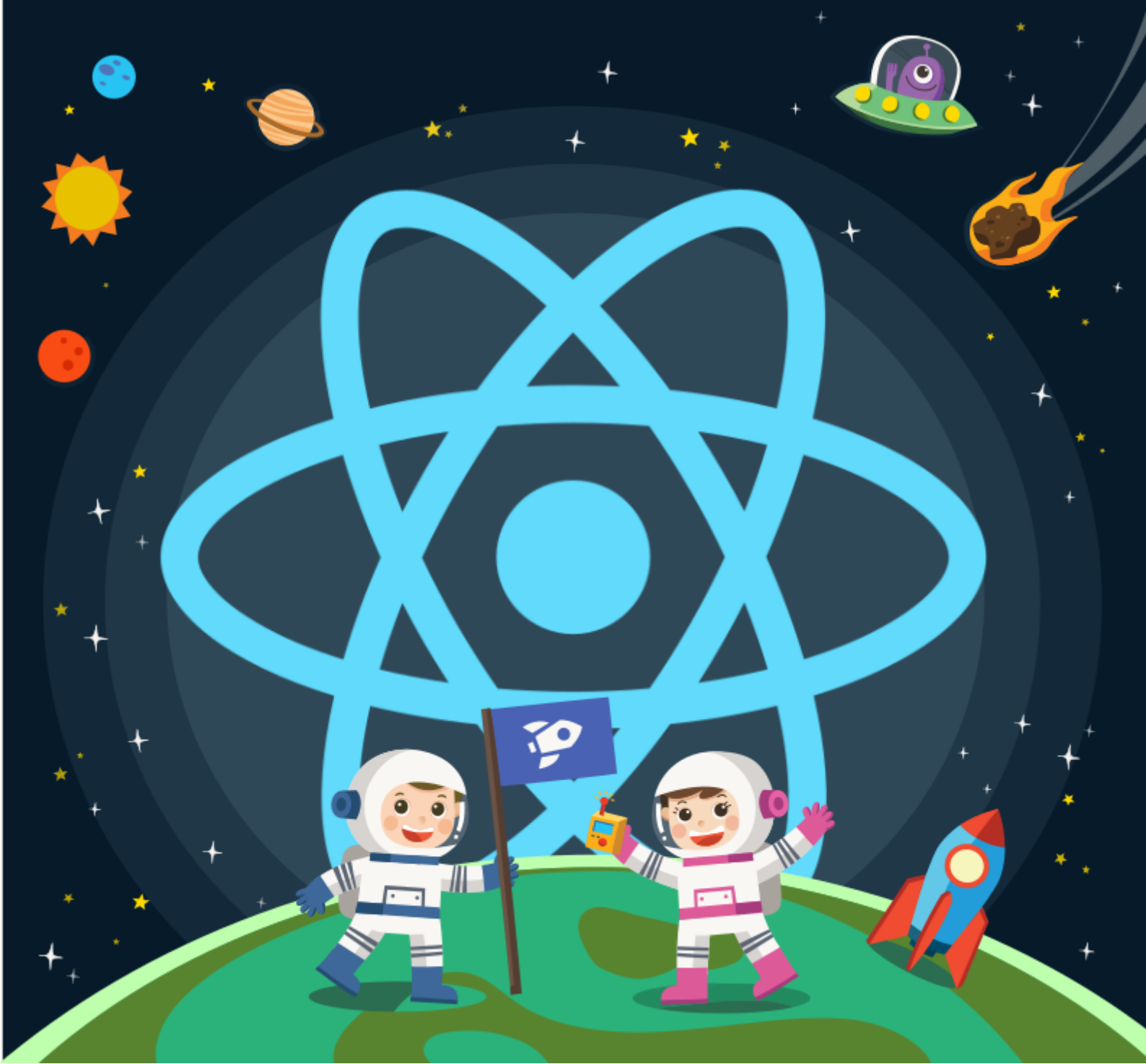


Nathan Sebastian

DIGESTING REACT

Learn how to build web apps with React



Digesting React

Learn how to build web applications with React

Nathan Sebastian

Copyright © 2020 Nathan Sebastian. All rights reserved.

While I did my best to take every precaution in the preparation of this book, I assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Please make your own best judgement, for things might change in the future after this publication

Contents

Foreword	7
Introduction to React	9
It's minimalist	9
It has small learning curve	10
It's unopinionated	10
Strong community support	10
Setting up your local computer environment	11
Installing VSCode	11
Installing Node.js	12
Installing Create React App	13
Installing React Developer Tools	15
Part 1: React Core Concepts	17
Introducing React components	18
Returning multiple elements	19
Rendering to the screen	21
Writing comments in React components	22
Your first exercise	23
JSX: The template language of React	25
Adding class attribute	26
Writing comments inside React JSX	26
Using JavaScript inside JSX	28
Assembling multiple components as one	32
Custom functions and this keyword	35
State and props	36
Passing down multiple props	37
Props are immutable	38
Binding a custom function	40
Using React DevTools to inspect state and props	41
Conclusion	43
React event handlers	44
Storing form input with onChange event handler	45
Preventing default event behavior	49
React component's lifecycle methods	52
The constructor function	54
render function	56

The componentDidMount function	56
The componentDidUpdate function	57
The componentWillUnmount function	60
Conclusion	62
Writing CSS for React Components	63
Inline styling	63
CSS stylesheets	66
CSS Modules	67
Styled components	70
Which one should you start with?	72
First Project: React Wizard Form	73
Building the app structure	78
Writing the functions for change and submit	86
Finishing child components	88
Wrapping up	91
Part 2: React function components and hooks	92
React function-based components	93
The useState hook	95
The useEffect hook	98
The componentWillUnmount effect	100
Running useEffect only once	101
The rules of hooks	102
Only call Hooks at the top level	102
Only call Hooks from function components	103
Conditional rendering with React	104
Partial rendering with a regular variable	104
Inline rendering with && operator	105
Inline rendering with conditional (ternary) operator	106
Second project: Finance Tracker	108
Composing the components	112
Implementing the application process	113
Transaction form rendering	114
Transaction form rendering	116
Writing the category form	119
Displaying categories in Header	121
Writing AddTransaction component	123

Listing transactions on TransactionTable component	127
Adding delete function to TransactionTable	131
Processing transactions with Chart.js	133
Activating category filter	138
Part 3: React component logics	142
Making AJAX calls in React	143
Make AJAX calls when the component has been mounted .	144
Using Axios in React	150
Using Axios in React	150
Requests example with Axios	152
Executing multiple concurrent requests	153
React Router	154
Making dynamic routing	156
Nested route	158
Passing props to Route component	162
React Router hooks	164
useParams hook	164
useLocation hook	167
useHistory hook	168
useRouteMatch hook	169
Should you use spread attributes when passing props?	172
Using propTypes and defaultProps	175
Passing data from child components to parent components	178
Third project: Github User Finder	181
The components composition	184
Writing the app structure	184
Writing the Header component	186
Writing the CSS	186
Writing the search page	190
Writing the SearchForm component	192
Writing the UsersList component	193
Writing the Profile component	196
Part 4: Context API	200
The Prop drilling pattern	201
The Context API	204
Providing context	205

Consuming context in function components	205
Consuming context in class components	207
Separating context from components	208
Fourth project: E-commerce App	210
Creating the layout for the application	213
Creating the Context API	216
Writing AddProduct component	218
Writing the Products component	222
Writing the Cart component	226
Persisting state into the Local Storage	229
Part 5: Closing	231
Afterword	232

Foreword

The year 2018 has been a great year for React. It wins the hearts of frontend developers and it also wins the job marketplace¹.

And React team itself doesn't seem to content with its position, with the new hooks feature being released as early as February 6th 2019. React is a revolutionary and very solid library for developing scalable and maintainable frontend, and it has proven track record from popular tech companies like AirBnB² and Netflix³.

(I don't need to mention Facebook⁴ now, do I?)

So yes, React is definitely at the top of the requirement list for many *nerd* jobs. And although learning React itself isn't that hard, the way to master React is complex and requires you to have a deep familiarity with its features.

For example, the hooks feature has been a great addition to React that opens up new possibilities of writing the code, but I think it also has made developers who want to learn React gets confused. Before hooks, using a JavaScript class is the only way to write a component that has business

¹[June 2018 Hacker News Hiring Trends](#)

²[Rearchitecting Airbnb's Frontend](#)

³[Netflix Likes React](#)

⁴[Does Facebook use React on Facebook.com?](#)

logic applied to it. Now with hooks, you can use both JavaScript class and function to write your component. Which one should you use?

Outside of React itself, the React ecosystem is filled with many libraries that strive to achieve the same goals, and they are all equally good. For example, you can use Redux to manage complex React state. But should you use Redux? What about the Context API? Wait, there's also Overmind and Recoil library. Which one should you use?

If you don't know what I'm talking about, then this book will be perfect for you. I will help you to learn both React and how to start navigating its huge ecosystem of tools and libraries in a pace that you can absorb.

To experience the full benefit of this guide, you need to have the following requirements:

- Know basic website technology like HTML and CSS
- You're familiar with basic JavaScript knowledge (variables, functions, conditionals)
- Familiar with code editors like VSCode and SublimeText

Finally, welcome to *Digesting React* where you will learn how to build web applications with React **without getting exhausted**.

Introduction to React

React is a very popular JavaScript front-end library that has received lots of love from developers around the world for its **simplicity** and **fast performance**.

React was initially developed by Facebook as a solution to front end problems they are facing:

- DOM manipulation is an expensive operation and should be minimized
- No library specialized in handling front-end library at the time (there is Angular, but it's an ENTIRE framework.)
- Using a lot of jQuery is causing spaghetti code

Why developers love React? As a software developer myself, I can think of a few reasons why I love it:

It's minimalist

React takes care of only ONE thing: the user interface and how it changes according to the data you feed into it. You can think of React as the “V” in an MVC framework.

It has small learning curve

The core concepts of React are easy to learn, and you don't need months or 40 hours of video lectures to learn about them.

It's unopinionated

React can be integrated with lots of different technologies. On the front-end, you can use different libraries to handle Ajax calls (Axios, Superagent, or just plain old Fetch.) On the back-end, You can use PHP/Rails/Go/Python or whatever language you prefer.

Strong community support

To enhance React's capabilities, open source contributors have build an amazing ecosystem of libraries that enables us to make even more powerful application. But most open source libraries for React is optional. You don't need to learn them until after you master React fundamentals.

The bottom line is that with a small learning curve, React gives you incredible power in making your UI **flexible, reusable** and **spaghetti-free**.

Setting up your local computer environment

This section will help you install all the necessary tools to start writing React application in your computer. While you can use browser-based code editor like [Code Sandbox](#) and [Codepen](#), I'd still recommend you to simply install things into your computer so that you won't be interrupted by bad internet connection or any downtime.

You will most likely use a local computer when working on a real project anyway, so let's do that now to make yourself comfortable. You can skip this section if you already have these tools installed:

- Code editor (I recommend VSCode)
- Node.js
- Create React App
- React Developer Tools

Installing VSCode

I assume you already know the benefits of using a code editor over a regular one, so I'm going to recommend you to install VSCode if you haven't. VSCode is a free code editor that's really popular with developers today. Head over to [VSCode website](#) and download the version for your system.

Installing Node.js

To be able to create React projects on your computer, you need to install Node.js. Head over to its website at nodejs.org and download the most recent LTS version for your computer:

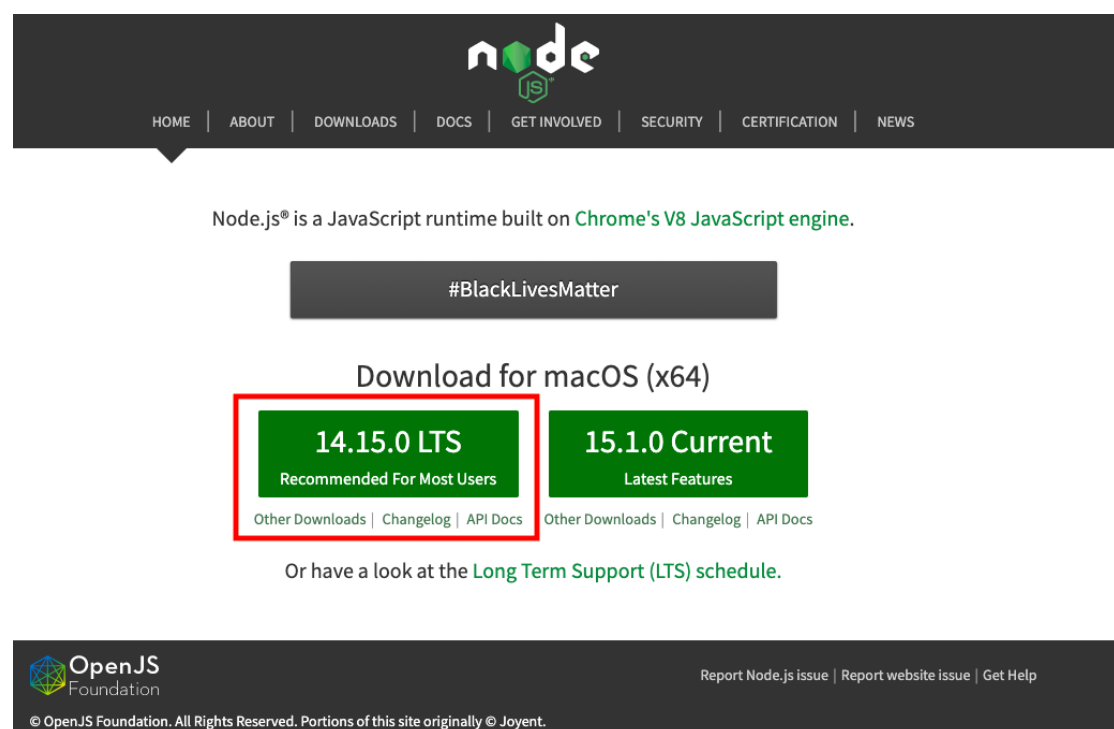


Figure 1: Node.js website

Once installed, please open your command line and type in the following command:

```
node --version && npm --version
```

You'll see the versions of Node and NPM that has been installed into your computer. Now you're ready to create React applications.

Installing Create React App

In order to make React run properly without any errors, it needs a lot of configurations. Yes, you need to configure Babel and Webpack at minimum so that React can run when you hit the browser. Yet configurations should not stand in the way of getting started, and that's why Facebook created a utility tool called [Create React App](#). It saves you from having to learn about configurations, and how to setup those configurations properly for React.

The only requirement to use this tool is that you have Node and NPM installed, so let's get one in your local computer by opening the Terminal and run the command:

```
npx create-react-app my-first-react-app
```

Once it's finished, navigate to the created directory and run the app using these commands:

```
cd my-first-react-app && npm start
```

A browser window will open and you'll be greeted with the index page of CRA apps:

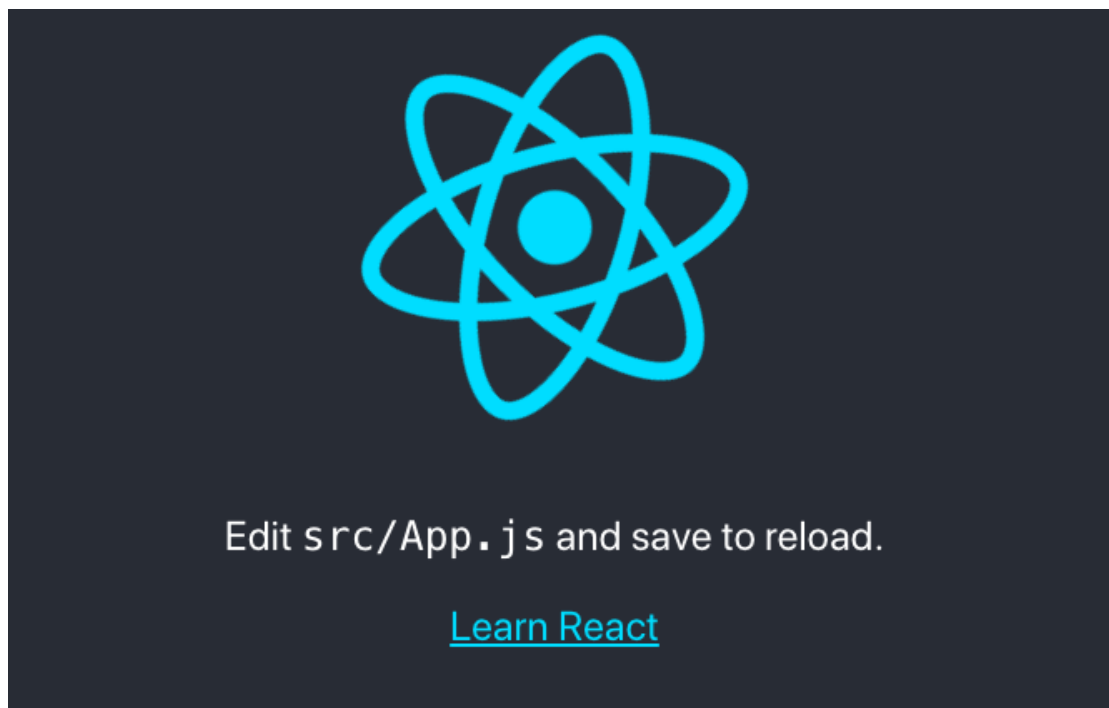


Figure 2: The index page of CRA apps

There's a lot going on under the hood of this app, but the most important thing for you to know is that the entire React app code can be found in `src/` directory, and it's hooked into the `public/index.html` file.

