Understanding React

The Simplest Practical Guide to Start Coding in React

★Includes:
React **Hooks** and React **Router**
+ **Full Stack** Authentication
& Authorization Flow

Enrique Pablo Molinari

Contents

About the Author

My name is Enrique Pablo Molinari. I have been working in the software industry for the last 22 years, working in different software projects from different companies as developer, technical lead and architect. I'm a passionate developer and also a passionate educator. In addition to my work on the software industry, I'm teaching Object Oriented Design and Advance Database Systems at Universidad Nacional de Río Negro.

Understanding React is my second book. I have also written Coding an Architecture Style, a book about hands-on software architecture. You can find more about my thoughts on software development at my blog: Copy/Paste is for Word. I would be very happy if you want to ping me by email at enrique.molinari@gmail.com to send thoughts, comments or questions about this book, the other or my blog.

What is this book about?

Every successful framework or library provides something unique which gives developers a new tool for writing better software. In the case of React, that tool is called **component**. You might be thinking that you have been reading about components as the solution to your spaghetti software nightmare for the last 15 years without any success. You are not wrong. However, React is an exception. It provides the constructions and tools to build highly cohesive components to assemble your next application. In this book, we will study React core concepts, to end up being very practical describing how to split an application into components and to fully implement it. But before that, it is necessary to study some Javascript concepts. Understanding these concepts will make you a better React developer. If you are already a Javascript developer, then you can just ignore the initial chapter. But if your experience is mainly on server side programming languages like Java, C#, PHP, etc, the initial chapter will give you the necessary basis.

Development Environment

There are many development environments out there, and you can choose the one you are more comfortable with. In any case if you don't have a preference, I recommend Visual Studio Code (VS Code). And to be more productive, especially if you are new to React, I suggest installing the extension VS Code ES7 React/Redux/React-Native/JS snippets which provides JavaScript and React snippets. I would also suggest installing Prettier, which is a JavaScript/React code formatter.

To install an extension, in Visual Studio Code, go to the File menu, then Preferences and then Extensions. You will see a search box that will allow you to find the extensions that you want to install.

Finally, I really recommend configuring VS Code to format your source files on save. You can do that by going to the File menu, then Preferences and then Settings. On the search box type Editor: Format On Save. This will format your code right after you save it.

Part I

Introduction

Chapter 1

Essential JavaScript Concepts

You can use React just by learning from React docs and you will also be able to build applications, without digging into JavaScript. However, if you want to master React and that means, understand why and how certain things work, you must learn some specific concepts from JavaScript.

In this chapter we will explain those JavaScript concepts and syntactical constructions needed to make a solid learning path to React. If you want to dig in more details on some of the topics explained here or others about JavaScript I recommend to visit the Mozilla[1] web site. Indeed, this section is based on learning paths and ideas taken from there. Having said that, let's begin.

From the Developer Mozilla JavaScript Documentation [1] JavaScript is defined as:

"JavaScript (JS) is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions. While it is most well-known as the scripting language for Web pages, many non-browser environments also use it, such as Node.js, Apache CouchDB and Adobe Acrobat. JavaScript is a prototype-based, multi-paradigm, single-threaded, dynamic language, supporting object-oriented, imperative, and declarative styles."

If you are a Java, C# or C++ developer that definition might sound a bit intimidating. The thing is that you do have to learn some concepts. Especially those that are not available in compiled languages (if your experience comes from there). To start with these concepts we will first explain basic language constructions and with that in place we will explain what it means for a language to have **first-class functions** and to be **prototype-based**, **multi-paradigm**, **single-threaded** and **dynamic**.

The JavaScript language is governed by a standard under the responsibility of ECMA[3]. ECMAScript is the name of the language specification. The standardisation allows vendors to write interpreters and developers to be able to run their programs on any vendor interpreter. So, it is a great thing. In 2015 there was a major release known as ES6, ECMAScript 6 or ECMAScript 2015. Most of the syntactical constructions that we will study in this chapter were implemented in this release.

Let's then begin learning. Any piece of code written in this chapter will run using the `node` interpreter. Install the latest LTS version of Node.js. Once installed, you can verify that it is working by open a console and type:

```
$ node -v
```

That will display the version installed. With this in place, you can execute a JavaScript file in the following way:

```
$ node yourjsfile.js
```

Using Visual Studio Code, you can go to the menu "Terminal" and then "New Terminal". It will open a small terminal window where you can run your scripts.

Printing text on the screen must be the very first thing you learn every time you start playing with a new programming language. This book is not the exception. You can print text on the screen using the `Console` object like the following example:

```
console.log("Coding in React!");
```

Let's then open VS Code to try this out. Open a console from your operating system, create a new folder called 'chapter1', and then type: `code chapter1`. That will open VS Code ready to be used inside of the 'chapter1' folder. Create a new file called `console.js`, copy and paste the previous snippet in that file, save it and execute it typing:

```
$ node console.js
```

The object `Console` was created mainly for debugging purposes, it should not be used in production. It gives you access to the Browser's debug console. It is also not part of the standard, but most modern browsers and nodejs supports it.

1.1 Variables

Let's move to **variables**. You can declare a variable using the `let` keyword:

```
let myFirstVariable;
```

Declaring a variable without initialising it will assign the undefined special value to it and that is what you will see if you print it. Try it!. Let's give it an initial string value:

```
let myFirstVariable = "Hello value!";
```

Now if you print it you will see the string. You can also declare a variable using the `const` keyword:

```
const myFirstConst = "Hello constant value!";
```

As you might have guessed, declaring a variable with `const` will not allow you to change the value of the variable once it has been initialised. If you do it the interpreter will throw an error. Try it!.

JavaScript is a **dynamic language**, among many other things that we will discuss later, that means that the type of a variable can be changed at runtime. Opposed to static (or compiled) languages where the type of a variable is defined at compile-time and cannot be changed during execution.

```
//my type is string
let changeMyType = "Hello String!";
//now it is number
changeMyType = 100;
```

1.2 Functions

Let's move now to **functions**. In the following code snippet we are declaring a function and after that we are calling it:

```
function saySomething(string) {
  console.log(string);
}

saySomething("Hello Function!");
```

If you just need to have a function that gets called as soon as it is declared, you can use the syntax below. This is called IIFE (Immediately-invoked Function Expression).

```
(function saySomething(string) {
  console.log(string);
})("Hello Function!");
```

In JavaScript functions always return something. If you don't explicitly return something from the function using the return keyword it will return undefined.

```
let x = saySomething("Hello Function!");
//x is undefined
```

In addition, functions are **first-class** objects. The most well known first class object of programming languages are *variables*. Variables are denominated first-class objects because it can be assigned, it can be passed as argument to a function or method, it can be returned from a function, etc. So, having functions as first-class objects means that all those things that you can do with variables are possible with functions too. See at the example below where on line 6 we are assigning the function to the variable `say`. And then on line 10 we are using it to invoke the function.

```
1  function returnSomething(string) {
2    return "This is it: " + string;
3  }
4
5  //assigning a function to a variable
6  let say = returnSomething;
7
8  //calling the function
9  returnSomething("Hello js!");
10 say("Hello again!");
```

Both `say` and `returnSomething` points to the same place which is the first statement in the body of the function. In the next example, on line 12 we are invoking a function and passing the `returnSomething` function as argument. Note how then is invoked on line 8 and its return value returned.

```
1  function returnSomething(string) {
2    return "This is it: " + string;
```

```
}

//receives a function as parameter
//invokes it and return the value
function saySomethingMore(fn) {
  return fn("Hey !");
}

//passing a function as argument
saySomethingMore(returnSomething); //"This is it: Hey !"
```

Functions can also be assigned to variables just in its declaration as the next example illustrate:

```
//assigning the function
const returnSomething = function (string) {
  return "This is it: " + string;
};

returnSomething("Hey !"); //"This is it: Hey !"
```

JavaScript provides another and a bit less verbose way to declare functions called **arrow functions**. Let's see some examples:

```
//arrow function with no parameters
const arrowf1 = () => {
  return "arrowf1 was invoked!";
};

//arrow function with one parameter
//parenthesis is not necessary here
const arrowf2 = param => {
  return "this is the argument: " + param;
};

//arrow functions with one statement
//in the body won't need return
const arrowf3 = (a, b) => a + b;
```

1.3 Arrays

Arrays are another very important construction that we will use massively. This is how you can declare an array:

```
//an empty array
let empty = [];

//an array
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];
```

The elements of an array can be accessed by its index, where the index of the first element is 0.

```
//an array
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];
family[0]; //Jóse
family[1]; //Nicolas
family[2]; //Lucia
family[3]; //Enrique
```

Adding an element at the end of the array:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];

//adding an element at the end of an array
family.push("Pablo");
```

And if you want to add the elements of an existing array to another array (empty or not), you can use what is known as *spread syntax*:

```
let myParents = ["EnriqueR", "Susana"];
let JoseParents = ["Eduardo", "Graciela"];
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];
let all = [...myParents, ...JoseParents, ...family];
//[
//  'EnriqueR', 'Susana', 'Eduardo', 'Graciela',
//  'Jóse', 'Nicolas', 'Lucia', 'Enrique'
// ]
```

Spread syntax is also available for functions to accept an indefinite number of arguments:

```
function restParams(param1, param2, ...params) {
 //params is [3, 4, 5]
}
restParams(1, 2, 3, 4, 5);
```

To simply iterate over an array you can use the following `for` construction:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];
for (let element of family) {
  console.log("regular for: ", element);
}
```

And in addition we have a set of very useful methods. Let's see first how we can iterate over an array using the `.forEach` method:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];

family.forEach(function (value, index, array) {
  //value is the element being processed
  //index is the index of the current value
  //array is the entire array
  console.log(value, index, array);
});
```

Note that the `.forEach` method accepts as parameter a function that accepts three parameters. `value` which is the element being processed, the `index` which is the index of the value being processed and `array` which is the array that we are looping. If you are only interested in the elements you can just do this:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];

family.forEach((value) => {
  //do something with the value here
});
```

Another very interesting method is `.filter`. Similar to the previous one, it receives a function with the same parameters. It will return a new array (shorter or equal than the original) with the elements that evaluates to true.

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];

const members = family.filter((member) => {
  return member.length > 5;
});

//members = ['Nicolas', 'Enrique']
```

Note that we are passing an arrow function to the `.filter` method with a condition testing the length of each element in the `family` array. Those elements whose length is greater than 5 will be part of the returned new array. Also note that the `family` array is not changed at all.

The last method we will see is `.map`. This method receives a function, same as the previous two methods, and it will return a new array with the result of applying the function to every element. It will always return an array of the same length as the one we are processing. As we will see later, `.map` is very used in React to add markup to the elements of arrays.

```
let numbers = [1, 2, 3, 4, 5, 6, 7];
const doubles = numbers.map((element) => {
  return element * 2;
});
//doubles = [2,  4,  6, 8, 10, 12, 14]
```

Array methods can be combined to produce the desired results. Look at the example below. We are first applying the `.filter` function to get an array only with odd numbers and then we are applying `.map` to transform it into an array of even numbers.

```
let numbers = [1, 2, 3, 4, 5, 6, 7];
const chain = numbers
  .filter((element) => {
    return element % 2 !== 0;
  }) //[1, 3, 5, 7]
  .map((element) => {
    return element * 2;
  });
//chain = [2, 6, 10, 14]
```

If you have an array with few elements, instead of working with indexes, there is a very convenient way called **destructuring** that allows you to assign each element of the array to named variables. See below:

```
let [one, two, three] = [1, 2, 3];
//one = 1
//two = 2
//three = 3

//same as the previous
let fewNumbers = [1, 2, 3];
```

```
[one, two, three] = fewNumbers;

//and here using spread syntax
let [a, b, ...rest] = [1, 2, 3, 4, 5];
//a = 1
//b = 2
//rest = [3, 4, 5]
```

1.4 Objects

There are several ways to create objects in JavaScript. We will study those that will be used further in the book when coding in React. The first way to create objects that we will see is called **Object Literal**. An object literal is created wrapping within curly braces a collection of comma-separated *property*:*value* pairs.

```
//an object literal
let mi = {
  name: "Enrique",
  surname: "Molinari",
  sports: ["football", "tennis"],
  address: {
    street: "San Martin",
    number: 125,
  },
  allSports: function () {
    console.log(this.sports);
  },
};
//this is an empty object
let obj = {};
```

As you can see an object literal can be composed not only of simple property-value pairs but also for arrays, other objects like address and functions (called methods). Additionally, since ES6, you can create object literals with what is called *computed* property names, like shown below on line 6:

```
1 let aproperty = "phone";
2 //an object literal with a computed property name
3 let mi = {
4   name: "Enrique",
```

```
  surname: "Molinari",
  [aproperty]: "+54 2920 259031"
};
```

Every time the JavaScript interpreter evaluates an object literal a new object is created. You can access the properties of an object using the **dot notation** as the example below shows:

```
console.log(mi.name); //Enrique
console.log(mi.sports[0]);//football
console.log(mi.address.street);//San Martin
console.log(mi.phone);//+54 2920 259031
mi.allSports(); //invoke the function and prints the sports array
```

You can add properties (and remove too) dynamically to an object. In the example below, on lines 3 and 4 we are adding the properties `x` and `y` (with their corresponding value) to the `obj` object.

```
1 let obj = {a: 1, b: 2};
2 //add properties to the obj object
3 obj.x = 3;
4 obj.y = 4;
```

The spread syntax also works with objects, see below:

```
let obj1 = {
  a: 1,
  b: 2,
};
let obj2 = {
  c: 3,
  d: 4,
};
let obj3 = { ...obj1, ...obj2 };
//obj3 = { a: 1, b: 2, c: 3, d: 4 }
```

And if you want to create an object from some declared variables, you can do this:

```
let a = 1,
  b = 2;
let obj4 = {
```

```
  a,
  b,
};
//obj4 = { a: 1, b: 2 }
```

So far we have seen object literal syntax and what you can do with it. But what if you don't know how many objects you will need to create? You need something like a *class* from class-based languages like Java or C++. In JavaScript, we have what is called **constructor functions**. As we will see later, JavaScript has added classes to the language, but they are just a syntactic sugar on top of functions.

A constructor function's name by convention starts with a capital letter. Lets see how to create and use them:

```
1  function Book(name, authors, publishedYear) {
2    this.name = name;
3    this.authors = authors;
4    this.publishedYear = publishedYear;
5    this.fullName = function () {
6      return this.name + " by " + this.authors + ". " + publishedYear;
7    };
8  }
9
10 thisBook = new Book("Understanding React",
11                         ["Enrique Molinari"], 2021);
12 thisBook.fullName(); //Understanding React by Enrique Molinari. 2021
13
14 archBook = new Book("Coding an Architecture Style",
15                         ["Enrique Molinari"], 2020);
16 archBook.fullName(); //Coding an Architecture Style by Enrique Molinari. 2020
```

As you can see, the function `Book` looks like a class's *constructor* of a class-based language, in which, in addition, we are able to declare right there methods like `fullName()`. We define properties and we initialise them with the function parameters on lines 2, 3 and 4. On line 5 we define a method. After that, on lines 10 and 14 we are creating two instances of two different books and then invoke the `fullName()` method.

1.4.1 A Prototype-Base Language

Now that we know how to create object literals, constructor functions and create instances from them, it is time to explain what it means for JavaScript to be a **prototype-based** language. Prototype-based languages are a style of object oriented programming in which objects are created without creating *classes*. That is why they are also called *classless* languages, in contrast to *class-based* object oriented languages (like Smalltalk, Java, C++ and C# to name a few). In prototype-based languages there are no classes, just objects. We don't have that difference between classes and objects. That difference between a *static* definition of a blueprint (a class) and their inheritance relationship (which cannot be changed at runtime) vs the *dynamic* instantiation (object creation). And not having the distinction between classes and objects becomes evident in some situations like the ones we will see next. Defining methods in constructors functions in the way we did before is not ideal due to for each instance that we will create we are adding the method `fullName()` to it. This can be illustrated with the code below:

```
thisBook = new Book("Understanding React",
                        ["Enrique Molinari"], 2021);
archBook = new Book("Coding an Architecture Style",
                        ["Enrique Molinari"], 2020);
//printing thisBook
//Book {
//  name: 'Understanding React',
//  authors: [ 'Enrique Molinari' ],
//  publishedYear: 2021,
//  fullName: [Function (anonymous)]
//}
//printing archBook
//Book {
//  name: 'Coding an Architecture Style',
//  authors: [ 'Enrique Molinari' ],
//  publishedYear: 2020,
//  fullName: [Function (anonymous)]
//}
```

As you can see in the previous example code, the two instances of the Book constructor function includes, in addition to the property names (and their values), the function implementation code. Source code is not shared across the instances like it is in class-based languages. This is an implementation detail of the language which if you are not aware of it might lead to inefficient programs. And what about *inheritance* which is a valuable language resource used by developers? If there are no classes, do we have inheritance? Yes, we have. The difference, among others, is that this relation is dynamic, meaning that the inheritance relationship in prototype-based languages can be changed at runtime (as opposed to class-based languages where inheritance is a static relationship that cannot change at runtime). Here is where we have to introduce the concept known as **prototype**.

Each object in JavaScript can have a **prototype** object, to *inherit* properties and methods from it. If you call a property or method in an object and is not defined there, it will **delegate** that call to its prototype. Since that prototype object might have a prototype object too, this delegation will follow until it is found or fails with an error. This is called a **prototype chain**.

Each constructor function has access to a special property called `prototype`, that can be accessed using dot notation: `Book.prototype`. And when you create an instance, you can also access (while in general is not necessary) to this property using: `thisBook.__proto__` or which is the same: `Object.getPrototypeOf(thisBook)`.

Knowing this we can improve our `Book` constructor function defined above in the following way:

```
function Book(name, authors, publishedYear) {
  this.name = name;
  this.authors = authors;
  this.publishedYear = publishedYear;
}
Book.prototype.fullName = function () {
  return this.name + " by " + this.authors + ". " + this.publishedYear;
};

thisBook = new Book("Understanding React",
                        ["Enrique Molinari"], 2021);
archBook = new Book("Coding an Architecture Style",
                        ["Enrique Molinari"], 2020);
//printing thisBook
//Book {
//  name: 'Understanding React',
//  authors: [ 'Enrique Molinari' ],
//  publishedYear: 2021
//}
//printing archBook
//Book {
//  name: 'Coding an Architecture Style',
```

```
23 //  authors: [ 'Enrique Molinari' ],
24 //  publishedYear: 2020
25 //}
```

In the example above we have defined the method `fullName()` in the prototype of the constructor function (line 6). After that, on lines 10 and 12 we are creating two instances and then printing them. Now as you can see the `fullName()` method is not there because it now belongs to their prototype object, shared by the two `Book` instances: `thisBook` and `archBook`. So, what happen if we execute the following statement:

```
thisBook.fullName();
```

JavaScript will try first to find the `fullName()` method in the `thisBook` instance. As it is not defined there, JavaScript will then look in their prototype object and because it's there it will be called.

Every prototype chain will end up pointing to `Object.prototype`. So, if you execute the following:

```
thisBook.valueOf();
```

JavaScript will try to find the method `valueOf()` in the `thisBook` instance. Then on their prototype and finally on the prototype object of their prototype, which is `Object.prototype`. The method is there, so it is called.

Let's now create a basic inheritance example. We are going to create a new `EBook` constructor function that will inherit from `Book`.

```
function EBook(filesize, name, authors, publishedYear) {
  Book.call(this, name, authors, publishedYear);
  this.filesize = filesize;
}
let eBook = new EBook(2048, "Understanding React", ["Enrique
↪ Molinari"], 2021);

//printing eBook:
//eBook:  EBook {
//  name: 'Understanding React',
//  authors: [ 'Enrique Molinari' ],
//  publishedYear: 2021,
//  filesize: 2048
//}
```

Above we have defined the EBook constructor function. On line 2 we are invoking the Book constructor function using the call method which allows us to set the value of this as the current object. This is, somehow, analogous to the use of super(...) inside a constructor in a class to instantiate the parent class and initialise their private members. On line 5 we are creating an instance of EBook and if we print the instance on the console we can see that now we have all the properties from the Book constructor functions on the eBook object. However, we don't yet have the fullName() method that was defined in the Book.prototype. To inherit that method in the EBook instances we have to set the Book.prototype object as the prototype of the eBook instance. We do that below:

```
Object.setPrototypeOf(eBook, Book.prototype);

//Another way of doing the same as above is this:
//thisEBook.__proto__ = Book.prototype;
//However __proto__ is deprecated
```

Object.setPrototypeOf is a method where the first parameter is the instance to have its prototype set and the second parameter is the prototype object to be set.

1.5 Classes

Yes! JavaScript has classes and their syntax is pretty similar to most of the class-based languages you might know. They were added to the language in 2015 as part of the EcmaScript 6. The thing is that classes in JavaScript are a syntactic sugar on top of constructor functions and prototype inheritance. Behind the scenes, everything works like a prototype-based language, even if you define instances from classes. That is why it is important to understand the previous sections.

We will implement our Book and EBook constructor functions with prototype inheritance from the previous section but using classes.

```
class Book {
  constructor(name, authors, publishedYear) {
    this.name = name;
    this.authors = authors;
    this.publishedYear = publishedYear;
  }

```

```
  //this method gets added to the Book.prototype
  fullName() {
    return this.name + " by " + this.authors + ". " + this.publishedYear;
  }
}

class EBook extends Book {
  constructor(filesize, name, authors, publishedYear) {
    super(name, authors, publishedYear);
    this.filesize = filesize;
  }
}
```

What we did in the above example with classes, the `EBook` and `Book` inheritance relationship is the same as what we did with the `EBook` and `Book` constructor functions in the section before. See the following code that demonstrate this:

```
1 let ebook = new EBook(2048, "Understanding React", ["Enrique, Molinari"], 2021);
2 //Book.prototype is the prototype of the ebook instance
3 console.log(Book.prototype.isPrototypeOf(ebook)); //true
4 //fullName method is found on the prototype
5 console.log(ebook.fullName()); //Understanding React by Enrique, Molinari. 2021
6 //EBook is a function not a class
7 console.log(typeof EBook); //function
```

In the example above, we first create an instance of `EBook` and then on line 3 we verify that `Book.prototype` is the prototype of the `ebook` instance. This means that the inheritance relationship was implemented as prototypes just like functions.

1.6 The Multiple Meanings of *this*

It is important to understand how `this` works in JavaScript. Its behaviour depends on where it is used. We have been using `this` in the examples from the previous sections in constructor functions and in classes. And we have not mentioned anything about it because in those examples it behaves just like you know from class-based languages like Java or C#. However, there are some details you should know specially if you use classes for your React components.

So far, if you use this in constructor functions and classes, and create the instances using the new keyword, this is bound to the object being instantiated. However, specifically with constructor functions this won't work as expected if you just call the function like in the next example:

```
1  function Constr(param) {
2    this.param = param;
3  }
4
5  Constr(2); //this is global object window
6  console.log(window.param); //prints 2
```

Calling the constructor function as we are doing on line 5 will bind this to the window global object. So, this example works perfectly, but what it does is probably something you don't expect. This example will end up adding the param property to the window object and assigning to it the value 2.

On classes, on the other hand, if you need to assign or pass as an argument a method, that method will lose the binding of this. Let's study the next example:

```
1  class Person {
2    constructor(name) {
3      this.name = name;
4    }
5    saySomething() {
6      console.log(this.name + " is talking...");
7    }
8  }
9  let enrique = new Person("Enrique");
10 enrique.saySomething(); //Enrique is talking...
11
12 let o = enrique.saySomething; //assigning to a variable
13 o(); //TypeError: Cannot read property 'name' of undefined
```

In the previous example we are defining a class Person. Then on line 9 we are creating an instance enrique and on line 10 we are invoking the saySomething() method. When the method is invoked since this is bound to the object enrique, this.name which was initialised to the "Enrique" string will work and prints "Enrique is talking...". However, on line 12 we are assigning the method to a variable and then using the variable to invoke the method on line 13. At the invocation of the `saySomething()` method (by the `o()` call), `this` is `undefined`, it is not bound to the object `enrique`. Then we will get a *TypeError* message saying that `name` is not a property of `undefined`.

In order to fix this, we have to explicitly bind the value of `this` as we see next:

```
1  class Person {
2    constructor(name) {
3      this.name = name;
4      this.saySomething = this.saySomething.bind(this);
5    }
6    saySomething() {
7      console.log(this.name + " is talking...");
8    }
9  }
10 let enrique = new Person("Enrique");
11 enrique.saySomething(); //Enrique is talking...
12
13 let o = enrique.saySomething; //assigning to a variable
14 o(); //Enrique is talking...
```

On line 4 above we are explicitly binding the value of `this` to the `saySomething` method. Inside the constructor the value of `this` is the object being instantiated. Now, when `saySomething` is invoked by the `o()` call, on line 14, it will work as expected. Another way to fix this is by declaring methods as *arrow functions*.

```
class Person {
  constructor(name) {
    this.name = name;
  }
  saySomething = () =>  {
    console.log(this.name + " is talking...");
  }
}
let enrique = new Person("Enrique");
enrique.saySomething(); //Enrique is talking...

let o = enrique.saySomething; //assigning to a variable
o(); //Enrique is talking...
```

Defining the method using the *arrow function* syntax will work because arrow functions retain the this value of the enclosing lexical scope which in this case is the class. However, arrow functions in classes will not get added to the prototype of the object being instantiated, which means, as we have already discussed, that every instance will have its own copy of the method.

1.7 Modules

The 6th edition of ECMAScript in 2015 has also added the possibility of defining modules. Before this release there were other options to create modular JavaScript programs using tools like RequireJS, among many others. Now we have this functionality supported natively by modern browsers.

Its use is pretty simple. You can define functions, classes, objects, constants in a JavaScript file and export those that you want other modules to be used. Additionally, the client module must import those abstractions that it wants to use. Let's see some code examples.

```
//this is my complex-module.js module

export function complexThing() {
  console.log("a complex thing has been executed...");
}

export let obj = {
  a: 1,
  b: 2,
};

export class ASimpleClass {
  constructor(name) {
    this.name = name;
  }

  print() {
    console.log("printing: ", this.name);
  }
}
```

In the module *complex-module.js* before we are exporting a function, an object and a class. Observe below, we rewrite the *complex-module.js* using a different syntax:

```
1  //this is my complex-module.js module
2
3  function complexThing() {
4    console.log("a complex thing has been executed...");
5  }
6
7  let obj = {
8    a: 1,
9    b: 2,
10 };
11
12 class ASimpleClass {
13   constructor(name) {
14     this.name = name;
15   }
16
17   print() {
18     console.log("printing: ", this.name);
19   }
20 }
21
22 export { obj, ASimpleClass, complexThing };
```

Everything can be `exported` at the end just like we are doing on line 22 above. Of course, you don't have to export everything from a module, just those abstractions that represent the public API of your module. Let's see below how from a module called *main-module.js* you can `import` the abstraction exported by the *complex-module.js*.

```
//this is my main-module.js module

import { complexThing, obj, ASimpleClass } from "./module/complex-module.mjs";

//calling the imported function
complexThing();

//printing the imported object
console.log(obj);

//instantiating the imported class
```

```
12 let o = new ASimpleClass("Enrique");
13 o.print();
```

As you can see on line 3 we are importing the three abstractions that the *complex-module.js* exports. We are able to use the abstractions imported as if they were declared in the *main-module.js* file.

There is a common practice to define a `default export` abstraction from a module in order that client modules can import those a bit easier. In the code below, on line 22, we are exporting the class as our default exported abstraction from the module.

```
1  //this is my complex-module.js module
2
3  function complexThing() {
4    console.log("a complex thing has been executed...");
5  }
6
7  let obj = {
8    a: 1,
9    b: 2,
10 };
11
12 class ASimpleClass {
13   constructor(name) {
14     this.name = name;
15   }
16
17   print() {
18     console.log("printing: ", this.name);
19   }
20 }
21
22 export default ASimpleClass;
23 export { obj, complexThing };
```

On line 3 below, note how we are importing the default exported abstraction with a different name (it is an alternative, but we can use the same name). Note also that there are no curly braces.

```
1 //this is my main-module.js module
2
```

```
import AClass from "./module/complex-module.mjs";
//This line below is the same as the one above
//import { default as AClass } from
↪  "./module/complex-module.mjs";
import { obj, complexThing } from
↪  "./module/complex-module.mjs";

let o = new AClass("Enrique");
o.print();

//... more code here
```

1.8 Single Thread Language

As per the definition about JavaScript we gave at the beginning of this chapter, we know that JavaScript is a **single threaded** language. This means that the execution of a program is one statement at a time. This might hurt the performance of the program if there are some statements that take some time to finish. For this reason, JavaScript supports *asynchronous* operations. So, how does this work? The JavaScript interpreter can delegate the execution of some statements to the Browser and continue executing the program without waiting for them to finish. After the Browser finishes the execution of a delegated statement, it is returned to the JavaScript interpreter as *callbacks*. Among the statements that the interpreter can delegate to the browser we have events (`onClick`, `onMouseOver`, etc), `fetch` and `XMLHttpRequest` (ajax calls), `setTimeout`, etc.

To handle this, the execution environment of JavaScript includes the following elements: the *call stack*, the browser's Web APIs, a *callback queue* and the *event loop*. Let's see the following simple example:

```
1 console.log("starting");
2
3 setTimeout(() => {
4   console.log("callback");
5 }, 1000);
6
7 console.log("finishing");
```

Below we can see the sequence of tasks to execute of the program above:

1. Statement on line 1 is pushed on to the *call stack* and executed. `"starting"` is printed on the console.

2. The `setTimeout` on line 3 is delegated to the browser's Web API to be executed. Which basically waits for one second. However execution of the program continues with the statement of line 7, it does not wait because the execution was delegated to the browser's Web API.

3. Statement on line 7 is pushed on to the *call stack* and executed. `"finishing"` is printed on the console.

4. After one second elapsed from the `setTimeout` function, the callback arrow function passed as the first argument was then pushed into the *callback queue*. Since there are no more statements to be executed on the *call stack*, the *event loop* gets from the top of the *callback queue* the arrow function and pushes it into the *call stack*. Finally it gets executed. `"callback"` is printed on the console.

Note that all the callback functions that end up in the *callback queue* get executed after the call stack is empty and not before (when the execution of the program ends with the last statement). To be very clear with this, look at the example below.

