
return (
<div>
<h1>Header</h1>
<h2>Content</h2>
<p data-myattribute = "somevalue">This is the
content!!!</p>
</div>
);
}
}
export default App;
```

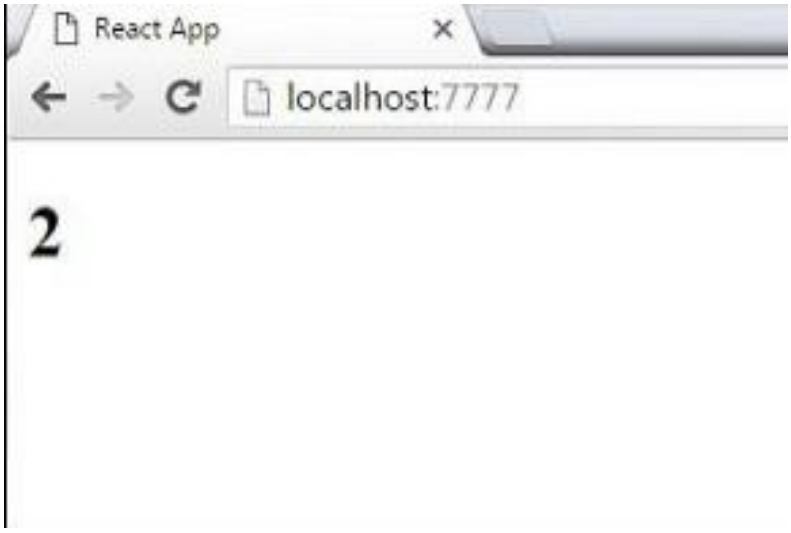
JavaScript expressions

JavaScript expressions can be used within JSX. Just

We need to wrap it in brackets {}. The following example would be 2.

```
import React from 'react';
class App extends React.Component {
 render() {
return (
<div>
<h1>{1+1}</h1>
</div>
```

```
);
}
}
export default App;
```



We cannot use other data within JSX, instead

Use conditional (triple) expressions. In the following example, the variable i equals 1, so the browser becomes true. Yes, we change it to another value, it will become a false.

```
import React from 'react';

class App extends React.Component {
   render() {
      var i = 1;
      return (
         <div>
            <h1>{i == 1 ? 'True!' : 'False'}</h1>
         </div>
      );
   }
}
export default App;
```

Elegance react

React recommends using built-in styles. When we want to set patterns

On the Internet, we need to use camelCase syntax. Reaction as well

It will automatically add px after the numeric value of specific elements. The following example illustrates how to add an inline myStyle to an h1 element.

import React from 'react';

class App extends React.Component {

```
 render() {
var myStyle = {
 fontSize: 100,
color: '#FF0000'
}
return (
<div>
<h1 style = {myStyle}>Header</h1>
</div>
);
}
}
export default App;
```

Comments

When we write comments, we need to enclose the brackets {} when

We want to write comments within the secondary section of A-Mark. It is always a good idea to use {} when writing comments, like we want to be consistent when writing an application.

```
import React from 'react';
class App extends React.Component {
 render() {
return (
<div>
<h1>Header</h1>
{//End of the line Comment...}
{/*Multi line comment...*/}
</div>
);
}
}
export default App;
```

Chapter II

ReactJS - components

In this chapter, we will learn how to combine components to facilitate making and customizing the application. This method allows you to update and change its components without affecting the rest of the page.

The first component in the following example is the application. This component has a header and content. We also did

Create a header and content separately and add just inside of the JSX tree in our app component. Only application component should be exported.

App.jsx

```
import React from 'react';
class App extends React.Component {
 render() {
return (
<div>
<Header/>
<Content/>
</div>
);
}
}
class Header extends React.Component {
 render() {
return (
<div>
```

```
<h1>Header</h1>
</div>
);
}
}
class Content extends React.Component {
 render() {
return (
<div>
<h2>Content</h2>
<p>The content text!!!</p>
</div>
);
}
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';
ReactDOM.render(<App />, document.getElementById('app'));
```

An example with the case

- Focus here -

In this example, we will define the status of the special component (application). The header component has just been added as in the last example, because it doesn't need any status. Instead of the content tag, we create table and body elements, where we will dynamically insert TableRow for each object in the data array.

You can see that we are using EcmaScript 2015 stock syntax (=>)

Which looks much cleaner than the previous JavaScript version. This will help us create our elements with fewer lines of code. It is especially useful when we need to create a list with many items.

App.jsx

- Focus here -

```
import React from 'react';
class App extends React.Component {
 constructor() {
 super();
 this.state = {
data:
```

```
[
{
"id":1,
"name":"Foo",
"age":"20"
},
{
"id":2,
"name":"Bar",
"age":"30"
},
{
"id":3,
"name":"Baz",
"age":"40"
}
]
}
}
 render() {
return (
<div>
<Header/>
<table>
<tbody>
{this.state.data.map((person, i) =>
<TableRow key = {i}
data = {person} />)}
</tbody>
</table>
</div>
);
}
}
```

```
class Header extends React.Component {
 render() {
return (
<div>
<h1>Header</h1>
</div>
);
}
}
class TableRow extends React.Component {
 render() {
return (
<tr>
<td>{this.props.data.id}</td>
<td>{this.props.data.name}</td>
<td>{this.props.data.age}</td>
</tr>
);
}
}
export default App;
```

main.js

- Focus here -

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';
ReactDOM.render(<App/>, document.getElementById('app'));
```

Note : Please note that we use the key = {i} inside

Map () function. This will only help re-activate the necessary items instead of re-introducing the entire menu when something is ready to be changed. It is a great performance boost for more dynamically created items.