If you want to learn about the configuration that's been setup by Create React App, you can read about my article [Step by step React configuration from scratch](#). That's exactly what Create React App has saved you from.

Installing React Developer Tools

React has a Developer Tool that allows you to inspect your React code at runtime from the browser. This is very useful to see changes in your React code as you'll see in the following chapters. To use it, you only need to download the right package for your environment:

- [Chrome extension](#)
- [Firefox extension](#)
- [Standalone app \(Safari, React Native, etc\)](#)

Opera users can [enable Chrome extensions](#) and then install the [Chrome extension](#).

I recommend you use Chrome or Firefox extension, since they are easier to work with compared to the standalone version. To bring up the Developer Tool, simply open your browser developer tool and a new 'Components' and 'Profiler' tabs will appear on sites using React:

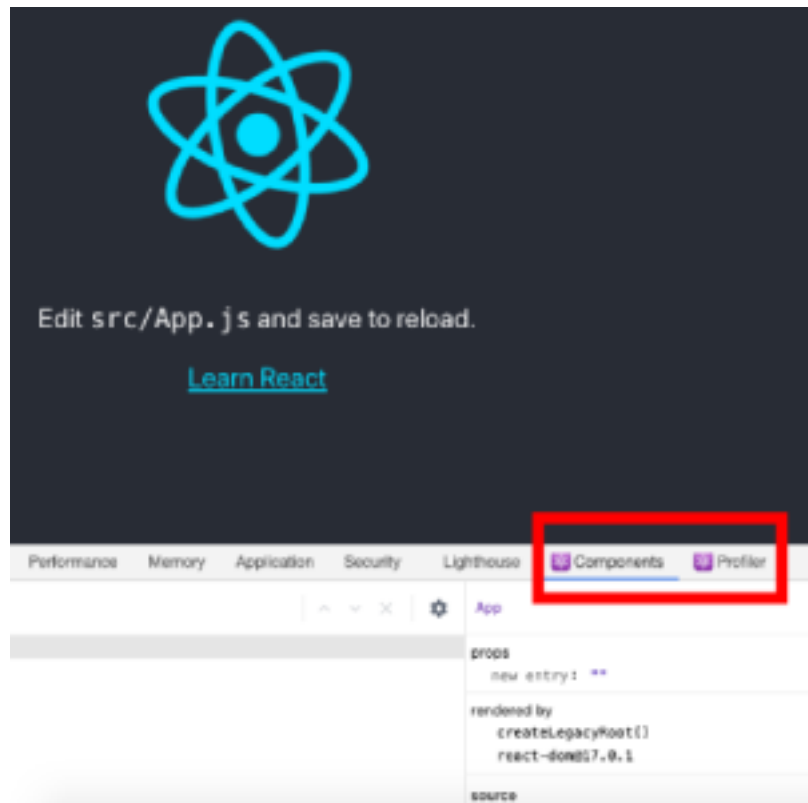


Figure 3: React dev tools in action

Similar to how you can inspect CSS in a regular HTML elements, you can inspect the state and props of React component using the developer tool. I will explain how you can use it later. Let's start learning React core concepts for now.

Part 1: React Core Concepts

This part will show you the building blocks of React applications. It will help you learn the idea of a **component** and how it can be used to split your application UI into small, manageable pieces.

You will also learn how your components know what to display on the screen based on the data you feed into it in the form of **props** and **state**.

To build a fully functional web application, you'll need more than one component, so you're going to learn how to **assemble multiple components** into a single application.

Next, you will learn about the **lifecycle** of a React class component, and how you can execute your code based on what lifecycle status your component is in.

Finally, you will learn how to **handle events** such as a button click with React. You will end part 1 with building a simple application that demonstrates how all these theories came together.

This part will focus on building components using JavaScript `class` syntax. You will learn about the `function` syntax in part two.

Introducing React components

A component in React is a single independent unit of a user interface.

What you write inside a component will determine what should appear on the browser screen at a given time. Each component in React must return at least one UI element in order to run properly.

You can create a React component using both JavaScript `class` and `function` syntax.

Here's how you declare a class component in React:

```
import React from 'react';

class MainComponent extends React.Component {
  render(){
    return <h1> Hello World </h1>
  }
}
```

A class component must always contain a `render()` function with a `return` statement or it will throw an error.

And here's how you declare a function component:

```
import React from 'react';

function MainComponent(){
  return <h1> Hello World </h1>
}
```

Just like class component, a function component must always have a `return` statement, but you don't need to create a `render()` function with it.

When you want a component to render nothing, you can return a `null` or `false` instead of an element.

```
import React from 'react';

function MainComponent(){
  return null
}

//or

class MainComponent extends React.Component {
  render(){
    return false
  }
}
```

Since React is a JavaScript library, all React component code goes under the `.js` file extension. You might find some public code use `.jsx` extension, but most compilers treat them as the same, so it's better to use `.js` extension because it's the universal JavaScript code format.

Returning multiple elements

A component must always return a single element. When you need to return multiple elements, you need to wrap all of it in a single element like a `<div>` :

```
import React from 'react';

class MainComponent extends React.Component {
  render(){
    return (
      <div>
        <h1>Hello World!</h1>
        <h2>Learning to code with React</h2>
      </div>
    )
  }
}
```

But this will make your application renders extra `<div>` nodes into the browser. To avoid cluttering your application, you can render an empty tag:

```
import React from 'react';

class MainComponent extends React.Component {
  render(){
    return (
      <>
        <h1>Hello World!</h1>
        <h2>Learning to code with React</h2>
      </>
    )
  }
}
```

With this, you won't render any extra nodes. You can do the same with React function components.

Rendering to the screen

A React component needs to be rendered into the screen by using the `ReactDOM.render()` function which takes two arguments: the component you want to render and the containing HTML element to render the component into. This means that you need to have an HTML file in your React project, or React won't know where to render the component. Usually, a very basic HTML document with a div is enough:

```
<div id="root"></div>
```

Next, you render the component into the `div`. Notice how ReactDOM is imported from `react-dom` package in the example below:

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

class MainComponent extends React.Component {
  render(){
    return <h1>Hello World</h1>
  }
}

ReactDOM.render(
  <MainComponent />,
  document.querySelector("#root")
)
```

The first argument is the `<MainComponent>` and the second argument is the `<div>` element with id “root”.

Here's a [render example in Code Sandbox](#) that you can tweak around with.

Writing comments in React components

Writing comments in React components can be done just like how you comment in regular JavaScript classes and functions. You can use the double forward-slash syntax `//` to comment any code:

```
class MainComponent extends React.Component {  
  render(){  
    // const name = "John"  
    // const age = 28  
    return <h1>Hello World</h1>  
  }  
}
```

Or you can use the forward-slash and asterisk format `/* */` like this:

```
class MainComponent extends React.Component {  
  render(){  
    /*  
    const name = "John"  
    const age = 28  
    */  
    return <h1>Hello World</h1>  
  }  
}
```

Generally, the forward-slash and asterisk format for comments is used for writing real comments like license and other documentation for developers to read instead of commenting out code:

```
/** @license React v16.13.1
 * react.development.js
 *
 * Copyright (c) Facebook, Inc. and its affiliates.
 *
 * This source code is licensed under the MIT license found in the
 * LICENSE file in the root directory of this source tree.
 */
```

But there is no strict rule that says you can't use it for commenting out code too.

Your first exercise

It's time to create your first React application. Open a terminal in your computer and create a new React application with Create React App.

Let's name this app `my-first-react-app` :

```
npm create-react-app my-first-react-app
```

Move into the folder and run the application with `npm start` :

```
cd my-first-react-app && npm start
```

Once the application is loaded into the browser, open your application with a code editor. Inside, you will see both `src/` and `public/` folders. The `src/` folder is where the code responsible for rendering your

application. Open the file `App.js` and you'll see the component that gets rendered by `index.js`.

Your assignment is very simple:

1. Change the text `Edit <code>src/App.js</code> and save to reload` into `Hello World from React!`
2. Remove the anchor element "Learn React" from the screen

You might see some unfamiliar syntax inside the `return` statement.

Don't worry about it, I will explain in the next chapter.

Once you make the changes, save the file and return to the browser. The script from Create React App will automatically refresh the browser and display the changes for you. Great job!

Next, you're going to learn about JSX, the template language of React.

JSX: The template language of React

In the previous chapter, you've learned that a component must always have a `return` statement that contains elements to render on the screen:

```
class MainComponent extends React.Component {  
  render(){  
    return <h1> Hello World </h1>  
  }  
}
```

The tag `<h1>` looks like a regular HTML tag, but it's actually a special template language included in React called JSX.

JSX is an extension of JavaScript that produces JavaScript powered React elements. It can be assigned to JavaScript variable and can be returned from function calls. For example:

```
class MainComponent extends React.Component {  
  render(){  
    const element = <h1> Hello World! </h1>  
    return element  
  }  
}
```

The example above is a valid JSX code.

Adding class attribute

Just like ordinary HTML, JSX can use `class` attribute with the `className` keyword (because the keyword `class` is reserved by JavaScript.)

```
const App = <h1 className='text-lowercase'>Hello World!</h1>;
```

Writing comments inside React JSX

Commenting inside React JSX syntax is a bit confusing because while JSX gets rendered just like normal HTML by the browser, JSX is actually an enhanced JavaScript that allows you to write HTML in React.

You can't write comments as you might do in HTML and XML with

`<!-- -->` syntax. The following example will throw `Unexpected token` error:

```
export default function App() {  
  return (  
    <div>  
      <h1>Hello World~ </h1>  
      <!-- <p>My name is Bob</p> -->  
      <p>Nice to meet you!</p>  
    </div>  
  );  
}
```

To write comments in JSX, you need to use JavaScript's forward-slash and asterisk syntax, enclosed inside a curly brace `{/* comment */}` .

Here's an example:

```
export default function App() {  
  return (  
    <div>  
      <h1>Commenting in React and JSX~ </h1>  
      { /* <p>My name is Bob</p> */ }  
      <p>Nice to meet you!</p>  
    </div>  
  );  
}
```

And here's for multiple lines of JSX:

```
export default function App() {  
  return (  
    <div>  
      { /* <h1>Commenting in React and JSX~ </h1>  
        <p>My name is Bob</p>  
        <p>Nice to meet you!</p> */ }  
    </div>  
  );  
}
```

It may seem very annoying that you need to remember two different ways of commenting when writing React code. But don't worry!

Most modern IDEs like [VSCode](#) and [CodeSandbox](#) already know about this issue. They will write the right comment syntax for you automatically when you press on the comment shortcut `CTRL+ /` or `command+ /` for macOS.

Using JavaScript inside JSX

You can embed JavaScript expression inside the element by using curly brackets `{}` :

```
const lowercaseClass = 'text-lowercase';
const text = 'Hello World!';
const App = <h1 className={lowercaseClass}>{text}</h1>;
```

This is what makes React element distinct from HTML element. You can't embed JavaScript directly by using curly braces in HTML.

Instead of creating a whole new templating language, you just need to use JavaScript functions to control what is being displayed on the screen.

For example, let's say you have an array of users that you'd like to show:

```
const users = [
  { id: 1, name: 'Nathan', role: 'Web Developer' },
  { id: 2, name: 'John', role: 'Web Designer' },
  { id: 3, name: 'Jane', role: 'Team Leader' },
]
```

You can use the `map()` function to loop over the array:

```
import React from "react"

function App() {
  const users = [
    { id: 1, name: 'Nathan', role: 'Web Developer' },
    { id: 2, name: 'John', role: 'Web Designer' },
    { id: 3, name: 'Jane', role: 'Team Leader' },
  ]
```

```
return (
  <>
    <p>The currently active users list:</p>
    <ul>
      {
        users.map(function(user){
          // returns Nathan, then John, then Jane
          return (
            <li> {user.name} as the {user.role} </li>
          )
        })
      }
    </ul>
  </>
)
```

Inside React, you don't need to store the return value of the `map` function in a variable. The example above will return a list element for each array value into the component.

While the above code is already complete, React will trigger a warning that you need to put “key” prop in each child of a list (the element that you return from `map` function):

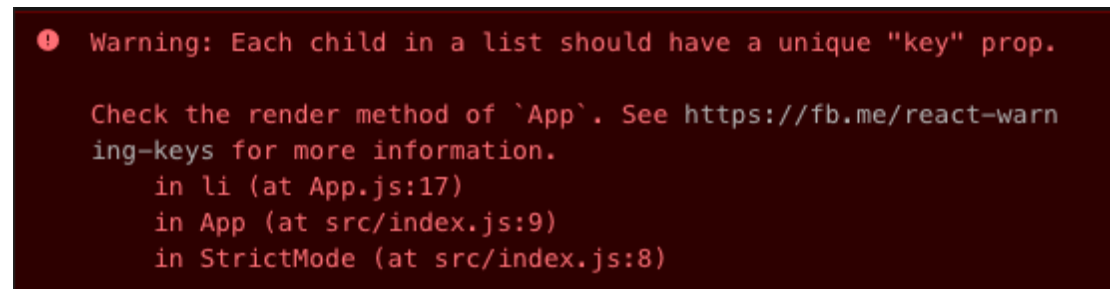


Figure 4: React needs a key inside each child

The “key” prop is a special prop that React will use to determine which child element have been changed, added, or removed from the list. you won’t use it actively in any part of your array rendering code, but since React needs the props, then let’s give it.

It is recommended that you put the unique identifier of your data as the key value. In the example above, you can put the `id` of each user as the key of each `` element:

```
return (  
  <li key={user.id}>  
    {user.name} as the {user.role}  
  </li>  
)
```

When you don’t have any unique identifiers for your list, you may use the array index as the last resort:

```
{
  users.map(function(user, index){
    return (
      <li key={index}>
        {user.name} as the {user.role}
      </li>
    )
  })
}
```

Now that you know how flexible and powerful JSX is, let's learn about composing components next.

Assembling multiple components as one

Up until this point, you've only rendered a single component into the browser. But applications build using React can comprise of tens and hundreds of components. **Composing or assembling components** is the process of forming the user interface by using loosely coupled components in a top-down approach.

It's kind of like making a robot out of lego blocks, as I will show you in the following demo:

```
class ParentComponent extends React.Component {
  render(){
    return (
      <>
        <UserComponent />
        <ProfileComponent />
        <FeedComponent />
      </>
    )
  }
}

class UserComponent extends React.Component {
  render(){
    return <h1> User Component </h1>
  }
}

class ProfileComponent extends React.Component {
  render(){
    return <h1> Profile Component </h1>
  }
}

class FeedComponent extends React.Component {
  render(){
```



```
    return <h1> Feed Component</h1>
  }
}
```

From the example above, you can see how the `<ParentComponent>` renders three children components:

- `<UserComponent>`
- `<ProfileComponent>`
- `<FeedComponent>`

Both class components and function components can compose components this way. You can even render a class component inside a function component and vice versa.

The composition of many components will form a single tree of React components, and a single component will be rendered into the DOM using `ReactDOM.render()` :

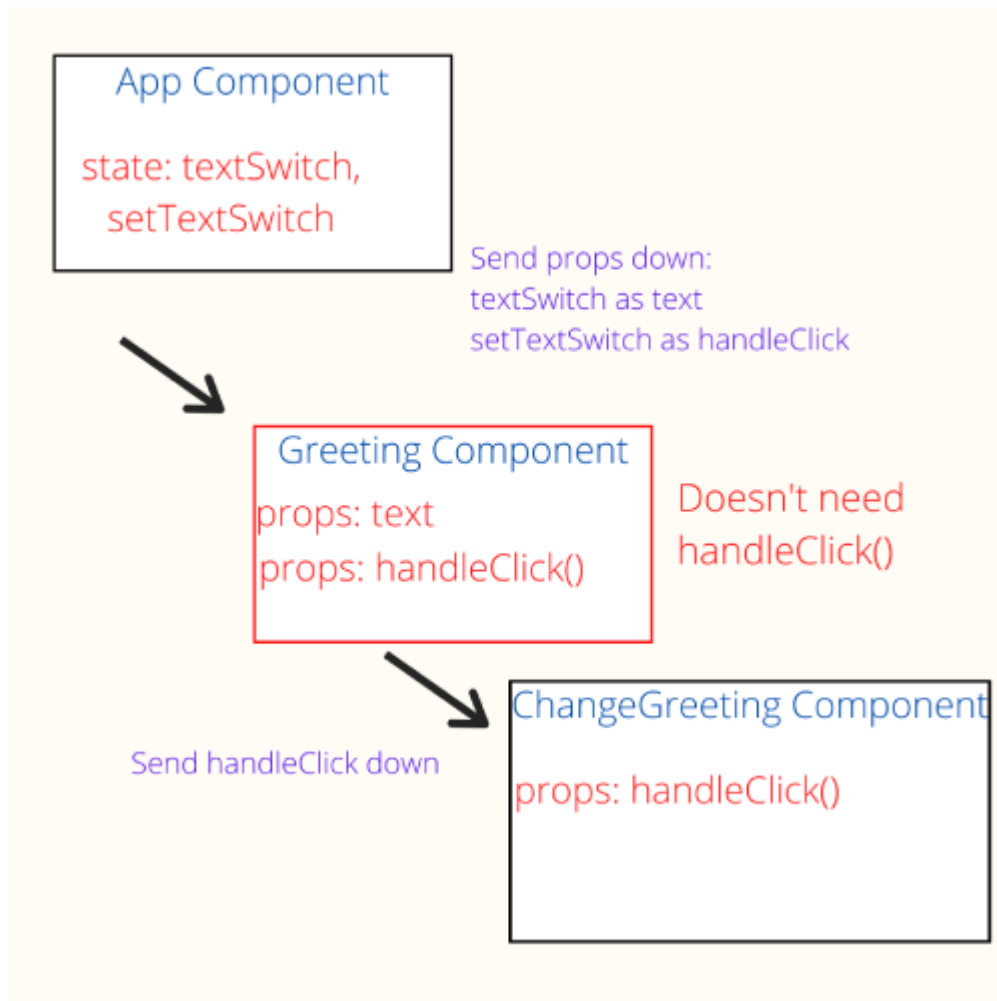


Figure 5: React tree of components

By composing multiple components, you can split the user interface into independent, reusable pieces, and develop each piece in isolation.

Custom functions and this keyword

You can create custom functions to run inside a component by declaring the function name without the `function` keyword. Here's an example:

```
import React from 'react';

class MainComponent extends React.Component {

  formatName(firstName, lastName) {
    return firstName + ' ' + lastName;
  }

  render(){
    const firstName = "Bruce"
    const lastName = "The Batman"
    return (
      <>
        <h1>Hello World!</h1>
        <h2>I'm {this.formatName(firstName, lastName)}</h2>
      </>
    )
  }
}
```

In the example above, the function `formatName()` is a custom function that gets called by the `render()` function using the `this.formatName()` syntax. The `this` keyword refers to the JavaScript class `MainComponent` which owns the function `formatName()`.

In the following chapters, you will find that React class components will use `this` keyword to access properties owned by the class.

State and props

In React library, props and states are both means to make a component more dynamic. Props (or properties) are inputs passed down from a parent component to its child component. On the other hand, states are variables defined and managed by the component.

For example, let's say we have a that calls a :

```
class ParentComponent extends React.Component {  
  return <ChildComponent />  
}
```

You can pass a prop from ParentComponent into ChildComponent by adding new attributes after the component name. For example, the `name` prop with value `John` is passed to ChildComponent below:

```
class ParentComponent extends React.Component {  
  return <ChildComponent name="John" />  
}
```

After that, the child component will accept assign the props into its own `this.props` object. What to do with the prop being passed down to the child is of no concern to the parent component. You can simply output the `name` prop like this:

```
class ChildComponent extends React.Component {  
  render(){  
    return <p> Hello World! my name is {this.props.name}</p>  
  }  
}
```

Passing down multiple props

You can pass as many props as you want into a single child component, just write the props next to the previous. Here's an example:

```
class ParentComponent extends React.Component {  
  render(){  
    return (  
      <ChildComponent  
        name="John"  
        Age={29}  
        isMale={true}  
        hobbies=["read books", "drink coffee"]  
        occupation="Software Engineer"  
      />  
    );  
  }  
}
```

It will all be passed accordingly into the ChildComponent's `this.props` object.

Also please remember: you need to **pass either a string or a**

JavaScript expression inside curly brackets as the value of props.

This is because your component call is in JSX syntax, and you need to use curly brackets to write an expression.

Props are immutable

Meaning that a prop's value can't be changed no matter what happens.

The following example will still output the value passed into it instead of "Mark":

```
class ChildComponent extends React.Component {  
  render(){  
    this.props.name = "Mark";  
    return <p> Hello World! my name is {this.props.name}</p>  
  }  
}
```

But what if you need variables that might change later? This is where state comes in. States are arbitrary data that you can define, but they are initialized and managed by a component. Here's how you define a class component state:

```
class ParentComponent extends React.Component {  
  state = {  
    name: "John"  
  }  
}
```

The state initialized in the class can be accessed from `this.state` object.

Please note that you mustn't change the state value by reassigning the variable. The following code is considered wrong even though it works:

```
class ParentComponent extends React.Component {
  state = {
    name: "John"
  }

  render() {
    this.state.name = "Mark"
    return <div>{this.state.name}</div>;
  }
}
```

You need to use the `setState` function which is inherited from React's `Component` class. You need to pass in an object just like when you declare the initial state:

```
class ParentComponent extends React.Component {
  state = {
    name: ""
  };

  render() {
    if (this.state.name === "") {
      this.setState({ name: "Mark" });
    }
    return <div>{this.state.name}</div>;
  }
}
```

Also, **never** call on the `setState` without a condition since it will cause an infinite loop. The following code will throw an error:

```
class ParentComponent extends React.Component {
  state = {
    name: ""
  };
}
```

```
render() {  
  this.setState({ name: "Mark" });  
  return <div>{this.state.name}</div>;  
}
```

Binding a custom function

You can pass state into any children component, but if you want to update the state from a child component, you need to create a custom function in the parent component and pass it down to the child component:

```
class ParentComponent extends React.Component {  
  state = {  
    name: "John"  
  };  
  
  setName(name) {  
    this.setState({ name: name });  
  }  
  
  render() {  
    return (  
      <ChildComponent  
        name={this.state.name}  
        setName={this.setName.bind(this)}  
      />  
    );  
  }  
}
```

The `bind` keyword used inside the `setName` props is needed because when you pass the function to other components, the function

will be called in a different context (in `ChildComponent` instead of `ParentComponent`)

When a child component needs the state to change, you can do so by calling on the `setName` function. In the following example, I put a button to change the value of `name` state when it gets clicked:

```
class ChildComponent extends React.Component {
  render() {
    return (
      <>
        <p> Hello World! my name is {this.props.name}</p>
        <button
          onClick={() => this.props.setName("Mark")}>
          Change name
        </button>
      </>
    );
  }
}
```

Notice how the `<button>` above has an `onClick` event attribute. You will explore that in the next chapter.

Using React DevTools to inspect state and props

To help ease your development, you can use React DevTools that you've installed previously to inspect the current state and props value of your components. Head over into the *components* tab and click on one of the

components. Here's an example of `ChildComponent` detail in the inspector:

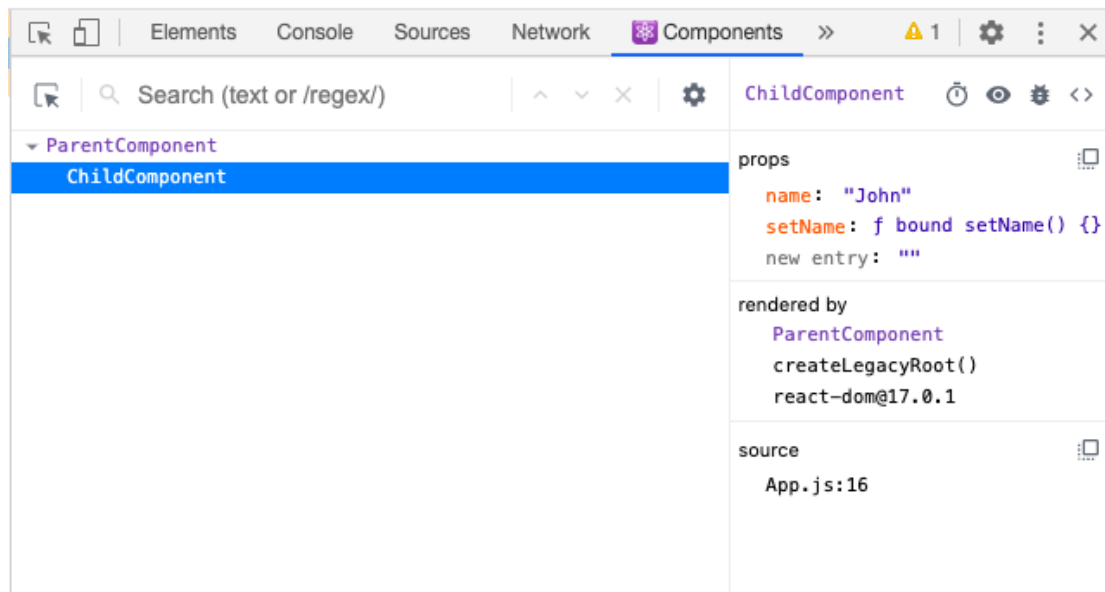


Figure 6: DevTools component inspector

When you click on the button, the `name` prop value will change accordingly. You can inspect the `ParentComponent` to view the value of state in it. This will be useful as you code your application in the exercise projects later.

Conclusion

You've just learned the difference between props and state in React. Both features are simply arbitrary variables that you can use to make your React components more dynamic. In most cases, states are initialized at top-level components and passed down as props into children components.

When you need to change the prop value, the child component can send a signal — usually function call when a button is clicked or something similar — to the parent component to change the state. This will make the props value being passed down from the parent component to child component changes.

The components will then render the user interface accordingly.

React event handlers

React has an internal event system that gets triggered every time a certain action is taken by the user. For example, an event can be triggered when you click on a button with the `onClick` prop:

```
class LogButton extends React.Component {
  handleClick = (event) => {
    console.log("Hello World!");
    console.log(event);
  };

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

When you click on the button above, the `event` variable will be logged as a `SyntheticBaseEvent` object in your console:

```
► SyntheticBaseEvent {_reactName: "onClick", _targetInst: null, type: "click", nativeEvent: MouseEvent, target: HTMLButtonElement...}
```

Figure 7: React's SyntheticBaseEvent log

The `event` argument passed into `handleClick` is React's own Synthetic event. It will always get send into your event handler function.

React's Synthetic events are basically wrappers around the [native DOM events](#). They are helper functions created to make sure the events have consistent properties across different browsers.

These specific events are baked into React library to help you in creating the proper responses to a user's actions. General use cases where you need to make use of these event handlers include listening to user inputs and storing form input data in React's state.

Storing form input with onChange event handler

The `onChange` event handler is a prop that you can pass into JSX's input elements. In React, `onChange` is used to handle user input in real-time.

If you want to build a form in React, you need to use this event to track the value of input elements.

Here's how you add the `onChange` handler to an input:

```
import React from "react"

class App extends React.Component {
  render(){
    return (
      <input
```

```
        type="text"
        name="firstName"
        onChange={ (event) => console.log("onChange is triggered") } />
    )
  }
}
```

Now whenever you type something into the input box, React will trigger the function that we passed into the `onChange` prop.

In regular HTML, form elements such as `<input>` and `<textarea>` usually maintain their own value:

```
<input id="name" type="text">
```

Which you can retrieve by using the `document` selector:

```
var name = document.getElementById("name").value;
```

In React however, it is encouraged for developers to store input values in the component's state object. This way, React component that renders the form elements will also control what happens on subsequent user inputs. First, you create a state for the input:

```
import React from "react"

class App extends React.Component {
  state = {
    name: ''
  }
}
```

Then, you create an input element and call the `setState` function to update the `name` state. Every time the `onChange` event is triggered, React will pass the `event` argument into the function that you define inside the prop:

```
import React from "react"

class App extends React.Component {
  state = {
    name: ''
  }

  render(){
    return (
      <input
        type="text"
        name="firstName"
        onChange={ (event) => this.setState({ name: event.target.value}) } />
    )
  }
}
```

Finally, you use the value of `name` state and put it inside the input's `value` prop:

```
return (
  <input
    type="text"
    name="firstName"
    onChange={
      (event) => this.setState({ name: event.target.value})
    }
    value={this.state.name} />
)
```

You can retrieve input value in `event.target.value` and input name

in `event.target.name` . You can also separate the `onChange` handler into its own function.

The `event` object is commonly shortened as `e`

```
import React, { useState } from "react"

class App extends React.Component {
  state = {
    name: ''
  }

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

  render(){
    return (
      <input
        type="text"
        name="firstName"
        onChange={ this.handleChange.bind(this) }
        value={this.state.name} />
    )
  }
}
```

The `bind` keyword is used when you call on the `handleChange` function so that you can call on `this.setState` inside it (`this` will still refers to `App` class instead of `handleChange` function)

This pattern of using React's `onChange` event and the component state will encourage developers to use state as the “single source of truth”. Instead of using Regular JavaScript to retrieve input values, you retrieve them from the state.

Preventing default event behavior

Since Synthetic events are just wrappers, the internal default behavior of the DOM object will still be triggered. One problem with the native DOM events is that it sometimes triggers a behavior that you don't need.

For example, a form's submit button in React will always trigger a browser refresh to submit the data into a backend system. This is bad because the behavior you defined in the `onSubmit` event function will be ignored by the browser. Try the following example:

```
import React from "react";

class App extends React.Component {
  state = {
    name: ''
  }

  handleSubmit(event) {
    console.log(this.state.name);
  };

  render(){
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input
            type="text"
            value={this.state.name}
            onChange={
              (event) => this.setState({name: event.target.value})
            }
          />
        </label>
        <input type="submit" value="Submit" />
      </form>
    )
  }
}
```

```
    );  
  }  
}
```

Because of the default DOM event behavior, the `handleSubmit` function will be ignored and your log will not get written on the console.

This may be good in the past where the entire form validation and processing happens in the backend, but modern web applications tend to run the form validation process on the client-side in order to save time and bandwidth. To do so, you need to run your own defined behavior.

To cancel the native behavior of the submit button, you need to use React's `event.preventDefault()` function:

```
handleSubmit(event) {  
  event.preventDefault();  
  console.log(name);  
};
```

And that's all you need. Now the default event behavior will be canceled, and any code you write inside `handleSubmit` will be run by the browser.

You can also write the `preventDefault()` function on the last line of your function:

```
handleSubmit(event) {  
  console.log(name);  
  console.log("Thanks for submitting the form!");  
  event.preventDefault();  
};
```

But for collaboration and debugging purposes, it's always better to write the prevent function **just below your function declaration**. That way you won't cause a bug by forgetting to put the prevent function too.

React component's lifecycle methods

All React components must have a `render` method, which returns some element that will be inserted into the DOM. Indeed, `ReactDOM.render` is called on a pure HTML element, which in most of our example so far, use the `<div>` tag with id `root` as its entry point.