```
1 console.log("starting");
2
3 setTimeout(() => {
4   console.log("callback");
5 }, 0);
6
7 console.log("finishing");
```

Note that on line 5 we are passing `0` seconds to the `setTimeout` function telling the Web API to not wait to push the callback arrow function into the *callback queue*. In any case, the result and the order of the console messages is the same as the example before: `"starting"`, `"finishing"`, `"callback"`.

We can expect exactly the same behaviour from the example below that perform an ajax call:

```
console.log("starting");
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then((response) => response.json())
  .then((json) => console.log(json));
console.log("finishing");
```

Ajax Call

`fetch` is delegated to the Web API which performs an ajax call. Once the server respond, the callback arrow functions are pushed into the *callback* *queue*. The first callback function, on line 3, transforms the response obtained from the server to json and the next callback function on line 4 prints that json in the console. Only after printing on the console the text `"finishing"` on line 5, is when those callbacks functions are pushed into the *call stack* and executed. You can find a more detailed explanation about the JavaScript interpreter and how it works, in the great talk by Philip Roberts[5].

1.9 The Promise Object and the async/await Keywords

The Promise Object was introduced in Javascript in ES2015. This object represents an asynchronous operation that might finish successfully or fail. See below how to create an instance of a Promise:

```
let p = new Promise(function (resolve, reject) {
  //function to be executed by the constructor
});

//do something with p
```

As we can see above, the Promise constructor receives a function, called *executor*, that will be invoked by the constructor. The *executor* function receives two additional functions as parameters: `resolve(value)` and `reject(reason)`. The body of the *executor* function performs, typically, an asynchronous operation and finish by calling the `resolve(value)` function if everything goes well or `reject(reason)` otherwise. See how this is done below:

```
1 let p = new Promise(function (resolve, reject) {
2   //long async operation
3   setTimeout(() => resolve("finished"), 1000);
4 });
```

In the example above using `setTimeout` (on line 3) we are simulating an operation that takes one second to finish. After that operation finishes it will call the `resolve` function passing as value the string `"finished"`. What can we do with that then? The Promise object have the `then(handleResolved)` method that receives a function that allows you to work with the parameter passed when you invoke the `resolve` function like we do on line 6 below:

```
1 let p = new Promise(function (resolve, reject) {
2   //long async operation
```

```
3     setTimeout(() => resolve("finished"), 1000);
4   });
5
6   p.then((value) => console.log(value));
```

As we can see above, on line 6 we are passing the `"finished"` string as a parameter named `value` to the arrow function passed to the `then(handleResolved)`. Then, that `value` is just printed (`"finished"` is printed on the console). What is important to note here is that the `handleResolved` function passed to the `then(...)` method is executed only once the promise is resolved.

Suppose that now, something goes wrong with the *executor* and the `reject` function is invoked. Then, it is possible to handle that in the following way:

```
1   let p = new Promise(function (resolve, reject) {
2     //long async operation
3     setTimeout(() => reject("can't be done..."), 1000);
4   });
5
6   p.then((value) => console.log("success: " + value))
7    .catch((value) => console.log(value));
```

Note that on line 3 now we are calling the `reject(reason)` function passing the `reason` value as the string `"can't be done"`. Then, on line 7 note that now we are using the `catch`, which is the one that will be invoked in this case.

During this book, and usually in React we don't write promises, but we use them frequently. By using the `fetch` method to retrieve data from an external API, we have to deal with a promise. Look at the example below:

```
1   function fetchPost() {
2    fetch("https://jsonplaceholder.typicode.com/posts/1")
3      .then((response) => response.json())
4      .then((json) => console.log(json));
5   }
6
7   fetchPost();
```

As you can see on line 2 above, the `fetch` method returns a promise, which allows us to call the `then` method on it, to work with the response data.

Another way to deal with promises is by using the `async` and `await` keywords. These keywords were added to Javascript on ES2017 allowing us to write asynchronous code in a synchronous way. Let's rewrite one of our previous examples to take advantage of these keywords. First, we will create a function that returns a promise. Functions that return promises are (usually) asynchronous:

```
function thePromise() {
  return new Promise(function (resolve, reject) {
    //long async operation
    setTimeout(() => resolve("finished"), 1000);
  });
}
```

See below how we can invoke the `thePromise()` function:

```
1 async function testingKeywords() {
2   console.log("before");
3   const data = await thePromise();
4   console.log("after");
5   console.log(data);
6 }
7
8 testingKeywords();
```

First note that we have to wrap the calling code in a function. And that function must be declared `async` (see line 1). Then, we use the `await` keyword before calling the `thePromise()` function, on line 3. `data` (the right hand side of the assignment on line 3) is actually the `"finish"` string, and not the promise returned by `thePromise()` function. Why? because the execution of the code inside the `async` function is paused until the promise is resolved (or rejected). Messages inside the `async` function are printed on the console in the same order as the order of the written statements. That is why we can say that using these keywords allow us to write asynchronous code that reads like synchronous.

Now we are going to rewrite the `fetchPost()` function to use these new keywords. See below:

```
1 async function fetchPost() {
2   let data = await
    ↪ fetch("https://jsonplaceholder.typicode.com/posts/1");
3   data = await data.json();
4   console.log(data);
5 }
6
7 fetchPost();
```

As you can see, again, we are declaring the function `async`, and in this case the `fetch` call on line 2, is prepended with the `await` keyword. Prepending the `await` keyword to the `fetch` function means that instead of returning a Promise object, it returns (if the promises resolve, you can use a try/catch block to handle errors) a Response object. That allows us to call on line 3 directly to the `json()` method of the Response object. As that method returns a Promise, we also prepend the sentence with the `await` keyword, giving us a Javascript object that is finally printed on the console.

It is important to note that `await` can only be used inside an `async` function. Let's see below how to handle errors in an `async/await` function:

```
async function fetchPost() {
 try {
  let data = await
   ↪ fetch("https://jsonplaceholder.typicode.com/posts/1");
  data = await data.json();
  console.log(data);
 } catch(error) {
  //handle the error here
 }
}

fetchPost();
```

And finally, `async` functions returns always a `Promise`. Suppose the example below where we have an `async` function that returns the json response from a `fetch` request. And with that, we would like to print it on the console. If we do it like below (see line 6), we will get printed `Promise { <pending> }` on the console.

```
1 async function fetchPost() {
2   let data = await
    ↪ fetch("https://jsonplaceholder.typicode.com/posts/1");
```

```
  return await data.json();
}

console.log(fetchPost());
```

Instead, we have to write:

```
fetchPost()
  .then((data) => console.log(data));
```

Part II

Understanding React

Chapter 2

Essential React Concepts

In this chapter we will learn the core concepts behind react. With different examples and with a different approach, we mainly follow the learning path suggested by the official docs[2], which is fantastic.

2.1 React Principles

I have to start saying that I *love* React! I love it because I love designing software in a professional way. With that I mean with practices and mainly with syntactical constructions that help keeping application's source code *modifiable* after several years of iterations. We say a software is modifiable, if I know on every change, where that change will impact. In React you design applications by assembling **components**, which are built by using plain JavaScript classes or functions. The guys behind React's design decisions have challenged very established patterns like MVC, where you have the display logic and the markup separated in different abstractions. They say that these elements are naturally coupled. In React you have the display logic (fetching data, event handling, etc) and the markup in the same abstraction: the component. And each component represents a fragment of functionality of your View. This is what makes React so great and the reason I love it. And among other features provided by React, understanding how to build applications by assembling components is the main goal of this book.

I really recommend you to see the explanation about react design principles by Pete Hunt: React - Rethinking Best Practices.

2.2 Creating a React Project

To start with React we will first create a React project using a tool called Create React App. With this tool, as you will see, you can create a starter React project without configuration. To do it, run the following command on the console:

```
$ npx create-react-app coding_in_react
```

When finished, if everything went fine, you will see a message like:

```
Success! Created coding_in_react at /home/react/coding_in_react
```

Run the following commands to start your application:

```
$ cd coding_in_react
$ npm start
```

That will open your default browser with the URL http://localhost:3000/, with the starter application running. Let's now open VS Code to see what is inside our project folder. To do that, you can type on your console the following:

```
$ cd coding_in_react
$ code .
```

This will open VS Code with the project folder `coding_in_react` ready to be used. Let's have a look at the project folder structure below. There is a brief explanation of what each item is.

```
coding_in_react
├── node_modules This folder contains packages installed by the
│                Create React App tool.Any package that we install
│                will end up here.
├── public This folder contains the index.html file (among others)
│          which creates the main skeleton layout of your web
│          application.
├── src This is the folder where your React source files will reside.
└── package-lock.json Here you will find the exact version of each
                      of the packages (and the dependencies of the
                      packages) that your project depends on.This file
                      should be changed only by using npm commands.
```

```
└─ package.json Here you will find the packages that your project
                depends on.This file should be changed only by
                using npm commands.
└─ README.md This is a Markdown file that contains documentation
             about the project.
```

In the next section we will start coding in React using this project. I will refer to the folders and files there when necessary.

2.3 React Components

React applications are built by creating components. Component is probably one of the most confusing terms in software engineering. So, we will provide our definition of what a component is in React. A React component is a self contained piece of functionality that is created by using a plain JavaScript `class` or `function`. It manages its own state and ideally performs a single task. It might collaborate with other components and knows how to paint itself on the browser.

In the previous paragraph I said that React components know how to paint itself in the browser and they can be defined using the `class` or the `function` syntactical constructions. If you use a `class` you have to add a method called `render`, which is the method invoked by React when the component requires it to be painted in the browser. And if you use a `function`, it is the function that gets invoked by React when the component requires it to be painted in the browser.

Let's create our first React component using a JavaScript `class`:

```
import { Component } from "react";

export default class Person extends Component {
  render() {
    return <p>This is a <strong>Person</strong>
    ↪  Component</p>;
  }
}
```

You first need to `import` the `Component` class from React Core, because your JavaScript classes must extend from it. And finally you have to define the `render` method that is invoked to paint the component on the browser. Note that this method just returns HTML (or at least looks like HTML as we will see later). Below we create the same component as the one above but using a `function` instead of a class:

```
export default function Person() {
  return <p>This is a <strong>Person</strong> Component</p>;
}
```

2.4 Rendering Components

I have my first component, how do I get it rendered into the DOM? To answer this question we will learn some React concepts.

Take a look again at the component we have created in section 2.3, the "HTML" snippet that the function component returns (or the render method in the class-based component) is not really HTML. It is called JSX, which stands for **JavaScript XML**. It is a *syntax extension* to JavaScript and what the React team recommends to use to paint your components on the browser.

In JSX, you can embed any valid JavaScript Expression between curly braces. As an example, below you can see the variable `name` declared and initialised (on line 2) and used inside the JSX syntax (line 6).

```
1 export default function Animal() {
2   let name = "Eze the Dog";
3
4   return (
5     <p>
6       This is <strong>{name}</strong>
7     </p>
8   );
9 }
```

Browsers do not understand JSX syntax. To make it work we have to translate JSX into JavaScript, using a compiler like Babel. If you create your React application using the `create-react-app` tool like we did before, you get this covered without needing to deal with it.

JSX gets translated into a JavaScript expression which at execution time, is evaluated along with the expressions defined by you in curly braces and painted on the browser (injecting in the DOM). As JSX are expressions, it is possible to see a JSX piece of code as a first class object. Which allows you to do, for instance, what you see below:

```
function passingAsArgument(jsx) {
  return jsx;
```

```
3  }
4
5  export default function Animal() {
6    let name = "Dog";
7    //assign a JSX block to a variable
8    let jsx = passingAsArgument(
9      <p>
10       This is a <strong>{name}</strong>
11     </p>
12   );
13
14   return jsx;
15 }
```

In the example above, we are calling a function passing a JSX expression as an argument and the return of that function (which is just this same argument) is assigned to the variable `jsx` on line 8.

As you might have noted, JSX is the tool you use in React to paint the components you create on the browser. Let's start writing some code. In the VS Code project we have created in section 2.2, create a JavaScript file called `src/Person.js` with the `Person` component we have created in section 2.3. Then, open the file `src/index.js`, delete their content and paste the following:

```
1  import React from "react";
2  import ReactDOM from "react-dom/client";
3  import "./index.css";
4  import Person from "./Person";
5
6  const root =
   ↪ ReactDOM.createRoot(document.getElementById("root"));
7  root.render(
8    <React.StrictMode>
9      <Person />
10   </React.StrictMode>
11 );
```

The `src/index.js` file is our JavaScript main module (the entry point). In this main file we call first the `ReactDOM.createRoot(document.getElementById("root"))` (line 6 above) passing as argument the DOM element in which a JSX expression will be rendered (injected). If you check the `public/index.html`, you will see the markup `<div id="root"></div>` we are referencing on line 6 above. The JSX expression that will be injected there is passed as argument to the `root.render` function (line 7 above).

> When you create a React project with create-react-app as we did, there are several things that happen unders the hood. One is the translation of JSX to JavaScript as we mentioned. But there are few others that are important to understand. create-react-app uses Webpack, a static module bundler to help improve the performance of your application by combining multiple static files into one, which reduces the number of requests the browser requires to do to the server to get static assets. Especially when you build a single page application with JavaScript tools like this becomes very important. So, Webpack under the hood, takes the `src/index.js` as the entry point to produce the bundler. It uses the `public/index.html` as the HTML template, which changes to inject a `<script>` tag with the path to the js bundler. This magic happens when you run `npm start` and `npm run build`. You can check this explanation directly from one of the React core developers.

On line 9, highlighted above, we are telling React to render the `Person` component. That will instantiate the `Person` `class` and execute the `render` method, in the case of a class-based component, or execute the `Person` `function` in the case of a function-based component. In either case, the output is inserted into the DOM, generating what is shown below:

```
1 <div id="root">
2  <p>This is a <strong>Person</strong> Component</p>
3 </div>
```

The `div` element on line 1 above comes from the `public/index.html` and the `p`, the paragraph with the text inside, comes from rendering the `Person` component. If you start the application as we explained in the section 2.2, you will see the results in the browser. You can also inspect the DOM with the browser's development tool.

Additionally, note that in the `src/index.js`, the `<Person/>` component is wrapped by the `<React.StrictMode>` component which helps us during development to inform us about potential problems with our React code. Have a look at strict mode in React official docs.

And finally, note that component names, in this case `Person`, start with a capital letter. React sees components starting with a capital letter as custom components (your components) and that requires the function or class definition to be in scope (that is why in the `src/index.js` we have to import the `Person` component). And React sees components starting with a lowercase letter as DOM tags, like `div`, `p`, `strong`, etc.

2.4.1 Styling

In this section, we are going to describe how you can add style to your React components. In the example below we are adding style to a component using a CSS file. Suppose the following CSS file called `StyleDemo.css`:

```css
.big {
  background-color: red;
  height: 200px;
  width: 200px;
  font-size: 30px;
  color: blue;
}

.small {
  background-color: green;
  height: 40px;
  width: 40px;
  font-size: 14px;
  color: black;
}
```

Below, we have created a component to use the `StyleDemo.css` file. It will paint on the browser two squares, one green and small and the other red and bigger:

```jsx
import "./StyleDemo.css";
import React, { Component } from "react";

export default class StyleAComponent extends Component {
  render() {
    return (
      <>
        <div className="small">Hello World!</div>
        <div className="big">Hello World!</div>
      </>
```

```
11        );
12      }
13    }
```

On line 1 we are first importing the CSS file which is located in the same directory of the component. Note that on lines 8 and 9 we are using the `className` attribute to assign specific style classes to `div` elements, instead of using the `class` attribute like in plain HTML. As we have mentioned, JSX is translated to JavaScript before execution, and `class` is a reserved word in JavaScript. That is the primary reason that forces React to use `className` in JSX instead of `class`. To try this example on VS Code, create the files `StyleDemo.css` and `StyleAComponent.js` under the `src` folder and then on the `src/index.js`, paste this:

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import StyleAComponent from "./StyleAComponent";

const root =
 ↪ ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <StyleAComponent />
  </React.StrictMode>
);
```

It is also possible, but not recommended, to use inline styling. The `style` attribute of JSX elements accepts a JavaScript object with *camelCased* properties, as demonstrated below:

```
import React, { Component } from "react";

export default class StyleInLine extends Component {
  render() {
    const colorBoxStyle = {
      width: "50px",
      height: "50px",
      border: "1px solid rgba(0, 0, 0, 0.05)",
      backgroundColor: "green",
    };

```

```
      return (
        <>
          <div style={colorBoxStyle}>Hello World!</div>
        </>
      );
    }
  }
```

2.5 Props

In React you can pass arguments to components. These arguments are transformed into a single JavaScript object with a special name: `props` (which stands for properties). Below you can see how we use `props` in the `Person` class-based component:

```
export default class Person extends Component {
  render() {
    return (
      <>
        My name is
        <strong>
         {this.props.name + " " + this.props.surname}
        </strong>
      </>
    );
  }
}
```

And now the equivalent but using the function-based component:

```
export default function Person(props) {
  return (
    <>
      My name is
      <strong> {props.name + " " + props.surname}</strong>
    </>
  );
}
```

You might note the use of an empty tag `<>...</>` in the components definition above. React components must always return an element.

> That forces you to wrap everything into a root element. In previous versions of React, it was necessary to use elements like <div> as root, but that introduces an extra node in the DOM. That is why React has introduced what is called fragments.

Note that in our `Person` component, the `props` object has two properties: `name` and `surname`. In the class-based component we accessed to them using `this.props` and in function-based component just use the argument of the function.

How do you then pass `name` and `surname` to the component? You have to define them as JSX attributes, lets see that below:

```
1  import React from "react";
2  import ReactDOM from "react-dom/client";
3  import "./index.css";
4  import Person from "./Person";
5
6  const root =
   ↪ ReactDOM.createRoot(document.getElementById("root"));
7  root.render(
8    <React.StrictMode>
9      <Person name="Enrique" surname="Molinari" />
10   </React.StrictMode>
11 );
```

When React sees attributes in a user-defined component, like what we have on line 9 above, it passes them as the single object `props`.

In the next example we have refactored the `StyleAComponent` component from the previous section, to accept as `props` the `className` value to use:

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import StyleAComponent from "./StyleAComponent";

const root =
↪ ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <StyleAComponent square="small" />
```

```
      <StyleAComponent square="big" />
    </React.StrictMode>
  );
```

```
1  import "./StyleDemo.css";
2  import React, { Component } from "react";
3
4  export default class StyleAComponent extends Component {
5    constructor(props) {
6      super(props);
7    }
8
9    render() {
10     return (
11       <>
12         <div className={this.props.square}>Hello
           ↪  World!</div>
13       </>
14     );
15   }
16 }
```

Note that now the `StyleAComponent` renders a single `div` element and the value of the `className` attribute is obtained from a prop called `square`. Additionally, if you define a `constructor` in a class-based component, like we did before on line 5, the constructor must receive `props` as argument and the first statement must be `super(props)`.

2.6 State

Suppose that we want to build a `CountDownLatch` component. Starting from a positive integer, on each second it is decremented until it arrives at zero. Using what we have learned so far, we can create the component below:

```
import React, { Component } from "react";

export default class CountDownLatch extends Component {
  render() {
    return <h1>{this.props.startFrom}</h1>;
  }
}
```

The component just paints the property `props.startFrom` wrapped in an `<h1>` tag on the browser. Then on the `src/index.js` we can do something like this:

```
1  import ReactDOM from "react-dom/client";
2  import React from "react";
3  import CountDownLatch from "./CountDownLatch";
4
5  const root = ReactDOM.createRoot(document.getElementById("root"));
6
7  function countDown(number) {
8    root.render(
9      <React.StrictMode>
10       <CountDownLatch startFrom={number} />
11     </React.StrictMode>
12   );
13   if (number === 0) {
14     clearInterval(intervalId);
15   }
16 }
17
18 let startFrom = 10;
19 let intervalId = setInterval(() => {
20   countDown(startFrom);
21   startFrom = startFrom - 1;
22 }, 1000);
```

The function on line 7 above, renders the `CountDownLatch` component on the browser passing the prop `number` to be painted on the browser. Then, we have starting on line 19 the `setInterval` JavaScript function which executes the arrow function (lines 20 and 21) every second. So, every second, the `CountDownLatch` component is repainted on the browser, each time with the `number` passed as prop decremented by one. Until it arrives at zero where the `clearInterval()` is executed, stopping the countdown (line 14).

And that works great. However, in the way we have implemented this, the logic that does the decrements, update the browser every second and stop the countdown is outside the component. We can implement our component in a much better way incorporating the logic *inside* the component. This will give us a much more reusable `CountDownLatch` component. In order to do this we must add **state** to our component. The **state** in React components is not like the classic state you know from objects in the object oriented paradigm. React components *react* to state changes performing the render again (also known as re-painted or re-rendered). Let's see how we add `state` to our `CountDownLatch` component:

```
1  import React, { Component } from "react";
2
3  export default class CountDownLatch extends Component {
4    constructor(props) {
5      super(props);
6      this.state = {
7        startNumber: this.props.startFrom,
8      };
9    }
10
11   render() {
12     return <h1>{this.state.startNumber}</h1>;
13   }
14 }
```

`state` is managed and controlled by the component. On line 12, we have changed the use of props to use the `state` instead. And on the constructor starting on line 4, we are initialising the `state` with a value coming in a `prop`. Note that `state` is a JavaScript object, but it has special meaning for React. Then on the `src/index.js` we add the JSX elements to paint our component on the browser:

```
1  import ReactDOM from "react-dom/client";
2  import React from "react";
3  import CountDownLatch from "./CountDownLatch";
4
5  const root = ReactDOM.createRoot(document.getElementById("root"));
6
7  root.render(
8    <React.StrictMode>
9      <CountDownLatch startFrom={10} />
10   </React.StrictMode>
11 );
```

As you can see on line 9 above we are passing the value `10` as `props` which is used to initialise the `state` in the component's constructor. So far, if we run this, we will have just this markup `<h1>10</h1>` painted on the browser. Now we have to add the logic that decrements our number. To do this, we will take advantages from what React calls **lifecycle** methods. These methods are present only in class-based components and they are hooks that you can use to plug your logic. Calling them hooks can be confusing because React Hooks is a big new topic that we will cover in the next section, but in object oriented frameworks literature hooks are called to those extension points that you have to customise the framework. And this is exactly that, they are a set of methods that React calls at *certain moments* and you can use to implement component's specific logic.

One of these lifecycle methods that we will use is `componentDidMount()`. This method is called by React after the `constructor` is executed and after `render` is executed. So, it is called just after our component is rendered into the DOM **for the first time**. It is called the *mounting* moment. This seems to be the perfect moment to set up the `setInterval` that performs the decrement. Let's see how this looks:

```
1  import React, { Component } from "react";
2
3  export default class CountDownLatch extends Component {
4    constructor(props) {
5      super(props);
6      this.state = {
7        startNumber: this.props.startFrom,
8      };
9    }
10
11   componentDidMount() {
12     this.intervalId = setInterval(() => {
13       this.setState((state) => ({
14         startNumber: state.startNumber - 1,
15       }));
16     }, 1000);
17   }
18
19   render() {
20     return <h1>{this.state.startNumber}</h1>;
21   }
22 }
```

On line 11 we have added the method `componentDidMount()` where we use to set up the decrement of the countdown. Note that each second the arrow function, passed as the first argument of the `setInterval` function, that decrements the `state.startNumber` by one is executed. To update the state we have to use the special `setState()` method (line 13 above), which triggers the *re-render* of the component. So, on each second the `this.state.startNumber` is decremented by one causing the component to be re-painted on the screen.

What is still missing in our component is to stop the countdown when it arrives at zero. To implement this, we have used another lifecycle method called `componentDidUpdate(prevProps, prevState)`. This method is executed when there is an update on `props` or `state` of the component, but after render the component on the DOM. By parameter receives the previous value from the props and the previous value from the state. Let's add then the logic to stop the countdown on this method to finish our component.

```
1  import React, { Component } from "react";
2
3  export default class CountDownLatch extends Component {
4    constructor(props) {
5      super(props);
6      this.state = {
7        startNumber: this.props.startFrom,
8      };
9    }
10
11   componentDidMount() {
12     this.intervalId = setInterval(() => {
13       this.setState((state) => ({
14         startNumber: state.startNumber - 1,
15       }));
16     }, 1000);
17   }
18
19   componentDidUpdate() {
20     if (this.state.startNumber === 0) {
21       clearInterval(this.intervalId);
22     }
23   }
24
25   render() {
```

```
26      return <h1>{this.state.startNumber}</h1>;
27    }
28  }
```

On line 19 above, we have added the lifecycle method `componentDidUpdate()`. As we can see on figure 2.1, after the *mounting* phase is finished, we have the *updating* phase, triggered by calling (among other ways) the `setState()` method, which first invokes the `render` and after that the `componentDidUpdate()`. There, if we arrive at zero we stop the countdown by calling the `clearInterval()` function (line 20, 21). Remember that the arrow function we have set up in the `setInterval()` on the `componentDidMount()` method will be executed every second, triggering the updating phase each time due to the call to the `setState()` method.

Let's try now our self-contained `CountDownLatch` component in action. In VS Code, create the file `src/CountDownLatch.js` and paste the source code from the `CountDownLatch` component above. Then, paste the next lines on the `src/index.js`[1] file.

```
import ReactDOM from "react-dom/client";
import React from "react";
import CountDownLatch from "./CountDownLatch";

const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(<CountDownLatch startFrom={10} />);
```

So far, we have seen how to use state and the lifecycle methods to create a self-contained and reusable component. And we have reviewed how changes in the component's state triggers the updating phase which among other things, re-render the components on the browser. There are some more lifecycle methods in React. Dan Abramov (from the React's team) has shared a picture to summarize all the lifecycle methods available. Note from that picture that there is a third phase called unmounting. That phase has a single lifecycle method called `componentWillUnmount()` which is executed before the component is unmounted from the DOM. This method is used for clean up or close resources: subscriptions, sockets, timers, etc.

[1] Note that below I'm not using React.StrictMode, this is because there is a strange bug that when using it causes the componentDidMount being called twice and that makes our countdown decrements by two each second.

Few important notes to manage state in the correct way:

- The constructor is the only place where you can change the state directly. In all other places use `setState(...)`.