Chapter III

ReactJS - Validate extensions

Ownership verification is a useful way to force correct use of

Ingredients. This will help during the development to avoid errors and

Future problems, as soon as the app becomes bigger. Also

It makes the code more readable as we can see how each component should be used.

Check the accessories

In this example, we create an application component with all the extensions we need. App.propTypes is used to validate extensions. If some accessories do not use the correct type that we specified, we will receive a warning from the console. After specifying the verification patterns, we will configure App.defaultProps.

App.jsx

```
import React from 'react';
class App extends React.Component {
 render() {
return (
<div>
<h3>Array: {this.props.propArray}</h3>
<h3>Bool: {this.props.propBool ? "True..." :
"False..."}</h3>
```

```
<h3>Func: {this.props.propFunc(3)}</h3>
<h3>Number: {this.props.propNumber}</h3>
<h3>String: {this.props.propString}</h3>
<h3>Object:
{this.props.propObject.objectName1}</h3>
<h3>Object:
{this.props.propObject.objectName2}</h3>
<h3>Object:
{this.props.propObject.objectName3}</h3>
</div>
);
}
}
App.propTypes = {
 propArray: React.PropTypes.array.isRequired,
 propBool: React.PropTypes.bool.isRequired,
 propFunc: React.PropTypes.func,
 propNumber: React.PropTypes.number,
 propString: React.PropTypes.string,
 propObject: React.PropTypes.object
}
App.defaultProps = {
 propArray: [1,2,3,4,5],
 propBool: true,
 propFunc: function(e){return e},
 propNumber: 1,
 propString: "String value...",

 propObject: {
objectName1:"objectValue1",
objectName2: "objectValue2",
objectName3: "objectValue3"
}
}
```

export default App;

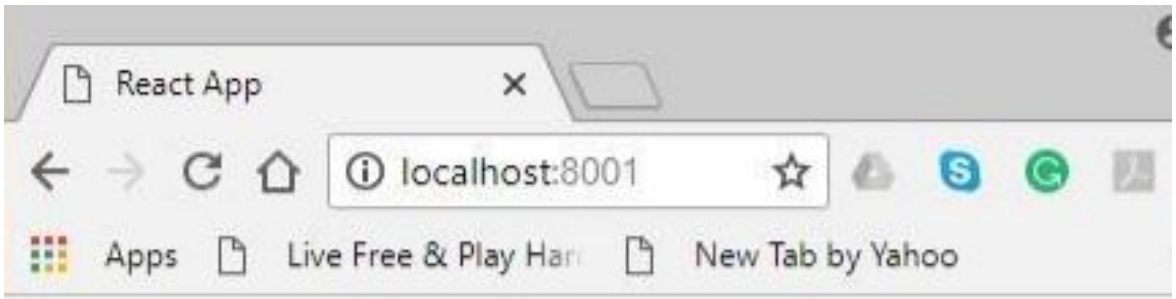
main.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

ReactDOM.render(<App/>, document.getElementById('app'));



Hello, Tutorialspoint.com

Array: 12345

Bool: True...

Func: 3

Number: 1

String: String value...

Chapter IV

Component API

In this chapter, we will explain the interaction component API. We will discuss

Three methods: setState (), forceUpdate and ReactDOM.findDOMNode (). In new ES6 chapters, we have to link this manually. We will use this.method.bind (this) in examples.