That's why when we do this:

```
class sampleComponent extends React.Component {
  render() {
    return (
      <h1>Hello World!</h1>
    );
  }
}

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

The `<h1>` element will be added into the DOM element with id `root` :

```
<div id='root'>
  <h1>Hello World</h1>
</div>
```

Even though you can't see it in the browser, there's a fraction of time before React component `render` or insert this `<h1>` element into the

browser and after it, and in that small fraction of time, you can run special functions designed to exploit that time.

This is what lifecycle functions in a React component do: it executes at a certain time *before* or *after* a component is rendered to the browser.

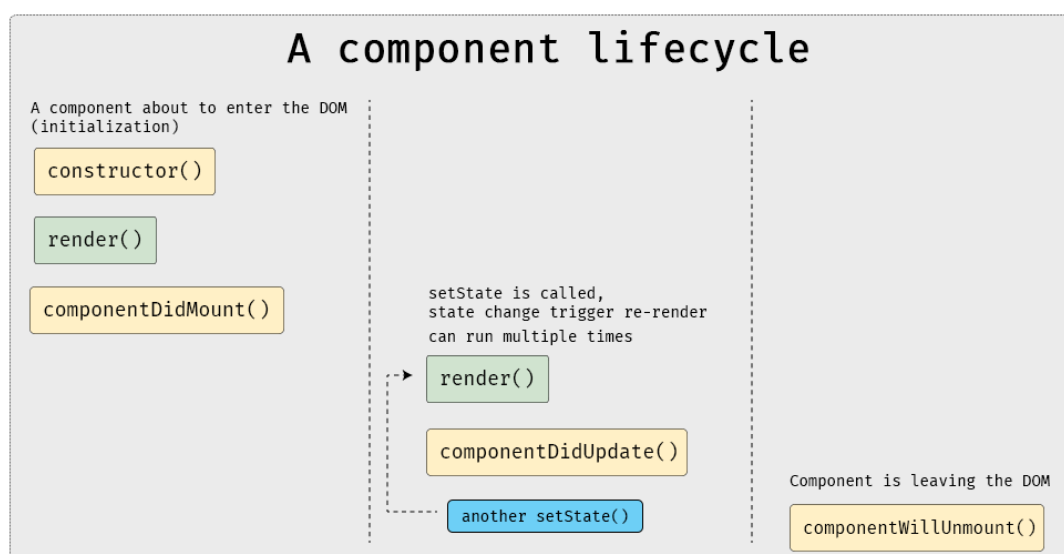


Figure 8: The lifecycle graph

When a component is first inserted into the DOM (or the `root` element), it will run the `constructor` method. At this point, nothing is happening in the browser.

Then React will run the component `render` method, inserting the JSX

you write into the DOM. After `render` is finished, it will immediately run the `componentDidMount` function.

When you call on `setState`, the `render` function will be called again after state is changed, with `componentDidUpdate` function immediately run after it.

`componentWillUnmount` function will run before the component rendered element is removed from the DOM.

The theory might seem complex, but as you will see in the following chapters, lifecycle functions are situational code, and they are used only for specific use cases.

The constructor function

The `constructor` function is run on the initialization of a React component. It is widely used as the place where state is initialized:

```
class sampleComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      number : 0
    }
  }
}
```

The function `super` will call on the parent `constructor` (specifically, the `React.Component` `constructor`) so that you can call on `this`:

```
class sampleComponent extends React.Component {  
  constructor(props) {  
    // this will cause error  
    this.state = {  
      number : 0  
    }  
    super(props);  
  }  
}
```

The `props` are being passed into `super` so that you can call on `this.props` on the constructor. If you're not using `props` in the constructor at all, you can omit it.

You might notice that on the previous chapters, you can also initiate state outside of the constructor:

```
class sampleComponent extends React.Component {  
  state = {  
    number: 0  
  }  
}
```

Both are valid state declaration, but the constructor style is widely adopted as the conventional style to class components, so you will find most React code use it.

The bottom line for `constructor` function — initialize your state there.

render function

You have seen this function in previous chapters, so it must be familiar for you. The `render` function is used to write the actual JSX elements, which is returned to React and hooked into the DOM tree.

Before returning JSX, you can write regular JavaScript syntax for operation such as getting state value, and embed it into the JSX:

```
render() {  
  const { name, role } = this.state;  
  return (  
    <div>My name is {name} and I'm a {role}</div>  
  )  
}
```

The componentDidMount function

The most common use of `componentDidMount` function is to load data from backend services or API. Because `componentDidMount` is called after render is finished, it ensures that whatever component manipulation you do next, like `setState` from fetched data, will actually update state from its initial value.

A data request to backend services might resolve faster than the component is inserted into the DOM, and if it did, you will do a `setState` faster than the `render` method finished. That will cause React to give

you a warning. The most common use of `componentDidMount` looks like this:

```
class sampleComponent extends React.Component {  
  
  componentDidMount() {  
    this.fetchData().then(response => {  
      this.setState({  
        data: response.data  
      });  
    });  
  }  
  
  fetchData = () => {  
    // do a fetch here and return something  
  }  
}
```

But `componentDidMount` is limited to running only once in a component lifecycle. To address this limit, let's learn about the next lifecycle function.

The `componentDidUpdate` function

Since `componentDidMount` is run only once in a component lifetime, it can't be used to fetch data in response to state change. Enter `componentDidUpdate` function. This function is always run in response to changes in the component, remember the diagram again:

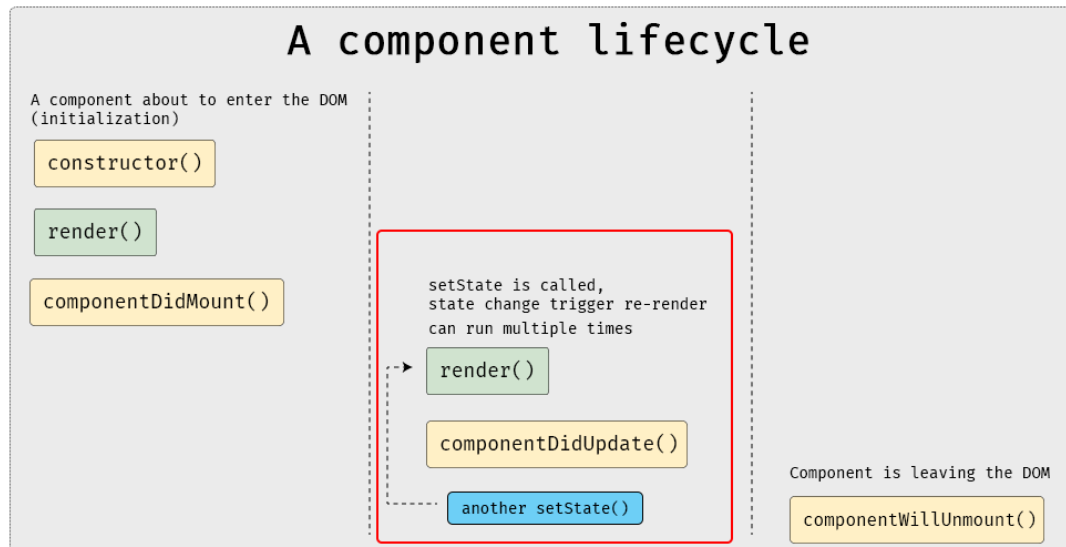


Figure 9: componentDidUpdate graph

An easy example would be to log the new state after a re-render.

```
class SampleDidUpdate extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      number: 0
    };
  }

  incrementState = () => {
    const { number } = this.state;
    this.setState({
      number: number + 1
    });
  };

  decrementState = () => {
```

```
    const { number } = this.state;
    this.setState({
      number: number - 1
    });
  };

  componentDidMount() {
    const { number } = this.state;
    console.log(`The current number is ${number}`);
  }

  componentDidUpdate() {
    const { number } = this.state;
    console.log(`The current number is ${number}`);
  }

  render() {
    const { number } = this.state;
    return (
      <>
        <div> The current number is {number}</div>
        <button onClick={this.incrementState}>Add number</button>
        <button onClick={this.decrementState}>Subtract number</button>
      </>
    );
  }
}
```

A demo is available [here](#). Notice how `didMount` and `didUpdate` is identical in everything but name. Since user can change the keyword after the component did mount into the DOM, subsequent request won't be run by `componentDidMount` function. Instead, `componentDidUpdate` will “react” in response to the changes after `render` function is finished.

The `componentWillUnmount` function

The final function `componentWillUnmount` will run when the component is about to be removed from the DOM. This is used to cleanup things that would be left behind by the component.

To try out this function, let's create two child component and one parent component.

```
class ChildComponentOne extends React.Component {
  componentWillUnmount() {
    console.log("Component One will be removed");
  }

  render() {
    return <div>Component One</div>;
  }
}

class ChildComponentTwo extends React.Component {
  componentWillUnmount() {
    console.log("Component Two will be removed");
  }

  render() {
    return <div>Component Two</div>;
  }
}
```

This child components will do a simple `div` render with `componentWillUnmount` function that logs a text into the console. Then the parent component will render one of them based on the current state it's in.

```
class ParentComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      number: 0
    };
  }

  switchState = () => {
    const { number } = this.state;
    this.setState({
      number: number === 0 ? 1 : 0
    });
  };

  render() {
    const { number } = this.state;
    let component = number ? <ChildComponentOne /> : <ChildComponentTwo />;
    return (
      <>
        {component}
        <button onClick={this.switchState}>Switch</button>
      </>
    );
  }
}
```

When you click on the Switch button, the component that will be removed from the DOM will log a message, then leave and be replaced with the new component. You can try it [here](#).

When to use it? It's actually very situational, and the best use of `componentWillUnmount` is to shut down some external service listener your component is subscribed into.

Conclusion

React's lifecycle methods are used for running codes that needs to be automatically run when the component is created, added, and removed from the DOM.

The lifecycle methods bring more control over what happens at each specific time during your component lifetime, from its creation to its destruction, allowing you to create dynamic applications in the process.

Writing CSS for React Components

There are four different ways to write CSS for React components. Let's learn what they are and which one you should start with.

Inline styling

React components are composed of JSX elements. But just because you're not writing regular HTML elements doesn't mean you can't use the old inline style method.

The only difference with JSX is that inline styles must be written as an object instead of a string.

Here is a simple example:

```
function App() {  
  return (  
    <h1 style={{ color: "red" }}>Hello World</h1>  
  );  
}
```

In the style attribute above, the first set of curly brackets will tell your JSX parser that the content between the brackets is JavaScript (and not a string). The second set of curly bracket will initialize a JavaScript object.

Style property names that have more than one word are written in camel-Case instead of using the traditional hyphenated style. For example, the usual `text-align` property must be written as `textAlign` in JSX:

```
function App() {  
  return (  
    <h1 style={{ textAlign: "center" }}>Hello World</h1>  
  );  
}
```

Because the style attribute is an object, you can also separate the style by writing it as a constant. This way, you can reuse it on other elements as needed:

```
const pStyle = {  
  fontSize: '16px',  
  color: 'blue'  
}  
  
export default function App() {  
  return (  
    <p style={pStyle}>The weather is sunny today.</p>  
  );  
}
```

If you need to extend your paragraph style further down the line, you can use the object spread operator. This will let you add inline styles to your already-declared style object:


```
const pStyle = {
  fontSize: "16px",
  color: "blue"
};

export default function App() {
  return (
    <>
      <p style={pStyle}>
        The weather has a small chance of rain today.
      </p>
      <p style={{ ...pStyle, color: "green", textAlign: "right" }}>
        When you go to work, bring your umbrella with you!
      </p>
    </>
  );
}
```

Inline styles are the most basic example of a CSS in JS styling technique.

One of the benefits in using the inline style approach is that you will have a simple component-focused styling technique. By using an object for styling, you can extend your style by spreading the object. Then you can add more style properties to it if you want.

But in a big and complex project where you have a hundreds of React components to manage, this might not be the best choice for you.

You can't specify pseudo-classes using inline styles. That means `:hover`, `:focus`, `:active`, or `:visited` go out the window rather than the component.

Also, you can't specify media queries for responsive styling. Let's consider another way to style your React app.

CSS stylesheets

When you build a React application using Create React App, you will automatically use webpack to handle asset importing and processing.

The great thing about webpack is that, since it handles your assets, you can also use the JavaScript `import` syntax to import a `.css` file to your JavaScript file. Just create a regular CSS file in your project folder:

```
/* style.css */
.paragraph-text {
  font-size: 16px;
  color: 'blue';
}
```

And you can import the CSS file and use the class name in JSX elements that you want to style, like this:

```
import React, { Component } from 'react';
import './style.css';

function App() {
  return (
    <>
      <p className="paragraph-text">
        The weather is sunny today.
      </p>
    </>
  );
}
```

This way, the CSS will be separated from your JavaScript files, and you can just write CSS syntax just as usual.

You can even include a CSS framework such as [Bootstrap](#) in your React app using this approach. All you need to is import the CSS file into your root component.

This method will enable you to use all of the CSS features, including pseudo-classes and media queries. But the drawback of using a stylesheet is that your style won't be localized to your component.

All CSS selectors have the same global scope. This means one selector can have unwanted side effects, and break other visual elements of your app.

Just like inline styles, using stylesheets still leaves you with the problem of maintaining and updating CSS in a big project.

CSS Modules

A [CSS module](#) is a regular CSS file with all of its class and animation names scoped locally by default.

Each React component will have its own CSS file, and you need to import the required CSS files into your component.

In order to let React know you're using CSS modules, name your CSS file using the `[name].module.css` convention.

Here's an example:

```
/* App.module.css */
.BlueParagraph {
  color: blue;
  text-align: left;
}
.GreenParagraph {
  color: green;
  text-align: right;
}
```

Then import it to your component file:

```
import React from "react";
import styles from "./App.module.css";

function App() {
  return (
    <>
      <p className={styles.BlueParagraph}>
        The weather is sunny today.
      </p>
      <p className={styles.GreenParagraph}>
        Still, don't forget to bring your umbrella!
      </p>
    </>
  )
}
```

When you build your app, webpack will automatically look for CSS files that have the `.module.css` name. Webpack will take those class names and map them to a new, generated localized name.

Here is [the sandbox](#) for the above example. If you inspect the blue paragraph, you'll see that the element class is transformed into

`_src_App_module__BlueParagraph` .

CSS Modules ensures that your CSS syntax is scoped locally.

Another advantage of using CSS Modules is that you can compose a new class by inheriting from other classes that you've written. This way, you'll be able to reuse CSS code that you've written previously, like this:

```
.MediumParagraph {  
  font-size: 20px;  
}  
.BlueParagraph {  
  composes: MediumParagraph;  
  color: blue;  
  text-align: left;  
}  
.GreenParagraph {  
  composes: MediumParagraph;  
  color: green;  
  text-align: right;  
}
```

Finally, in order to write normal style with a global scope, you can use the `:global` selector in front of your class name:

```
:global .HeaderParagraph {  
  font-size: 30px;  
  text-transform: uppercase;  
}
```

You can then reference the global scoped style like a normal class in your JavaScript file:

```
<p className="HeaderParagraph">Weather Forecast</p>
```

Styled components

Styled Components is a library designed for React and React Native. It combines both the CSS in JS and the CSS Modules methods for styling your components.

Let me show you an example:

```
import React from "react";
import styled from "styled-components";