```
//only do this inside the constructor
this.state.startNumber = 4;
```

- State Updates may be asyc: React may batch multiple `setState()` calls into a single update for performance. Because `this.props` and `this.state` may be updated asynchronously, you should not rely on their values for calculating the next one. In our `CountDownLatch` component we are updating the state on line 14 based on their previous value. In these cases you should not trust in the state current value. Use the one passed as argument on the `setState()` method, like below:

```
//Don't do this
this.setState(() => ({
  startNumber: this.state.startNumber - 1,
}));

//Do this
this.setState((state, props) => ({
  startNumber: state.startNumber - 1,
}));
```

- In the `componentDidUpdate(prevProps, prevState)` lifecycle method you might need to call the `setState()` method. When this is needed, make sure you wrap that call in a condition to avoid an infinite loop. Usually, the condition is based on the previous values of the props and/or state with the current values of them.

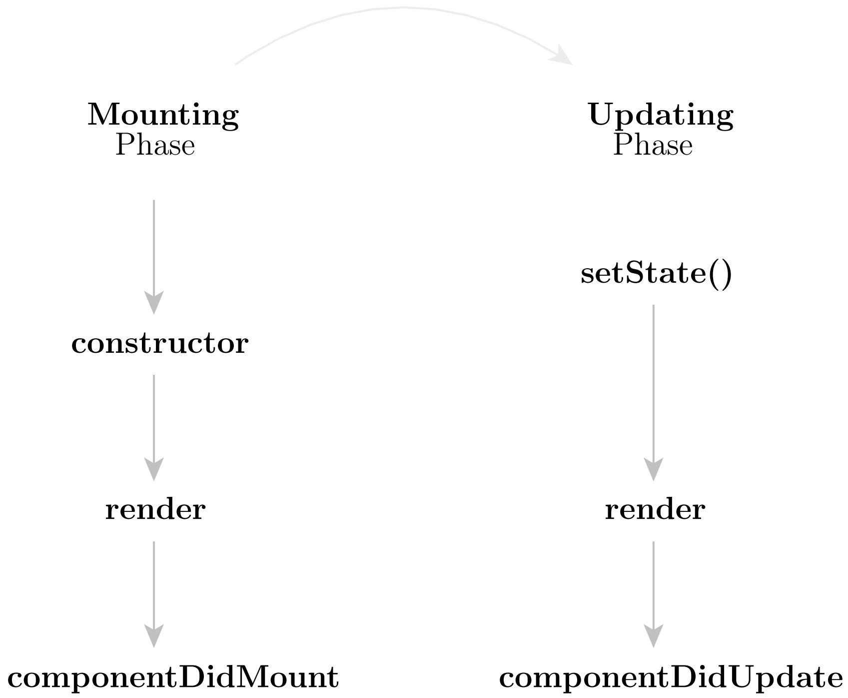


Figure 2.1: Order of execution of lifecycle methods used in the CountDownLatch component

2.7 Dealing with Events

Defining events in React is not that different from defining them on HTML. The big different is that in React you define the events on JSX, which is then translated to JavaScript, so there are two important things you should know:

- Event names in JSX are defined in *camelCase*.
- You have to pass a function to handle the event.

Look at the example below:

```
<button onClick={clickMe}>Click me</button>
```

Note that the event name `onClick` is written in *camelCase*. As we have mentioned, in JSX between curly braces you can define any valid JavaScript expression. In this case we are passing the function `clickMe` to be used to handle the event. As we have seen in section 1.2, functions are first-class objects in JavaScript allowing us to do that. Let's create a function-based component to use some events:

```
1  export default function EventExample() {
2    function onOver() {
3      console.log("onOver...");
4    }
5
6    function clickmeLink(e) {
7      //this is necesary to prevent the default
8      //link behavior
9      e.preventDefault();
10     console.log("link clicked...");
11   }
12
13   return (
14     <div>
15       {/*events are camelCase*/}
16       {/*and they receive a function not a string*/}
17       <a href="#" onMouseOver={onOver}
          ↪  onClick={clickmeLink}>
18         click this link
19       </a>
20     </div>
21   );
22 }
```

On line 17 above we have defined an anchor element, with two events: `onMouseOver` and `onClick`, each of them receives the function that will handle these events. These handlers are defined on lines 2 and 6, which just print on the console some text. Note that on the function `clickmeLink(e)` on line 6 we are receiving as argument an instance of the event, but is not the browser's native event, it is an instance of SyntheticEvent, a React wrapper to make events work identically across browsers. And in this case we use it to prevent the default behaviour of clicking an anchor element.

Now, lets see how this example can be translated into a class-based component:

```
import React, { Component } from "react";

export default class EventExample extends Component {
  constructor(props) {
    super(props);
```

```
6      this.onOver = this.onOver.bind(this);
7      this.clickmeLink = this.clickmeLink.bind(this);
8    }
9
10   onOver() {
11     console.log("onOver...");
12   }
13
14   clickmeLink(e) {
15     //this is necesary to prevent the default
16     //link behavior
17     e.preventDefault();
18     console.log("link clicked...");
19   }
20
21   render() {
22     return (
23       <div>
24         {/*events are camelCase*/}
25         {/*and they receive a function not a string*/}
26         <a href="#" onMouseOver={this.onOver}
           ↪  onClick={this.clickmeLink}>
27           click this link
28         </a>
29       </div>
30     );
31   }
32 }
```

Since event handlers are class methods, on line 26 we can see that they are passed using `this`. However, as we have explained in section 1.6, if we assign or pass as an argument a method, we lose the binding of `this`, in this case, to the component instance. That is why we have to explicitly set this binding as we do on lines 6 and 7. Other than that, it is pretty similar to what we did on the function-based component.

Here is another example, in this case we change the `state` on the event handler:

```
import React, { Component } from "react";

```

```
export default class ColorSelect extends Component {
  constructor(props) {
    super(props);
    this.state = {
      value: "white",
    };
    this.colorChanged = this.colorChanged.bind(this);
  }

  colorChanged(e) {
    this.setState({
      value: e.target.value,
    });
  }

  render() {
    const colorBoxStyle = {
      width: "30px",
      height: "30px",
      border: "1px solid rgba(0, 0, 0, 0.05)",
      backgroundColor: this.state.value,
    };

    return (
      <>
        <label for="colors">Choose your favourite color: </label>
        <select name="colors" onChange={this.colorChanged}>
          <option>Options...</option>
          <option value="blue">Blue</option>
          <option value="red">Red</option>
          <option value="green">Green</option>
          <option value="yellow">Yellow</option>
        </select>
        <div style={colorBoxStyle}></div>
      </>
    );
  }
}
```

In the example above we have a drop down list with colors. When you choose one color, the `onChange` event (line 29) is triggered, calling the method `colorChanged()` (line 12). The `state` gets updated with the color selected. Changing the `state` makes the render to be executed, painting the square box (line 36) now colored with the chosen color.

It is important to clarify that React is very efficient in updating the DOM. Only the exact portion of the DOM that has changed is the one that is updated. The rest remain untouched. You can check what I'm saying by running this component inspecting the DOM with the Browser's DevTool. This is achieved by React using what is called Virtual DOM and the reconciliation.

We will look at more events later in the book. Here is the full list of supported events.

2.8 Hooks

Hooks were added in React 16.8, allowing you to use `state` and to plug your code in the lifecycle of your function-based components. Before React 16.8 was released there was no way to use the `state` in function-based components. As soon as you needed `state` you had to move to class-based components. That seems kind of obvious, right? Functions allocate memory at call time to store local variables and parameters, but that allocation is removed after the function returns. On the other hand, classes has state (instance members) shared across their methods, allocated at instantiation time and removed, somehow explicitly (C++) or by a garbage collector at some point. As we explained, in React when the `state` changes the component gets re-rendered. With classes, the language gives you the tool to do this, that is why `state` is just an instance variable and `setState()` ends up calling the `render()` method to have the component re-rendered. So, to implement this in function-based components, React had to take care of maintaining a shared state between function calls. That was a bit of implementation details. Let's move to understand how to use hooks and we will delve into these implementation details later.

Hooks are just functions. Reacts provides some built-in hooks and in this section we will explain the two most used ones:

- `useState(initialState)`
- `useEffect(callback[, dependencies])`

These are the hooks that allow you to have state and lifecycle inside function-based components.

2.8.1 useState

As mentioned, hooks are special functions that allow you to use React features. `useState(initialState)` is the hook that gives you the ability to add `state` to a function-based component. To start learning this hook we will migrate the `ColorSelect` class-based component from the previous section to a function-based component.

We first start below importing the `useState` function from React, and then call it from inside our component:

```
1 import { useState } from "react";
2
3 export default function ColorSelect() {
4   //declaring the color state variable
5   const [color, setColor] = useState("white");
6   .
7   .
8   .
9 }
```

On line 5 above, by calling the `useState` function we are declaring a `state` variable. This function returns an array with two values: the *current state value*, and a function to *update* that *value*. Note that we use the destructuring initialisation we showed in section 1.3. The current state value is set into the variable called: `color` and the `setColor` function is what we have to use to update it. `useState` receives as a parameter the initial value for the `state` variable. In our case, `color` is initialised with `"white"`.

See below the full implementation of the `ColorSelect` using a function-based component.

```
import { useState } from "react";

export default function ColorSelect() {
  const [color, setColor] = useState("white");

  function colorChanged(e) {
```

```
7       setColor(e.target.value);
8     }
9
10    const colorBoxStyle = {
11      width: "30px",
12      height: "30px",
13      border: "1px solid rgba(0, 0, 0, 0.05)",
14      backgroundColor: color,
15    };
16
17    return (
18      <>
19        <label for="colors">Choose your favourite color:
            ↪ </label>
20        <select name="colors" onChange={colorChanged}>
21          <option>Options...</option>
22          <option value="blue">Blue</option>
23          <option value="red">Red</option>
24          <option value="green">Green</option>
25          <option value="yellow">Yellow</option>
26        </select>
27        <div style={colorBoxStyle}></div>
28      </>
29    );
30  }
```

So, when this function-based component is mounted on the DOM, which is the first time the component is rendered (or what is the same, the first time the function component is invoked), the `color` `state` variable is initialised with `"white"` (line 4). After that, every time the user chooses another color from the select input the event handler `colorChanged()` (line 6) is executed, calling the `setColor` function (line 7). That method is the one that *changes* the `state` which provokes the re-render of the component on the browser. And when `state` (the `color` variable declared on line 4) is changed, here is what happens in React: the entire function-based component is called again (in order to perform the re-render), which end up calling the `useState` hook again (line 4), which in this case, reads the current `state` value (the one that was set as part of the event handler on line 7), instead of initialising it with `"white"`. That value is then assigned to the `colorBoxStyle.backgroundColor` (line 14), which is finally used as the style of the JSX element (line 27) which draws a square.

You can declare as many `state` variables as you need. Suppose now that we want to extend our `ColorSelect` component, printing how many times the user changes the color. See below how we have implemented this, adding another state variable:

```
1 import { useState } from "react";
2
3 export default function ColorSelect() {
4   const [color, setColor] = useState("white");
5   const [changes, setChanges] = useState(0);
6
7   function colorChanged(e) {
8     setChanges((prev) => prev + 1);
9     setColor(e.target.value);
10   }
11
12   const colorBoxStyle = {
13     width: "30px",
14     height: "30px",
15     border: "1px solid rgba(0, 0, 0, 0.05)",
16     backgroundColor: color,
17   };
18
19   return (
20     <>
21       <label for="colors">Choose your favourite color:
          </label>
22       <select name="colors" onChange={colorChanged}>
23         <option>Options...</option>
24         <option value="blue">Blue</option>
25         <option value="red">Red</option>
26         <option value="green">Green</option>
27         <option value="yellow">Yellow</option>
28       </select>
29       <div style={colorBoxStyle}></div>
30       <p>Count: {changes}</p>
31     </>
32   );
33 }
```

On line 5 we have added the declaration of another `state` variable called `changes`. Note that each `state` variable has its own function to update it. Then, when the `onChange` event is triggered, the handler which starts on line 7, will call the update function of both state variables. The first one on line 8 increments the number of changes. Note that the `setChanges` update function is receiving as an argument a function which as parameter (`prev`) receives the current `state` value, and returns the new calculated value. This is mandatory when the new `state` is calculated based on the current one. As we mentioned in section 2.6 when explaining the `state` in class-based components, the mutator function that `useState` gives you might be asynchronous too. So, if you need to set a new `state` value based on the previous one, it must be passed as parameter as we are doing on line 8 above.

Do you want to test your knowledge? As an exercise change the `ColorSelect` component above to fire the event even when the same color is chosen from the select input element. Once you do that, you will note that the `changes` `state` variable gets updated even when the same color is picked. Change that to only increment the `changes` `state` variable each time the current and the chosen color differs.

There are two rules you must follow in order to use hooks without problems. And these rules apply to any hook, not only to the one we are studying in this section.

- Call hooks at the top level of the React Function: Don't call them inside loops or conditions.
- Call hooks from React Functions: Only call hooks from React function-based components, not from plain JavaScript functions.

By following these rules, you ensure that hooks are called in the same order each time a component renders, so that React can, among other things, keep track of the state the component is associated with. This is necessary as it was explained, there is state being kept between function calls. Some notes about how this is implemented in React can be found in the official doc faq.

We will see more complex examples using this hook in the next section.

2.8.2 useEffect

By convention, all hook names start with the *use* prefix and then what best describes what the function does or is about. In this case, the term *effect* refers to *side effects*. We call side effects to any change we do to variables that do not belong to the scope of a function. Functions use props and state to calculate the output, any change to a data outside of that is called side effect. Examples of side effects in React components are Ajax requests, set up timers, changes to the DOM directly, etc. Whenever you need to do any of these in a function-based component, you need to use this hook.

This hook gives you the possibility to attach behaviour to the component lifecycle, in function-based components, in a similar way you have with class-based components. As the official documentation explains, this hook can be seen as the `componentDidMount`, `componentDidUpdate` and the `componentWillUnmount` combined in one special function.

This hook accepts 2 arguments:

- `useEffect(callback[, dependencies])`

  `callback`: the implementation of the side effect.

  `dependencies`: an optional array of `props` and `state` variables.

Let's study our first example below to understand how this hook works:

```
import { useEffect, useState } from "react";

export default function UseEffectExample() {
  const [state, setState] = useState("initial State");

  useEffect(() => {
    console.log("useEffect is called");
  });

  return (
    <div>
      <input
        type="text"
        value={state}
        onChange={(e) => setState(e.target.value)}
      />
    </div>
  );
}
```

On line 6 we call the `useEffect` function, just passing the first argument, the `callback`. In this case, just print on the console once the `callback` is executed. Our function-based component also has an `state` variable, initialised by calling the `useState` hook on line 4. And as part of the JSX that is painted on the browser we have an input text (line 12) where every time their value changes it will set that value to the `state` variable (see the `onChange` event on line 15). This way of calling `useEffect` (without passing any `dependency` as a second argument) is called *by default*. In this case, the `callback` function will be invoked always **after** *every render* (or if you like, after mount and after update). Note that in this case, every time the end user type something on the input text, will change the state variable, triggering the re-render and after that calling to the `useEffect` callback.

If you need that your `callback` effect function gets called **only** on mounting, as a *dependency*, second parameter, you have to pass an empty array, like below. See on line 8:

```
1  import { useEffect, useState } from "react";
2  
3  export default function UseEffectExample() {
4    const [state, setState] = useState("initial State");
5  
6    useEffect(() => {
7      console.log("only invoked on mounting");
8    }, []);
9  
10   return (
11     <div>
12       <input
13         type="text"
14         value={state}
15         onChange={(e) => setState(e.target.value)}
16       />
17     </div>
18   );
19 }
```

Let's consider another more complex example. Inside the effects `callback`, you might require to change the `state`. In general, this is required to display data that is fetched from a remote service. In the component example below, we will display a list of passengers and their number of trips. This information is retrieved from a service using the native `fetch` API.

```
1  import { useEffect, useState } from "react";
2
3  export default function UseEffectExample2() {
4    const [pass, setPass] = useState({ passengers: { data: []
     ↪  }});
5
6    useEffect(() => {
7     fetch("https://api.instantwebtools.net/v1/passenger")
8     .then((response) => response.json())
9     .then((pasData) => {
10      setPass({ passengers: pasData });
11    });
12   });
13
14   return (
15     <div>
16       <table>
17         <thead>
18           <tr>
19             <th>name</th>
20             <th>trips</th>
21           </tr>
22         </thead>
23         <tbody>
24           {pass.passengers.data.map((element) => (
25             <tr key={element._id}>
26               <td>{element.name}</td>
27               <td>{element.trips}</td>
28             </tr>
29           ))}
30         </tbody>
31       </table>
32     </div>
33   );
34 }
```

On line 4 above, we are defining a `state` variable that will hold the passengers list, initialised with an object, with an empty array called `data`. In the first render, the `pass.passengers.data` state array is empty, so only the table header and the structure is displayed on the browser. After that, the `useEffect` is executed, fetching the data from a service (line 7), then transforming the response into *json* on line 8, and then updating the `state` variable with the received data on line 10. The data we get from the service has the following structure:

```
{
   "totalPassengers":7365,
   "totalPages":1,
   "data":[
      {
         "_id":"5f1c59c7fa523c3aa793bea6",
         "name":"nbmbjkhjhk",
         "trips":19,
      },
   ]
}
```

Updating a `state` variable, will produce the function component to be called again to perform the re-render, and after that (after every render), `useEffect` is called again, fetching data and updating the `state` variable and everything starts again. Did you notice what happened? we have produced an infinite loop. How can we do this then? The way we have to fix this is by using the *dependencies* parameter of the effects hook. In this case, just passing an empty array is what we need to fix our infinite loop due to in that case the effect callback is called only once (on mounting).

> You might have noticed that on the component `UseEffectExample2` on line 25 we have added a property called `key` to the `tr` element. That is a special property that must be added when you create a list of elements like in our case above. Without that property in cases like this a warning message is generated. The `key` property helps React to identify which item has changed, added or removed when re-render is executed.

Let's see another example. We will improve our component to fetch data page by page from the service. We will call the same API but now we will pass the page number that we want to retrieve, and we will also add a button to retrieve the next page. Let's study our example below:

```
import { useEffect, useState } from "react";

export default function UseEffectExample2() {
```

```
  const [state, setState] = useState({ passengers: { data: [] }});
  const [page, setPage] = useState(0);

  useEffect(() => {
    fetch(
      "https://api.instantwebtools.net/v1/passenger?page=" + page + "&size=10"
    )
      .then((response) => response.json())
      .then((pasData) => {
        setState({ passengers: pasData });
      });
  }, [page]);

  function handleClick() {
    setPage((currentPage) => currentPage + 1);
  }

  return (
    <div>
      <table>
        <thead>
          <tr>
            <th>name</th>
            <th>trips</th>
          </tr>
        </thead>
        <tbody>
          {state.passengers.data.map((element) => (
            <tr key={element._id}>
              <td>{element.name}</td>
              <td>{element.trips}</td>
            </tr>
          ))}
        </tbody>
      </table>
      <button onClick={handleClick}>Next Page >></button>
    </div>
  );
}
```

To make the paging work, on line 5 we have defined the `page` `state` variable. The `useEffect` callback is the same as before with the exception that now to fetch the data from the API it passes as argument the page number (line 9). An the other difference is that the *dependency* argument of the effect hook (line 15) is: `[page]`. This means that the effects callback will be called **only** if the `page` `state` variable changes. And this variable will change every time we press the "Next Page" button. So, using the *dependency* argument of the effect hook is the way you have to control **when** the effect hook callback gets called.

So far, we have seen how to use the effects hook to be executed only on the mounting phase. We have seen that not using the *dependencies* argument will make the effect callback to be executed every after render. And finally, we have seen that by using the *dependencies* argument we can control when the effects run after render. Now, we are going to see how to deal with the unmount phase. This is normally used for cleanup resources, as we have seen there are some side effects that require cleanup, like timers. To do this, you have to return a function from the effects callback, like shown below:

```
  useEffect(() => {
    //effects here

    return () => {
      //clean resources here
    };
  }, dependencies);
```

This function returned by the callback is executed by React before every new render and on the unmount moment. On mounting, where the render is done for the first time, this function is not invoked. On any subsequent render, before invoking the effects callback, the cleanup function is invoked in order to clean the previous effects execution. And finally, is then invoked on the un-mount phase. To demonstrate how this is used we will re implement the `CountDownLatch` component we shown in section 2.6. The implementation of the countdown behaviour requires to set up a timer and to clear it up. In addition, we will explore another built-in hook called `useRef`. Let's explore how this component is implemented.

```
import React, { useState, useEffect, useRef } from "react";

export default function CountDownLatch(props) {
```

```
4      const [startNumber, setStartNumber] =
        ↪  useState(props.startFrom);
5      const intervalId = useRef(0);
6
7      useEffect(() => {
8        if (startNumber === props.startFrom) {
9          intervalId.current = setInterval(() => {
10           setStartNumber((startNumber) => startNumber - 1);
11         }, 1000);
12       }
13       return () => {
14         if (startNumber === 1) {
15           clearInterval(intervalId.current);
16         }
17       };
18     }, [startNumber, props.startFrom]);
19
20     return <h1>{startNumber}</h1>;
21   }
```

On line 4, we define the `startNumber` state variable to hold the counter which is initialised with the `props.startFrom`. On line 5, we call to the `useRef` hook which returns a mutable object with a `.current` property initialised in 0. That mutable object is assigned to the `intervalId` variable (on line 9), which is required later for the cleanup. The mutable object returned by the `useRef` hook can be seen as instance variables like we have in objects. The initialisation is only done in the first call of the function component and the value is persistent across the subsequent calls. It is just an instance variable in a function-based component. Then, on line 7 we define the `useEffect`. Note that with the condition on line 8 we are setting up the timer only in the first call of the component. The return value of the `setInterval` is assigned to our `intervalId.current` "instance" variable. Then, on line 13 we return a function that will clear the timer only when the counter value is one (when the countdown is finished). Each second, the `setStartNumber` is called decrementing the `startNumber` state value, executing the render. And only when the `startNumber` state value is 1 the `clearInterval` is executed, stopping the timer. Confused? Let's review this step by step. Suppose we call our `CountDownLatch` component starting with 3, like below:

```
import ReactDOM from "react-dom/client";
import React from "react";
```

```
import CountDownLatch from "./CountDownLatch";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<CountDownLatch startFrom={3} />);
```

On Mounting:

1. `startNumber` is initialised with 3.
2. render is called, painting 3 on the browser.
3. timer is set up
4. `intervalId.current` is assigned with the identifier returned from `setInterval`
5. the cleanup function is not called but somehow remembers that `startNumber` == 3.

On the first run of the timer:

1. `startNumber` is decremented, now the value is 2.
2. render is called, painting 2 on the browser.
3. the cleanup function is called from the previous effects but since `startNumber` == 3 `clearInterval()` is not executed. And somehow remembers the new value of `startNumber`.

On the second run of the timer:

1. `startNumber` is decremented, now the value is 1 (line 10).
2. render is called, painting 1 on the browser.
3. The cleanup function is called from the previous effects but since `startNumber` == 2 `clearInterval()` is not executed. And remembers the new value of `startNumber`.

On the third run of the timer:

1. `startNumber` is decremented, now the value is 0.
2. render is called, painting 0 on the browser.
3. the cleanup function is called from the previous effects (`startNumber` == 1). Here the `clearInterval` is executed, stopping the timer.

I hope that gives you some clarity. In the next chapter we will continue using these hooks while learning more React concepts.

Part III

Practical React

Chapter 3

A Simple CRUD Application

In this chapter you will learn how to build a simple CRUD application. The application will support the classic CRUD operations of users data. Along the implementation you will learn several React concepts and deal with common problems you will face when coding in React. We will see how to create and submit forms, display data in tabular format (data grids), open modals and also very specific React concepts like the children prop and conditional rendering.

To code the application we will use Material UI[6]. It provides a rich set of React components, like forms, grids, modals and many more, styled with the material design theme.

The full source code of this application is available at github/crud. We will go in detail explaining every component we have created. However, I recommend you to follow the steps here to install and run the application, in order you can play with it while reading the chapter.

3.1 Material UI

To create this project we have executed the following commands. First creating the project folder:

```
$ npx create-react-app react_simple_crud
```

Then, we install Material UI core, icons and data-grid components:

```
$ cd react_simple_crud
$ npm install @material-ui/core
$ npm install @material-ui/icons
$ npm install @material-ui/data-grid
```

3.2 Identifying Components

In this section we will start describing the functionality of the application and then describe the components we will create to implement that functionality. We will present a set of figures that illustrate what we are going to build. We will refer to these figures as mockups.

The figure 3.1 illustrates the application's home page. Observe that it presents a classic layout with a left panel with menu items (Welcome, User Lists, Add User), a blue top bar and a center panel (the main page). Figure 3.2 shows a users data grid displayed in the center panel. Each data grid row is editable and selectable. If you select a row in the grid, you can delete a user by pressing the "Delete Selected User" button. By double clicking on a cell you will be able to edit their content and commit the changes by pressing the Enter key. In addition, clicking on the "More..." button you will get more details from the user like depicted in the figure 3.3. Finally, we have the figure 3.4 that shows a form for creating new users.

Now, it is time to decompose the functionality explained into components. As shown in the figures 3.5, 3.6, 3.7 and 3.8, we have identified the following components:

1. A Layout (red square).
2. A Left Menu (blue square).
3. A Welcome box (violet square).
4. A Users Grid (green square).
5. A User Detail (orange square).
6. A User Form (yellow square).

Note that each component implements a specific piece of functionality of the application. As explained beautifully in the official documentation Thinking in React, once we identify the components we have to arrange them into a hierarchy. Below, we present the components hierarchy of our simple CRUD application:

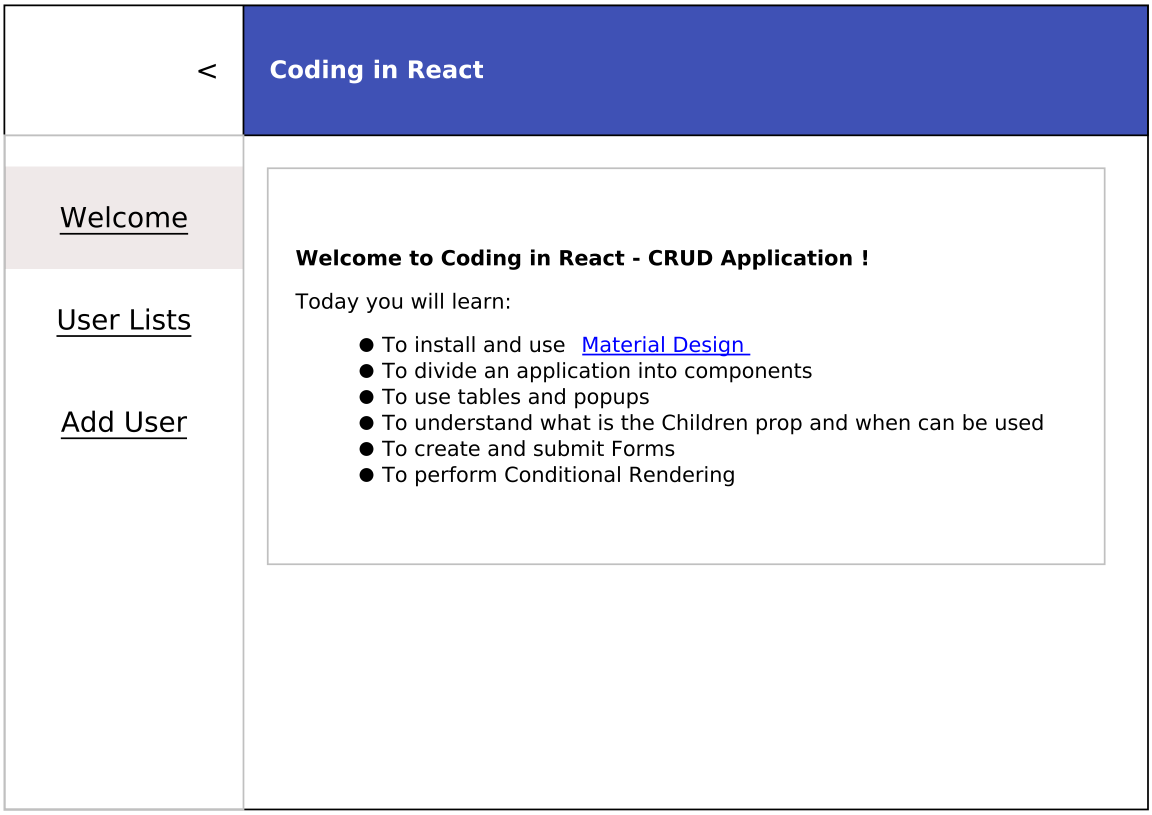
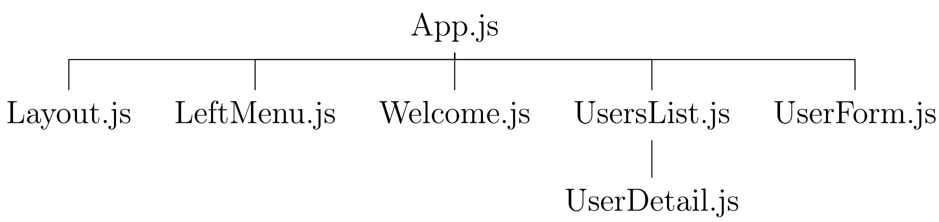


Figure 3.1: Welcome / Home Page



Note that we have the `App.js` root component, which has as children `Layout.js`, `LeftMenu.js`, `Welcome.js`, `UsersList.js` and `UserForm.js`. `UserDetail.js` is a child of `UsersLists.js`. Data in React **flows** from the top level component to the bottom one and is **one-way**. In the next sections, we will be explaining how each component is implemented. During this process we will learn new React concepts and typical problems found in this type of application. We will first start implementing the root component `App.js` and with that we will learn the concept called **children prop**.

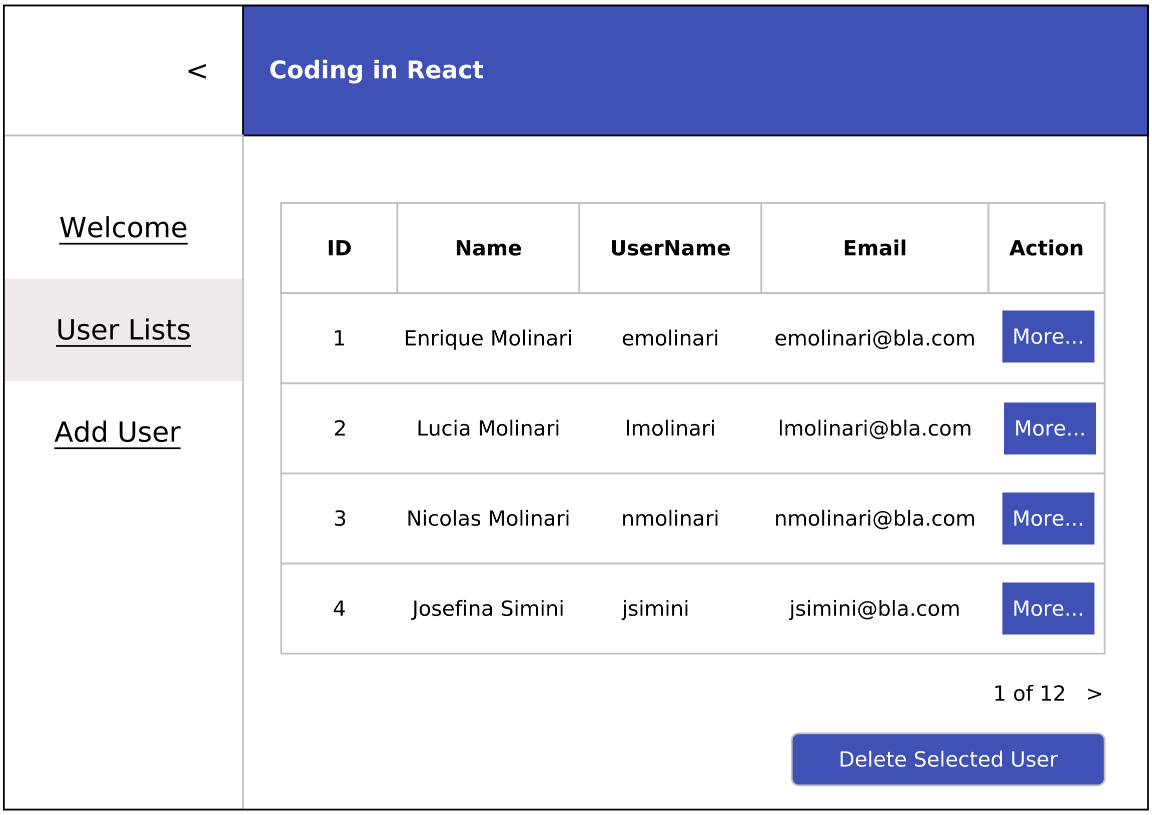


Figure 3.2: Users Navigation

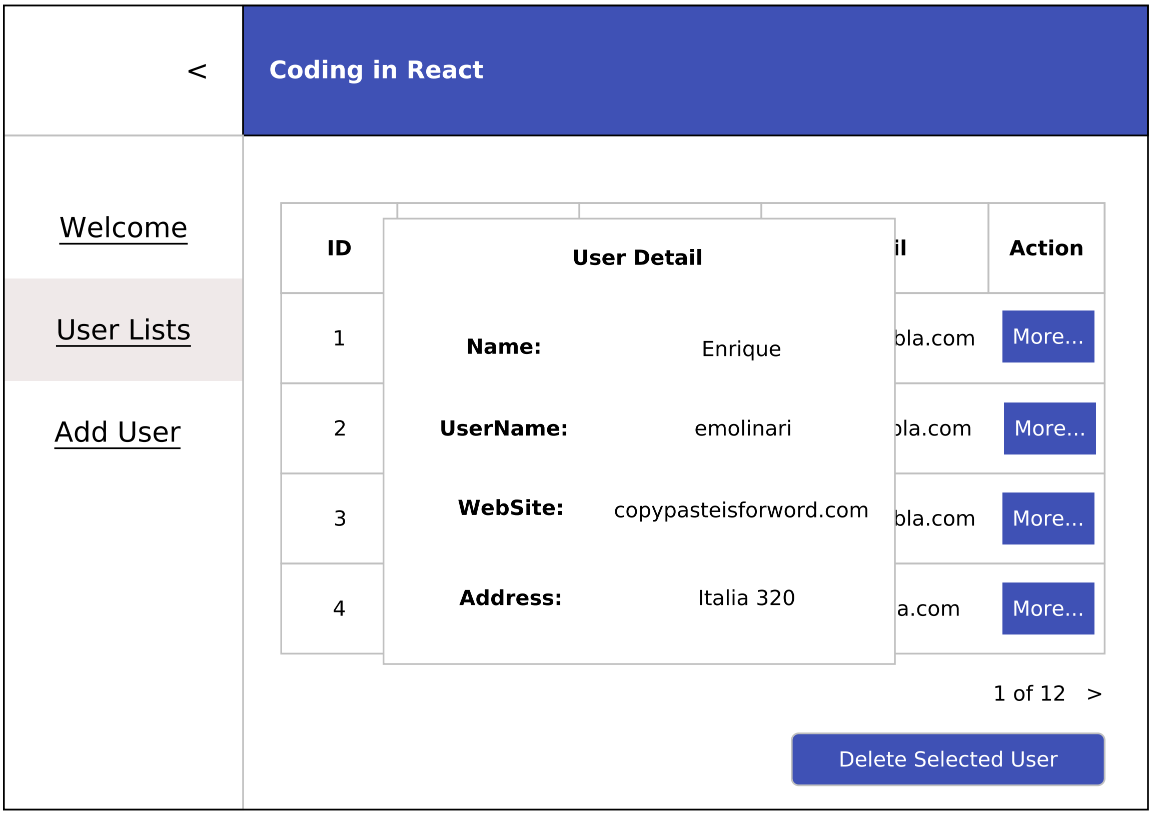


Figure 3.3: User Details

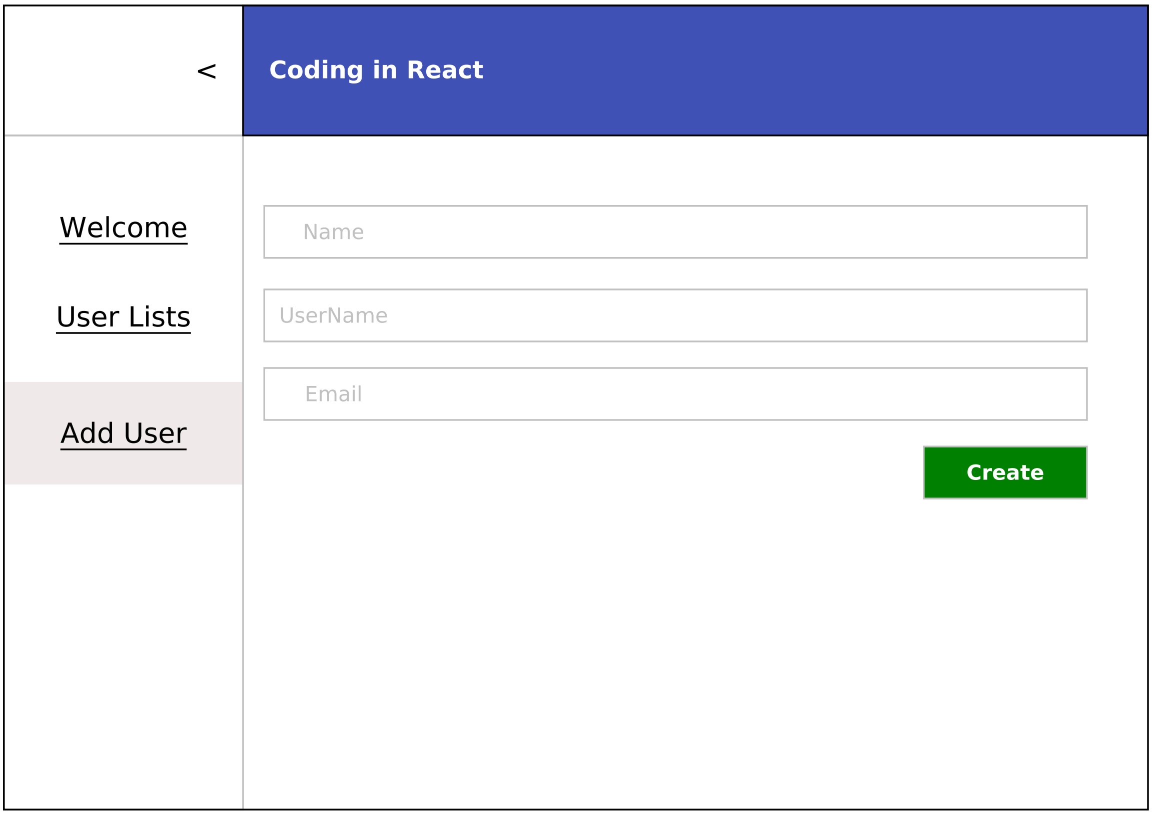


Figure 3.4: Add New User Form

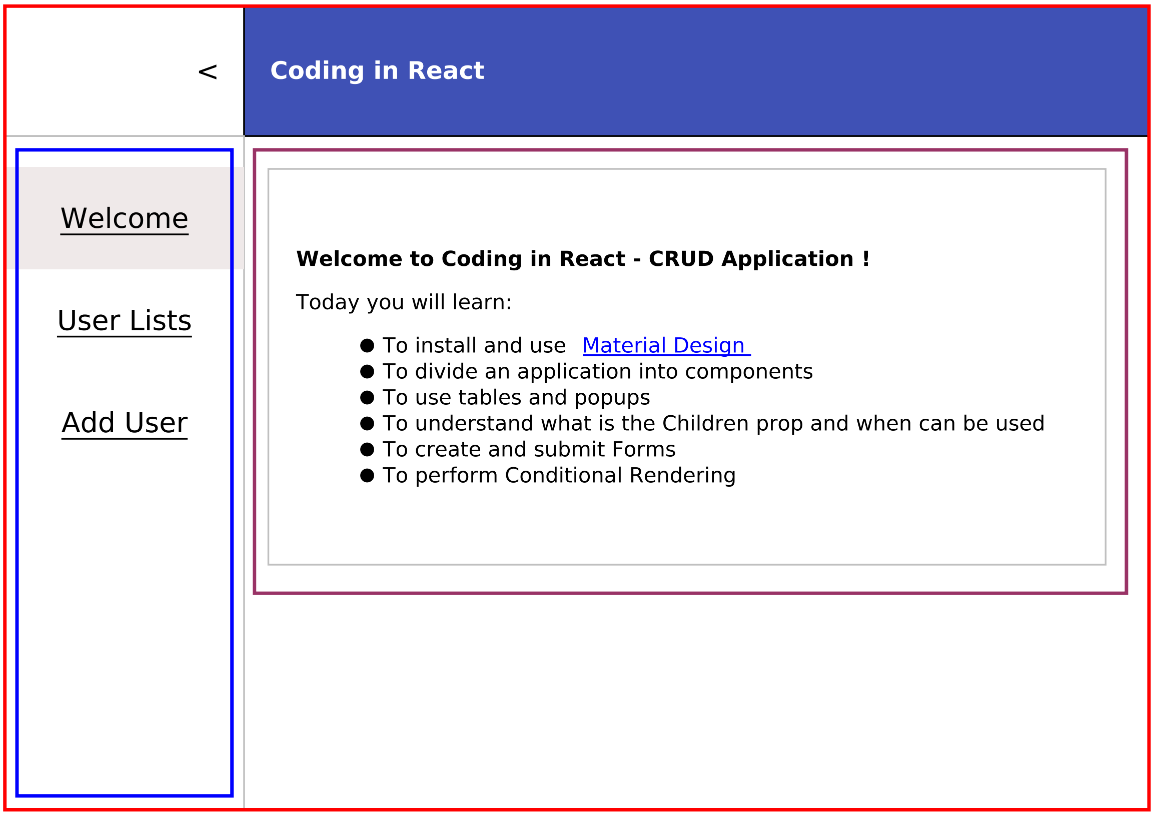


Figure 3.5: Layout (red), Left Menu (blue) and Welcome (violet) component

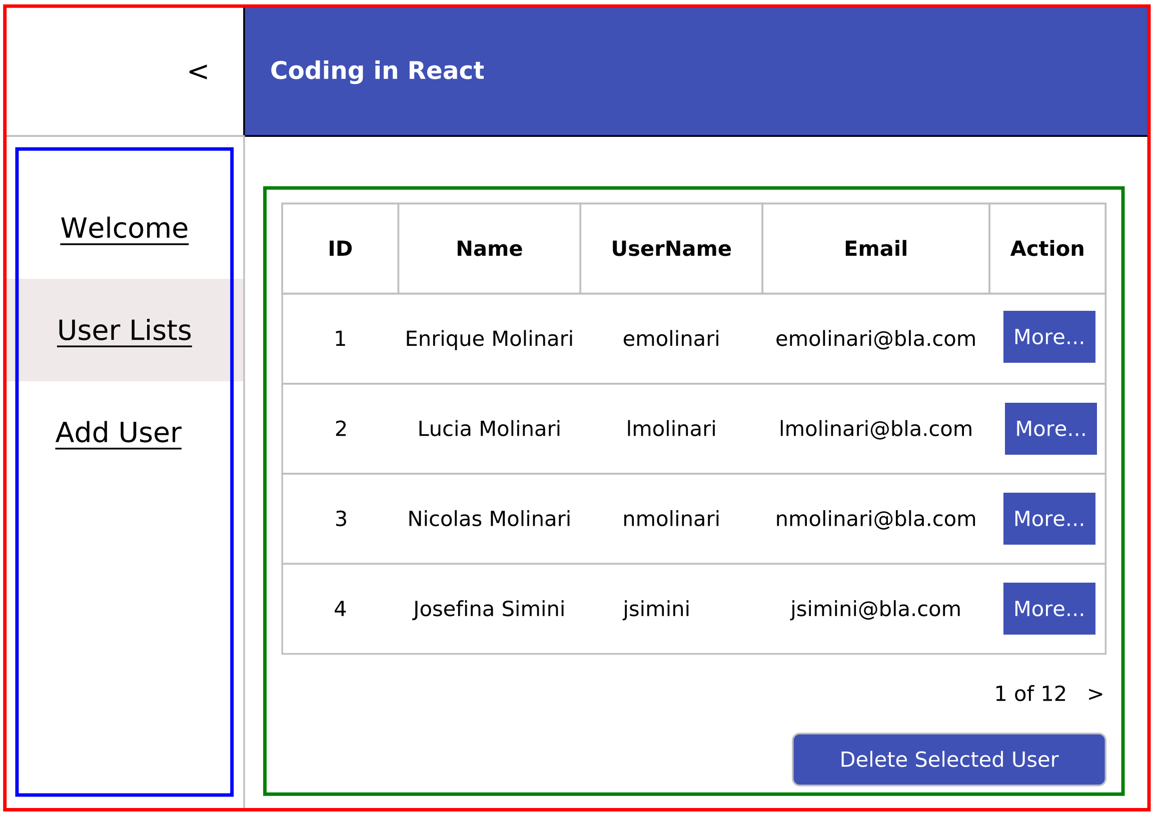


Figure 3.6: User list component (green)

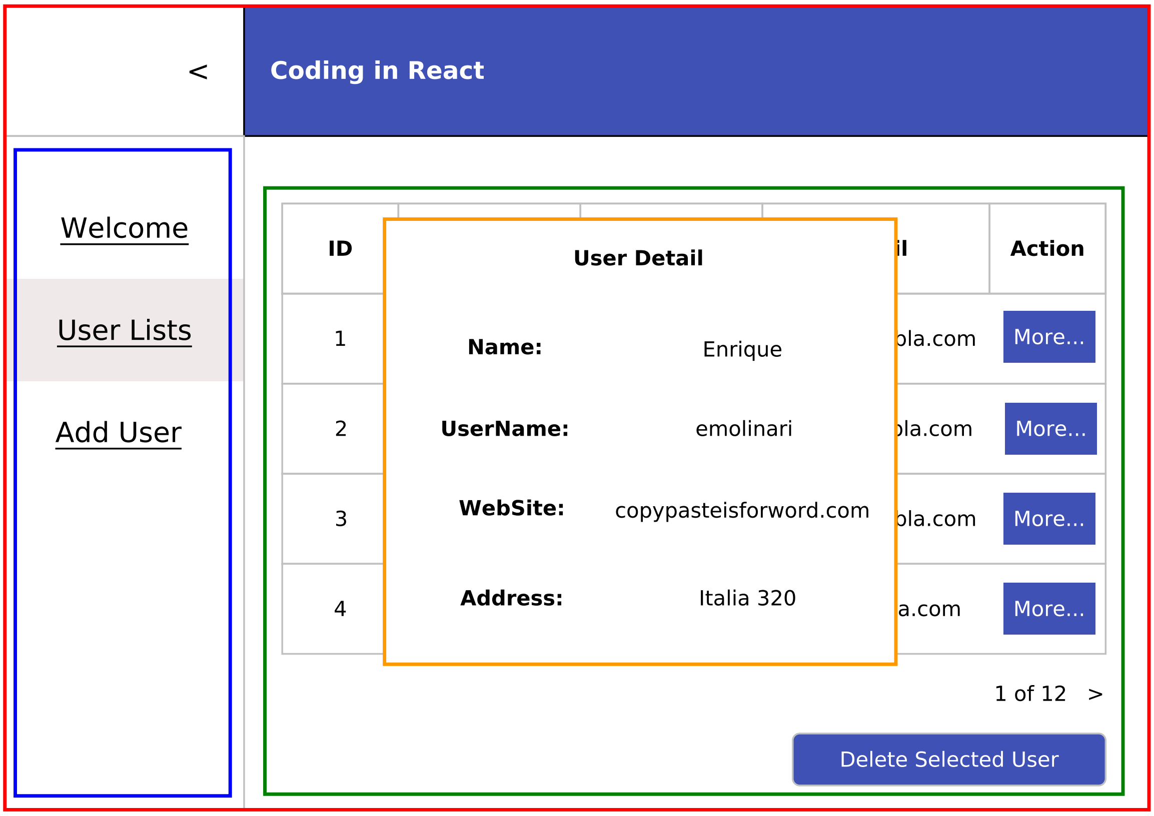


Figure 3.7: User Details component (orange)

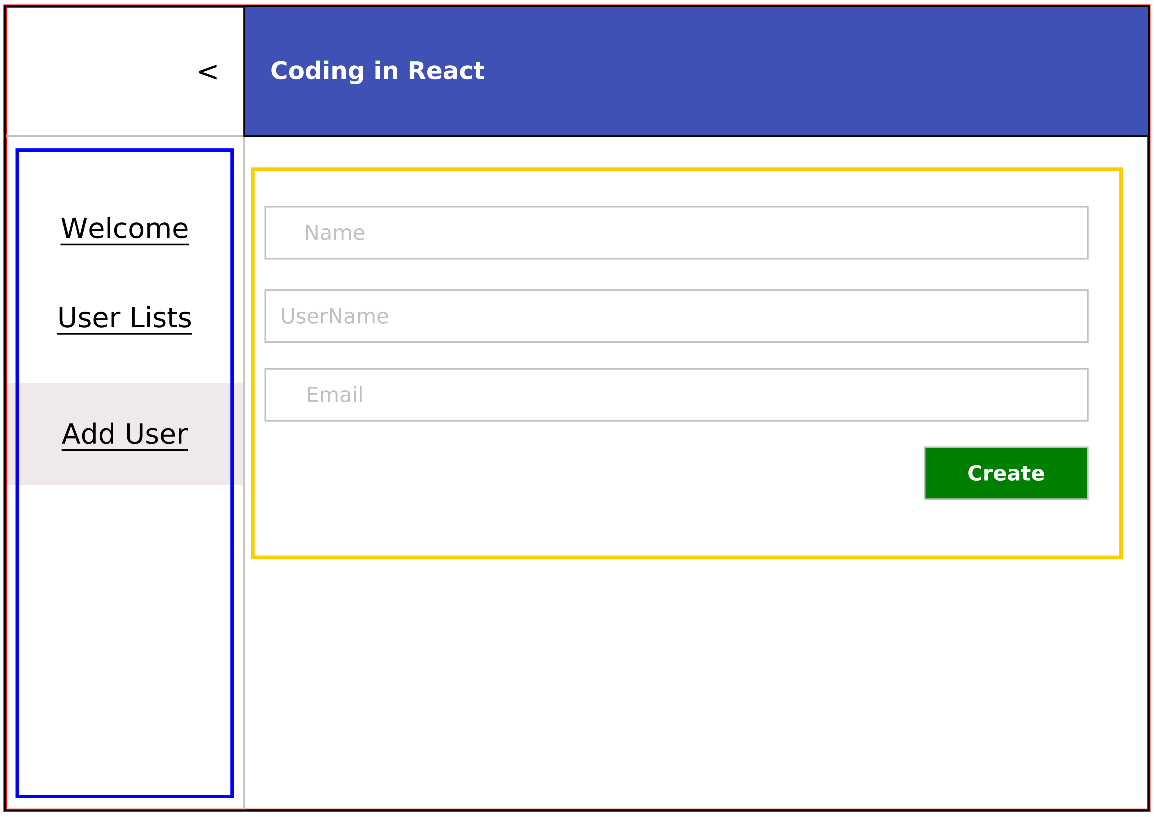


Figure 3.8: User Form Component (yellow)

3.3 Children Prop

To understand what the children prop is, let's see below how we have implemented the `App.js` root component and the `Layout.js` component:

```
import React from "react";
import Layout from "./Layout.js";
import LeftMenu from "./LeftMenu.js";
import Welcome from "./Welcome.js";

export default function App() {
  return (
    <Layout left={<LeftMenu />}>
      <Welcome />
      <UsersList />
      <UsersForm />
    </Layout>
  );
}
```

As can be seen in the `App.js` component above, described in JSX we can see the component hierarchy we mentioned in the previous section (with the exception of the `UserDetail.js` component which we will describe later). And below, we describe the `Layout.js` component. The implementation of this component was taken from the Material UI Dashboard Template example. If you want to see the full implementation of this component, check it on github: Layout.js. The source code presented below was simplified to make the explanation of the children prop concept easier.

```
export default function Layout(props) {
  ...
  return (
    <div>
      <CssBaseline />
      <AppBar position="absolute">
        <Toolbar>
          ...
        </Toolbar>
      </AppBar>
      <Drawer variant="permanent" open={open}>
        <div>
          <IconButton onClick={handleDrawerClose}>
```

```
14              <ChevronLeftIcon />
15            </IconButton>
16          </div>
17          <Divider />
18          {props.left}
19        </Drawer>
20        <main>
21          <Container maxWidth="lg">
22            <Grid item xs={12}>
23              <Paper>{props.children}</Paper>
24            </Grid>
25          </Container>
26        </main>
27      </div>
28    );
29  }
```

The `Layout.js` component consists of the blue top bar (see figure 3.1) painted by the *AppBar* component on line 6 above. Then, the Left panel painted by the *Drawer* on line 11 above, and the *Main* or center panel wrapped by the *main* element on line 20. That will generate the structure of the layout we see on the mockups above.

Note that on the `App.js` component, on line 8 we are passing the `LeftMenu.js` component as a prop called `left` to the `Layout.js` component. This is used in the `Layout.js` on line 18. This means that we are rendering the `LeftMenu.js` component, that paints the menu options (Welcome, User Lists and Add User), just there on the *Drawer*. On the other hand, if you look one more time at the `App.js` component above, you will note that the `Welcome.js`, `UsersList.js` and `UserForm.js` are defined as children of the `Layout.js` component. In the `Layout.js` component source code we can refer to these components using the **special prop** called `children`. This special prop is used on the highlighted line 23 above. This is how you can render these three components in the *main* panel.

3.4 Conditional Rendering

Let's have a look one more time to our `App.js` component below:

```
import React from "react";
import Layout from "./Layout.js";
```

```
import LeftMenu from "./LeftMenu.js";
import Welcome from "./Welcome.js";

export default function App() {
  return (
    <Layout left={<LeftMenu />}>
      <Welcome />
      <UsersList />
      <UserForm />
    </Layout>
  );
}
```

In this way, the `App.js` is rendering the `Layout.js` plus the `LeftMenu.js`, but also, all these three components `Welcome.js`, `UsersList.js` and `UserForm.js`, one after another. And that will render something like what is shown on figure 3.9.

In order to avoid this behaviour we have to use what is known as **Conditional Rendering**. Conditional rendering isn't really different than to apply conditions in Javascript, but in this case to render or not a React component. From the mockups, we know that depending on the menu item clicked on the left panel, is the component that we have to render on the main panel. The items on the menu are the conditions to render or not a specific component.

To implement conditional rendering, we have to change the `App.js` component. Below is how it looks like with conditional rendering.

```
export default function App() {
  const MENU_ITEMS = {
    WELCOME: 0,
    USERSLIST: 1,
    USERFORM: 2,
  };
  const [itemClicked, setItemClicked] =
  ↪  React.useState(MENU_ITEMS.WELCOME);

  function handleClick(item) {
    setItemClicked(item);
  }

  return (
```

```
14      <Layout
15        left={<LeftMenu items={MENU_ITEMS}
          ↪  handleMenu={handleClick}
          ↪  valueItem={itemClicked}/>}>
16        {itemClicked === MENU_ITEMS.WELCOME && <Welcome />}
17        {itemClicked === MENU_ITEMS.USERSLIST && <UsersList
          ↪  />}
18        {itemClicked === MENU_ITEMS.USERFORM && <UsersForm />}
19      </Layout>
20    );
21  }
```

On line 2 above we have declared an object with one constant per menu item (Welcome, User Lists and Add User). Then, on line 7 we have declared the state variable `itemClicked` initialised with `MENU_ITEMS.WELCOME`. And now on the JSX starting on line 14 we render the components depending on the value of the `itemClicked` state variable (lines 16, 17 and 18). As you can see, due to `itemClicked` is initialised with `MENU_ITEMS.WELCOME`, only the `Welcome` component is rendered. We are using the logical && operator to create conditional expressions, it is a syntactic shortcut that I like. However, there are other options that you can check in the official docs about conditional rendering.

Let's study now how the value of the `itemClicked` is changed, triggering the re-render of a different component on the main panel. The `itemClicked` is passed as prop (`valueItem`) to the `LeftMenu.js` component (line 15 above). And in addition, we are passing as a prop (`handleMenu`) a handler function that changes the value of the `itemClicked` (function defined on line 9 above). Let's see below how the `LeftMenu.js` component uses these props to change which component is rendered.

```
export default function LeftMenu(props) {
  function handleListItemClick(item) {
    props.handleMenu(item);
  }

  return (
    <List>
      <ListItem
        selected={props.valueItem === props.items.WELCOME}
        button
```

```
11          onClick={() =>
             ↪  handleListItemClick(props.items.WELCOME)}>
12          <ListItemIcon>
13            <Home />
14          </ListItemIcon>
15          <ListItemText primary="Welcome" />
16        </ListItem>
17        <ListItem
18          selected={props.valueItem === props.items.USERSLIST}
19          button
20          onClick={() =>
             ↪  handleListItemClick(props.items.USERSLIST)}>
21          <ListItemIcon>
22            <PeopleIcon />
23          </ListItemIcon>
24          <ListItemText primary="User Lists" />
25        </ListItem>
26        <ListItem
27          selected={props.valueItem === props.items.USERFORM}
28          button
29          onClick={() =>
             ↪  handleListItemClick(props.items.USERFORM)}>
30          <ListItemIcon>
31            <AddBox />
32          </ListItemIcon>
33          <ListItemText primary="Add User" />
34        </ListItem>
35      </List>
36    );
37  }
```

The `LeftMenu.js` component above uses the List Component from Material UI, to create the menu items you have seen on the mockups above. The `props.valueItem` is used on the *ListItem* component in the *selected* property (lines 9, 18 and 27). The *selected* property is responsible to paint the background grey color on the item that was recently clicked. On line 2, there is a handler function being called when clicking on any of the items from the menu (see the `onClick` event defined on lines 11, 20 and 29). Note that when each item is clicked it calls this handler with the appropriate constant value passed as the `item` parameter. This calls then the `props.handleMenu(item)` which invokes the handled defined on line 9 of the parent component (`App.js`), which finally sets the new value to the `itemClicked` `state` variable. And as we know, when the state changes, the render is executed, producing the painting on the screen of the appropriate component on the main panel.

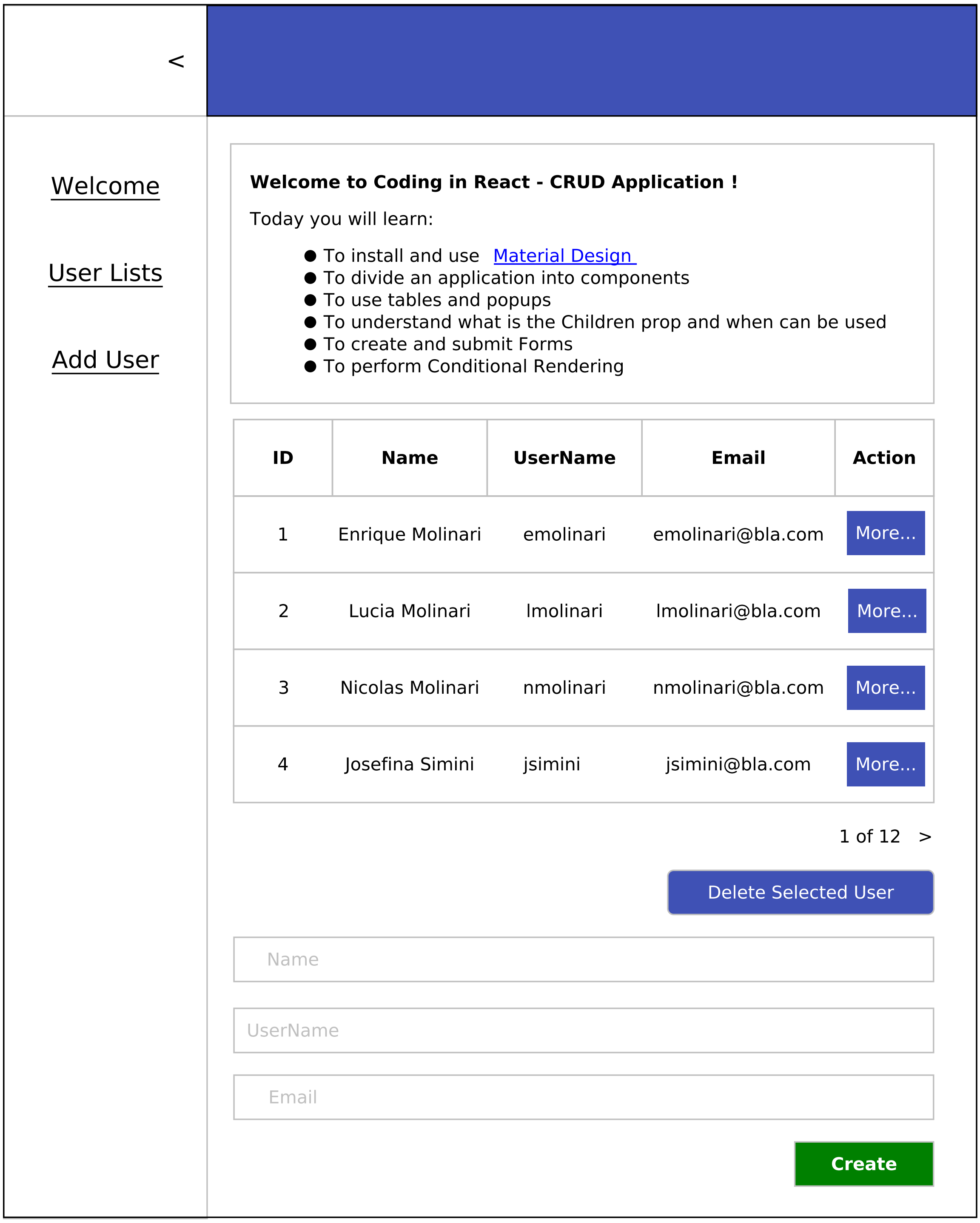


Figure 3.9: Everything Rendered

3.5 Components Communication

The two most common ways of communication between components is from **parent to child** using **props** and from **child to parent** using **callbacks**. These two ways of communication were used in our example from the previous section between the `App.js` and the `LeftMenu.js` components. To study this concept in more detail see the diagram in figure 3.10.

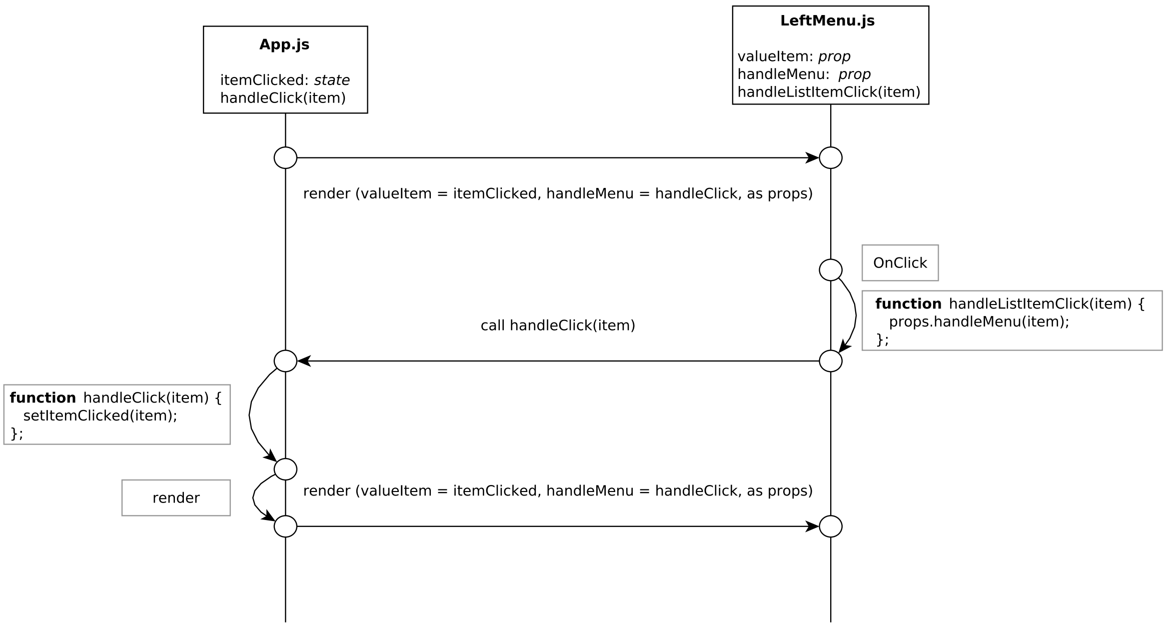


Figure 3.10: Communication between Components

As we can see on figure 3.10, the `App.js` component defines a `state` variable called `itemClicked` and a handler function called `handleClick`, which receives the `item` clicked to change the `itemClicked` state variable with that value. When we render the `App.js` component, that triggers the render of the `LeftMenu.js` passing the `itemClicked`, initialised with `MENU_ITEMS.WELCOME` as `valueItem` prop and the `handleClick` handler function as `handleMenu` prop. That is our **parent to child** communication.

Then, when the user clicks on any element of the menu that belongs to the `LeftMenu.js` component, the `handleListItemClick` function is called, calling the `prop.handleMenu` callback. That invokes the `handleClick` handler function from the `App.js` parent component. And there we have the **child to parent** communication. Then, this change the `itemClicked` state variable, triggering the re-render of the App.js which triggers the re-render of the LeftMenu.js and all the other child components in the hierarchy. Observe that in this second re-render of the LeftMenu.js component and in any other rendering after that, the props are passed again. In this case, the itemClicked will have a new value, the one that was clicked by the user. For the LeftMenu.js this will make the background color of the item clicked, be changed from white to grey (selected color). It is important to note that even when the click event is triggered and handled on the LeftMenu.js component, the grey background of the item clicked is set only after the App.js component gets re-rendered (because that triggers the re-render of the LeftMenu.js component passing this new value of the itemClicked by prop).

3.6 Custom Environment Variables

The create-react-app tool gives you a built-in environment variable called NODE_ENV which informs you about the environment where your app is running. Its value will be 'development', when you run npm start. It will be 'test' when you run npm test and it will be 'production' when you create the bundle for prod using: npm run build. This value is available from Javascript by using process.env.NODE_ENV.

We are also able to create custom environment variables. For our example projects, created in this and next chapter, we need a variable to store the URL of the APIs that our components will consume. To do that we have created a file called .env in the root folder of the application (check it on github). The name of the custom environment variable must start with REACT_APP_. In our case the variable is named REACT_APP_API_URL.

As a design rule we have defined that the value of this variable must be referenced only from our root component, which in our case is App.js. And its value is passed by prop to any component that requires it.