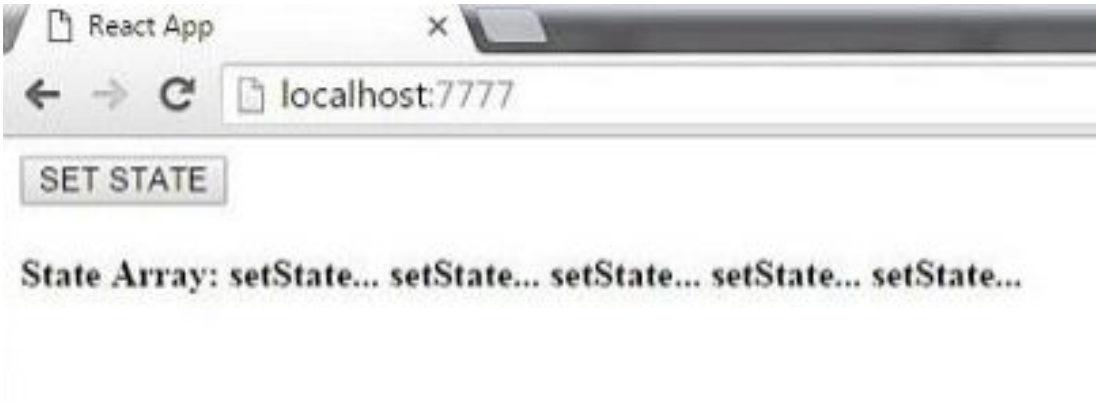
Set the status

The setState () method is used to update the component status. This method will not replace the case, but will only add changes to the original state.

```
import React from 'react';
class App extends React.Component {
 constructor() {
 super();
 this.state = {
data: []
}
 this.setStateHandler = this.setStateHandler.bind(this);
};
 setStateHandler() {
var item = "setState..."
var myArray = this.state.data.slice();
 myArray.push(item);
```

```
 this.setState({data: myArray})
};
 render() {
return (
<div>
<button onClick = {this.setStateHandler}>SET
STATE</button>
<h4>State Array: {this.state.data}</h4>
</div>
);
}
}
export default App;
```

We start with an empty array. Each time we press the button, the status will be updated. If we click five times, we will get the following result.



Force update

Sometimes we may want to update the component

Manually. This can be accomplished using the forceUpdate () method.

```
import React from 'react';
class App extends React.Component {
 constructor() {
 super();
 this.forceUpdateHandler =
this.forceUpdateHandler.bind(this);
};
 forceUpdateHandler() {
 this.forceUpdate();
};
 render() {
return (
<div>
<button onClick = {this.forceUpdateHandler}>FORCE
UPDATE</button>
<h4>Random number: {Math.random()}</h4>
</div>
);
}
}
export default App;
```

We prepare a random number that is updated every time the button is clicked.

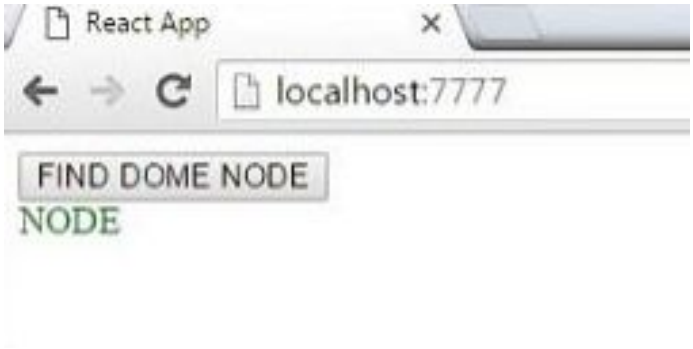
Look for the Dom node

To manipulate the DOM, we can use the ReactDOM.findDOMNode () method. First we need to import the reaction.

```
import React from 'react';
import ReactDOM from 'react-dom';
class App extends React.Component {
 constructor() {
 super();
 this.findDomNodeHandler =
this.findDomNodeHandler.bind(this);
};
 findDomNodeHandler() {
var myDiv = document.getElementById('myDiv');
 ReactDOM.findDOMNode(myDiv).style.color = 'green';
}
 render() {
return (
<div>
<button onClick = {this.findDomNodeHandler}>FIND
DOME NODE</button>
<div id = "myDiv">NODE</div>
</div>
);
```

```
}
}
export default App;
```

The color of myDiv changes to green with just the click of a button



Note : As of Update 0.14, most of the older components API methods are deprecated or removed to accommodate ES6.

Chapter V

Component life cycle

In this chapter we will discuss life cycle methods

Ingredients.

Life cycle methods

*** The elementWillMount is triggered before submission, either on the server or on the client side.**

*** ElementDidMount is played only after premiere on the client side. This is where AJAX and DOM requests or status updates should occur. This method is also used to integrate with other JavaScript frameworks and any snooze functionality, such as setTimeout or setInterval. We use it to update status so we can activate other life cycle methods.**

*** ElementWillReceiveProps is called once the extensions are updated before another call Submit. We activate it from setNewNumber when we update the status.**

*** shouldComponentUpdate should return a true or false value. This will determine whether or not the component will be updated. This is set to true by default. If you are sure that the component does not need to be displayed after updating the status or extensions, you can return a wrong value.**

*** ComponentWillUpdate is called right before the show.**

*** ComponentDidUpdate is called immediately after rendering.**

*** ComponentWillUnmount is called after unloading component from dom. We dismantle our component in main.js. In the following example, we'll set the initial state in the constructor function. SetNewnumber is used to update the status. All life cycle methods within the content component.**

App.jsx

```
import React from 'react';
class App extends React.Component {
constructor(props) {
super(props);

 this.state = {
data: 0
}
 this.setNewNumber = this.setNewNumber.bind(this)
};
 setNewNumber() {
 this.setState({data: this.state.data + 1})
}
 render() {
return (
<div>
<button onClick =
{this.setNewNumber}>INCREMENT</button>
<Content myNumber = {this.state.data}></Content>
</div>
);
}
}
class Content extends React.Component {
 componentWillMount() {
 console.log('Component WILL MOUNT!')
}
 componentDidMount() {
```