// Create a Title component
// that renders an <h1> tag with some styles
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;

function App() {
  return <Title>Hello World!</Title>;
}
```

When you write your style, you're actually creating a React component with your style attached to it. The funny looking syntax of `styled.h1` followed by backtick is made possible by utilizing JavaScript's tagged template literals.

Styled Components were created to tackle the following problems:

- **Automatic critical CSS:** Styled-components keep track of which components are rendered on a page, and injects their styles and nothing else automatically. Combined with code splitting, this means your users load the least amount of code necessary.
- **No class name bugs:** Styled-components generate unique class names for your styles. You never have to worry about duplication, overlap, or misspellings.
- **Easier deletion of CSS:** It can be hard to know whether a class name is already used somewhere in your codebase. Styled-components makes it obvious, as every bit of styling is tied to a specific component. If the component is unused (which tooling can detect) and gets deleted, all of its styles get deleted with it.
- **Simple dynamic styling:** Adapting the styling of a component based on its props or a global theme is simple and intuitive, without you having to manually manage dozens of classes.
- **Painless maintenance:** You never have to hunt across different files to find the styling affecting your component, so maintenance is a piece of cake no matter how big your codebase is.
- **Automatic vendor prefixing:** Write your CSS to the current standard and let styled-components handle the rest.

You get all of these benefits while still writing the same CSS you know and love – just bound to individual components.

If you'd like to learn more about styled components, you can visit the [documentation](#) and see more examples.

Which one should you start with?

As you will practice building some React projects in the next chapter, I recommend you to go with **CSS stylesheets** method because it's probably most familiar to you.

CSS Modules and Styled Components are better methods to handle complex front-end styling, but since these are practice projects, you don't need to think about scaling your CSS for large React projects.

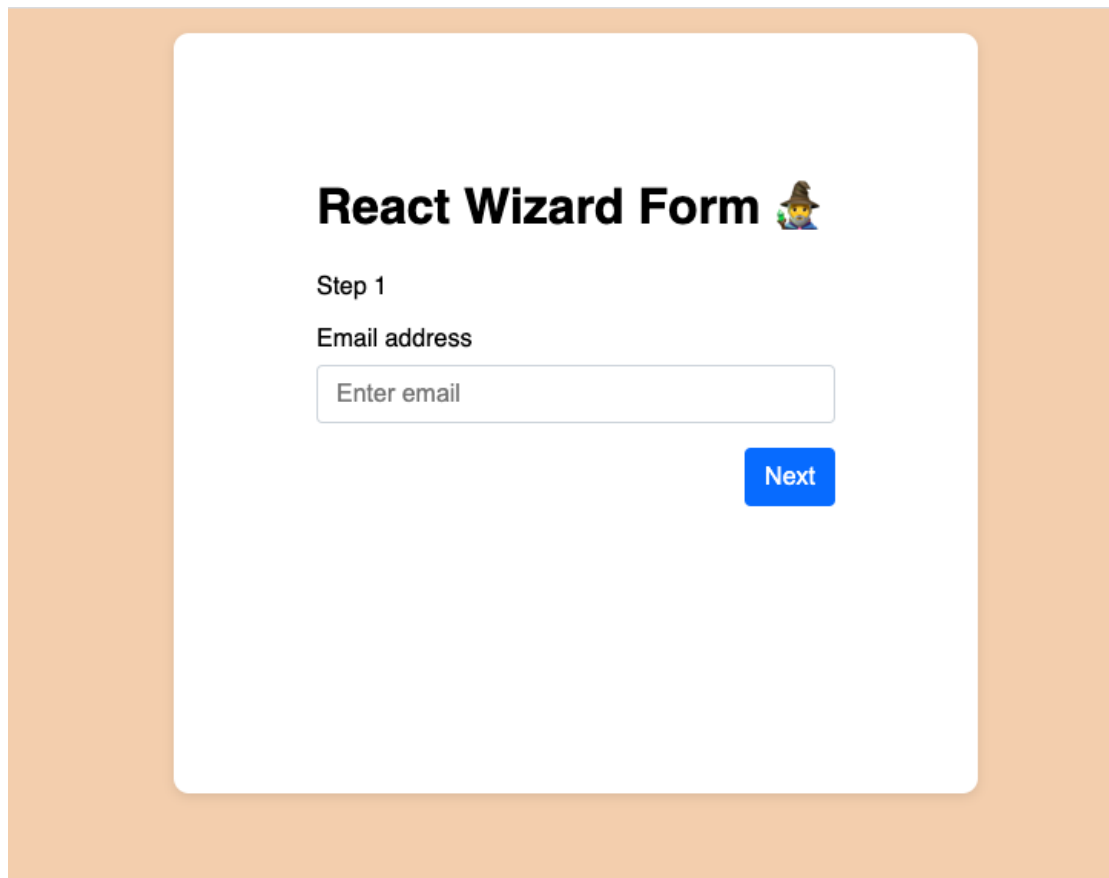
First Project: React Wizard Form

Alright, it's time to put what you've learned into practice!

In this module, you're going to build a wizard form using React.

You've probably seen something similar when you register for a new bank account. A wizard form is a multi-step form designed to ease the filling process for a long and complex form. By only showing a few inputs on a screen, users will feel encouraged to fill the blank inputs rather than feeling overwhelmed and potentially abandoning the form.

Here's the initial screen of your application:

The image shows a wizard form titled "React Wizard Form" with a wizard emoji. It is labeled "Step 1" and asks for an "Email address". There is a text input field with the placeholder "Enter email" and a blue "Next" button to its right. The form is centered on a light orange background.

React Wizard Form 🧙

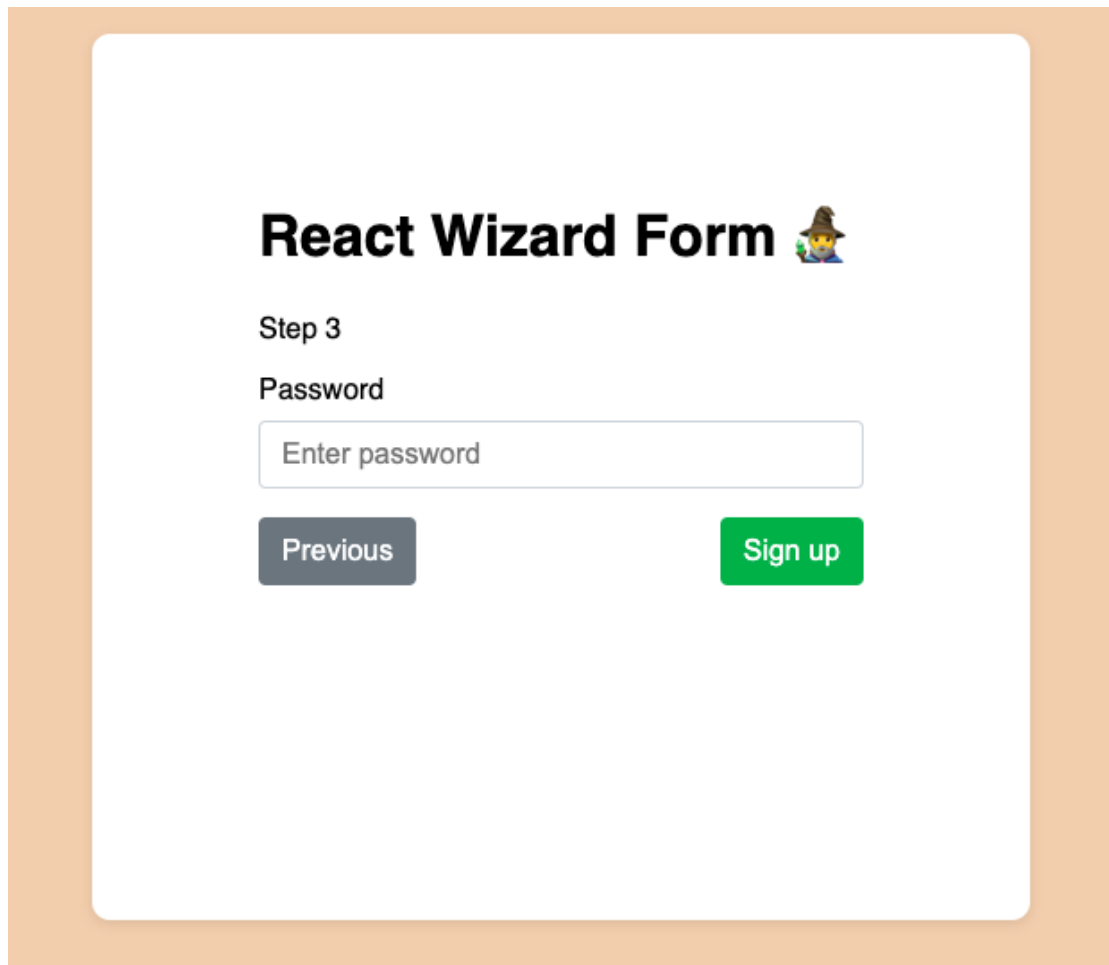
Step 1

Email address

Next

Figure 10: Wizard form initial screen

By pressing on the next button, you can jump into the other parts of the form:



The image shows a wizard form titled "React Wizard Form" with a wizard emoji. It is at "Step 3" and asks for a "Password". There is a text input field with the placeholder "Enter password". Below the input field are two buttons: a grey "Previous" button and a green "Sign up" button.

Figure 11: Wizard form step three

When you click on the submit button, an alert will be triggered and your inputs will be displayed:

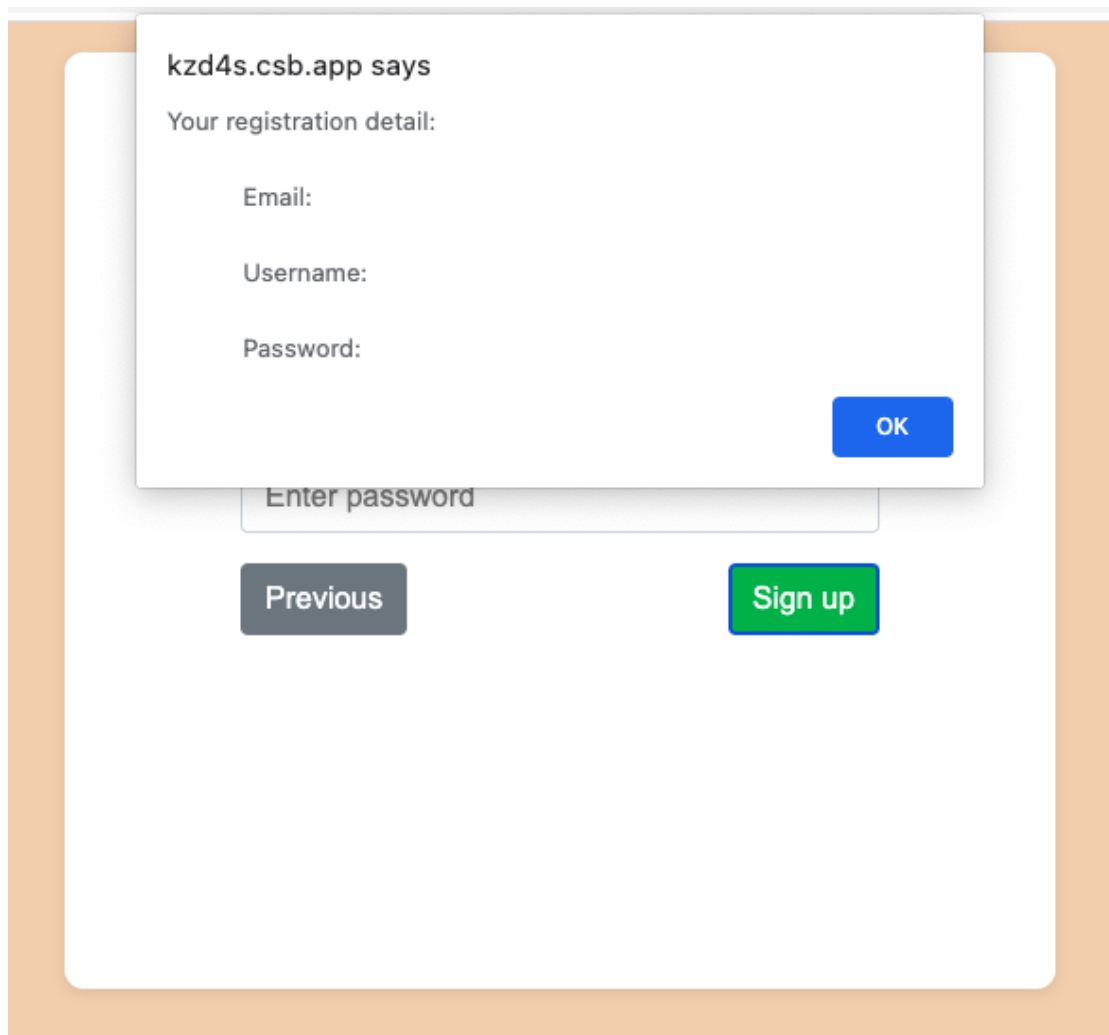


Figure 12: Wizard form submission

To keep things simple, the form will only have one input for each step and you don't need to validate the data. You can see the demo of [the app here](#).

Given the UI demo above, you basically need four components for this app: three components for each of the wizard form step and one for the wrapper component that change the display based on user click.

You can use both Create React App on your local computer or Code Sandbox on the web for this project.

Now that you know what you're going to build, let's create the first component in the next section.

Building the app structure

The easiest way to create a multi step form is to create a container form element, which contains all the wizard step component inside of it. This diagram will help you understand the mechanics of the form:

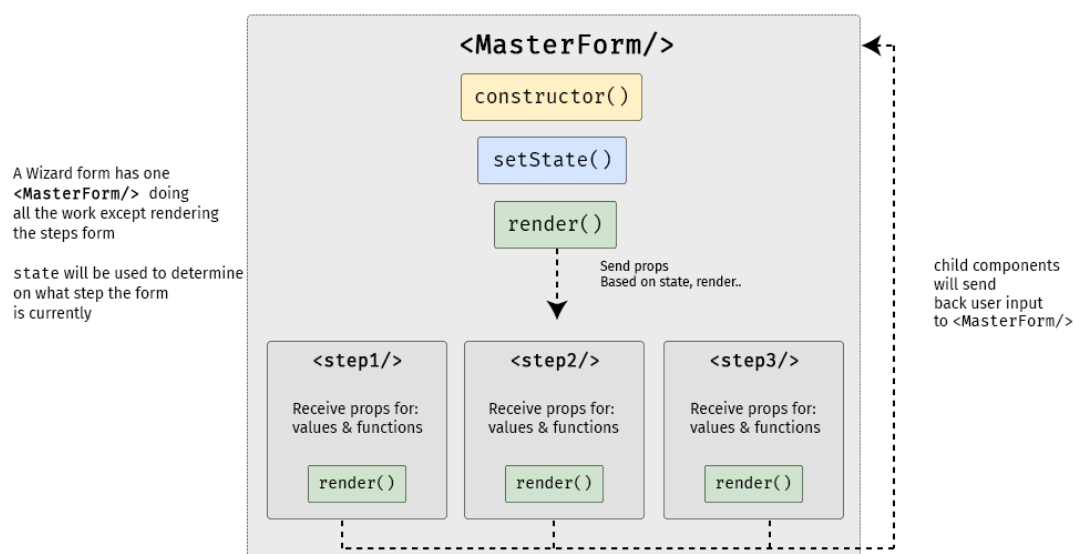


Figure 13: React wizard form mechanics

Instead of having one form component, we will have one parent component and three child components. In the diagram above, `<MasterForm/>` component will send data and functions to children components via props, and children components will trigger `handleChange()` function to set

values in the state of `<MasterForm/>` . You will need a function to move the form from one step to another as well.

In total, you need to build four components:

- `<MasterForm/>` will be the main container of the other form components
- `<Step1/>` component will render email address input
- `<Step2/>` will render username input
- `<Step3/>` will render password input and a submit button

Alright, let's start with making the wizard form mechanism. First, create three simple components like the following:

```
function Step1() {  
  return <h1>Step 1 Component</h1>;  
}  
  
function Step2() {  
  return <h1>Step 2 Component</h1>;  
}  
  
function Step3() {  
  return <h1>Step 3 Component</h1>;  
}
```

Next, let's start with making the `<MasterForm/>` component. Based on the demo that we've seen, the form will have three inputs, each for email, username and password. You need to track the current step in your state as well, so let's put four state properties for the component.

Open your `App.js` file and write the following code:

```
class MasterForm extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      currentStep: 1,
      email: '',
      username: '',
      password: '',
    }
  }
}
```

Next, write the render function as follows:

```
render() {
  return (
    <div className="wizard">
      <h1>React Wizard Form</h1>
      <p>Step {this.state.currentStep} </p>
      <form>
        <Step1 />
        <Step2 />
        <Step3 />
      </form>
    </div>
  );
}
```

Now you see that the `<MasterForm>` has all the children components rendered on the screen at the same time. You need to create a mechanism to display only one child component at a time. You can do that by sending the `currentStep` prop into the children components.

If the `currentStep` value equals the step number, the child component will render the element, else it will render `null` :


```
function Step1(props) {
  if (props.currentStep !== 1) {
    return null;
  }
  return <h1>Step 1 Component</h1>;
}

function Step2(props) {
  if (props.currentStep !== 2) {
    return null;
  }
  return <h1>Step 2 Component</h1>;
}

function Step3(props) {
  if (props.currentStep !== 3) {
    return null;
  }
  return <h1>Step 3 Component</h1>;
}
```

Then send the `currentStep` into each of these components:

```
<Step1 currentStep={this.state.currentStep} />
<Step2 currentStep={this.state.currentStep} />
<Step3 currentStep={this.state.currentStep} />
```

If you try to run the app again, you'll see that only one child component returns the `<h1>` element because the other will return `null` depending on the `currentStep` value.

Let's create a mechanism to change the `currentStep` value. To do that, you need to write two functions, one to increase the step and the other to decrease the step. Let's call these functions `next()` and `previous`. Place them just below the `constructor` call:

```
next = () => {
  let currentStep = this.state.currentStep;
  currentStep = currentStep >= 2 ? 3 : currentStep + 1;
  this.setState({
    currentStep: currentStep
  });
};

prev = () => {
  let currentStep = this.state.currentStep;
  currentStep = currentStep <= 1 ? 1 : currentStep - 1;
  this.setState({
    currentStep: currentStep
  });
};
```

The functions will make sure that the step value will be between one to three. Now you need to create two buttons. One button for the next step and one for the previous step. Let's put them inside a function like this:

```
previousButton() {
  let currentStep = this.state.currentStep;
  if (currentStep > 1) {
    return (
      <button
        className="btn btn-secondary"
        type="button"
        onClick={this.prev}
      >
        Previous
      </button>
    );
  }
  return null;
}

nextButton() {
  let currentStep = this.state.currentStep;
  if (currentStep < 3) {
    return (
```

```
    <button
      className="btn btn-primary float-right"
      type="button"
      onClick={this.next}
    >
      Next
    </button>
  );
}
return null;
}
```

The next button will only be visible when the `currentStep` is smaller than three, while the previous button appears when `currentStep` is bigger than one.

Any custom functions in your component can also return an element to render. You just need to call these functions inside your `render()` function:

```
<Step1 currentStep={this.state.currentStep} />
<Step2 currentStep={this.state.currentStep} />
<Step3 currentStep={this.state.currentStep} />
{this.previousButton()}
{this.nextButton()}
```

With that, you have the wizard form mechanism ready. Let's complete this section by writing the CSS for your form. These CSS classes are referenced from Bootstrap, but I only pick the part that you need to complete the project:

```
body{
  margin: 1em;
  font-family: sans-serif;
  background-color: #EDCEB0;
}

label {
  display: inline-block;
  margin-bottom: .5rem;
}

.wizard {
  margin-left: auto;
  margin-right: auto;
  position: relative;
  background: #fff;
  height: 424px;
  width: 338px;
  padding: 71px 93px 0;
  border-radius: 10px;
  box-shadow: 0 2px 7px 0 rgba(0,0,0,.1);
}

.form-group {
  margin-bottom: 1rem;
}

.form-control {
  display: block;
  width: 100%;
  padding: .375rem .75rem;
  font-size: 1rem;
  line-height: 1.5;
  color: #495057;
  background-color: #fff;
  background-clip: padding-box;
  border: 1px solid #ced4da;
  border-radius: .25rem;
  box-sizing: border-box;
  transition: border-color .15s ease-in-out, box-shadow .15s ease-in-out;
}

.float-right {
  float: right !important;
}
```

```
.btn-primary {
  color: #fff;
  background-color: #007bff;
  border-color: #007bff;
}

.btn-secondary {
  color: #fff;
  background-color: #6c757d;
  border-color: #6c757d;
}

.btn-success {
  color: #fff;
  background-color: #28a745;
  border-color: #28a745;
}

.btn {
  display: inline-block;
  font-weight: 400;
  text-align: center;
  white-space: nowrap;
  vertical-align: middle;
  -webkit-user-select: none;
  -moz-user-select: none;
  -ms-user-select: none;
  user-select: none;
  border: 1px solid transparent;
  padding: .375rem .75rem;
  font-size: 1rem;
  line-height: 1.5;
  border-radius: .25rem;
  transition: color .15s;
}
```

You can see the [Code Sandbox](#) here if you miss a step.

Writing the functions for change and submit

With the wizard form mechanism ready, it's time to write the functions for change and submit event. Don't worry! It's actually very simple. Here's the function for `handleChange` and `handleSubmit` respectively.

Instead of using the regular `function` keyword, you can also use the ES6 arrow function syntax. This way, you don't need to bind the function context when calling it:

```
handleChange = (event) => {  
  const { name, value } = event.target;  
  this.setState({  
    [name]: value  
  });  
};  
  
handleSubmit = (event) => {  
  event.preventDefault();  
  const { email, username, password } = this.state;  
  alert(`Your registration detail: \n  
    Email: ${email} \n  
    Username: ${username} \n  
    Password: ${password}`);  
};
```

You're going to pass the `handleChange` function to each of the child components, while `handleSubmit` needs to be passed into the `<form>` element in the parent component:

```
render() {  
  return (  
    <div className="wizard">  
      <h1>React Wizard Form</h1>  
      <p>Step {this.state.currentStep} </p>  
  
      <form onSubmit={this.handleSubmit}>  
    {/*  
      code omitted for brevity  
    */}  
  )  
}
```

Great! Now all you need to do is to finish the children components. You'll do that in the next section.

Finishing child components

It's time to write the form input components on each step. First, let's send additional props to the child components. You need to send the state required by the child along with the `handleChange` function. Here's the updated call to the components:

```
<Step1
  currentStep={this.state.currentStep}
  handleChange={this.handleChange}
  email={this.state.email}
/>
<Step2
  currentStep={this.state.currentStep}
  handleChange={this.handleChange}
  username={this.state.username}
/>
<Step3
  currentStep={this.state.currentStep}
  handleChange={this.handleChange}
  password={this.state.password}
/>
```

Now let's write the steps component, starting with **Step1** component.

The component will render a form input, with its value retrieved from the `email` prop. The `handleChange` function will be triggered when a user types into the text input:

```
function Step1(props) {
  if (props.currentStep !== 1) {
    return null;
  }
  return (
    <div className="form-group">
```



```
    <label htmlFor="email">Email address</label>
    <input
      className="form-control"
      id="email"
      name="email"
      type="text"
      placeholder="Enter email"
      value={props.email}
      onChange={props.handleChange}
    />
  </div>
);
}
```

You need to do the same for the **Step2** component, only it's for `username` instead of `email` :

```
function Step2(props) {
  if (props.currentStep !== 2) {
    return null;
  }
  return (
    <div className="form-group">
      <label htmlFor="username">Username</label>
      <input
        className="form-control"
        id="username"
        name="username"
        type="text"
        placeholder="Enter username"
        value={props.username}
        onChange={props.handleChange}
      />
    </div>
  );
}
```

For the **Step3** component, you need to also add the submit button so that the user can submit the form:

```
function Step3(props) {
  if (props.currentStep !== 3) {
    return null;
  }
  return (
    <React.Fragment>
      <div className="form-group">
        <label htmlFor="password">Password</label>
        <input
          className="form-control"
          id="password"
          name="password"
          type="password"
          placeholder="Enter password"
          value={props.password}
          onChange={props.handleChange}
        />
      </div>
      <button
        className="btn btn-success float-right"
        >
        Sign up
      </button>
    </React.Fragment>
  );
}
```

And now your wizard form is done! Here's [the complete app](#) to compare with your own.

Wrapping up

Congratulations on finishing your first React project! This application really helps you to get the idea of create a web applicatio by composing components together. Although a wizard form looks complex, it's actually pretty easy when you have the steps mechanism in place.

You also learned how to pass data back and forth between parent and child components. The pattern of calling a component and sending props, including the function to update the state, is pretty common in React applications.

Let's continue to part two, where you will learn all about React function-based component.

Part 2: React function components and hooks

Previously, React features such as state and lifecycle functions are available only for class-based components. Function-based components are referred to as dumb, skinny, or just presentational components because they can't have access to state and lifecycle functions.

But since the release of **React Hooks**, function-based components have been elevated into first-class citizens of React. It has enabled function components to compose, reuse, and share React code in new ways.

As a React developer, you need to learn how to write React components using function components because you will meet both class and function based components in real React projects across the globe.

And since function based components will be the common way to write React components, you need to know how it works. Let's start right away.

React function-based components

React allows you to write components using both JavaScript `class` and `function` syntax. In the past, writing components using the `class` syntax is required if you want your components to use React features like `state` and the lifecycle functions.

But in February 2019, React team released a feature called hooks that allows you to write a fully-featured React components with just JavaScript functions.

In part 1 of this book, you've seen how you can write components using both `class` and `function` syntax:

```
import React from 'react';

class MyClassComponent extends React.Component {
  render(){
    return <h1> Hello World </h1>
  }
}

function MyFunctionComponent(){
  return <h1> Hello World </h1>
}
```

Unlike class components, function components doesn't inherit from React's `Component` class. This is why a functional component doesn't know about state and can't execute code when the component enters or

leave the screen. Another difference is that while a class component receive props in `this.props` object, a function component receive props as the first argument:

```
function MyFunctionComponent(props){  
  return <h1> Hello World </h1>  
}
```

To wrap up the introduction, here's how a class component calls a function component:

```
import React from 'react';  
  
class MyClassComponent extends React.Component {  
  render(){  
    return <MyFunctionComponent name="John"/>  
  }  
}  
  
function MyFunctionComponent(props){  
  return <h1> Hello World {props.name}</h1>  
}
```

Next, you're going to learn about React hooks and how it enabled function components to compose, reuse, and share React code in a convenient way.

The `useState` hook

React hooks are JavaScript functions that you can import from React package in order to add features into your components. Hooks are available only for function based components, so they can't be used inside a class component.

React provides you with 10 function hooks, but only 2 of these hooks are going to be used frequently when you write function components. They are `useState` and `useEffect` hooks. Let's learn about `useState` first.

The `useState` hook is a function that takes one argument, which is the initial state, and it returns two values: the current state and a function that can be used to update the state. Here's the hook in action:

```
import React, { useState } from 'react'

function UserComponent() {
  const [name, setName] = useState('John')
}
```

Notice the use of square brackets when state variable is declared. This is the ES6 [array destructuring syntax](#), and it means we're assigning the first element of the array returned by `useState` to `name` and the second element to `setName` variable.

So this means we have a state named `name` and we can update it by calling on `setName()` function. Let's use it on the return statement:

```
import React, { useState } from 'react'

function UserComponent() {
  const [name, setName] = useState('John')

  return <h1> Hello World! My name is {name} </h1>
}
```

Since function components don't have the `setState()` function, you need to use the `setName()` function to update it. Here's how you change the name from "John" to "Luke":

```
import React, { useState } from 'react'

function UserComponent() {
  const [name, setName] = useState('John')

  if(name === "John"){
    setName("Luke")
  }

  return <h1> Hello World! My name is {name} </h1>
}
```

When you have multiple states, you can call the `useState` hook as many times as you need. The hook receives all valid JavaScript data types such as string, number, boolean, array, and object:


```
import React, { useState } from 'react'

function UserComponent() {
  const [name, setName] = useState('Jack')
  const [age, setAge] = useState(10)
  const [isLegal, setLegal] = useState(false)
  const [friends, setFriends] = useState(["John", "Luke"])

  return <h1> Hello World! My name is {name} </h1>
}
```

And that's all there is to it. The `useState` hook basically enables function components to have its own internal state.

The useEffect hook

The `useEffect` hook is the combination of `componentDidMount`, `componentDidUpdate` and `componentWillUnmount` class lifecycle methods. This hook is the ideal place to setup listener, fetching data from API and removing listeners before component is removed from the DOM.

Let's look at an example of `useEffect` in comparison with class lifecycle methods. Normally in class component, we write this kind of code:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'Nathan',
    };
  }

  componentDidMount() {
    console.log(
      `didMount triggered: Hello I'm ${this.state.name}`
    );
  }

  componentDidUpdate() {
    console.log(
      `didUpdate triggered: Hello I'm ${this.state.name}`
    );
  }

  render() {
    return (
      <div>
        <p>`Hello I'm ${this.state.name}`</p>
        <button
          onClick={() =>
            this.setState({ name: 'Gary'})
          }
        />
      </div>
    );
  }
}
```

```
        >  
        Change me  
      </button>  
    </div>  
  );  
}  
}
```

Since `componentDidMount` is run only once when the component is inserted into the DOM tree structure, subsequent render won't trigger the method anymore. In order to do run something on each render, you need to use `componentDidUpdate` method.

Using `useEffect` hook is like having both `componentDidMount` and `componentDidUpdate` in one single method, since `useEffect` runs on every render. It accepts two arguments:

- (mandatory) A function to run on every render
- (optional) An array of state variables to watch for changes.

`useEffect` will be skipped if none of the variables are updated.

Rewriting the above class into function component would look like this:

```
const Example = props => {  
  const [name, setName] = useState('Nathan');  
  
  useEffect(() => {  
    console.log(`Hello I'm ${name}`);  
  });  
  
  return (  
    <div>
```

```
    <p>{`Hello I'm ${name}`}</p>
    <button
      onClick={() => {
        setName('Gary')
      }}>
      Change me
    </button>
  </div>
)
}
```

The function component above will run the function inside of `useEffect` function on each render. Now this isn't optimal because the state won't be updated after the first click. This is where `useEffect` second argument come into play.

```
useEffect(() => {
  console.log(`Hello I'm ${name} and I'm a ${role}`);
},
[name]);
```

The second argument of `useEffect` function is referred as the “dependency array”. When the variable included inside the array didn't change, the function passed as the first argument won't be executed.

The `componentWillUnmount` effect

If you have code that needs to run when the component will be removed from the DOM tree, you need to specify a `componentWillUnmount`

method by writing a `return` statement into the first argument function.

Here is an example:

```
useEffect(() => {  
  console.log(`useEffect function`);  
  
  return () => { console.log("componentWillUnmount effect"); }  
}, [name] );
```

Running useEffect only once

To run `useEffect` hook only once like `componentDidMount` function, you can pass an empty array into the second argument:

```
useEffect(  
  () => {  
    console.log(`useEffect function`);  
  },  
  [] );
```

The empty array indicates that the effect doesn't have any dependencies to watch for change, and without a trigger it won't be run after the component is mounted.

The rules of hooks

When using hooks, you need to follow these two rules:

Only call Hooks at the top level

Don't call hooks inside loops, conditions, and nested functions. When you want to use some hooks conditionally, write the condition inside those hooks.

Don't do this:

```
if (name !== '') {  
  useEffect(function logger() {  
    console.log(name)  
  });  
}
```

Instead, do this:

```
useEffect(function logger() {  
  if (name !== '') {  
    console.log(name)  
  }  
});
```

This rule will ensure that Hooks are called in the same order each time a component renders. That's what allows React to correctly preserve the state of Hooks between multiple `useState` and `useEffect` calls.

Only call Hooks from function components

Don't call hooks from regular JavaScript functions. Call hooks only from function components or custom hooks. By following this rule, you make sure that all stateful logic in a component is clearly visible from the source code.

Conditional rendering with React

You can control what output is being rendered by your React component by implementing a conditional rendering in your JSX. For example, let's say you want to switch between rendering login and logout button, depending on the availability of the `user` state:

```
import React from "react"

function App(props) {
  const {user} = props

  if (user) {
    return <button>Logout</button>
  }
  return <button>Login</button>
}

export default App
```

You don't need to add an `else` into your component, because React will stop further process once it reaches a `return` statement. In the example above, React will render the logout button when `user` value is truthy, and the login button when `user` is falsy.

Partial rendering with a regular variable

There might be a case where you have want to render a part of your UI dynamically in a component. For example, you might want to render a button below a header element:


```
import React from "react"

function App(props) {
  const {user} = props

  let button = <button>Login</button>

  if (user) {
    button = <button>Logout</button>
  }

  return (
    <>
      <h1>Hello there!</h1>
      {button}
    </>
  )
}

export default App
```

Instead of writing two return statements, you just need to store the dynamic UI element inside a variable (in this case, the `button` variable)

This is convenient because you just have to write the static UI element only once.

Inline rendering with `&&` operator

It's possible to render a component only if a certain condition is met and render null otherwise. For example, let's say you want to render a dynamic message for users when they have new emails in their inbox:

```
import React from "react"

function App() {
  const newEmails = 2

  return (
    <>
      <h1>Hello there!</h1>
      {newEmails > 0 &&
        <h2>
          You have {newEmails} new emails in your inbox.
        </h2>
      }
    </>
  )
}

export default App
```

Inline rendering with conditional (ternary) operator

It's also possible to use a ternary operator in order to render the UI dynamically. Take a look at the following example:

```
import React from "react"

function App(props) {
  const {user} = props

  return (
    <>
      <h1>Hello there!</h1>
      { user? <button>Logout</button> : <button>Login</button> }
    </>
  )
}

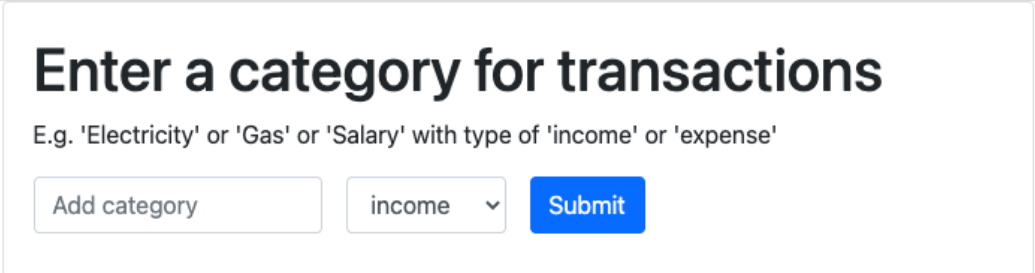
export default App
```

Instead of using a variable to hold the button element, you can simply use the ternary operator on the `user` value and render Logout button when it's falsy, or Login button when it's truthy.