```
export default function App() {
  const MENU_ITEMS = {
    WELCOME: 0,
    USERSLIST: 1,
    USERFORM: 2,
  };
  const [itemClicked, setItemClicked] =
   ↪ React.useState(MENU_ITEMS.WELCOME);
```

```
8    const apiUrl = process.env.REACT_APP_API_URL;
9
10   function handleClick(item) {
11     setItemClicked(item);
12   }
13
14   return (
15     <Layout
16       left={
17         <LeftMenu
18           items={MENU_ITEMS}
19           handleMenu={handleClick}
20           valueItem={itemClicked}
21         />
22       }
23     >
24       {itemClicked === MENU_ITEMS.WELCOME && <Welcome />}
25       {itemClicked === MENU_ITEMS.USERSLIST && <UsersList apiUrl={apiUrl} />}
26       {itemClicked === MENU_ITEMS.USERFORM && <UsersForm apiUrl={apiUrl} />}
27     </Layout>
28   );
29 }
```

Note that the environment variable is read on line 8 and then is passed as prop to child components on lines 25 and 26.

It is not recommended to store in source code any secret value. In the official documentation: Environment Variables, you will find other ways of creating environment variables.

3.7 Data Grids

In this section we will study how to display data in a grid-like format. We will use the Material UI Data Grid component. Our `UserList.js` component is the one that wraps the Material UI Data Grid. The Material Data Grid is a very powerful component allowing, among other things, to do selection, updates, filtering, ordering, etc, only using this component. In the Data Grid API docs you will find the full list of props that this component accepts. In this chapter, we will explain the ones that we have used to implement the application.

Let's start looking at an initial version of the `UserList.js` component. Below, is the version of the component that paints the grid on the browser, and allows you to do paging, like is shown on figure 3.11.

```
export default function UsersList(props) {
  const [users, setUsers] = useState({ result: { data: [] } });
  const [page, setPage] = useState(0);
  const [pageSize, setPageSize] = useState(5);

  useEffect(() => {
    fetchUsers();
  }, [page, pageSize]);

  async function fetchUsers() {
    let response = await fetch(
      props.apiUrl +
        "?_page=" +
        //first page is 1 for the json server API
        //Material DataGrid first page is 0
        (page + 1) +
        "&_limit=" +
        pageSize
    );
    let json = await response.json();
    response = {
      total: response.headers.get("x-total-Count"),
      data: json,
    };
    setUsers({ result: { total: response.total, data: response.data } });
  }

  const columns = [
    { field: "id", headerName: "ID", width: 60 },
    { field: "name", headerName: "Name", width: 180, editable: true },
```

```
31      { field: "username", headerName: "UserName", width: 200, editable: true },
32      { field: "email", headerName: "Email", width: 250, editable: true },
33    ];
34
35    return (
36      <>
37        <div style={{ height: 420, width: "100%" }}>
38          <DataGrid
39            rows={users.result.data}
40            columns={columns}
41            pageSize={pageSize}
42            paginationMode="server"
43            page={page}
44            onPageChange={(params) => {
45              setPage(params.page);
46            }}
47            onPageSizeChange={(params) => {
48              setPageSize(params.pageSize);
49            }}
50            rowsPerPageOptions={[3, 5]}
51            rowCount={parseInt(users.result.total)}
52          />
53        </div>
54      </>
55    );
56  }
```

We start the `UserList.js` component defining some state variables on line 2. The `users` state variable contains the list of users to be displayed on the grid. `page` is where we store the current page that is displayed on the grid. And finally the `pageSize` is where we store the current number of elements (rows) that each page contains.

After that, we have on line 6 the `useEffect` hook which fetches from the server (calling the `async fetchUsers()` function), the list of users to then update the `users` state variable (triggering the re-render). Note that the effects callback will be invoked when either the `page` or `pageSize` change (dependencies parameter on line 8). If the end user using the grid navigate to the next or previous page or change the size of the number of rows per

Figure 3.11: UserLists with paging

page, the callback is called and everything re-painted.

Then, we will jump directly to the JSX from the return statement of the component. The `DataGrid` component on line 38 is wrapped in a `div` element to define its height and width. Then on line 39 and 40 we have the props to feed the grid with the *rows* data and the definition of the *columns*. The *rows* is the JSON fetched from a server API, as explained before, and accessible from `users.result.data`. Its structure is shown below:

```
[
 { "id": "1", "name": "Enrique", "username": "emolinari", "email": "..."},
 { "id": "2", "name": "Josefina", "username": "jsimini", "email": "..."},
 { "id": "3", "name": "Lucía", "username": "lmolinari", "email": "..."}
```

```
]
```

And we have the definition of the *columns* on line 28 of the `UserList.js` component above. The columns are defined as an array of objects respecting the GridColDef interface. Note that with the property `field` we define in which column each property of a row will be painted. The value specified by each `field` property in the *columns* array and the property names specified in the *rows* (the JSON obtained from the API) must match to have this working. For instance, the `field: "id"` matches with the `"id"` property of the response JSON. Then, additionally on the *columns* we have the `headerName`, the `width` and the `editable` property, to specify the label to name each column, their width and if it is editable or not.

After the *rows* and *columns* props on the `DataGrid` component (line 38), we have the following (in order):

- pageSize: The number of rows for a page. Note that we set this prop with our `pageSize` state variable defined on line 4.
- paginationMode: If the paging is calculated on the *client* or on the *server*.
- page: The current displayed page. Note that we set this prop with our `page` state variable defined on line 3.
- onPageChange: The callback function to be invoked when the user navigates to the next or previous page. It receives the page to go, and that is used to change our `page` state variable, triggering the render and the `useEffect` hook.
- onPageSizeChange: The callback function to be invoked when the user changes the number of rows per page. Here we are changing the `pageSize` state variable triggering the render and the `useEffect` hook.
- rowsPerPageOptions: The options for the user to choose how many rows per page wants to have. In our case, we are displaying three or five rows per page.
- rowCount: The total number of rows.

To implement the delete operation, we will ask the user to select a row on the grid and press the "Delete Selected User" button. Note that on figure 3.12 we show a selected row of the grid and the blue "Delete Selected User" button. In addition, when the button is pressed we will show a spinner while the operation is in progress. When finished, we will display a success message. The JSON Placeholder API that we are using fakes the insert, update and delete operations. It always returns success that is why we are not showing how to handle the failures. Below we show the `UserLists.js` component just with the code to deal with the delete operation. The full source code of this component can be found here.

```
const StyledBox = styled(Box)({
  height: 40,
  display: "flex",
  justifyContent: "flex-end",
  marginTop: 10,
});

export default function UsersList(props) {
  const [loading, setLoading] = useState(false);
  const [showAlert, setShowAlert] = useState(false);
  const [alertMsg, setAlertMsg] = useState("");

  let userIdSelected = 0;

  async function handleDelete() {
    if (userIdSelected === 0) {
      setAlertMsg("Please, select a row of the grid first");
      setShowAlert(true);
      return;
    }

    setLoading(true);

    await fetch(props.apiUrl + "/" + userIdSelected, {
      method: "DELETE",
    });

    setLoading(false);
    setAlertMsg("User Deleted Successfully");
    setShowAlert(true);
    //refresh the grid data after delete
    fetchUsers();
  }

```

```
35    function handleCloseAlert() {
36      setShowAlert(false);
37    }
38
39    return (
40      <>
41        <div style={{ height: 420, width: "100%" }}>
42          <DataGrid
43            ...
44            onRowSelected={(e) => (userIdSelected = e.data.id)}
45          />
46        </div>
47        <div>
48          <StyledBox component="div">
49            <Button variant="contained" color="primary" onClick={handleDelete}>
50              {loading && <CircularProgress color="inherit" size={24} />}
51              {!loading && "Delete Selected User"}
52            </Button>
53          </StyledBox>
54        </div>
55        <Snackbar
56          anchorOrigin={{
57            vertical: "top",
58            horizontal: "center",
59          }}
60          open={showAlert}
61          autoHideDuration={3000}
62          onClose={handleCloseAlert}
63        >
64          <Alert severity="success">{alertMsg}</Alert>
65        </Snackbar>
66      </>
67    );
68  }
```

We have added some more state variables. On line 9 we have the `loading` to handle the spinner functionality while the delete operation is in progress. Then we have `showAlert` and `alertMsg`, both used to display a success message to the user after the delete operation completes.

Then on the JSX, on the `DataGrid` component (line 42), we have used the prop `onRowSelected` which triggers the callback when a row of the grid is selected. Note that in this case the callback initialises the variable `userIdSelected` (declared on line 13) with the `id` value of the selected row. This `id` (the unique identifier of the user), as we will see next, is sent to the server when triggering the delete operation.

On line 49 we have the Button component to be used to trigger the delete operation. Its label's value depends on the `loading` state variable. If `loading` is `true` we will see the spinner, using the CircularProgress component, if not the text "Delete Selected User". The button is wrapped in a Box component (a div in this case) styled (using the style function) to be placed on the bottom right corner (the `StyledBox` is defined on line 1).

Next to that, we have the Snackbar component (line 55) used to inform the user about the success or failure of an operation. In this case, we will use it to inform the user about the delete and edit operations. Note that by default the Snackbar is closed (not visible) and this is handled by their `open` prop (line 60). This prop uses the value of the `showAlert` state variable which is initialised in `false`. The Snackbar gets closed automatically after three seconds, by using the prop `autoHideDuration` (line 61) and also offers a manual close which when clicked will call to the function passed on the `onClose` prop. We are passing the `handleCloseAlert` function (line 62). The `handleCloseAlert` function is defined on line 35 and just changes to `false` the `showAlert` state variable.

When the user presses the "Delete Selected User" button the `handleDelete` function on line 15 is called. After a validation that verifies that there is a row selected (line 16), it first sets the `loading` state variable to `true` and then performs the fetch request (line 24) to delete the selected user. After that, change to false the `loading` state variable (which hides the spinner), then, on lines 29 and 30, sets the `alertMsg` and `showAlert` state variable to make the Snackbar visible to inform the user about the success of the operation. Finally, to refresh the grid, it performs a call to the `fetchUsers` functions (line 32).

Let's move now to show how the edit operation is implemented. If you specify in the *columns* definition the property editable in true, the DataGrid component when the user double click on a cell of that column, will place the cell value inside an input text, allowing the user to modify that value. Let see next the `UserLists.js` component with the elements necessary to perform the edition of a cell.

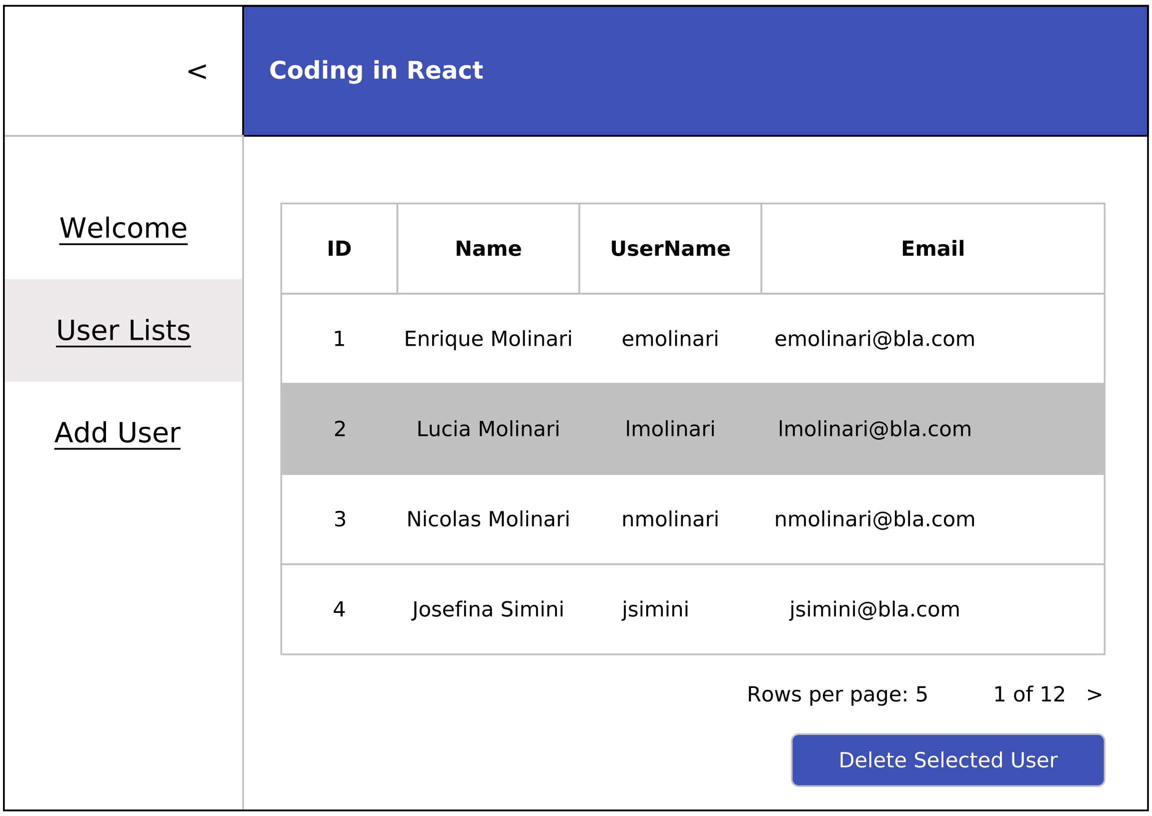


Figure 3.12: UserLists: Delete User

```
export default function UsersList(props) {
  const [showAlert, setShowAlert] = useState(false);
  const [alertMsg, setAlertMsg] = useState("");

  const columns = [
    { field: "id", headerName: "ID", width: 60 },
    { field: "name", headerName: "Name", width: 180,
    ↪  editable: true },
    { field: "username", headerName: "UserName", width: 200,
    ↪  editable: true },
    { field: "email", headerName: "Email", width: 250,
    ↪  editable: true },
```

```
  ];

  function handleCloseAlert() {
    setShowAlert(false);
  }

  async function handleEditing(params) {
    await fetch(props.apiUrl + "/" + params.id, {
      method: "PUT",
      body: JSON.stringify({
        id: params.id,
        [params.field]: params.props.value,
      }),
      headers: {
        "Content-type": "application/json; charset=UTF-8",
      },
    });
    setAlertMsg("User Updated Successfully");
    setShowAlert(true);
  }

  return (
    <>
      <div style={{ height: 420, width: "100%" }}>
        <DataGrid
          ...
          onEditCellChangeCommitted={(params) => handleEditing(params)}
        />
      </div>
      <Snackbar
        anchorOrigin={{
          vertical: "top",
          horizontal: "center",
        }}
        open={showAlert}
        autoHideDuration={3000}
        onClose={handleCloseAlert}
      >
        <Alert severity="success">{alertMsg}</Alert>
      </Snackbar>
```

```
50      </>
51    );
52  }
```

As you can see above, we use the **showAlert** and **alertMsg** state variables to use them in the Snackbar component in the same way as we explained for the delete operation. After that, on the columns definition starting on line 5 you can see the editable property which allows, by double click on a cell, to edit it. On line 36 in the DataGrid component, we use the **onEditCellChangeCommitted** prop to pass a callback that is called when you submit the edition. That submission might be done with the enter or tab keys. When that occurs, the **handleEditing** function is called passing as argument the values of the row where the cell is edited (including the new value). The function **handleEditing** starting at line 16, use the fetch function to consume an API. Note the **params** argument that receives which is an object with the fields:values of the row where the cell being edited belongs. That argument is used on line 17 to create the PUT URL to that specific user Id and on line 19 to build the body of the request.

Finally, we will show how we have implemented the action "More..." to display a dialog box with more information about a user. In the figure 3.2 you can see the action column containing the "More..." button. And on figure 3.3 you will see the dialog box open after clicking the "More..." button. Below you will see the source code from the component `UsersList.js` dedicated to illustrate how the dialog box works. Full source code of this component can be seen here.

```
export default function UsersList(props) {
  const [userId, setUserId] = useState(0);
  const [showDetail, setShowDetail] = useState(false);

  const columns = [
    { field: "id", headerName: "ID", width: 60 },
    { field: "name", headerName: "Name", width: 180,
      editable: true },
    { field: "username", headerName: "UserName", width: 200,
      editable: true },
    { field: "email", headerName: "Email", width: 250,
      editable: true },
    {
      field: "action",
```

```
        headerName: "Action",
        width: 250,
        renderCell: (params) => (
          <Button
            variant="contained"
            color="primary"
            size="small"
            style={{ marginLeft: 16 }}
            onClick={() => openDetails(params.row.id)}
          >
            More...
          </Button>
        ),
      },
    ];

    function openDetails(rowId) {
      setUserId(rowId);
      setShowDetail(true);
    }

    function closeDetails() {
      setShowDetail(false);
    }

    return (
      <>
        <div style={{ height: 420, width: "100%" }}>
          <DataGrid
           ...
          />
        </div>
        <div>
          <StyledBox component="div">
            ...
          </StyledBox>
        </div>
        <UserDetails
          apiUrl={props.apiUrl}
          userId={userId}
          show={showDetail}
```

```
53          handleClose={closeDetails}
54        />
55        <Snackbar>
56          ...
57        </Snackbar>
58      </>
59    );
60  }
```

On the component above, the first thing to note is how we paint the "More..." button on each row of the grid. To do this, starting on line 14 we have used the *renderCell* property of the column definition interface called GridColDef. The callback passed to the *renderCell* property receives a parameter (`params`) of the interface GridCellParams. Additionally, the button, on line 20, defines an `onClick` event which calls the `openDetails` function passing as an argument the identifier of the user. As we explained and is illustrated on figure 3.7 the dialog box with the details was written creating the component `UserDetails.js`. On line 49 we render this component. We have a communication from the parent (`UserLists.js`) to child (`UserDetails.js`), by passing via prop the identifier of the user (`userId` prop on line 51), in order that the child component can request additional information from that user. It also passes the `showDetail` boolean value (a state variable defined on line 3) that is used to show or hide the dialog box. Note that additionally we are passing the prop `handleClose` (line 53) with the handler function `closeDetails`. As we see later, that represents our communication from child to parent.

3.8 Dialog Box

To show more information about each user from the grid presented in the previous section, we have chosen to use a dialog box. The dialog box is created by implementing the component `UserDetails.js`, as the figure 3.7 illustrates. From the previous source code listing we have seen that the `UserDetails.js` component is rendered every time the `UsersList.js` is rendered. On line 49, we can see that there is no condition. However, we pass the `show` prop (line 52), which by default is `false`, to the component, which is used to perform the conditional rendering inside that component.

Below, we have the source code of the `UserDetails.js` component. It receives as props, the `userId`, the `show` boolean value and the `handleClose` function.

```
export default function UserDetails(props) {
  const [userData, setUserData] = useState({ result: {} });
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    if (props.userId <= 0) return;

    const fetchUser = async () => {
      let response = await fetch(props.apiUrl + "/" + props.userId);
      response = await response.json();
      setUserData({ result: { response } });
      setLoading(false);
    };
    fetchUser();

    return setLoading(true);
  }, [props.userId]);

  return (
    <Dialog open={props.show}>
      <DialogTitle id="alert-dialog-title">User Details</DialogTitle>
      <DialogContent>
        <DialogContentText id="alert-dialog-description">
          {loading ? (
            <CircularProgress />
          ) : (
            <table>
              <tbody>
                <tr>
                  <td>Name: </td>
                  <td>{userData.result.response.name}</td>
                </tr>
                <tr>
                  <td>User Name: </td>
                  <td>{userData.result.response.username}</td>
                </tr>
                <tr>
                  <td>Website: </td>
```

```
39                    <td>{userData.result.response.website}</td>
40                  </tr>
41                  <tr>
42                    <td>Address: </td>
43                    <td>
44                      {userData.result.response.address.street +
45                        " - " +
46                        userData.result.response.address.city}
47                    </td>
48                  </tr>
49                </tbody>
50              </table>
51            )}
52          </DialogContentText>
53        </DialogContent>
54        <DialogActions>
55          <Button onClick={props.handleClose} color="primary" autoFocus>
56            Close
57          </Button>
58        </DialogActions>
59      </Dialog>
60    );
61  }
```

On line 2, we have defined the `userData` state variable to store user details fetched from an API. And on line 3, we have defined the boolean `loading` state variable for our spinner.

On line 5, we have the `useEffect` hook defined. As can be seen we fetch the user details from the API when we have a `userId` value greater than 0. Every time we receive as a prop a different `userId`, this hook will be executed. Note that we have defined a clean up function, as explained in section 2.8.2. This function is executed before, except for the first time, the execution of the `useEffect`. Without this clean up, when the user opens the dialog box, as the `useEffect` is executed after rendering and there is a delay to fetch the data from the remote API, the user will notice that the previous user data is shown and suddenly change with the fresh data. Using this clean up, the spinner is shown before painting the dialog box with the fresh data.

On line 20 you can see that we use the Dialog component from Material UI. The prop `open` is the one used to display or not the dialog box. Finally, on line 55 we have the close button that defines the `onClick` event, which receives the handler function `props.handleClose`, that belongs to the parent. That is our child to parent communication, as we previously explained. When this button is pressed the `closeDetails` function from the parent is executed which sets the `showDetail` state variable to false. Hiding the dialog box.

3.9 Forms

There are two ways to implement forms in React, the controlled way or the uncontrolled way. In the controlled way, the React component controls the input's value of the form by using the **state**. While in the uncontrolled way the input's value is handled by the DOM, like in plain HTML. To implement the `UserForm.js` component, we have used the controlled way which is the one that the official React docs recommend.

As shown by the figure 3.4, the form that we have to build has three input text. For the input text we use the TextField component from Material UI. It makes the form validation easier, but the way of managing the happy path is the same as using the plain HTML input element.

We will start this section by showing what it means in code to implement a controlled form in React. Then, we will continue about how to do the submission and finally how we can do validation. Below we present a version of the `UserForm.js` component to study how to implement a controlled form in React. The full source code of this component can be found here.

```
export default function UsersForm(props) {
  const [inputsValue, setInputsValue] = useState({
    name: "",
    username: "",
    email: "",
  });

  function handleSubmit(e) {
    e.preventDefault();

```

```
      console.log(e);
    }

    function handleChange(e) {
      const name = e.target.name;
      const value = e.target.value;
      setInputsValue((inputsValue) => {
        return { ...inputsValue, [name]: value };
      });
    }

    return (
      <>
        <form noValidate autoComplete="off" onSubmit={handleSubmit}>
          <div className="form">
            <TextField
              id="name"
              name="name"
              label="Name"
              fullWidth={true}
              value={inputsValue.name}
              onChange={handleChange}
            />
          </div>
          <div className="form">
            <TextField
              id="username"
              name="username"
              label="User Name"
              fullWidth={true}
              value={inputsValue.username}
              onChange={handleChange}
            />
          </div>
          <div className="form">
            <TextField
              id="email"
              name="email"
              label="EMail"
              fullWidth={true}
```

```
51              value={inputsValue.email}
52              onChange={handleChange}
53            />
54          </div>
55          <div>
56            <Button type="submit" variant="contained"
               ↪  color="primary">
57              Submit
58            </Button>
59          </div>
60        </form>
61      </>
62    );
63  }
```

On the JSX above, starting at line 22, we have the form definition with the three input text using the TextField component. After that, on line 56, the submit Button. As mentioned, to implement the form in a controlled way, we have to define a `state` variable for each of the input text. As a general rule, for each element of a form that can hold a value, a state variable must be defined and that value must be kept in sync. The component, at the point of requiring to read the value of any element of the form, will get it from the corresponding state variable. The component's state is the source of trust. Having said that, on line 2, we have defined the `inputsValue` state variable, which is an object with three properties, one for each of the input text.

Now, we have to keep the input value and their corresponding state variable in sync. We do that by adding to each of the form elements the `onChange` event. On lines 32, 42 and 52, you will see the event with the `handleChange` handler. On line 14 you can see the handler function defined. Every event handler in React will receive as a parameter an instance of SyntheticEvent. It is a wrapper around the browser's native event. Using their `e.target` property we get access to the DOM input element, and from there we can retrieve their *name* and their *value*, like is shown above on lines 15 and 16. Then, on line 17, we update the state variable. So, any change on any of the input elements of the form gets copied to their corresponding state variable. This is how we can keep in sync the value of the form inputs with the component's state. Note that the function passed as argument to the `setInputsValue` returns an object literal which contains the current properties and values of the `inputsValue` object (using spread syntax, see section 1.4), plus the new property/value that has just changed (using computed property names like described in section 1.4).

Now that we know how a form in a controlled way is implemented, you might have guessed how the submission of the form is done. Below we show the complete source code of the `UserForm.js` component:

```
export default function UsersForm(props) {
  const [inputsValue, setInputsValue] = useState({
    name: "",
    username: "",
    email: "",
  });
  const [loading, setLoading] = useState(false);
  const [errorInputs, setErrorInputs] = useState({});
  const [showSuccess, setShowSuccess] = useState(false);

  function handleSubmit(e) {
    e.preventDefault();
    setLoading(true);
    setErrorInputs({});
    fetch(props.apiUrl, {
      method: "POST",
      body: JSON.stringify({
        name: inputsValue.name,
        userName: inputsValue.username,
        email: inputsValue.email,
      }),
      headers: {
        "Content-type": "application/json; charset=UTF-8",
      },
    })
      .then((response) => response.json())
      .then((json) => {
        setLoading(false);
        checkResponse(json);
      });
  }

  function handleClose() {
    setShowSuccess(false);
  }
```

```

  function handleChange(e) {
    const name = e.target.name;
    const value = e.target.value;
    setInputsValue((inputsValue) => {
      return { ...inputsValue, [name]: value };
    });
  }

  function checkResponse(json) {
    if (json.name && json.userName && json.email) {
      setShowSuccess(true);
      return;
    }
    if (!json.name) {
      setErrorInputs((errorInputs) => ({
        ...errorInputs,
        name: "This field is required",
      }));
    }
    if (!json.userName) {
      setErrorInputs((errorInputs) => ({
        ...errorInputs,
        username: "This field is required",
      }));
    }
    if (!json.email) {
      setErrorInputs((errorInputs) => ({
        ...errorInputs,
        email: "This field is required",
      }));
    }
  }

  return (
    <>
      <form noValidate autoComplete="off" onSubmit={handleSubmit}>
        <div className="form">
          <TextField
            id="name"
```

```
              name="name"
              label="Name"
              error={typeof errorInputs.name !== "undefined"}
              helperText={errorInputs.name ? errorInputs.name : ""}
              required={true}
              fullWidth={true}
              value={inputsValue.name}
              onChange={handleChange}
            />
          </div>
          <div className="form">
            <TextField
              id="username"
              name="username"
              label="User Name"
              error={typeof errorInputs.username !== "undefined"}
              helperText={errorInputs.username ? errorInputs.username : ""}
              required={true}
              fullWidth={true}
              value={inputsValue.username}
              onChange={handleChange}
            />
          </div>
          <div className="form">
            <TextField
              id="email"
              name="email"
              label="EMail"
              error={typeof errorInputs.email !== "undefined"}
              helperText={errorInputs.email ? errorInputs.email : ""}
              required={true}
              fullWidth={true}
              value={inputsValue.email}
              onChange={handleChange}
            />
          </div>
          <div>
```