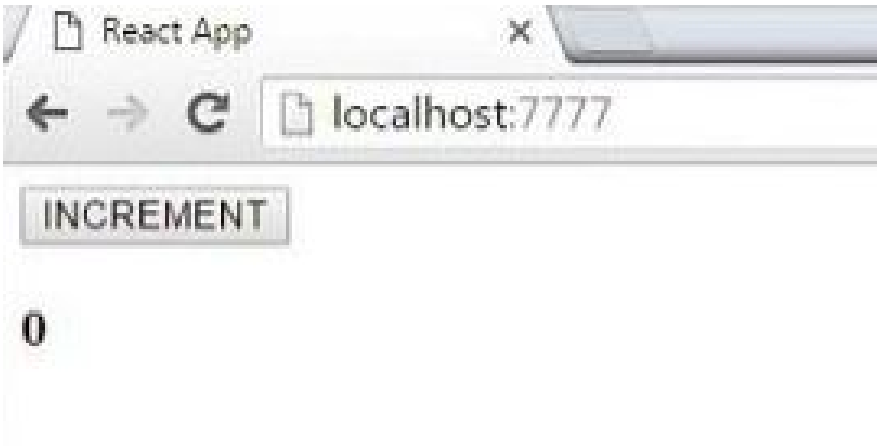
```
 console.log('Component DID MOUNT!')
}
 componentWillReceiveProps(newProps) {
 console.log('Component WILL RECIEVE PROPS!')
}
 shouldComponentUpdate(newProps, newState) {
return true;
}
 componentWillUpdate(nextProps, nextState) {
 console.log('Component WILL UPDATE!');
}
 componentDidUpdate(prevProps, prevState) {
 console.log('Component DID UPDATE!')
}
 componentWillUnmount() {
 console.log('Component WILL UNMOUNT!')
}
 render() {
return (
<div>
<h3>{this.props.myNumber}</h3>
</div>
);
}
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
```

```
import App from './App.jsx';
ReactDOM.render(<App/>, document.getElementById('app'));
setTimeout(() => {

ReactDOM.unmountComponentAtNode(document.getElementById('app'
));}, 10000);
```



Remembering the clock example from the Object Rendering section in React, we learned in that section only one way to update the UI by calling the function ReactDOM.render () to change the result:

```
function tick () {
  const element = (
    <div>
      <h1> Hello world! </h1>
      <h2> The time now is {new Date (). toLocaleTimeString ()}. </h2>
    </div>
  );
  ReactDOM.render (
    element,
```

```
    document.getElementById ('root')
  );
}

setInterval (tick, 1000);
```

In this section, we will learn how to make the Clock component truly reusable and encapsulate within itself, where it sets its own time counter and updates itself every second.

We can start by wrapping the watch shape:

```
function Clock (props) {
  return (
    <div>
      <h1> Hello world! </h1>
      <h2> The time now is {props.date.toLocaleTimeString ()}. </h2>
    </div>
  );
}

function tick () {
  ReactDOM.render (
    <Clock date = {new Date ()} />,
    document.getElementById ('root')
```

```
  );
}

setInterval (tick, 1000);
```

However, this component lacks a prerequisite: that setting the clock for the timer and updating the UI every second should be an internal detail of the Clock component.

We ideally want to write this code once and have the Clock component update itself:

```
ReactDOM.render (
  <Clock />,
  document.getElementById ('root')
);
```

To do this we need to add a state to the Clock component.

The state is similar to props, but it is private and is controlled by the component.

We have previously mentioned in the Components and Properties section that components defined as classes have additional features. Local State is one of these features and is only available for items.

Converts the function to a class

You can convert function components like Clock into classes in five steps:

Create a class with the same name that extends (extends) to React.Component.

Add a single empty function to this class named render ().

Move the function object to the render () function.

Switch props to this.props in the render object.

Delete the remainder of the empty function statement.

```
class Clock extends React.Component {
  render () {
    return (
      <div>
        <h1> Hello world! </h1>
        <h2> The time now is {this.props.date.toLocaleTimeString ()}.
</h2>
      </div>
    );
  }
}
```

The Clock component is now defined as a class instead of a function.

The render () sequence will be called each time an update occurs, but as long as we assign (Render) the <Clock /> element to the same DOM node,

only one instance of the Clock class will be used. This allows us to use additional features such as local Component life.

Add the local status of the item

We will move the date from the props to the status in three steps:

First, toggle this.props.date with this.state.date in the render () continuation:

```
class Clock extends React.Component {
  render () {
    return (
      <div>
        <h1> Hello world! </h1>
        <h2> The time now is {this.state.date.toLocaleTimeString ()}.
</h2>
      </div>
    );
  }
}
```

Second, add a constructor that sets the initial value of this.state:

```
class Clock extends React.Component {
  constructor (props) {
    super (props);
    this.state = {date: new Date ()};
  }
```

```
  render () {
    return (
      <div>
        <h1> Hello world! </h1>
        <h2> The time now is {this.state.date.toLocaleTimeString ()}.
</h2>
      </div>
    );
  }
}
```

Notice how we passed the props properties to the builder:

```
constructor (props) {
    super (props);
    this.state = {date: new Date ()};
  }
```

Class components should always call the builder function with props.

Third, remove the date property from the <Clock /> element:

```
ReactDOM.render (
  <Clock />,
```

```
  document.getElementById ('root')
);
```

Later, we'll re-add the timer code to the same component. The result looks like this:

```
class Clock extends React.Component {
  constructor (props) {
    super (props);
    this.state = {date: new Date ()};
  }

  render () {
    return (
      <div>
        <h1> Hello world! </h1>
        <h2> The time now is {this.state.date.toLocaleTimeString ()}.
</h2>
      </div>
    );
  }
}

ReactDOM.render (
  <Clock />,
  document.getElementById ('root')
```

);

We will now have the Clock component set its own time counter and update itself every second.

Add life cycle dependencies to the class

In applications that have many components, it is important that we release the resources reserved by these components when they are destroyed.

We want to set the time counter as soon as the Clock component is initialized to the DOM for the first time, this is called a mounting in React. We also want to clear this counter as soon as the DOM generated by the Clock component is removed, this is called React in unmounting.

We can declare special functions in the component components to execute some codes when the component is connected and disconnected:

```
class Clock extends React.Component {
  constructor (props) {
```

```
    super (props);
    this.state = {date: new Date ()};
  }

  componentDidMount () {

  }

  componentWillUnmount () {

  }

  render () {
    return (
      <div>
        <h1> Hello world! </h1>
        <h2> The time now is {this.state.date.toLocaleTimeString ()}.
</h2>
      </div>
    );
  }
}
```

These functions are called lifecycle hooks. The componentDidMount () hook works after the component output is set to DOM, so it's a convenient place to set the timer:

```
componentDidMount () {
   this.timerID = setInterval (
    () => this.tick (),
    1000
   );
 }
```

Notice how we saved the timer ID with this.

Since this.props are set up by React itself and this.state has a special meaning, you are free to manually add additional fields to the class if you need to store something that is not involved in the data flow (such as the timer ID).

We will finish the time counter in the life cycle hook componentWillUnmount ():

```
componentWillUnmount () {
   clearInterval (this.timerID);
 }
```

Finally, we'll add a function called tick () that the Clock component executes every second. This.setState () will be used to schedule updates to the local state of the component:

```
class Clock extends React.Component {
  constructor (props) {
   super (props);
```

```
  this.state = {date: new Date ()};
 }

 componentDidMount () {
  this.timerID = setInterval (
   () => this.tick (),
   1000
  );
 }

 componentWillUnmount () {
  clearInterval (this.timerID);
 }

 tick () {
  this.setState ({
   date: new Date ()
  });
 }

 render () {
  return (
   <div>
    <h1> Hello world </h1>
    <h2> The time now is {this.state.date.toLocaleTimeString ()}.
</h2>
```

```
      </div>
    );
  }
}

ReactDOM.render (
  <Clock />,
  document.getElementById ('root')
);
```

The clock now rings every second.

Let us quickly summarize what is going on and recall the order in which the functions are called:

*** When the <Clock /> element is passed to ReactDOM.render (), React calls the Clock component builder. Since Clock needs to show the current time it will initialize this.state with an object that includes the current time, and later update this state.**

*** React then calls the render () of the Clock component, so React learns what should be displayed on the screen. React then updates the DOM to match the Clock output.**

*** When the clock output is entered into the DOM, React calls the componentDidMount () lifecycle hook, and inside it asks the browser clock to set the time counter to call the component's tick () function once every second.**

*** Every second the browser calls the tick () function, and inside the Clock component schedules an update of the user interface by calling setState () with an object that contains the current time. Thanks to the call to setState (), React learns that the state has changed so that it calls the render () again to know what should be on the screen. . React updates the DOM.**

*** If the Clock component is removed from the DOM, React calls the componentWillUnmount () lifecycle hook so that the time counter stops.**

Use the case correctly

There are three things you should know about setState ().

Do not modify the status directly

For example, the following example will not reassign the component:

// Wrong way

this.state.comment = 'Welcome';

Use setState () instead:

// the right way

this.setState ({comment: 'Hi'});

The only place you can set this.state is the builder.

Status updates may be out of sync

React may combine several calls to the setState () sequence in one update for improved performance.

Since this.props and this.state may be updated asynchronously, they should not rely on their values to calculate the next state.

For example, the following code may fail to update the time counter:

```
// Wrong way
this.setState ({
  counter: this.state.counter + this.props.increment,
});
```

To fix this, use another form of setState () that accepts a function instead of an object. This function receives the preceding state as its first argument, and the props attributes at the time the update is applied as its second argument:

```
// the right way
this.setState ((prevState, props) => ({
  counter: prevState.counter + props.increment
}));
```

We used sagittal functions in the previous example, but they also work with regular functions:

```
// the right way
this.setState (function (prevState, props) {
  return {
    counter: prevState.counter + props.increment
  };
});
```

React incorporates status updates

When you call setState (), React merges the object you provide it with the current state.

For example, your status may contain several independent variables:

```
constructor (props) {
   super (props);
   this.state = {
     posts: [],
     comments: []
   };
 }
```

You can then update it independently with separate setState () function calls:

```
componentDidMount () {
   fetchPosts (). then (response => {
     this.setState ({
       posts: response.posts
     });
   });

   fetchComments (). then (response => {
     this.setState ({
       comments: response.comments
     });
   });
 }
```

This consolidation is minimal, so updating the status of this.setState comments ({comments}) does not affect the status of posts ie this.state.posts, but it completely replaces this.state.comments comments status.