```
113           <Button type="submit" variant="contained"
               ↪  color="primary">
114             {loading && <CircularProgress color="inherit"
                 ↪  size={24} />}
115             {!loading && "Submit"}
116           </Button>
117         </div>
118       </form>
119       <Snackbar
120         anchorOrigin={{
121           vertical: "top",
122           horizontal: "center",
123         }}
124         open={showSuccess}
125         autoHideDuration={3000}
126         onClose={handleClose}
127       >
128         <Alert severity="success">User Created Successfully
             ↪  !</Alert>
129       </Snackbar>
130     </>
131   );
132 }
```

To implement the form submission, we have added the `onSubmit` event to the `form` element, on line 72. And the handler function `handleSubmit`, which is defined on line 11. On line 7 we define the `loading` state variable that will allow us to show a spinner while the user is waiting for the response after submission. On line 8 we define the `errorInputs` state variable that we will use, in case of validation errors, to store the error message that will be displayed to the user. And on line 9 we define the `showSuccess` state variable used to show or hide a success or error message after receiving the response from the submission.

Now, if the user presses the submit button, on line 113, the `handleSubmit` function will be invoked. The first statement of the function, on line 12, will disable the default browser behaviour which is executed when a submission is done. In this default or native behaviour the browser will perform a request reloading the page. We don't want that, as we will perform the request ourselves using the `fetch` function. On line 13, we set to true the `loading` state variable. That immediately will show a spinner as the value of the submit button. Observe on line 114 the conditional render to show the spinner or the "Submit" value. On line 14, we initialise (cleaning it from the previous submission) the `errorInputs` state variable. Then, on line 15 we invoke the `fetch` function that performs the request. This is a POST request, as specified on line 16. On line 17 we define the body of the POST request, using the `JSON.stringify` function which transforms an object or a Javascript value into a JSON string. Note that we get the values from the `inputsValue` state variable, and not from the DOM elements.

Finally, after the request finishes, we stop the spinner (on line 28) and then we inspect the response to verify if there is an error or everything is fine. That is done in the function `checkResponse`. As mentioned, we are using the JSON Placeholder API, which fakes inserts, updates or deletes. It always responds with success, plus the exact body that you send in the request, plus the *id* of the new inserted element. In order to show how forms validation works in React we will follow the next strategy. If the user does not complete the three inputs in the form and do the submission, the response will tell us which input was not completed, and we will use that to inform the user of the "field required" message, as shown in figure 3.13. That is basically what the `checkResponse` function on line 45 does.

The `errorInputs` state variable is an object with the same structure as the `inputsValue` state variable. One property for each of the input elements of the form. So, have a look at the `checkResponse` function again. It first checks if in the response I have the three input elements, if that is the case it means that the user has completed the form correctly. In that case we just display a success message using the Snackbar component. If any property is missing, we set it in the `errorInputs` with the property as the name and the value is the error message that we want to show to the user, in our case this is "This field is required".

If that happens, if `errorInputs` is updated, as we know, it will re-render the component and with that, we want to show the error message to the user as shown in figure 3.13. To display the error to the user, the TextField component has two props: `error` and `helperText`. Note that we use them on lines 78, 79, 91, 92, 104 and 105. The `helperText` is the message to display associated with the input, which in our case is "This field is required". And the `error` is a boolean prop. If true, the input label, their borders and the `helperText` message becomes red. As I mentioned, there is no real validation on this fake API. However, what we showed is the way you can follow to implement a decent form validation. You just need to talk with your API provider (maybe it is just another mate on your team or yourself) which will be the format of the response to identify inputs and their error message.

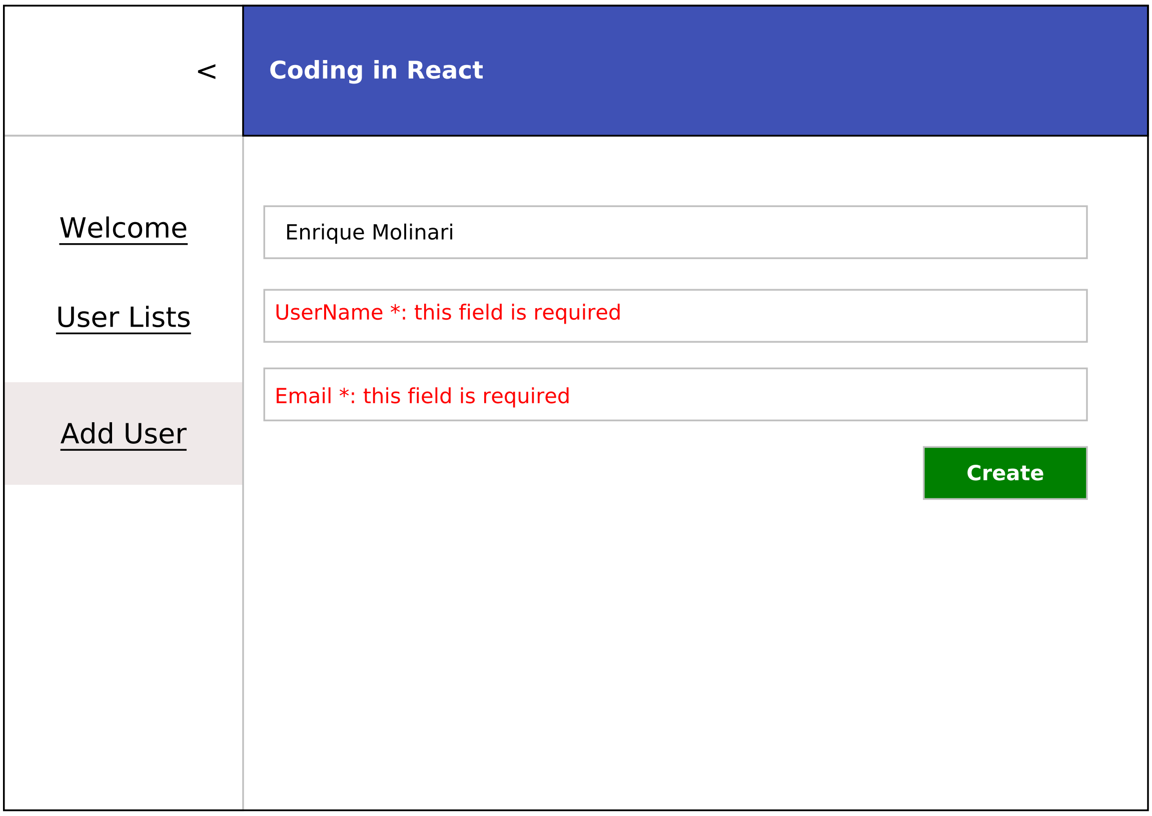


Figure 3.13: Form Validation

3.10 Re Write with Class-Based Components

For those who prefer class-based components, I have rewritten this application transforming the function-based components into class-based ones. Full source code can be seen here: github/crud.

Chapter 4

Creating a Blog

In this chapter we will design and build a blog application that we call **react-blog**. The full source code is available on github following the link: react-blog. To see the react-blog application running, I have written a small back-end application that can be downloaded and installed from github too, following the link: blog-api. The blog-api back-end application exposes the set of APIs required by the react-blog application. They provide the content for the blog.

To build the react-blog application I have used the html theme called Editorial from html5up.net[7]. From the original sources I have just removed JQuery, as we are only interested in the HTML markup and the CSS files.

Let's start by describing what functionality the blog application will have. Like in the previous chapter, we will present a set of figures that illustrate what we are going to build.

4.1 Identifying Components

The figure 4.1 shows the homepage of the blog application. As can be appreciated, the blog will have two main panes. On the left pane, there is a search box, a menu with a list of author names and the number of posts they have published, a contact information and the license. The right hand pane will be used to display different views of the blog. Specifically on figure 4.1, on the right pane you are seeing at the top the title of the blog and a page explaining what this blog is about. And on the bottom you have the latest posts published. Note that you can click on the *Read More* buttons to get the full text.

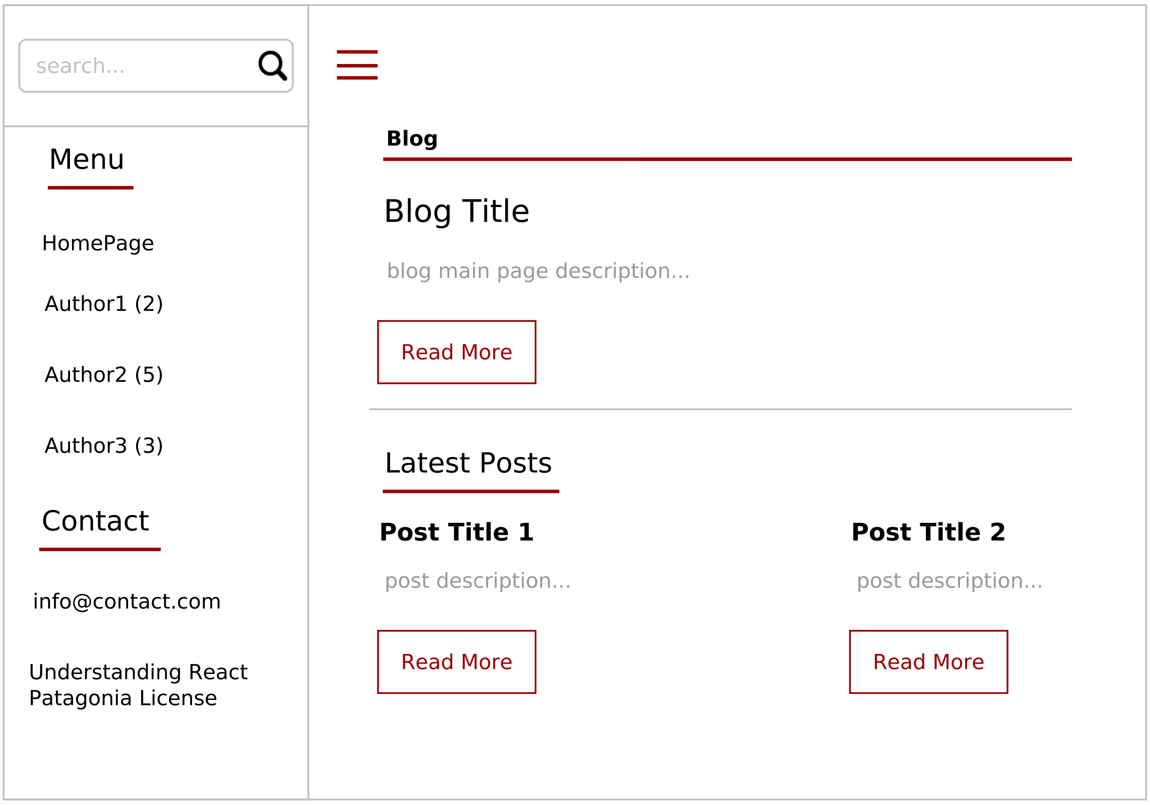


Figure 4.1: Blog Home Page

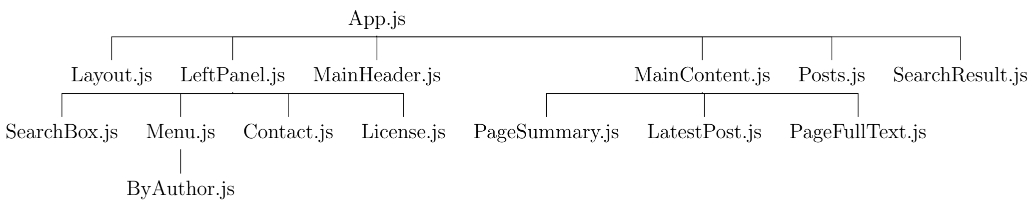
Then, figure 4.2 shows the full text of the main page of the blog. On figure 4.3 you can see the full text of a post and finally on figure 4.4 you can see the search result list.

We have identified 14 components which are depicted on figures 4.5, 4.6, 4.7, 4.8. Below is the list:

1. Layout (orange square) on figure 4.5.
2. Left Pane (black square) on figure 4.5.
3. SearchBox (red square) on figure 4.5.
4. Menu (green square) on figure 4.5.
5. ByAuthor (light green square) on figure 4.5.
6. Contact (yellow square) on figure 4.5.

7. License (turquoise square) on figure 4.5.
8. Main Header (gray square) on figure 4.5.
9. Main Content (blue square) on figure 4.5.
10. Page Summary (orange square) on figure 4.5.
11. Latest Posts (violet square) on figure 4.5.
12. Page Full Text (blue square) on figure 4.6.
13. Posts (orange square) on figure 4.7.
14. Search Result (green square) on figure 4.8.

And the component hierarchy illustrated below:



There are, of course, some components that are conditionally rendered, but we will study that in the next section. Let's review now the APIs required to make the blog alive.

4.1.1 Components and APIs

To understand better how this blog is implemented, we will review the APIs that some components consume to display data.

Retrieving a Blog Page by Id

This API is invoked by the `MainContent.js` component (blue box on figure 4.5) and the response is passed by prop to the `PageSummary.js` child component (orange box on figure 4.5).

The URL of this GET endpoint is: `https://{host}/pages/{Id}`, and respond with the output:

```
[
   {
      "_id":{
         "$oid":"..."
      },
      "title":"...",
      "text":"...",
      "author":"...",
      "date":{
         "$date":"..."
      }
   }
]
```

Retrieving a Blog Post by Id

This API is invoked by the component `Posts.js` and the response is rendered in this same component. The visual representation of this component can be seen on figure 4.7.

The URL of this GET endpoint is: https://{host}/posts/{Id}, and respond with the output:

```
[
   {
      "_id":{
         "$oid":"..."
      },
      "title":"...",
      "resume":"...",
      "text":"...",
      "tags":"...",
      "relatedlinks": "...",
      "author":"...",
      "date":{
         "$date":"..."
      }
   }
]
```

Retrieving a Blog Post by Author Name

This API is invoked by the component `Posts.js` and the response is rendered in this same component. The visual representation of this component can be seen on figure 4.7. As we are retrieving posts filtered by author name it might respond with more than one blog post.

The URL of this GET endpoint is: https://{host}/posts/author/{name}, and respond with the output:

```
[
   {
      "_id":{
         "$oid":"..."
      },
      "title":"...",
      "resume":"...",
      "text":"...",
      "tags":"...",
      "relatedlinks": "...",
      "author":"...",
      "date":{
         "$date":"..."
      }
   }
]
```

Retrieving Latest's Blog Posts

This API is invoked by the component `LatestPost.js` and the response is rendered in this same component. The visual representation of this component can be seen on figure 4.5, in the violet box.

The URL of this GET endpoint is: https://{host}/posts/latest, and respond with the output:

```
[
   {
      "_id":{
         "$oid":"..."
      },
      "title":"...",
      "resume":"...",
   },
   {
      "_id":{
         "$oid":"..."
```

```
        },
        "title":"...",
        "resume":"...",
    },
]
```

Retrieving Total Posts Grouped by Author Name

This API is invoked by the component `ByAuthor.js` and the response is rendered in this same component. The visual representation of this component can be seen on figure 4.5, in the light green box.

The URL of this GET endpoint is: https://{host}/byauthor, and respond with the output:

```
[
    {
        "_id": "...",
        "count": 2,
    },
    {
        "_id": "...",
        "count": 12,
    },
]
```

Retrieving Search Results

This API is invoked by the component `SearchBox.js` (red square on figure 4.5) and the response is passed by prop to the `SearchResult.js` component (green box on figure 4.8).

The URL of this GET endpoint is: https://{host}/search/{searched text}, and respond with the output:

```
[
    {
        "_id":{
            "$oid":"..."
        },
        "title":"...",
        "resume":"...",
        "text":"...",
```

```
        "tags":"...",
        "relatedlinks": "...",
        "author":"...",
        "date":{
           "$date":"..."
        }
     },
  ]
```

4.2 React Router

To implement conditional rendering, this time we are going to use React Router. React Router is a module with a collection of components that allows you to do, in an elegant way, and among other things, conditional rendering.

4.2.1 Defining Routes

The first thing you have to do to use React Router, is to wrap your root component with the BrowserRouter component. The best place to do this is on the `index.js` file as shown below:

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import App from "./App";
import { BrowserRouter } from "react-router-dom";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
```

With this in place we are ready to start studying the Routes and Route components. Let's review how these components can be used on the `App.js` root component on the example below:

```
1  export default function App() {
2    const apiUrl = process.env.REACT_APP_API_URL;
3    const pageId = process.env.REACT_APP_MAIN_PAGE_ID;
4
5    return (
6      <Layout
7        leftPane={<LeftPanel apiUrl={apiUrl} />}
8        mainPane={
9          <>
10           <MainHeader />
11           <Routes>
12             <Route
13               path="/posts/author/:name"
14               element={<Posts apiUrl={apiUrl} />}
15             />
16             <Route path="/posts/:postId" element={<Posts
                ↪ apiUrl={apiUrl} />} />
17             <Route
18               path="/*"
19               element={<MainContent apiUrl={apiUrl}
                  ↪ pageId={pageId} />}
20             />
21             <Route path="/search/result"
                ↪ element={<SearchResult />} />
22           </Routes>
23         </>
24       }
25     />
26   );
27 }
```

As can be seen in the previous example, each route is defined by the `Route` component, and it is mandatory to wrap them all using the `Routes` component. The `path` property of the `Route` component, is compared against the URL that is currently being navigated. If the URL and the `path` match, the component specified in the `element` property get rendered. The `path="/*"` property on line 18 indicates that there are descendant routes, in this case, routes defined in the `MainContent` component. Let's see how that component looks like:

```
export default function MainContent(props) {
  const [mainPage, setMainPage] = useState([]);
```

```
3
4      useEffect(() => {
5        fetch(props.apiUrl + "pages/" + props.pageId)
6          .then((response) => response.json())
7          .then((response) => {
8            setMainPage(response);
9          });
10     }, []);
11
12     return (
13       <>
14         <section id="banner">
15           <Routes>
16             <Route index element={<PageSummary page={mainPage}
                ↪  />} />
17             <Route path="/page/full" element={<PageFullText
                ↪  page={mainPage} />} />
18           </Routes>
19         </section>
20         <Routes>
21           <Route index element={<LatestPost
              ↪  apiUrl={props.apiUrl} />} />
22         </Routes>
23       </>
24     );
25   }
```

`<Route path>` is relative, which means that it automatically build on their parent route's path. In our case, the parent is just `"/"`. The `index` property on `Routes` on lines 16 and 21 indicates to use the path from the parent, no additional path are appended. On the other hand if we navigate to `"/page/full"` the `PageFullText` component gets rendered.

4.2.2 Navigation

So far we have studied how to configure the routes to do conditional rendering. Now, let's study how to navigate, which means to change the current URL to make some components to be rendered and others to be removed from the DOM. This can be done using the `Link` component. Below is the implementation of the `Menu.js` component.

```
1  export default function Menu(props) {
2    return (
3      <nav id="menu">
4        <header className="major">
5          <h2>Menu</h2>
6        </header>
7        <ul>
8          <li>
9            <Link to="/">Homepage</Link>
10         </li>
11         <li></li>
12       </ul>
13       <ByAuthor apiUrl={props.apiUrl} />
14     </nav>
15   );
16 }
```

Note on figure 4.5, green box, how this component is rendered on the browser. We have a homepage link and then the render of the `ByAuthor.js` component. When using React Router, links are created by using the `Link` component. You can see this on line 9 of the previous example. `Link` accepts the property `to` which is used to specify the route you want to navigate. On line 9 above, we are navigating to the / route which triggers the rendering of the `MainContent.js`. Similar to `<Route path>`, the `<Link to>` is relative too.

It is also possible to send *path* parameters on the URL and obtain their values on the rendered components. This is implemented by using a combination of the `Link` component, the `Route` component and the useParams hook. First it is necessary to define on the `Route path` property the parameter name and position. We have already done this on the `App.js` component. See below an extraction from that component source code:

```
1  <Route path="/posts/author/:name" element={<Posts apiUrl={apiUrl} />}/>
2  <Route path="/posts/:postId" element={<Posts apiUrl={apiUrl} />} />
```

On lines 1 and 2 above we have defined the `path` property which ends up with a parameter name. These routes are used to render the `Posts.js` component which displays blog posts, obtaining them by filtering them by author or by Id. To use one of these routes, we have to use the `Link` component. On the `ByAuthor` component, illustrated on figure 4.5 (light green box), we render a list of authors where each item on the list is a link. Let's see below how we have implemented this:

```
1 export default function ByAuthor(props) {
2   const [results, setResults] = useState([]);
3
4   useEffect(() => {
5     fetch(props.apiUrl + "byauthor")
6       .then((response) => response.json())
7       .then((response) => {
8         setResults(response);
9       });
10   }, []);
11
12   return (
13     <>
14       <ul>
15         {results.map((item) => (
16           <li key={item._id}>
17             <Link to={"/posts/author/" + item._id}>
18               {item._id + " (" + item.count + ")"}
19             </Link>
20           </li>
21         ))}
22       </ul>
23     </>
24   );
25 }
```

This component consumes the /byauthor API, as described in the previous section, and renders the response as a list. Each item on the list links to the URL /posts/author/:name. Note that starting on line 15, we are iterating over the list of authors obtained from the API and generating the value of the `Link` `to` property dynamically.

And finally, we will show how to obtain the path parameter value. As we showed before from the implementation of the `App.js` component, the URL /posts/author/:name renders the `Posts.js` component passing the *name* parameter. Let's have a look at the source code of the `Posts.js` component below:

```
export default function Posts(props) {
  const [posts, setPosts] = useState([]);
  const { postId } = useParams();
  const { name } = useParams();

  useEffect(() => {
    let uri = "posts/";
    if (postId) uri += postId;
    if (name) uri += "author/" + name;

    fetch(props.apiUrl + uri)
      .then((response) => response.json())
      .then((response) => {
        setPosts(response);
      });
  }, [postId, name]);

  return (
    <span key={name}>
      {posts.map((post) => (
        <section key={post._id.$oid}>
          <header className="main">
            <h1>{post.title}</h1>
          </header>
          <h3>{post.resume}</h3>
          <p>{post.text}</p>
          <h4>Related Links</h4>
          <ul className="alt">
            {post.relatedlinks.map((link, index) => (
              <li key={index}>{link}</li>
            ))}
          </ul>
          <h4>Tags</h4>
          <ul>
            {post.tags.map((tag, index) => (
              <li key={index}>{tag}</li>
            ))}
          </ul>
          <h4>Author</h4>
          <p>{post.author}</p>
        </section>
```

```
42          ))}
43          <p>
44            <Link to="/">Back to Home Page</Link>
45          </p>
46        </span>
47      );
48    }
```

On lines 3 and 4 we use the `useParams` hook from React Router to get the value of the path parameters: `postId` or `name`. The `Posts.js` component is used to render full text blog posts, but they are obtained by using either the /posts/:postId API or the /posts/author/:name API, depending on the parameter received. Note that on lines 7, 8 and 9 we are building the URL of the API based on the parameter obtained from the path. Once the URL is built we do the API call on line 11 and the response is rendered.

4.2.3 Programmatically Navigation

The search functionality of the blog is implemented by two components: `SearchBox.js` and `SearchResult.js`. The visual representation of the `SearchBox.js` component is illustrated by the figure 4.5, red square. And the visual representation of the `SearchResult.js` component is illustrated by the figure 4.8, green square.

The `SearchBox.js` renders the form with the input text. On the submission event, the /search/{searched text} API is consumed and with that done we have to *somehow* navigate to the /search/result URL, passing the response (the search result response object) as parameter. It is the `SearchResult.js` component which renders the results.

Note that we have to do this inside of an event handler function which means that the `Link` component cannot be used. This time we have to use the `useNavigate` hook. Let's review the source code of the `SearchBox.js` component:

```
export default function SearchBox(props) {
  const [textSearch, setTextSearch] = useState("");
  const navigate = useNavigate();

  function handleSubmit(e) {
    e.preventDefault();
```

```
7
8       fetch(props.apiUrl + "search/" + textSearch)
9         .then((response) => response.json())
10        .then((response) => {
11          navigate("/search/result", { state: response });
12        });
13    }
14
15    function handleChange(e) {
16      setTextSearch(e.target.value);
17    }
18
19    return (
20      <section id="search" className="alt">
21        <form method="get" onSubmit={handleSubmit}>
22          <input type="text" onChange={handleChange}
              ↪  placeholder="Search" />
23        </form>
24      </section>
25    );
26  }
```

On line 3 above we obtain the navigate function by calling the `useNavigate` hook. This function allows us to change the URL to *navigate* to an specific path (same as the `Link` component).

The `SearchBox.js` component implements the search form in a controlled way as explained in section 3.9. The state variable `textSearch` holds the search text typed by the end user. On the `handleSubmit` handler on line 5, note that we use that state variable to create the URL of the request (line 8). With the response, on line 11, we `navigate` to the `"/search/result"` URL. The URL of the new location is the first parameter of the `navigate` function (which is mandatory), and we use the second parameter (which is optional), to pass data to the rendered component. In this case we pass the entire response obtained from the API, as a property called `state`.

Calling the navigate function **triggers** the re-render, and based on the location URL passed as the first parameter we know that the `SearchResult.js` component will get rendered.

Now, let's study how we can obtain the data sent by the `navigate` function call, by looking at the implementation of the `SearchResult.js` component:

```
1 export default function SearchResult() {
2   const location = useLocation();
3 
4   return (
5     <div className="table-wrapper">
6       <table>
7         <thead>
8           <tr>
9             <th>Author</th>
10            <th>Title</th>
11            <th>Resume</th>
12            <th>Post</th>
13          </tr>
14        </thead>
15        <tbody>
16          {location.state.map((result, index) => (
17            <tr key={index}>
18              <td>{result.author}</td>
19              <td>{result.title}</td>
20              <td>{result.resume}</td>
21              <td>
22                <Link to={"/posts/" + result._id.$oid}>Read more...</Link>
23              </td>
24            </tr>
25          ))}
26        </tbody>
27      </table>
28      <ul className="actions">
29        <li>
30          <Link to="/">Back To Home Page</Link>
31        </li>
32      </ul>
33    </div>
34  );
35 }
```

On line 2, we use the `useLocation` hook to obtain the location object. The location object represents the current location URL and has the following structure:

```
{
  pathname: '/alocation',
  search: '?some=querytring',
  hash: '#hashx',
  state: null,
  key: 'a key'
}
```

Note that it represents mainly all the information that a URL might have. But what is most important from there right now is the `state` property. There, is where we will have the data sent by the `navigate` function call (second optional parameter). Knowing this now, if you look at the implementation of the `SearchResult.js` component above, on line 16, you will see that we can call to the `location.state.map` to render the JSON structure described in section 4.1.1.

We have explained the most important components of React Router. You might have noted that it is a very clean and simple module. React and their component design architecture makes this possible. Now, you are ready to start your next application using React Router.