Data flows to lower levels

Components do not tell parents, not even children, whether a specific component has or is not, and should not care whether it is defined as a function or class.

This is why the case is called local or packaged, as it is inaccessible by any component other than the one that owns it and assigns it.

The component may choose to pass its status as props attributes to its lower-level child elements:

<h2> The time now is {this.state.date.toLocaleTimeString ()}. </h2>

This example also works with user-defined components:

<FormattedDate date = {this.state.date} />

The FormattedDate component receives the date date in its properties and will not know whether this date came from the Clock component's status, Clock properties, or manually written:

```
function FormattedDate (props) {
  return <h2> The time now is {props.date.toLocaleTimeString ()}. </h2>;
}
```

This is usually called data flow from the top-down or unidirectional. Any state owned by a specific component, and any data or user interface derived from that state, can only affect the components underneath in the component tree.

If the component tree is imagined as a cascade of characteristics, each component is like an additional source of water that joins it at a random point and flows downward.

To demonstrate that all the components are really isolated, we can create the App component that shows a tree of <Clock> components:

```
function App () {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}

ReactDOM.render (
  <App />,
  document.getElementById ('root')
);
```

Each Clock component sets its own meter and updates it separately.

In React applications, when you use a component within another component, the component is an operational breakdown of the other component, whether the component has a stateful or stateless state and can change over time.You can use the components without the state within the components of the state and vice versa.

Conditional rendering in React

In React, you can create unique components that encapsulate the behavior you want, and then only portions of them depending on the status of your application.

Conditional rendering in React works in the same way as conditional expressions in JavaScript, where you can use JavaScript operators such as if or the conditional operator to create elements that represent the current state, and then let React update the user interface to match them.

Consider these two components:

```
function UserGreeting (props) {
  return <h1> Welcome back! </h1>;
}

function GuestGreeting (props) {
  return <h1> Please register at the site </h1>;
}
```

We will create a Greeting component that displays one of these components depending on the user's status if they are logged in or not:

```
function Greeting (props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

ReactDOM.render (
  // Try changing to isLoggedIn = {true}
  <Greeting isLoggedIn = {false} />,
  document.getElementById ('root')
);
```

This example shows a different welcome depending on the value of the isLoggedIn property.

Item variables

You can use variables to store items. This helps to condition a part of the component so that the rest of its output does not change.

Consider these two new components, which represent the Logout and Login buttons:

```
function LoginButton (props) {
  return (
    <button onClick = {props.onClick}>
      log in
    </button>
  );
}

function LogoutButton (props) {
  return (
    <button onClick = {props.onClick}>
      sign out
    </button>
  );
}
```

In the previous example, we will create a state component and call it LoginControl. This component will either <LoginButton /> or

<LogoutButton/> depending on its current state, it will also <<Greeting /> from the previous example:

```
class LoginControl extends React.Component {
  constructor (props) {
    super (props);
    this.handleLoginClick = this.handleLoginClick.bind (this);
    this.handleLogoutClick = this.handleLogoutClick.bind (this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick () {
    this.setState ({isLoggedIn: true});
  }

  handleLogoutClick () {
    this.setState ({isLoggedIn: false});
  }

  render () {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick = {this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick = {this.handleLoginClick} />
    }

    return (
      <div>
```

```
      <Greeting isLoggedIn = {isLoggedIn} />
      {button}
     </div>
   );
  }
}

ReactDOM.render (
  <LoginControl />,
  document.getElementById ('root')
);
```

Defining a variable and using the if statement is a good way to conditionally configure a component, but sometimes you may want to use a more concise syntax. There are some ways to put conditional sentences inline in JSX that we will now explain.

Clause If Inline with Logical Operator &&

You can include any expressions in JSX by enclosing them in parentheses. This includes the && logical operator in JavaScript, where it is useful to conditionally include an element:

```
function Mailbox (props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1> Welcome! </h1>
      {unreadMessages.length> 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
```

```
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re: Re: React'];
ReactDOM.render (
  <Mailbox unreadMessages = {messages} />,
  document.getElementById ('root')
);
```

This example works because the true && expression expression always evaluates to the expression value in JavaScript, and the expression && expression always evaluates to false. Therefore, if the condition is true, it will appear within the result of the element after the && operator, and if the condition is false, React ignores it and skips it.

In-line If-Else clause with conditional operand

Another method used for conditional rendering is the use of a triple conditional operator in JavaScript, which is condition? true: false.

We'll use it in the following example to conditionally make a small block of text:

```
render () {
  const isLoggedIn = this.state.isLoggedIn;
```

```
  return (
    <div>
      User <b> {isLoggedIn? 'Not': 'Currently'} </b> is signed in.
    </div>
  );
}
```

It can also be used for larger expressions although there is little clarity about what happens in this example:

```
render () {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn? (
        <LogoutButton onClick = {this.handleLogoutClick} />
      (:)
        <LoginButton onClick = {this.handleLoginClick} />
      )}
    </div>
  );
}
```

As with JavaScript, you have the freedom to choose the appropriate format that you and your team consider more readable. Keep in mind that when conditional sentences become too complex, it is more appropriate to extract the component.

Prevent component from rendering

In rare cases, you may want the component to hide itself even though it was created by another component, and to do this set the value null instead of the output.

In the following example, <WarningBanner/> is made based on the value of the property called warn, if it is false then the component is not:

```
function WarningBanner (props) {
  if (! props.warn) {
    return null;
  }

  return (
    <div className = "warning">
      Warning!
    </div>
  );
}

class Page extends React.Component {
  constructor (props) {
    super (props);
    this.state = {showWarning: true};
    this.handleToggleClick = this.handleToggleClick.bind (this);
  }

  handleToggleClick () {
    this.setState (prevState => ({
```

```
      showWarning:! prevState.showWarning
    }));
  }

  render () {
    return (
      <div>
        <WarningBanner warn = {this.state.showWarning} />
        <button onClick = {this.handleToggleClick}>
          {this.state.showWarning? 'Hide': 'Show'}
        </button>
      </div>
    );
  }
}

ReactDOM.render (
  <Page />,
  document.getElementById ('root')
);
```

A null return from the render function of the component does not affect the release of the component's lifecycle dependencies, for example, calling the componentDidUpdate function will not be affected.

Chapter VI

Forms

In this chapter, we will learn how to use models in React.

Simple example In the following example, we'll set an input form with the value = {this.state.data}. This allows the status to be updated every time the value of the entry changes. We use the onChange event which will monitor input changes and update status accordingly.

App.jsx

```
import React from 'react';
class App extends React.Component {
constructor(props) {
super(props);

 this.state = {
data: 'Initial data...'
}
 this.updateState = this.updateState.bind(this);
};
 updateState(e) {
 this.setState({data: e.target.value});
}
 render() {
return (
```

```
<div>
<input type = "text" value = {this.state.data}
 onChange = {this.updateState} />
<h4>{this.state.data}</h4>
</div>
);
}
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';
ReactDOM.render(<App/>, document.getElementById('app'));
```

Example

- Focus here -

In the following example, we'll see how to use sub-component models. OnChange will trigger the status update to pass it to the value of the secondary entry and display it on the screen. A similar example is used in the Events chapter. Whenever we need to update the status of the subcomponent, we must pass the function that will handle update (updateState) as an extension (updateStateProp).

App.jsx

- Focus here -

```
import React from 'react';
class App extends React.Component {
constructor(props) {
super(props);

 this.state = {
data: 'Initial data...'
}
 this.updateState = this.updateState.bind(this);
};
 updateState(e) {
 this.setState({data: e.target.value});
}
 render() {
return (
<div>
<Content myDataProp = {this.state.data}
 updateStateProp =
{this.updateState}></Content>
```

```
</div>
);
}
}
class Content extends React.Component {
 render() {
return (
<div>
<input type = "text" value =
{this.props.myDataProp}
 onChange = {this.props.updateStateProp} />
<h3>{this.props.myDataProp}</h3>
</div>
);
}
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';
ReactDOM.render(<App/>, document.getElementById('app'));
```

Chapter VI I

Fields in React

Form elements work slightly differently than other DOM elements in React because field elements normally retain their own internal state. For example, this field in HTML accepts a single name:

```
<form>
  <label>
    The name:
    <input type = "text" name = "name" />
  </label>
  <input type = "submit" value = "submit data" />
</form>
```

This field has the same default behavior for HTML fields as moving to a new page when the user presses the Submit button.If you just want this behavior in React, it will work well with you, but most often it is more convenient to have a JavaScript function With data submission and has access to the data entered by the user in the field. The standard way to achieve this is by using a technique called controlled components.

Control ingredients

Field elements in HTML such as <input>, <textarea>, and <select> maintain and update their own state according to user input. In React, an editable state is maintained within the state property of the components and updated only by the setState () function.

We can combine them by making the React state the only source of truth, so the React component that becomes the field also controls what happens in that field with the user input sequence. The input field element whose value is controlled by React is called the configured component.

For example, if in the previous example we wanted to display the name after it was submitted, we could type the field as a configured component:

```
class NameForm extends React.Component {
  constructor (props) {
    super (props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind (this);
    this.handleSubmit = this.handleSubmit.bind (this);
  }

  handleChange (event) {
    this.setState ({value: event.target.value});
  }

  handleSubmit (event) {
```

```
    alert ('Provide name:' + this.state.value);
    event.preventDefault ();
  }

  render () {
    return (
      <form onSubmit = {this.handleSubmit}>
        <label>
          The name:
          <input type = "text" value = {this.state.value} onChange =
{this.handleChange} />
        </label>
        <input type = "submit" value = "submit data" />
      </form>
    );
  }
}
```

Since the value property is set by the field element, its displayed value will always be this.state.value, making React the only source of truth. Since the handleChange function is executed at each click of a user to update the React state, the displayed value will be updated as the user types.