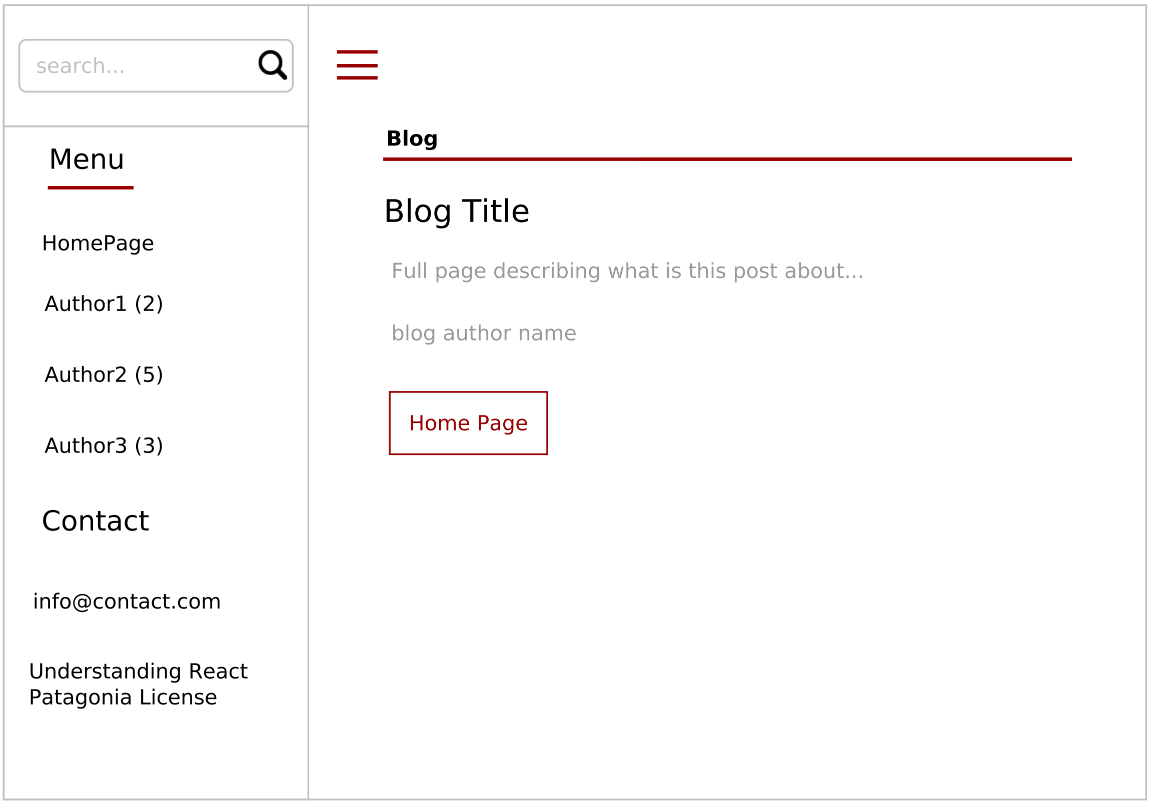


Figure 4.2: Blog Main Page Full Text

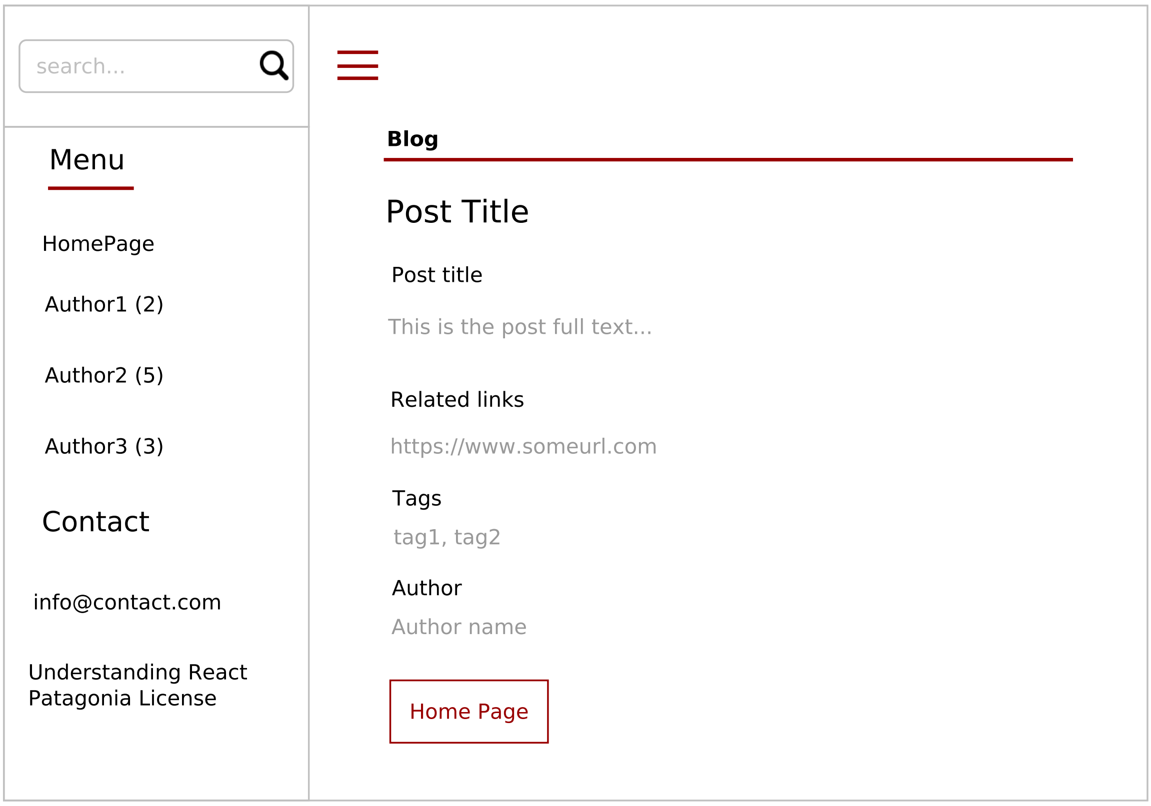


Figure 4.3: Blog Posts Full Text

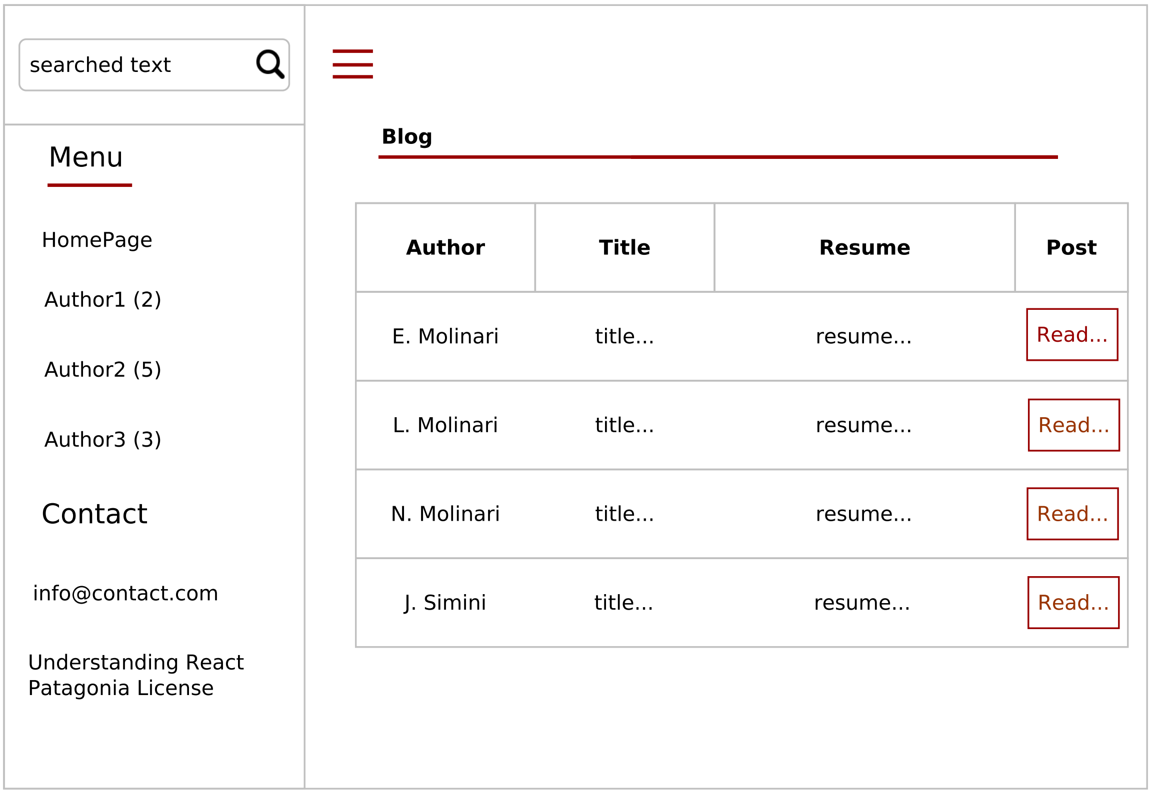


Figure 4.4: Blog Search Results

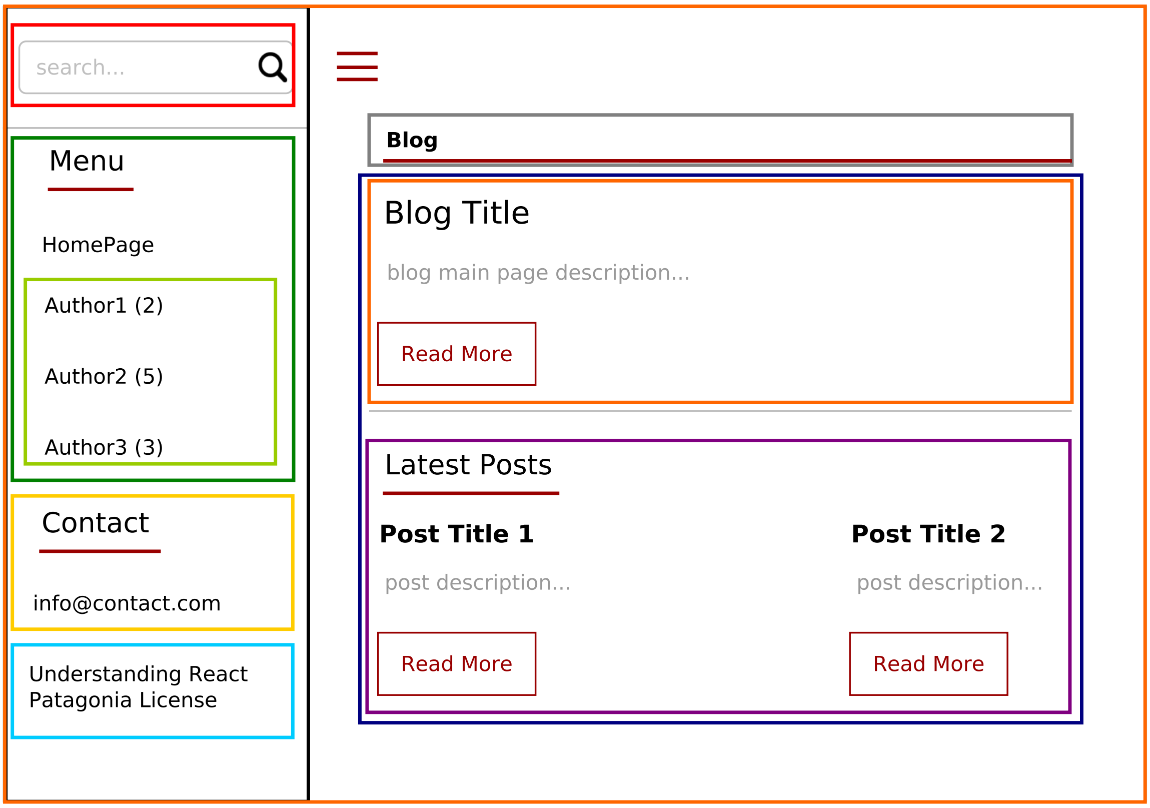


Figure 4.5: Layout (orange), LeftPane (black), SearchBox (red), Menu (green), ByAuthor (light green), Contact (yellow), License (turquoise), MainHeader (gray), MainContent (blue), PageSummary (orange), LatestPosts (violet).

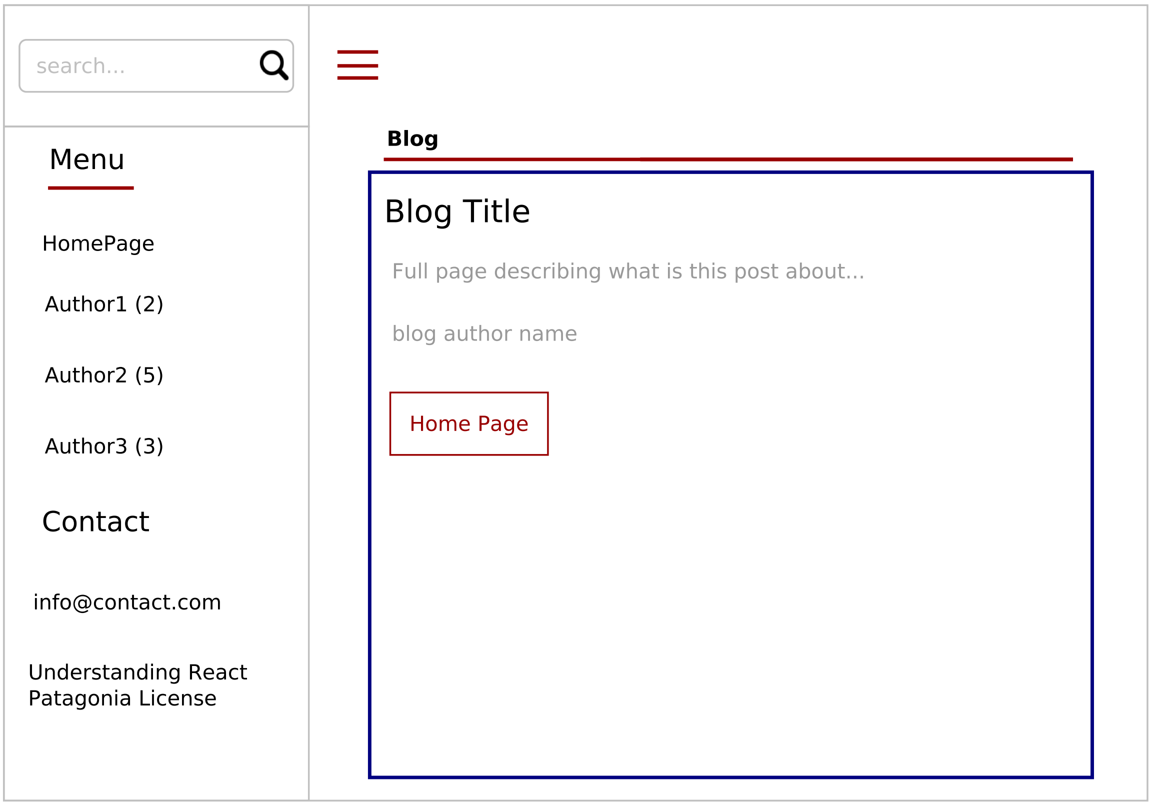


Figure 4.6: PageFullText (blue)

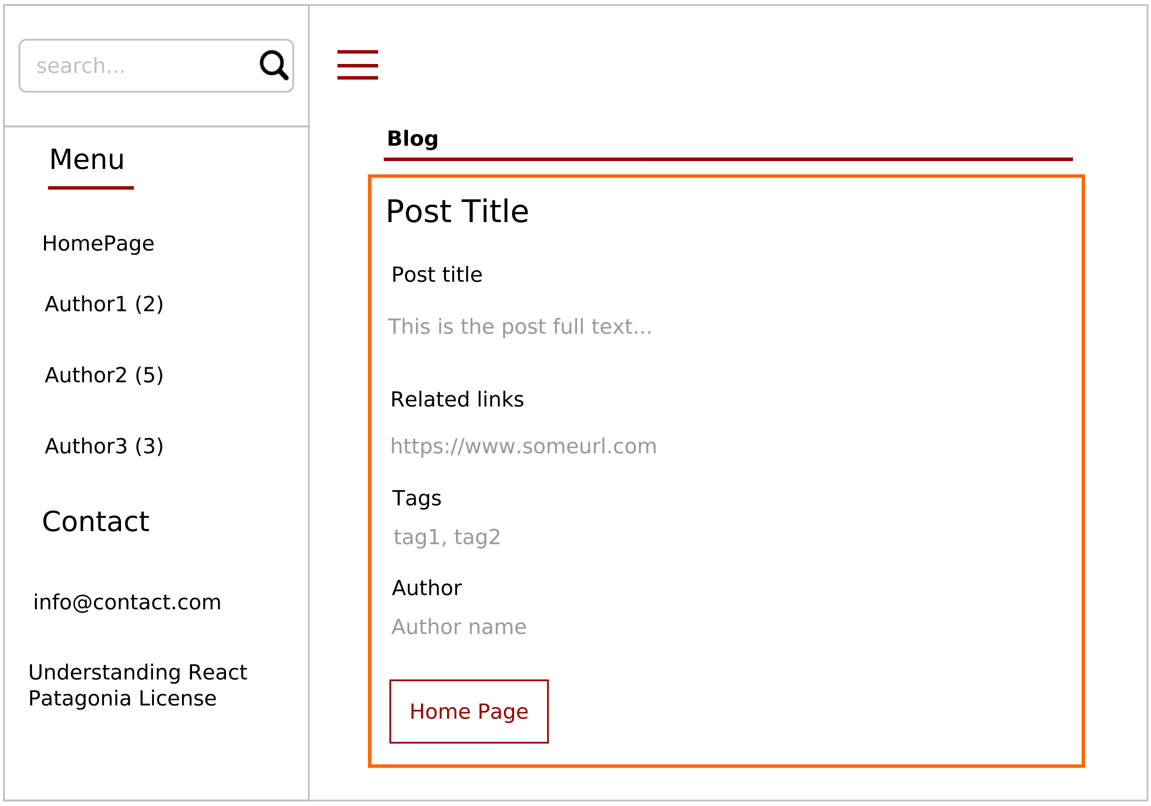


Figure 4.7: Posts (orange)

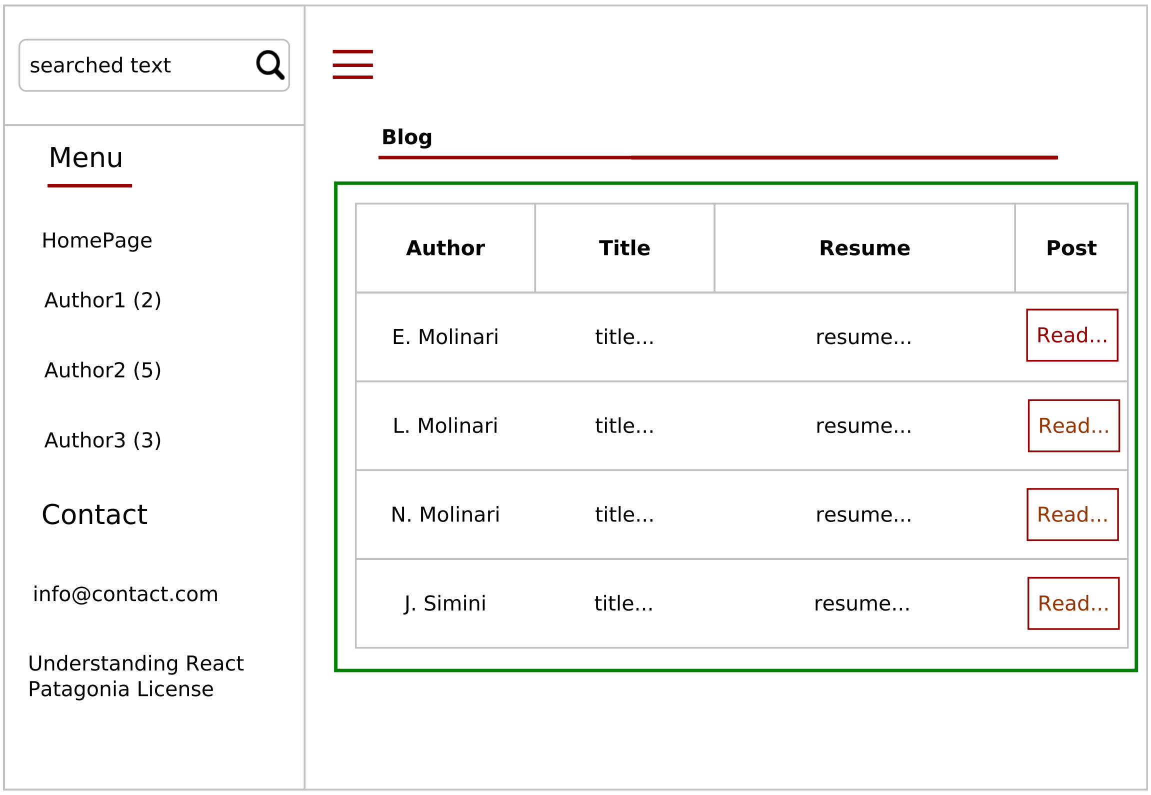


Figure 4.8: SearchResult (green)

Chapter 5

Authentication and Authorization

During this chapter I will explain how to implement a **secure** authentication and authorization flow for First-party Applications. First-party applications are those where the same organization provides both the API and the application that consumes that API. I would like you to know that every day new vulnerabilities are found, which means that what today is secure it might not be secure tomorrow. Make sure you frequently check the OWASP Web site (OWASP Cheatsheet).

Having said that, let's begin. As you might have guessed, I like to explain concepts by showing running applications, and this is not an exception. To explain the concepts of this chapter I have created an application called **Task List**.

5.1 Task List Application

The Task List application was built using the microservice architecture style. We have two back-end services: the authentication microservice called UserAuth and the Task List microservice. These back-end services are consumed by the Task List UI React application. All source code is available following the links.

The figure 5.1 presents the architecture of the solution using a container[1] diagram (C4 Model). As you can see, we have 7 containers. The **Task List UI**, which is the React application, that consumes services from the two microservices: **User Auth** and **Task List**. Note that the request from the UI to the microservices goes through an **API Gateway**. The microservices

[1]Not docker. A container in the C4 Model represents an application or a data store

use the Derby embedded (in memory) database. This will allows you (the reader) to easily start them without installing or starting any additional service.

Finally, we have a **Web Server** to serve the React application. Again, the request of the Task List UI application goes through the **API Gateway**. Having the React application talking only to the **API Gateway** (reverse proxy) has an important implication on the security of the authentication proposal that we want to share in this chapter. We will get back to this point later.

Figures 5.2 and 5.3 shows the login screen and a task list main screen respectively. In order to access to their task lists a user must type first their username and password (in the login screen 5.2). That will generate a request to the UserAuth microservice which validates user's credentials, and if successful it will return an access token. The access token allows the user to access their tasks consuming the Task List services. Where to store in the browser the access token is the topic of the next section, as this decision has security implications.

Once authenticated, the user can retrieve their tasks. They are presented in a list as shown in figure 5.3 with their expiration date. Depending how close to the deadline they are, they will get a different background color. If the user click on the checkbox, the task is marked as done (as shown in figure 5.3, second task). You can also delete tasks and add new tasks.

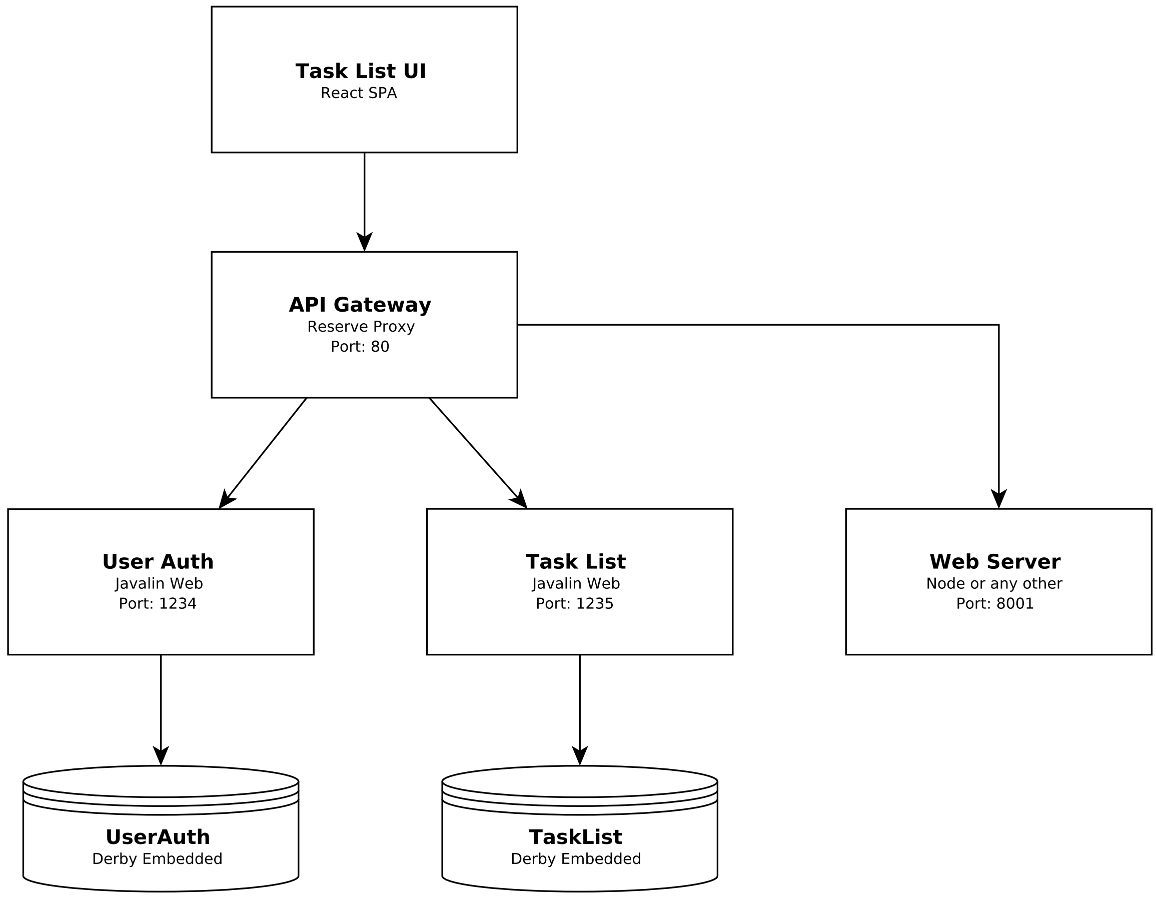


Figure 5.1: Task List - Microservice Architecture

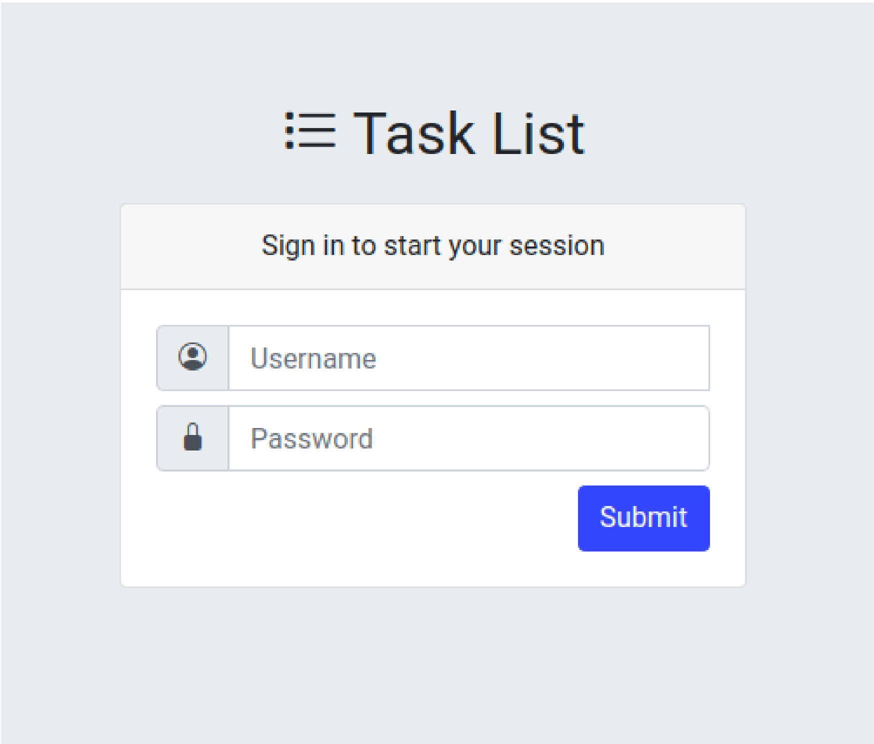


Figure 5.2: Login Page - Task List

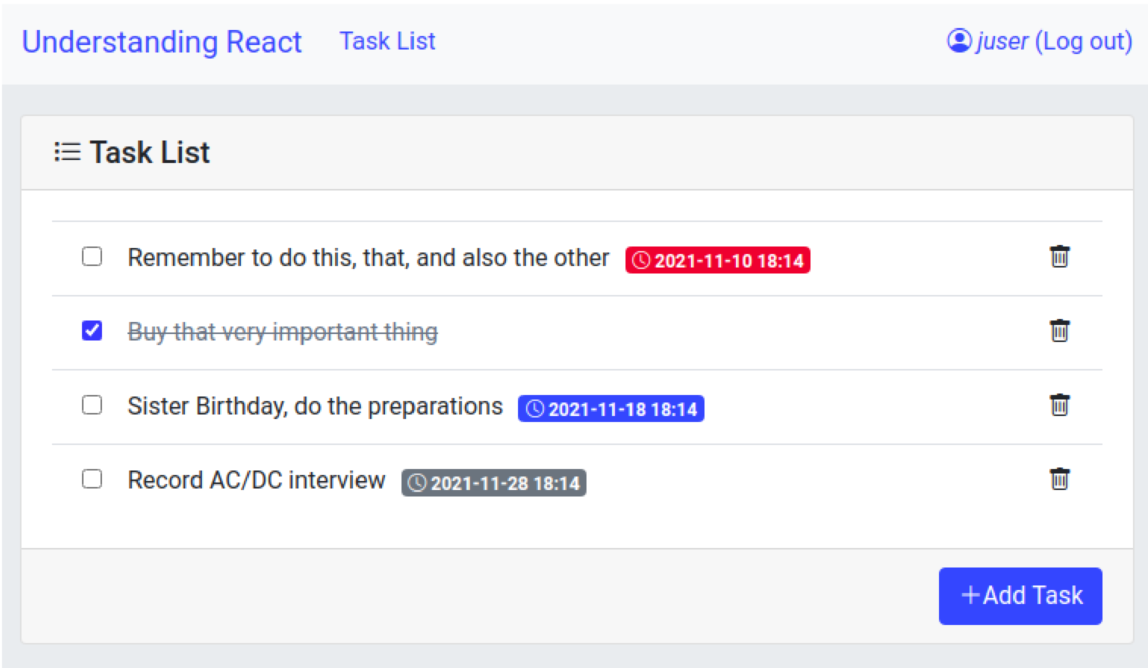


Figure 5.3: juser's Task List

5.2 XSS and CSRF

In this section, we are going to briefly describe two major security exploits that you must understand to know how to prevent it. These two exploits attempt to gain access to the user's authentication details, like the access token. Where to store the token and how to protect it is an important part of this chapter.

A more complete view about Web application security, can be found by visiting the OWASP (Open Web Application Security Project) web site. Specifically, the Top 10 security exploits they have published and up to date. The OWASP web site is full of details that will definitely help you understand the most common security exploits and the actions you can follow as remedy. Let's start by explaining specifically these two exploits.

XSS (Cross-Site Scripting) is a security exploit in which an attacker is able to inject malicious scripts into an application. As described by OWASP, XSS attacks occur when:

- Untrusted data, coming often in a Web request, enters into a Web application
- That untrusted data is sent to a browser without being validated for malicious content.

Untrusted data is the data that might come in an HTTP request: query parameters, form input fields, headers and cookies. Suppose the following back-end code that belongs to the fictitious myweb application:

```
value = request.param("q1");
response.add(value);
```

The back-end code above might be invoked passing a malicious scripts as the value of the `q1` parameter, like this:

`https://myweb.com/query?q1='a very malicious script'`

Note, from the back-end code, that there is no validation performed on the `value` received from the untrusted source before that is sent into the response to the browser. The browser will end up executing the malicious script. However, you might wonder, why would someone create such a malicious URL that will end up being executed on its own computer? The real problem come with an attacker creating that malicious URL and somehow (probably using social engineering) send that URL to a myweb authenticated user. For instance, the victim might receive that URL hidden behind a link like "Click to win!" in an HTML email page. If the authenticated user visit that link the malicious script will be executed on their own computer. That malicious script could be intended, among other things, to steal authentication tokens or sessions IDs.