In the controlled components, each state change has a corresponding function to deal with. This makes it easy to modify and verify user income. For example, if you want to force the user to type capital letters, we will write the handleChange function as follows:

```
handleChange (event) {
  this.setState ({value: event.target.value.toUpperCase ()});
}
```

The textarea element

In HTML, the text of the <textarea> element is directly defined as follows:

```
<textarea>
  Hello, This is some text in the textarea element
</textarea>
```

In React, the <textarea> element uses the value property instead. This way, the field that uses <textarea> can be written similar to the field that uses the <input> input element:

```
class EssayForm extends React.Component {
  constructor (props) {
    super (props);
    this.state = {
```

```
    value: 'Please type an article about your favorite DOM'
  };

  this.handleChange = this.handleChange.bind (this);
  this.handleSubmit = this.handleSubmit.bind (this);
}

handleChange (event) {
  this.setState ({value: event.target.value});
}

handleSubmit (event) {
  alert ('Article:' + this.state.value);
  event.preventDefault ();
}

render () {
  return (
    <form onSubmit = {this.handleSubmit}>
      <label>
        the article:
        <textarea value = {this.state.value} onChange =
{this.handleChange} />
      </label>
      <input type = "submit" value = "submit article" />
    </form>
```

```
    );
  }
}
```

Note that we have configured this.state.value with an initial value in the constructor, thus ensuring that there is text within the <textarea> element from the beginning.

select

In HTML, the <select> element creates a drop-down list. For example, this code creates a drop-down list of some fruit names:

```
<select>
  <option value = "Banana"> Banana </ option>
  <option value = "apple"> Apple </ Option>
  <option selected value = "Orange"> Orange </strong> option
  <option value = "mango"> و مانج  </ option>
</select>
```

Note that the initial option here is orange because there is a property selected next to it, but in React instead of using the selected property we use the value property under the <select> element, which is easier in the constituent components because you will need to modify it only in one place. For example:

```
class FlavorForm extends React.Component {
```

```
  constructor (props) {
    super (props);
    this.state = {value: 'orange'};

    this.handleChange = this.handleChange.bind (this);
    this.handleSubmit = this.handleSubmit.bind (this);
  }

  handleChange (event) {
    this.setState ({value: event.target.value});
  }

  handleSubmit (event) {
    alert ('Your favorite fruit is:' + this.state.value);
    event.preventDefault ();
  }

  render () {
    return (
      <form onSubmit = {this.handleSubmit}>
        <label>
          Choose your favorite fruit
          <select value = {this.state.value} onChange = {this.handleChange}>
             <option value = "banana"> Banana </ option>
<option value = "apple"> Apple </ Option>
<option value = "orange"> Orange </strong> option
<option value = "mango"> و مانج  </ option>
          </select>
        </label>
        <input type = "submit" value = "submit data" />
      </form>
    );
  }
}
```

The <input type = "text">, <textarea>, and <select> elements work similarly, all accepting the value property that we use to implement the configured component.

Note: You can pass an array to the value property, which allows you to select several options in the <select> element:

<select multiple = {true} value = {['B', 'C']}>

File input element

In HTML, the <input type = "file"> element allows a user to select one or more files from his or her device to upload to the server or work with them via JavaScript via the file API:

<input type = "file" />

Since its value is read-only, it is an uncontrolled component in React, we will discuss this component with the other uncontrolled components in its own section.

Handling multiple entries

When you need to manipulate multiple input elements set, you can add the name property to each element and let the event handler choose what to do based on the event.target.name value.

```
class Reservation extends React.Component {
 constructor (props) {
  super (props);
  this.state = {
   isGoing: true,
   numberOfGuests: 2
  };

  this.handleInputChange = this.handleInputChange.bind (this);
 }

 handleInputChange (event) {
  const target = event.target;
  const value = target.type === 'checkbox'? target.checked: target.value;
  const name = target.name;

  this.setState ({
   [name]: value
  });
 }

 render () {
  return (
   <form>
    <label>
     Gone:
     <input
      name = "isGoing"
```

```
          type = "checkbox"
          checked = {this.state.isGoing}
          onChange = {this.handleInputChange} />
        </label>
        <br />
        <label>
          number of guests:
          <input
          name = "numberOfGuests"
          type = "number"
          value = {this.state.numberOfGuests}
          onChange = {this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

Note how we used the formatting of the property name calculated in ES6 to update the status key to match the entered name:

```
this.setState ({
  [name]: value
});
```

The previous code in ES5 is equivalent to:

var partialState = {};

partialState [name] = value;

this.setState (partialState);

Since the setState () function automatically integrates a partial state with the current state we will only need to call it with the variable parts.

Justified entries with Null value

Setting the value property in the configured components prevents the user from changing inputs unless desired. If you set the value and the input element remains editable, you may inadvertently set the value to undefined or null.

The following code illustrates this (the input element is initially locked and then becomes editable after a short period of time):

```
ReactDOM.render (<input value = "hi" />, mountNode);

setTimeout (function () {
  ReactDOM.render (<input value = {null} />, mountNode);
}, 1000);
```

Alternatives to controlled ingredients

Using tuned components can sometimes become tedious because you need to write an event handler for each method your data may change and direct all inputs through the React component. This becomes especially annoying when you convert your existing code to React or when you merge React with another library. In these cases, you may want to use uncontrolled components, an alternative technique for handling input fields.

Integrated solutions

If you are looking for a complete solution that includes validation, tracking visited fields, and processing field submission, Formik is one of the most popular options. However, it is based on the same principles of control and case management components, so don't ignore it and miss it.

><

.