As an advantage, Reacts prevents XSS attacks by escaping any data that you embed in JSX. See the next snippet of a React component:

```
1  export default function Xss() {
2   //...
3
4   fetch("{URI}/tasks", {
5    method: "POST",
6    body: JSON.stringify({
7      date: inputForm.date,
8      text: inputForm.text,
9    }),
10   headers: {
11     "Content-type": "application/json; charset=UTF-8",
12   },
13  })
14  .then((r) => r.json())
15  .then((json) => setStateVariable(json.text));
16
17  //...
18
19  //JSX escapes the stateVariable before render
20  return (
21    <h1>{stateVariable}</h1>
22  );
23 }
```

Note from the component above, that on line 4 we are making a request to a remote API. As part of this request, on lines 6, 7, 8 and 9 using the request's `body`, we are sending untrusted data from two form inputs: `date` and `text` (the form and state variables are not shown). Suppose now that the server-side code that gets executed when calling to `"{URI}/task"`, retrieve the values from the request's body, do some processing with them and as part of the response data, it includes the `text` value, without applying any kind of validation. Then, on the client-side, on the React component at line 14 and 15, we obtain from the response the `text` value (untrusted) and we use it sets the `stateVariable`. As you know, that update triggers the re paint of the component. Note that on line 21 we are asking React to render the now untrusted `stateVariable`. As mentioned, in this case, the `stateVariable` is escaped by React before rendering. Which is a very good thing for us.

However, there are many places in which you might want to render untrusted data where React won't do any escaping nor validation, like for instance:

```
<a href={stateVariable}>link text</a>
```

In the above case (as the value of the `href` attribute), if `stateVariable` contains a script (malicious or not) it will get executed. There is a great discussion thread on stackoverflow I recommend to read to have a better understanding about XSS vulnerabilities in React.

On the other hand, CSRF (Cross-Site Request Forgery) is a security exploit where an attacker forces an end user to execute an action (a request) in a Web application in which they are authenticated. The typical examples are for bank applications. An attacker can create a malicious URL to transfer money from the victim's account to the attacker account, like this:

```
https://bank.com/transfer?acc=my&amount=100
```

Again, using social engineering the victim might be redirected to a malicious Web site that present that link, see the HTML below:

```
<a href="https://bank.com/transfer?acc=my&amount=100">Click to
↪ win!</a>
```

The victim might click the "Click to win!" link in a moment in which she is authenticated in the bank application, transferring the money to the attacker.

This vulnerability can be exploited also in POST request, using for instance, a hidden form like below:

```
<body onload="document.forms[0].submit()">
...

<form action="https://bank.com/transfer" method="POST">
```

```
5      <input type="hidden" name="acc" value="my"/>
6      <input type="hidden" name="amount" value="100"/>
7     </form>
8
9     ...
10
11    </body>
```

Note the line 1 from the HTML form above, that means that as soon as the browser render the page, the submission is done without requiring any user action. CSRF attack requires the victim to visit the attacker web site, and from there the malicious request is performed. The problem occurs when the victim goes to the attacker web site when is authenticated in the bank application, because the browser will do the submission including the authentication cookies. The bank application cannot distinguish between a valid request and a malicious request, and will perform the action.

5.2.1 Same-Origin Policy

An attacker might also redirect the victim to their malicious site to perform a similar attack but using an Ajax script like the one below:

```
<script>
function send() {
 fetch("http://bank.com/transfer", {
   method: "POST",
   body: JSON.stringify({
     acc: "my",
     amount: 100,
   }),
   headers: {
     "Content-type": "application/json; charset=UTF-8",
   },
  });
}
</script>
<body onload="send()">
//...
</body>
```

Additionally, with Ajax is possible to use others HTTP methods like PUT or DELETE. Fortunately, this Ajax request is not executed by the browser thanks to the same-origin policy restriction. Due to the malicious script runs on the attacker's web site (origin 1) and the request points to the victim's web site (origin 2), the browser detects this difference and the request is cancelled. Two URLs have the same origin if the protocol, port (if specified), and host are the same for both.

5.2.2 Cross-Origin Resource Sharing

The same-origin policy is enabled by default as the main barrier to CSRF. However, if becomes necessary (i.e. requiring to call to third-party APIs) it can be relaxed. The way to relax this policy is by using what is known as CORS (Cross-Origin Resource Sharing). CORS allows servers to specify from which **origins** (protocol, domain or port) the browser can perform request to (other than its own). It is done by using specifics HTTP response headers which are read by the browser to determine if the back-end service allows CORS or not.

How does it work? If it is the server who needs to authorise the browser to perform the request, how does the browser do the initial CORS request?

The browser deals with different types of request and depending on which groups they belong to will work differently. There are three groups of request: **Simple Request**, **Non-Simple Request** and **Requests with Credentials** that we are detailing below:

Simple Request The requests in this group are supposed to be harmless (idempotent or without side-effects) to the back-end server that is why there is no so much control performed by the browser, they are just executed. They are considered harmless by the CORS definition, so you as a back-end service developer, must be aware of it in order to not write non idempotent APIs that meet these conditions. Why? Because the browser will perform the request and your back-end will execute it (no one is stopping that). However, the request will include the **origin** in a header and the server, after processing it, will attach the *access-control-allow-origin* to the response header indicating if the received origin is allowed or not (if the response header does not contain the *access-control-allow-origin* it will be assumed as not allowed). If it is allowed, the browser will allow the requested script to use the response data. If not, the response data will not be allowed to be used by the requested script. Simple Request are executed (hence the importance of being idempotent or not having side-effects), but depending on the CORS response header, the response data may or may not be accessible by the client script. The conditions a request must have to be in this group are below:

- Methods are GET, HEAD or POST
- No custom HTTP headers are sent, just CORS safelisted ones.
- Content-type header can only have the values: application/x-www-form-urlencoded, multipart/form-data and text/plain.
- If XMLHttpRequest is used, there is no listener attached to XMLHttpRequest.upload.
- ReadableStream object is not used.

Non-Simple Request Any request that doesn't meet the simple request conditions are considered non-simple ones. For the requests that belong to this group, the browser send what is known as **preflight** request, before performing the actual request. The goal of the preflight request is to understand if the server can handle CORS and allows or not CORS request. How does it work? Before sending a non-simple request, as they are considered back-end services with side-effects, the browser will send an OPTIONS request (called preflight request), including the *origin*, the *access-control-request-method* (the HTTP method of the actual request) and the *access-control-request-headers* (the HTTP headers of the actual request). The server must decide if accepts or not the request by including in the response the headers: *access-control-allow-origin* (the origins allowed), *access-control-allow-methods* (the methods allowed), *access-control-allow-headers* the headers allowed, *access-control-max-age* (seconds to cache the OPTIONS response). If CORS headers are not added in the response or the back-end service does not allow the request, then, the actual request is not performed by the browser. Otherwise, the browser will perform the actual request.

Requests With Credentials This is a particular case (stricter) of a non-simple request. By default in cross-origin request, the browser will not send credentials in Ajax request. Credentials can be cookies or authorization headers (plus TLS client certificates). In order to make this work in a cross-origin request, both, the client must specify that requires to include credentials and the back-end service must respond (as part of the preflight response) with the header *access-control-allow-credentials* set to true. The following example shows how to perform a cross-site `fetch` request including credentials.

```
fetch('https://bank.com/transfer', {
  credentials: 'include'
})
```

5.2.3 Token Storage

As mentioned before, when you submit from the login screen your credentials, if they are correct, the back-end service will respond with an access token which allows your React application to perform subsequent requests to token's protected APIs. So, where to store the access token in the browser is what we have to decide. The options are described below:

localStorage or sessionStorage As described by OWASP HTML5 CheatSheet, "Do not store session identifiers in local storage as the data is always accessible by JavaScript". Any XSS exploit can be used to steal the access token.

cookie Cookies are currently the best choice. In the last years they were improved to accommodate them to new security vulnerabilities. Now, cookies can be configured to be **httpOnly** (not accessible by JavaScript), **secure** (only transmitted in secure channels) and **same-site** = strict or lax (only sent by the browser in same-origin request). The full stack application we have written as part of this chapter uses a cookie to store the access token.

5.2.4 Best Practices

In this section we are just summarising what are the recommendations or best practices to be protected against XSS and CSRF exploits. The list below is strict. It might occur, depending on the use cases you have to support, that you are not able to follow all the recommendations below. In this case, you will have to relax some of them, and based on which one you relax, you will know what are the possible vulnerabilities you are exposing your application. However, as we will see on the section 5.3, if you are developing a first-party application you will be able to follow everyone on the list.

Avoid any secret to be accessible via JavaScript As it was mentioned on the previous section, by storing access tokens on the **localStorage** or the **sessionStorage** you suffer the risk of getting stolen if XSS flaws are exploited. If you don't have other alternative, make the access tokens to expire in a short time and constantly review your code to make sure untrusted data is validated before including them in the response. By no means, you have to store third-party API keys on the browser, store them on the server, always.

Validate untrusted data Before rendering, untrusted data must be encoded. In addition, and especially if access tokens are stored in places accessible via JavaScript, it is recommended to sanitize untrusted data server-side before anything else. OWASP provides libraries in several languages to do this job (HtmlSanitizer).

Use JSON APIs As we have studied, AJAX calls are CORS-restricted by default. And there is no way for an HTML `<form>` to invoke a JSON API (passing inputs as a JSON body).

CORS not enabled Try at all cost to not enable CORS. If you require to call APIs on different origins, use a back-end service to forward to them. For instance, a reverse proxy or an API Gateway can be used for this purpose.

Simple-request must be idempotent Make sure that all the simple-request your React application performs don't have server-side side-effects. Remember that simple-request are submitted by the browser, without asking if CORS is allowed.

Verify that only newer browsers are used Prepare a barrier on the back-end that allows you to stop requests from certain older browsers. Browsers are updated frequently and sometimes the fixes are related to security flaws.

5.3 Login and Private Routes

In this section we are going to describe the implementation of the authentication and authorization flow of the Task List application. As mentioned, the back-end of the application was built using the microservice architecture style. The system has two services: the authentication microservice UserAuth and the Task List microservice. Both are written in Java using the Javalin lightweight web framework. In any case, you can implement your back-end in the language of your choice, as what is explained here is applicable to any other programming languages. Finally, we have the React application Task List UI. Following the links you will have full access to the source code.

As we will see during this section, the implementation follows the best practices described in section 5.2.4. Let's start describing the authentication flow illustrated in a sequence diagram in figure 5.5. Each rectangle from the diagram represents a container[2] from the (C4 Model). The first one, the **Task List UI** represents the browser executing the React application. Note that the React application talk to the **API Gateway** and not directly to the microservices or web server in this case. We use that approach mainly to be able to start the back-end services with any domain name and/or port. The browser will always make request to the API Gateway and is this one who forward the request to the appropriate back-end service. In this way, we don't have to enable CORS.

In the first request/response flow, highlighted with a green rectangle (labelled as React App on the top left corner) on figure 5.5, we are requesting the react application (entering a URL like http://localhost:port/ in the browser). Note that the static files from the application are downloaded from a Web server running on port 8001 (forwarded there from the API Gateway). The browser starts executing the application by rendering the `App.js` component. See below how this component looks like:

```
function App() {
  const apiGwUrl = process.env.REACT_APP_API_GW;

  return (
    <Routes>
      <Route
        path={"/"}
        element={
          <>
            <Menu apiGwUrl={apiGwUrl} />
            <Welcome />
          </>
        }
      />
      <Route
        path={"/tasklist"}
        element={
          <>
            <Menu apiGwUrl={apiGwUrl} />
            <PrivateRoute
              component={<TasksList apiGwUrl={apiGwUrl} />}
              requiredRoles={["SIMPLE", "ADMIN"]}
            />
```

[2]Not docker. A container in the C4 Model represents an application or a data store

```
24              </>
25            }
26          />
27          <Route path={"/login"} element={<Login apiGwUrl={apiGwUrl}
            ↪  />} />
28        </Routes>
29      );
30    }
31
32    export default App;
```

From the component above we can see that we are using React Router and when the route is the path / the rendered components are `Menu.js` and `Welcome.js`. That will paint on the browser what is shown on figure 5.4.

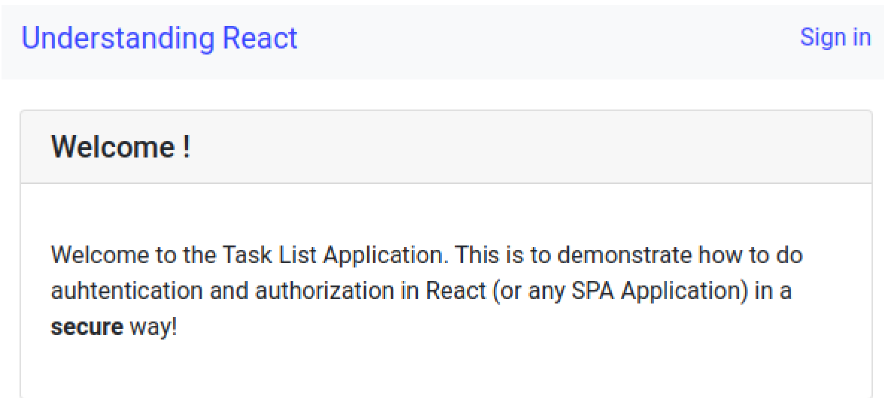


Figure 5.4: Welcome - Task List

After that, if the user clicks on the "Sign in" link it will request the `/login` route, which ends up rendering the `Login.js` component (see line 27 on the `App.js` component above). Let's discuss the `Login.js` component presented below (the full source code of the component can be found here, below is a fragment with the most relevant parts):

```
export default function Login(props) {
  const [loginForm, setLoginForm] = useState({
    user: "",
    pass: "",
  });
  const [errorResponse, setErrorResponse] = useState({
    msg: "",
```

```
    error: false,
  });
  const navigate = useNavigate();

  function handleSubmit(e) {
    e.preventDefault();

    new User(props.apiGwUrl)
      .login(loginForm.username, loginForm.password)
      .then((v) => {
        setErrorResponse({
          msg: "",
          error: false,
        });

        navigate("/");
      })
      .catch((v) => {
        setErrorResponse(v);
      });
  }

  return (
    <div className="login">
      <div className="login-logo">
        <i className="bi bi-list-task" />
        <b> Task List</b>
      </div>
      <Card>
        <Card.Header className="login-msg">
          Sign in to start your session
        </Card.Header>
        <Card.Body>
          <Form onSubmit={handleSubmit}>
            <InputGroup className="mb-2">
              <InputGroup.Prepend>
                <InputGroup.Text>
                  <i className="bi bi-person-circle"></i>
                </InputGroup.Text>
              </InputGroup.Prepend>
              <Form.Control
                name="username"
                type="text"
```

```
51                  placeholder="Username"
52                  onChange={handleChange}
53                  isInvalid={errorResponse.error}
54                />
55                <Form.Control.Feedback type="invalid">
56                  {errorResponse.msg}
57                </Form.Control.Feedback>
58              </InputGroup>
59
60              <InputGroup className="mb-2">
61                <InputGroup.Prepend>
62                  <InputGroup.Text>
63                    <i className="bi bi-lock-fill"></i>
64                  </InputGroup.Text>
65                </InputGroup.Prepend>
66                <Form.Control
67                  name="password"
68                  type="password"
69                  onChange={handleChange}
70                  placeholder="Password"
71                  isInvalid={errorResponse.error}
72                />
73              </InputGroup>
74
75              <Button variant="primary" type="submit">
76                Submit
77              </Button>
78            </Form>
79          </Card.Body>
80        </Card>
81      </div>
82    );
83  }
```

There are two important parts to highlight. First is that the login form is painted using React Bootstrap (Card, Form, InputGroup, etc, are components that belong to React bootstrap). The login screen is presented on figure 5.2. And second, when the form is submitted the `handleSubmit` function on line 12 is invoked. The request flow that is triggered when the login form is submitted is illustrated on figure 5.5 (green rectangle labelled Authentication on the top left corner). Please, note from the figure 5.5, that if the login is successful the response contains the user id, their name and roles and most important, the `httpOnly`, `secure` and `same-sate=strict` cookie. This cookie contains a paseto access token. Why paseto and not jwt? Among other things that are very well described here, Paseto supports the creation of encrypted tokens while jwt does not. If you want to see how the cookie and the paseto tokens are created look at the source code of the UserAuth microservice.

Note on line 15 above that the login request is delegated to the `User` object. Let's then study what this object does. See the source code below:

```
const STOREUNAME = "username";
const STOREUROLES = "roles";
const STOREUID = "id";

export default class User {
  constructor(apiUrl) {
    this.apiUrl = apiUrl;
  }

  userId() {
    return sessionStorage.getItem(STOREUID);
  }

  userName() {
    return sessionStorage.getItem(STOREUNAME);
  }

  hasRole(role) {
    let userRoles = sessionStorage.getItem(STOREUROLES);
    return role.includes(userRoles);
  }

  static current(apiUrl) {
    return new User(apiUrl);
  }

  logout() {
    return fetch(this.apiUrl + "/auth/logout", {
      method: "POST",
      headers: {
        "Content-type": "application/json; charset=UTF-8",
      },
    })
      .then((response) => response.json())
```

```
35       .then((json) => {
36         sessionStorage.clear();
37         return Promise.resolve();
38       });
39   }
40 
41   login(userName, password) {
42     return fetch(this.apiUrl + "/auth/login", {
43       method: "POST",
44       body: JSON.stringify({
45         user: userName,
46         pass: password,
47       }),
48       headers: {
49         "Content-type": "application/json; charset=UTF-8",
50       },
51     })
52       .then((response) => {
53         if (response.status === 401) {
54           return Promise.reject({
55             msg: "Username or password incorrect...",
56             error: true,
57           });
58         }
59         return response.json();
60       })
61       .then((json) => {
62         if (json.result === "success") {
63           sessionStorage.setItem(STOREUNAME, json.user.name);
64           sessionStorage.setItem(STOREUROLES, json.user.roles);
65           sessionStorage.setItem(STOREUID, json.user.id);
66           return Promise.resolve();
67         } else {
68           return Promise.reject({
69             msg: json.message,
70             error: true,
71           });
72         }
73       });
74   }
75 }
```

On line 41 above, you can see the implementation of the `login` method. The method performs a POST request passing the username and password

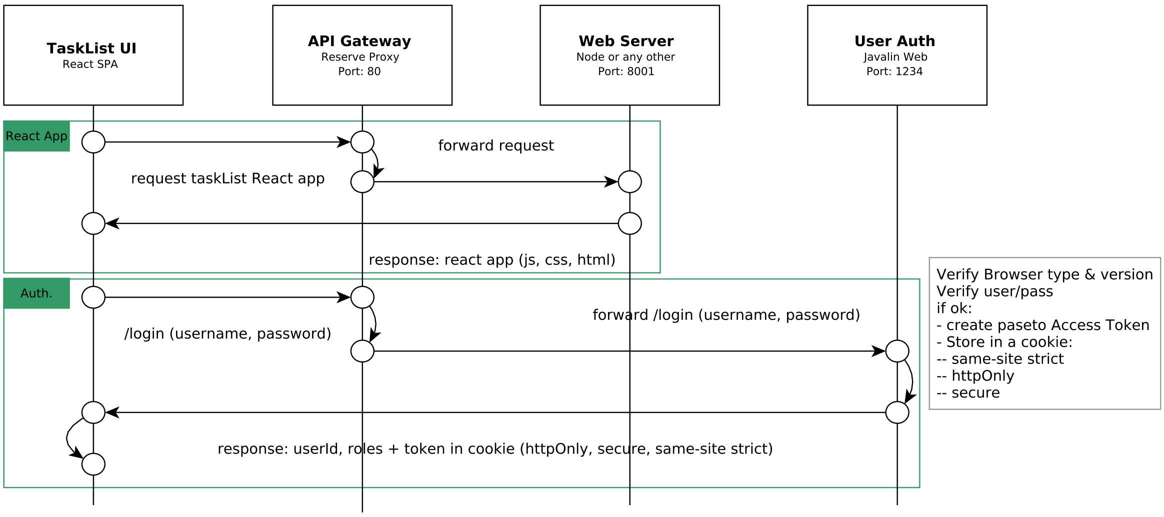


Figure 5.5: Login Flow

in the body. If the response is 401 (unauthorized) it will return a rejected Promise object with the error message to be displayed to the user. If the response is success it will store in the browser's session storage the id of the user, her name and her roles and it will return a resolved Promise. The data in the session storage is used to show or hide links from the UI to improve the user experience (UX). By no means, this is enough to control user authorization. As was mentioned before, the session storage is accessible by JavaScript (and pretty easy to change by browser's development tools) so it is just for UX, authorization must be done server-side. In addition, in a successful response we will receive the access token in a cookie. This token contains the same information (user id, name and roles), but it can't be changed by the user or an attacker. It is encrypted and signed and is used to perform authorization server-side. Using tokens that can store information is a convenient authentication and authorization method for the microservice architecture. Authentication and Authorization is a cross-cutting concern which means that all microservices somehow need to deal with it. The advantage of using tokens like JWT or Paseto, that can carry user's identity and roles, is that the microservices have in them all the data necessary to perform the corresponding validations, instead of having user's identity and roles replicated in each microservice database.

The cookie received in a successful response is **httpOnly**, which means that it cannot be accessed by JavaScript. It is **secure** to be only transmitted on secure channels and it is **same-site strict**, which means that it will be added to a request's header only to those that conform with the same-origin policy. Once authenticated, any `fetch` request that we do from the React application will contain the cookie with the access token.

Going back to the React application, on a successful authentication request the `User.login()` method returns a resolved Promise and then the `Login.js` component, on line 23, performs a `navigate("/")` that triggers the re render. If you look again at the source code of the `App.js` component above, the `Menu.js` and `Welcome.js` components are rendered. Let's have a look at the `Menu.js` component source code:

```
export default function Menu(props) {
  const userName = User.current().userName();
  const navigate = useNavigate();

  function handleLogout(e) {
    e.preventDefault();
    User.current(props.apiGwUrl)
      .logout()
      .then(() => navigate("/login"));
  }

  return (
    <Navbar bg="light" expand="sm">
      <Navbar.Brand href="#">
        <Link to="/">Understanding React</Link>
      </Navbar.Brand>
      <Navbar.Toggle aria-controls="basic-navbar-nav" />
      <Navbar.Collapse id="basic-navbar-nav">
        <Nav className="mr-auto">
          <Nav.Link href="#">
            <PrivateRoute
              component={<Link to="/tasklist">Task List</Link>}
              requiredRoles={["SIMPLE", "ADMIN"]}
            ></PrivateRoute>
          </Nav.Link>
        </Nav>
        <Nav>
          {!userName && <Link to="/login">Sign in</Link>}
          {userName && (
            <a href="#task" onClick={handleLogout}>
```

```
31                  <i className="bi bi-person-circle"> {userName}</i>
                     ↪  (Log out)
32                </a>
33              )}
34            </Nav>
35          </Navbar.Collapse>
36        </Navbar>
37      );
38    }
```

From the source code of the `Menu.js` component above, I wanted to highlight the component used on line 21. The `PrivateRoute.js` component, receives two `props`, the component to render and the roles required to render that component. It is a pretty simple component, you can see the source code below:

```
import User from "./User";

export default function PrivateRoute({ component, requiredRoles })
↪  {
  if (!User.current().userId() ||
  ↪  !User.current().hasRole(requiredRoles))
    return null;

  return component;
}
```

Since now the user is authenticated, user id, names and roles are stored in the session storage accessible by the `User` object. So, the `Link` component passed as prop to the `PrivateRoute` component on line 22 above is rendered. You can can see now on figure 5.6 the link highlighted in a red box.

Now, if the user clicks on the Task List link, the `TaskList.js` component will be rendered. On mounting, a simple GET request is performed by the component to retrieve user's task (`/tasks` API from the TaskList microservice), but this time, since the user is authenticated and the request goes to the same origin, the cookie is included by the browser (nothing additional is required by the developer). The flow is described in figure 5.7. At server-side (see source code here) the token is obtained from the cookie (if there is no cookie, the request is rejected as 401 unauthorized) and verified, if it was changed it will be discarded as invalid. After that, the roles that the token contains are compared against the roles required to execute the `/tasks` API. If everything success the user's task are returned, and the React application shows them as you can see on figure 5.3.

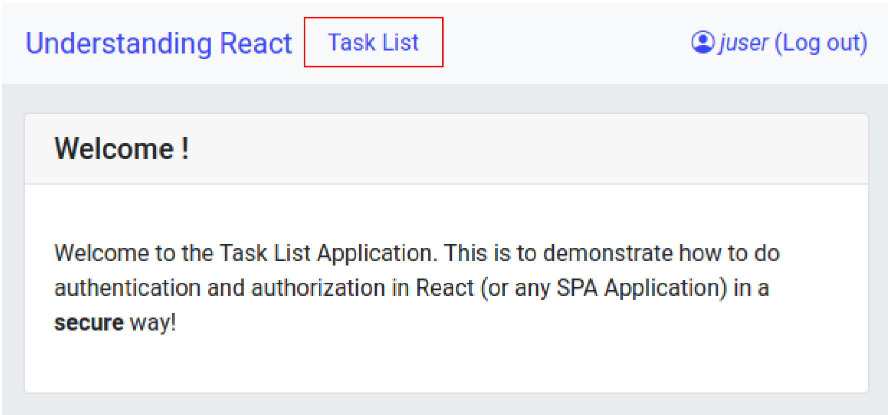


Figure 5.6: Authenticated Welcome Page - Task List

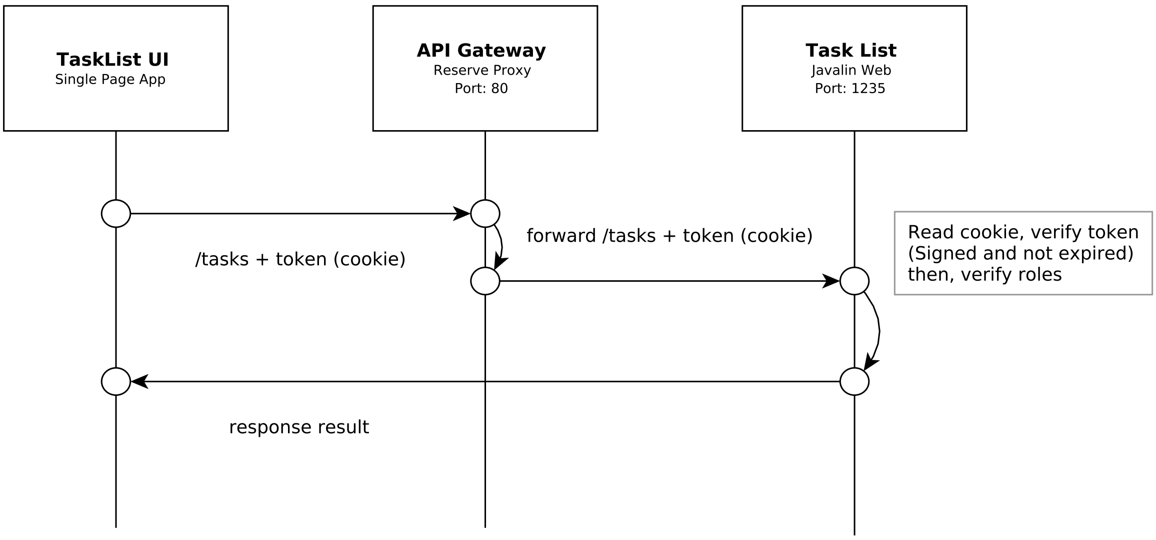


Figure 5.7: Authenticated Path

5.4 Logout

To do a proper logout, two things must happen: session storage must be deleted, and then and more importantly, the cookie must be deleted. The deletion of the cookie must be done server-side, and for that the UserAuth microservice exposes the `/logout` API. You can see above, the `handleLogout` function on the `Menu.js` component on line 5, that calls the `User.logout()` method which performs a POST request to the `/logout` API. On success, it just clear the session storage (line 36 on `User` object) and returns a resolved Promise. Finally, the login screen is shown. The logout flow is shown on figure 5.8.

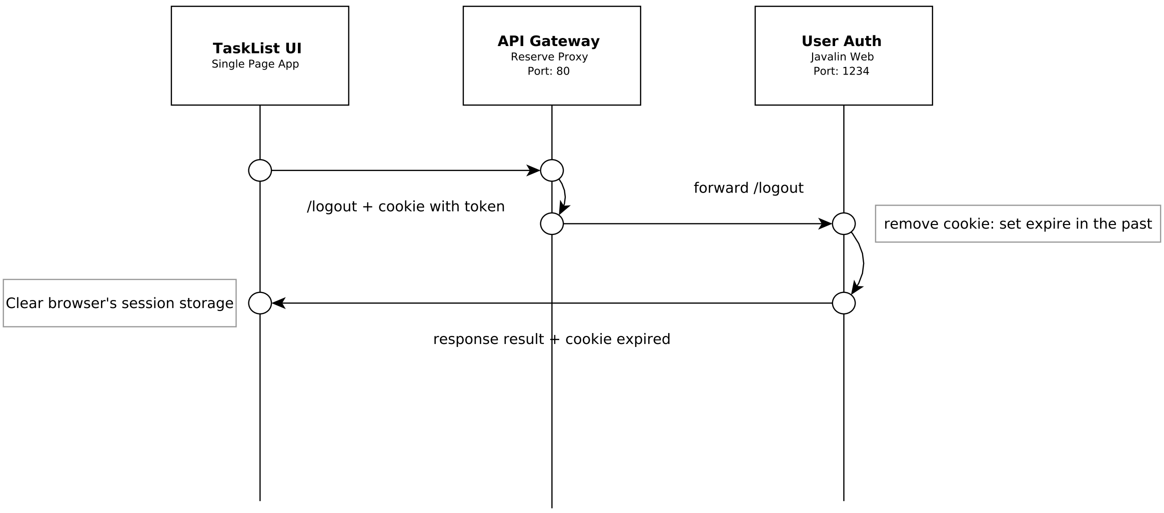


Figure 5.8: Logout Flow

Bibliography

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript

[2] https://reactjs.org/docs/getting-started.html

[3] https://www.ecma-international.org/

[4] https://nodejs.org/

[5] http://latentflip.com/loupe/

[6] https://material-ui.com/

[7] https://html5up.net/