Second project: Finance Tracker

Now that you’ve finished learning about React functional components, it’s time to put your new knowledge into practice. In this section, you’re going to build a finance tracker application, where you can input an amount and mark it as expense or income.

On render, the application will first ask you to input a transaction category and its type: whether an “income” or “expense”:



Enter a category for transactions

E.g. 'Electricity' or 'Gas' or 'Salary' with type of 'income' or 'expense'

Figure 14: Finance tracker category form

Once you submit your first category, you’ll have the main screen displaying the category as the header, with a transaction table and a chart below it:

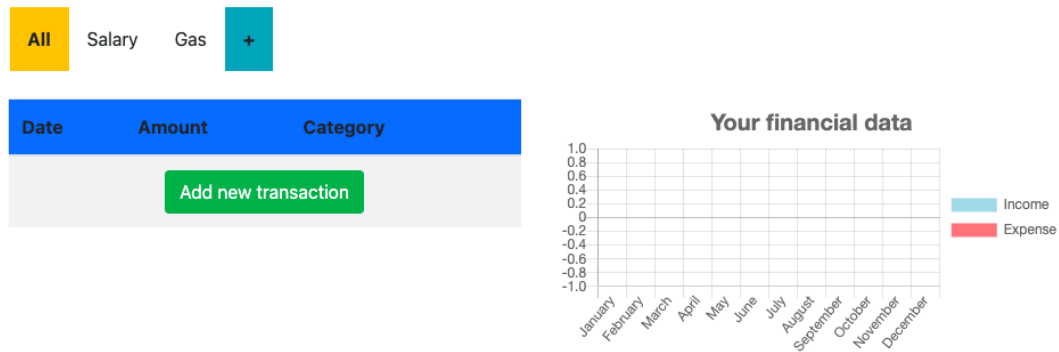


Figure 15: Finance tracker main screen

When you click on the add new transaction button, you'll get another form to input the detail of your transaction:

The screenshot shows a form titled 'Enter a new transaction'. Below the title is a subtitle: 'Enter the date, amount and category of the transaction'. The form contains three input fields: a date field with the value '11/24/2020', a category dropdown menu showing 'Salary', and an amount field with the value '100'. An 'Add' button is located to the right of the amount field.

Figure 16: Adding new transaction

Finally, the new transaction that you added will be displayed in the transaction table along with the updated chart, showing visualization of your income and expense amount. You can add as many category and transaction into the application, but the chart will only display transactions for this year:

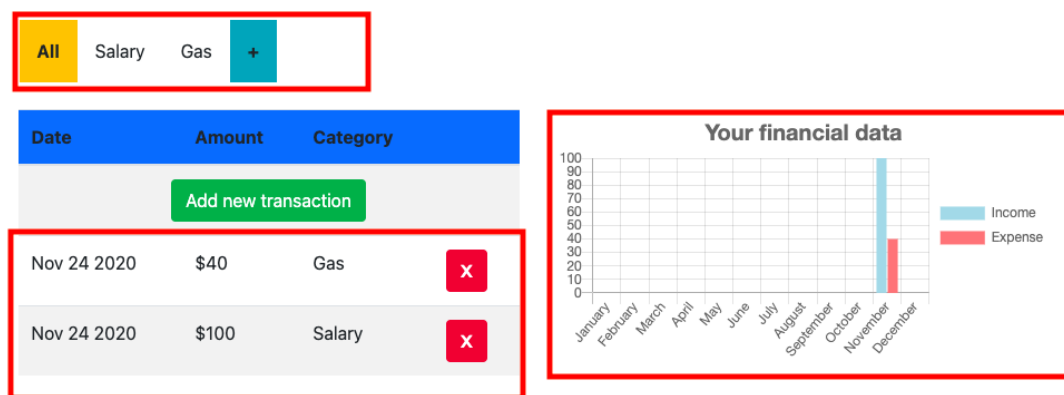


Figure 17: The complete finance tracker app

Here's a [live demo for this application](#).

You will only use function components and hooks so that you'll get used to the functional component pattern. The styling of the application use the Bootstrap framework, so you need to install that into your project as well.

Before moving on, take some time to play with the demo. Can you guess the pattern and how many components you need to build the application?

I'll see you in the next section.

Composing the components

Just like the last project, you need to start by thinking about the application structure and how many components you need to write. Looking at the demo, the application could be split into the following components:

- The category form will be one component
- The transaction form will be one component

The main screen can be one component, but since it has more parts than the other, let's split it into three components:

- The header where the categories are displayed
- The table for transactions
- The chart for transactions visualization

Following this structure, you will need to build five components that will interact with each other. Let's create these components before anything else. Create a new `components` folder in your app and write the following components:

- `AddCategory.js`
- `AddTransaction.js`
- `Header.js`
- `TransactionTable.js`
- `Chart.js`

Give a placeholder element for each of these components, like this:

```
import React from "react";

export default function Header(){
  return <h1>Header component</h1>
}
```

You can do the same with the rest of the components.

Implementing the application process

With the components in place, let's place them all accordingly in the

`App` component, which acts as the container:

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

import Header from "../components/Header";
import TransactionTable from "../components/TransactionTable";
import Chart from "../components/Chart";
import AddCategory from "../components/AddCategory";
import AddTransaction from "../components/AddTransaction";

export default function App(){
  return (
    <>
      <Header />
      <TransactionTable />
      <Chart />
      <AddCategory />
      <AddTransaction />
    </>
  )
}
```

Next, you need to write the logic for conditionally rendering these components.

Transaction form rendering

Back to the demo, The application will first display only the category form, and after one category is added, it will display the `Header` , `TransactionTable` , and `Chart` component.

You need to implement a logic to change the component being rendered by React. Do you remember how to do it? You've done the same thing with the wizard form.

That's right, you need to create a state that acts as the indicator for React which component to render to the screen. Let's name this state `showAddCategory` and set its default value to `true` :

```
function App(){  
  const [showAddCategory, setShowAddCategory] = useState(true);  
}
```

And implement a conditional rendering using that state value :

```
if (showAddCategory) {  
  return <AddCategory />;  
} else {  
  return <Header />  
}
```

Because `AddCategory` must be able to switch back into the main view, you need to pass the `setShowAddCategory` as a prop into it:

```
return <AddCategory setShowAddCategory={setShowAddCategory} />;
```

Then inside `AddCategory` component, let's create a button to set the state back to `false` :

```
function AddCategory({ setShowAddCategory }) {  
  return (  
    <>  
      <h1> AddCategory Component </h1>  
      <button onClick={() => setShowAddCategory(false)}>  
        Return to main screen  
      </button>  
    </>  
  );  
}
```

Next, pass `setShowAddCategory` to `Header` component so that you can switch from the main screen into the category form:

```
if (showAddCategory) {  
  return <AddCategory setShowAddCategory={setShowAddCategory} />;  
} else {  
  return <Header setShowAddCategory={setShowAddCategory} />  
}
```

Add a button to `Header` component to set the state to `true` :

```
function Header({ setShowAddCategory }) {  
  return (  
    <>  
      <h1> Header Component </h1>  
      <button  
        onClick={() => setShowAddCategory(true)}>  
        Add new Category  
      </button>  
    </>  
  );  
}
```

You can now switch back and forth from category form and the main screen. Great job!

Transaction form rendering

You need to do the same for transaction form, so let's create another Boolean state called `showAddTransaction` . The default value will be `false` :

```
const [showAddCategory, setShowAddCategory] = useState(true);  
const [showAddTransaction, setShowAddTransaction] = useState(false);
```

If `showAddTransaction` is `true` , you need to render the `AddTransaction` component. Also, pass `setShowAddTransaction` to both `AddTransaction` and `TransactionTable` component:

```
if (showAddCategory) {
  return (
    <AddCategory
      setShowAddCategory={setShowAddCategory} />
  );
}
if (showAddTransaction) {
  return (
    <AddTransaction
      setShowAddTransaction={setShowAddTransaction} />
  );
}

return (
  <>
    <Header
      setShowAddCategory={setShowAddCategory} />
    <TransactionTable
      setShowAddTransaction={setShowAddTransaction} />
    <Chart />
  </>
);
```

Finally, add a button to `AddTransaction` component for returning to the main screen:

```
function AddTransaction({ setShowAddTransaction }) {
  return (
    <>
      <h1> AddTransaction Component </h1>
      <button
        onClick={() => setShowAddTransaction(false)}>
        Return to main screen
      </button>
    </>
  );
}
```

And inside `TransactionTable` component for going to the transaction form screen:

```
function TransactionTable({ setShowAddTransaction }) {  
  return (  
    <>  
      <h1>TransactionTable Component</h1>  
      <button onClick={() => setShowAddTransaction(true)}>  
        Add new transaction  
      </button>  
    </>  
  );  
}
```

With that, you can switch back and forth from the main screen to the transaction form.

Here's the [complete structure demo](#) if you miss any step.

There you go. With the basic structures of your application in place, you now have a clear separation of responsibility for each component. This will help you to stay productive and avoid getting confused with the component details later.

In the next section, you're going to start writing the `AddCategory` component first.

Writing the category form

Since the category form will add new category for the application, that means you need to store an array of categories somewhere inside your application. Let's store it as a state inside the `App` component:

```
const [categories, setCategories] = useState([]);
```

Then, pass `setCategories` into the `AddCategory` component:

```
<AddCategory  
  setCategories={setCategories}  
  setShowAddCategory={setShowAddCategory} />
```

This category form will simply ask you to input a transaction category and its type. There are two types for the category, between income and expense, so you need to use a `<select>` input for that. You can write the `types` as an array constant:

```
function AddCategory({ setCategories, setShowAddCategory }) {  
  const types = ["income", "expense"];  
  const [name, setName] = useState("");  
  const [selectedType, setSelectedType] = useState("income");
```

Then, write a `handleSubmit` function to process the form submission. You need to save the category as an object with property of `name` and `type`. After updating the categories, return to the main screen by calling `setShowAddCategory(false)`:

```
const handleSubmit = (e) => {
  e.preventDefault();
  if (!name) {
    alert("Enter a category");
    return;
  }
  const category = {
    name,
    type: selectedType
  };

  setCategories((currentState) => [...currentState, category]);
  setShowAddCategory(false);
};
```

Now you just need to write the `return` statement. The core form element is simply one input element, one select element, and one submit button:

```
<div className="container">
  <div className="card">
    <div className="card-body">
      <h1>Enter a category for transactions</h1>
      <p>
        E.g. 'Electricity' or 'Gas' or 'Salary' with type of 'income' or
        'expense'
      </p>
      <form
        className="form-inline"
        onSubmit={handleSubmit}>
        <div className="form-group mb-2">
          <input
            className="form-control"
            value={name}
            placeholder="Add category"
            onChange={(e) => setName(e.target.value)}
          />
        </div>
        <div className="form-group mx-sm-3 mb-2">
          <select
            className="form-control"
```



```
        value={selectedType}
        onChange={(e) => setSelectedType(e.target.value)}
      >
      {types.map((type, index) => {
        return (
          <option key={index} value={type}>
            {type}
          </option>
        );
      })}
    </select>
  </div>
  <button
    type="submit"
    className="btn btn-primary mb-2">
    Submit
  </button>
</form>
</div>
</div>
</div>
```

Now your `categories` state will receive new category object from `AddCategory` form. Let's display those categories inside the `Header` next.

Displaying categories in Header

Inside `App.js`, send the `categories` prop into the `Header`:

```
return (
  <div className="container">
    <div className="row">
      <Header
        categories={categories}
        setShowAddCategory={setShowAddCategory}
      />
    </div>
  </div>
);
```

Next, open `Header.js` file and grab the `categories` from props.

Also, write a `ul` element inside your `return` statement:

```
export default function Header({ categories, setShowAddCategory }) {  
  return (  
    <ul className="navbar navbar-expand flex-row w-100 list-unstyled">  
      </ul>  
  );  
}
```

After that, you need to `map()` over the `categories` and return a `li` for each array element:

```
{categories.map((category, index) => {  
  return (  
    <li className="p-3 nav-item" key={index}>  
      {category.name}  
    </li>  
  );  
}}}
```

Finally, write another `li` just below your `map()` that represents a button to add new category:

```
<li  
  className="font-weight-bold p-3 nav-item bg-info"  
  onClick={() => setShowAddCategory(true)}  
>  
  +  
</li>
```

Here's [the state of the application](#) until this point. Be sure to checkout the `Header` and `AddCategory` component if you face any error.

In the next section, you will write the `AddTransaction` component.

Writing AddTransaction component

Just like the `AddCategory` component, the `AddTransaction` component will add new transactions into the application, so let's create a state to store the data. Add the `transaction` state inside `App.js` file:

```
const [transactions, setTransactions] = useState([]);
```

Each transaction must have a category assigned to it, so pass both `categories` and `setTransactions` into `AddTransaction` :

```
if (showAddTransaction) {  
  return (  
    <AddTransaction  
      categories={categories}  
      setTransactions={setTransactions}  
      setShowAddTransaction={setShowAddTransaction}  
    />  
  );  
}
```

Now it's time to write the form. Head over to `AddTransaction` component and destructure the newly added props:

```
export default function AddTransaction({  
  categories,  
  setTransactions,  
  setShowAddTransaction,  
})
```

This form will have one date input, one select input for category, one input for amount, and the submit button. Let's use the

`react-datepicker` library to render a date input instead of writing one from scratch. Install it with NPM:

```
npm install react-datepicker
```

And import both the library and the style into your component:

```
import DatePicker from "react-datepicker";  
import "react-datepicker/dist/react-datepicker.css";
```

Next, write the states you need for the input elements:

```
const [amount, setAmount] = useState(0);  
const [selectedDate, setSelectedDate] = useState(new Date());  
const [selectedCategory, setSelectedCategory] = useState(0);
```

And then write the `handleSubmit` function for the form. It's very similar to the submit function of `AddCategory` except that the data object has three properties instead of two:

```
const handleSubmit = (e) => {  
  e.preventDefault();  
  if (!amount) {  
    alert("Enter amount");  
    return;  
  }  
  const finance = {  
    date: selectedDate,  
    category: categories[selectedCategory],  
    amount: amount  
  };  
  setTransactions((currentState) => [...currentState, finance]);  
  setShowAddTransaction(false);  
};
```

Finally, write the form inside the `return` statement:

```
<div className="container">
  <div class="card">
    <div class="card-body">
      <h1>Enter a new transaction</h1>
      <p>Enter the date, amount and category of the transaction</p>
      <form className="form-inline" onSubmit={handleSubmit}>
        <div className="form-group mb-2">
          <DatePicker
            className="form-control"
            selected={selectedDate}
            onChange={(date) => setSelectedDate(date)}
          />
        </div>
        <div className="form-group mx-sm-3 mb-2">
          <select
            className="form-control"
            value={selectedCategory}
            onChange={(e) => setSelectedCategory(e.target.value)}
          >
            {categories.map((category, index) => {
              return (
                <option key={index} value={index}>
                  {category.name}
                </option>
              );
            })}
          </select>
        </div>
        <div className="form-group mr-3 mb-2">
          <input
            className="form-control"
            value={amount}
            placeholder="Set amount"
            onChange={(e) => setAmount(e.target.value)}
          />
        </div>
        <button type="submit" class="btn btn-primary mb-2">
          Add
        </button>
      </form>
    </div>
  </div>
</div>
```

You can view the complete code in [this Code Sandbox](#).

With that, you have new transactions added into the `transactions` state. Let's render those transactions inside `TransactionTable` component next.

Listing transactions on TransactionTable component

First, pass the `transactions` state into the `TransactionTable` :

```
<TransactionTable
  transactions={transactions}
  setShowAddTransaction={setShowAddTransaction}
/>
```

Then head over to the `TransactionTable.js` file. Destructure your new prop at the function definition:

```
export default function TransactionTable({
  transactions,
  setShowAddTransaction
})
```

After that, write a table styled using Bootstrap classes:

```
<table className="table table-striped border">
  <thead className="bg-primary">
    <tr>
      <th scope="col">Date</th>
      <th scope="col">Amount</th>
      <th scope="col">Category</th>
      <th scope="col"></th>
    </tr>
  </thead>
  <tbody>
    <tr className="p-4 bg-blue-200 text-center">
      <td colSpan="4">
        <button
          className="btn btn-success"
          onClick={() => setShowAddTransaction(true)}
        >
          Add new transaction
      </td>
    </tr>
  </tbody>
</table>
```

```
        </button>
      </td>
    </tr>
  </tbody>
  {transactions.map((transaction, index) => {
    return (
      <tr className="p-4" key={index}>
        <td>{format(transaction.date, "MMM d yyyy")}</td>
        <td>${transaction.amount}</td>
        <td>{transaction.category.name}</td>
        <td>
          <button className="btn btn-danger">X</button>
        </td>
      </tr>
    );
  })}
</tbody>
</table>
```

Inside the `table` element, write the `thead` to represent the table header row. You will have four columns:

- One for transaction date
- One for the amount
- One for the category
- And the last one to place the delete transaction button

You're going to write the delete button in the next section. For now, let's focus on rendering the transactions:

```
<table className="table table-striped border">
  <thead className="bg-primary">
    <tr>
      <th scope="col">Date</th>
      <th scope="col">Amount</th>
      <th scope="col">Category</th>
```



```
    <th scope="col"></th>
  </tr>
</thead>
</table>
```

Then write the `tbody` element just below `thead` to represent the table body. The first row will be used to place the “Add new transaction” button:

```
<tbody>
  <tr className="p-4 bg-blue-200 text-center">
    <td colspan="4">
      <button
        className="btn btn-success"
        onClick={() => setShowAddTransaction(true)}
      >
        Add new transaction
      </button>
    </td>
  </tr>
</tbody>
```

Finally, loop over the `transactions` array just below the `tr` element:

```
{transactions.map((transaction, index) => {
  return (
    <tr className="p-4" key={index}>
      <td>{transaction.date}</td>
      <td>${transaction.amount}</td>
      <td>{transaction.category.name}</td>
      <td>
        <button
          className="btn btn-danger"
          onClick={() => removeTransaction(index)}
        >
          X
        </button>
      </td>
    </tr>
  )
})}
```

```
    </td>
  </tr>
);
})}
```

If you try to run the code, you'll see an error saying "Objects are not valid as a React child". This is because `react-datepicker` use the `date-fns` library to return a Date object as the value. To render the date correctly, you need to format the Date object into a string using `date-fns` . Install the library with NPM:

```
npm install date-fns
```

And import the `format` function from the library into your component:

```
import { format } from "date-fns";
```

Now you just need to format the date. Let's use `MMM d yyyy` to format the date as "Nov 21 2020":

```
<td>{format(transaction.date, "MMM d yyyy")}</td>
```

Here's the [Code Sandbox demo](#). Try to add as many transactions as you like.

It's time to write the delete button code.

Adding delete function to TransactionTable

Since the transaction is an array, you can use the array index value and the `filter` function to return each element whose index doesn't match the selected index. Create the `removeTransaction` function inside

`App.js` file:

```
const removeTransaction = (index) => {
  const newTransactions = transactions.filter((transaction, idx) => {
    return idx !== index;
  });
  setTransactions(newTransactions);
};
```

Pass the function into the `TransactionTable` component:

```
<TransactionTable
  setShowAddTransaction={setShowAddTransaction}
  removeTransaction={removeTransaction}
  transactions={transactions}
/>
```

Head to the `TransactionTable` and destructure the newly added prop:

```
export default function TransactionTable({
  removeTransaction,
  transactions,
  setShowAddTransaction
})
```

Finally, pass the function into the delete button `onClick` event:

```
<button
  className="btn btn-danger"
  onClick={() => removeTransaction(index)}
>
  X
</button>
```

With that, your delete button is now fully working.

Processing transactions with Chart.js

With the transactions table finished, you can now focus on building the `Chart` component.

To display a graphical chart representing the transactions data, you can use either [D3.js](#) or [Chart.js](#) library.

I decided to write the chart using Chart.js library because there's already a React - Chart.js binding library called `react-chart-js-2`. To use the library, you need to install it first:

```
npm install chart.js react-chart-js-2
```

Inside `App.js` file, pass the `transactions` state into the `Chart` component:

```
<Chart transactions={transactions} />
```

Looking at the demo, the app seems to use the bar chart, so let's import `Bar` from the `react-chart-js-2` library:

```
import { Bar } from "react-chart-js-2"

function Chart({transactions}){
}
```

Now you need to do some calculation using the available `transactions` data so that you have the right format to pass to the `Bar` component.

The data for the chart is as follows:

```
const chartData = {
  labels: ['January', 'February', 'March',
           'April', 'May'],
  datasets: [
    {
      label: "Income",
      backgroundColor: "lightblue",
      data: [65, 59, 80, 81, 56]
    },
    {
      label: "Expense",
      backgroundColor: "lightcoral",
      data: [25, 11, 29, 47, 60]
    }
  ]
}
```

The `labels` array and the `data` array is connected to each other.

From the example above, the month January will have income amount of `65` and expense amount of `25`. This means you need to sum all transactions for each month before passing it to the `Bar` component. First, let's create the `labels` for the graph:

```
const labels = [
  "January",
  "February",
  "March",
  "April",
  "May",
  "June",
  "July",
]
```

```
"August",  
"September",  
"October",  
"November",  
"December",  
];
```

Next, write the function to process the `transactions` array. It will accept two parameters, the `transactions` and the `type` that you want to sum, whether income or expense:

```
const processTransactions = (transactions, type) => {
```

Inside the function, create a new array with 12 elements, each with the value of 0. This will represent the amount for each month:

```
const monthsWithTxns = new Array(12).fill(0);
```

Then you need to loop over the `transactions` and return the total amount of income or expense for each month, depending on the `type` parameter being passed into it. Here's the full code for the function:

```
const processTransactions = (transactions, type) => {  
  const monthsWithTxns = new Array(12).fill(0);  
  
  for (const transaction of transactions) {  
    if (!isThisYear(transaction.date)) {  
      continue;  
    }  
    if (transaction.category.type !== type) {  
      continue;  
    }  
  }  
}
```

```
    }  
    const monthName = format(transaction.date, "MMMM");  
    const indexOfMonth = labels.indexOf(monthName);  
    monthsWithTx[indexOfMonth] += Number(transaction.amount);  
  }  
  
  return monthsWithTx;  
};
```

The two `if` block will check for the transaction `date` and `type` value. The loop will skip if:

- The transaction is not dated the current year
- The transaction doesn't have the same type as specified

The function `isThisYear` and `format` is imported from the `date-fns` library, so import them at the top of your component file:

```
import React from "react";  
import { Bar } from "react-chartjs-2";  
import { isThisYear, format } from "date-fns";
```

Next, you need to put the labels and the transactions into the right format under a new variable. I'll name it as `chartData`. Notice how the function `processTransactions` is assigned into the `data` property below:

```
const chartData = {  
  labels,  
  datasets: [  
    {
```



```
    label: "Income",
    backgroundColor: "lightblue",
    data: processTransactions(transactions, "income")
  },
  {
    label: "Expense",
    backgroundColor: "lightcoral",
    data: processTransactions(transactions, "expense")
  }
]
};
```

Finally, you just need to pass the data into the `Bar` component inside the `return` statement. I'll pass along some options to make the chart more informative:

```
return (
  <Bar
    data={chartData}
    options={{
      title: {
        display: true,
        text: "Your financial data",
        fontSize: 20
      },
      legend: {
        display: true,
        position: "right"
      }
    }}
  />
);
```

Here's a [working demo](#) for the Chart.

And that's it. Now you have a bar chart showing the total amount of income and expense for each month in the current year.

Activating category filter

You're almost done with this application. All you need to do now is to filter the transactions based on the category being clicked from the header.

First, add a new state called `activeCategory` into `App.js` file:

```
const [activeCategory, setActiveCategory] = useState("");
```

And pass both variables into the `Header` component:

```
<Header
  activeCategory={activeCategory}
  setActiveCategory={setActiveCategory}
  categories={categories}
  setShowAddCategory={setShowAddCategory}
/>
```

Head over to `Header.js` file and modify the `li` elements. Add a new `li` for displaying transactions from all categories before the the categories:

```
<li
  className={`font-weight-bold p-3 nav-item ${
    !activeCategory ? "bg-warning" : ""
  }}
  onClick={() => setActiveCategory("")}
>
  All
</li>
```

The “All” element will set the active category into empty string. It will also have a background color of yellow representing the current active category.

Inside `map()` function, you need to add the `onClick` event and the background color as well:

```
{categories.map((category, index) => {  
  return (  
    <li  
      className={`p-3 nav-item ${  
        activeCategory === category.name ? "bg-warning" : ""  
      }}  
      key={index}  
      onClick={() => setActiveCategory(category.name)}  
    >  
      {category.name}  
    </li>  
  );  
})}
```

If you test the application, you’ll see that the header already change the color of the active category when clicked. However, the transactions displayed both in the table and the chart doesn’t change at all.

You need to filter the transactions based on the currently active category. Add the following function inside your `App` component:

```
const filterTransactions = () => {  
  return transactions  
    .filter((transaction) =>  
      activeCategory ? transaction.category.name === activeCategory : true  
    )  
}
```

```
.sort((a, b) => (new Date(a.date) < new Date(b.date) ? 1 : -1));  
};
```

Also, let's chain the `filter` with `sort` function so that you'll have the transactions sorted from the most recent to the oldest:

```
.filter((transaction) =>  
  activeCategory ? transaction.category.name === activeCategory : true  
)  
.sort((a, b) => (new Date(a.date) < new Date(b.date) ? 1 : -1));
```

Finally, call the function when you pass `transactions` into `TransactionTable` and `Chart` components:

```
<TransactionTable  
  setShowAddTransaction={setShowAddTransaction}  
  removeTransaction={removeTransaction}  
  transactions={filterTransactions(transactions)}  
</>  
  
<Chart transactions={filterTransactions(transactions)} />
```

Now the transactions will be filtered accordingly when you click on the header list. One thing I noticed is that the mouse cursor doesn't change to `pointer` when you hover on the header element. You can fix this by adding a custom css to the `.nav-item` class:

```
.nav-item {  
  cursor: pointer;  
}
```

Here's the [complete app demo](#) again.

Part 3: React component logics

Now that you know how to write components in both `class` and `function` syntax, it's time to learn about logics that you can apply to React components. This part will include the most common tasks you will do when creating applications with React. You will learn about the following topics:

- How to make an AJAX call in React
- Using Axios library to create a server request
- Conditionally rendering component output
- Checking component props with `PropType`
- Context API and the `useContext` hook
- Passing data from child component to parent component

When you're done with this part, you'll be ready to build even more advanced React applications. Let's start by learning how to write AJAX calls.

Making AJAX calls in React

One of the most asked questions by people learning React is:

How do I code an AJAX call in React?

This is a valid question because modern web-based applications tend to have a modular architecture, where the back-end is separated from the front-end. The front-end app will need to access data from a remote end-point or server.

But here's the interesting point: React doesn't tell you how you should send your AJAX calls. The library only focuses on rendering UI with data from state and props. This means you can use any AJAX-based libraries to fetch your data.

Some of the most popular libraries include: * [Fetch](#) * [Axios](#) * [Axios React](#) * [Superagent](#)

These HTTP request libraries have a slight of different features, but it really doesn't matter which one you choose.

Once you've chosen a library to use with your project, let's learn about when you should make an HTTP request.

Make AJAX calls when the component has been mounted

AJAX calls in React should be done **right after your component has been rendered into the browser for the first time**. This is so you can update your state when the data has been retrieved from the end-point.

If you write components using the class approach, you place your AJAX request in `componentDidMount()` lifecycle function. When the data is retrieved, you will need to assign it to state with `setState` function.

Here's an example of fetching GitHub users data from its API. Notice how the code is written inside `componentDidMount()` function:

```
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      data: []
    };
  }

  componentDidMount() {
    fetch("https://api.github.com/users?per_page=3")
      .then((res) => res.json())
      .then(
        (data) => {
          this.setState({
            data: data
          });
        },
        (error) => {
          console.log(error)
        }
      );
  }
}
```



```
}

render() {
  const { data } = this.state;
  return (
    <div className="App">
      <h1>React AJAX call</h1>
      <ul>
        {data.map((item) => (
          <li key={item.id}>{item.login}</li>
        ))}
      </ul>
    </div>
  );
}
```

When you use the function approach, you place the request inside `useEffect()` hook with an empty array `[]` as its second parameter, so that **the hook will run only once at first render**.

The code below is the same as the above:

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

export default function App() {
  const [data, setData] = useState([]);

  useEffect(() => {
    fetch("https://api.github.com/users?per_page=3")
      .then((res) => res.json())
      .then(
        (data) => {
          setData(data);
        },
        (error) => {
          console.log(error);
        }
      );
  }, []);
}
```

```
return (  
  <div className="App">  
    <h1>React AJAX call</h1>  
    <ul>  
      {data.map((item) => (  
        <li key={item.id}>{item.login}</li>  
      ))}  
    </ul>  
  </div>  
)  
);  
}
```

While React will execute both examples above without any errors, you might notice that there's a slight delay before React renders the data that you're fetching from the remote endpoint. You can clarify this by adding a delay before your AJAX call is executed:

```
componentDidMount() {  
  setTimeout(() => {  
    fetch("https://api.github.com/users?per_page=3")  
      .then((res) => res.json())  
      .then(  
        (data) => {  
          this.setState({  
            data: data  
          });  
        },  
        (error) => {  
          console.log(error);  
        }  
      )  
    );  
  }, 5000);  
}
```

You can do the same with the function component:

```
useEffect(() => {
  setTimeout(() => {
    fetch("https://api.github.com/users?per_page=3")
      .then((res) => res.json())
      .then(
        (data) => {
          setData(data);
        },
        (error) => {
          console.log(error);
        }
      );
  }, 5000);
}, []);
```

It will be so much better when the application can tell your users that the data is still being retrieved. You can do this by adding a state that returns `true` when the data is loading and `false` when the data has finished loading. Notice how `isLoading` state is added in the following example:

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

export default function App() {
  const [data, setData] = useState([]);
  const [isLoading, setLoading] = useState(true);

  useEffect(() => {
    fetch("https://api.github.com/users?per_page=3")
      .then((res) => res.json())
      .then(
        (data) => {
          setData(data);
          setLoading(false);
        },
        (error) => {
          console.log(error);
          setLoading(false);
        }
      );
  });
}
```

```
    }  
    );  
  }, []);  
  
  if (isLoading) {  
    return <h1>Loading data... </h1>;  
  } else {  
    return (  
      <div>  
        <h1>React AJAX call</h1>  
        <ul>  
          {data.map((item) => (  
            <li key={item.id}>{item.login}</li>  
          ))}  
        </ul>  
      </div>  
    );  
  }  
}
```

Finally, you can also render an error message when your AJAX call returns an error. You just need to add another state for storing the error:

```
import React, { useState, useEffect } from "react";  
  
export default function App() {  
  const [data, setData] = useState([]);  
  const [isLoading, setLoading] = useState(true);  
  const [error, setError] = useState(false);  
  
  useEffect(() => {  
    fetch("https://api.github.com/users?per_page=3")  
      .then((res) => res.json())  
      .then(  
        (data) => {  
          setData(data);  
          setLoading(false);  
        },  
        (error) => {  
          setError(error);  
          setLoading(false);  
        }  
      )  
  })  
}
```

```
    );  
  }, []);  
  
  if (error) {  
    return <div>Fetch request error: {error.message}</div>;  
  } else if (isLoading) {  
    return <h1>Loading data... </h1>;  
  } else {  
    return (  
      <div>  
        <h1>React AJAX call</h1>  
        <ul>  
          {data.map((item) => (  
            <li key={item.id}>{item.login}</li>  
          ))}  
        </ul>  
      </div>  
    );  
  }  
}
```

See a [working codesandbox example here](#).

As you can see in the examples above, React is a library that focuses on rendering user interface to the browser through components. It doesn't even have a mechanism to create a request.

The way to make AJAX calls (or HTTP API requests) in React is to put your own request code into the `componentDidMount()` function or creating a `useEffect()` hook that gets executed only once after the component has been rendered to the browser.

Next, let's look at how to use Axios library inside React.

Using Axios in React

Axios is a JavaScript library for creating HTTP requests. It's similar to the native `fetch` API, but has more useful features, including:

- Make XMLHttpRequests from the browser
- Make http requests from node.js
- Supports the Promise API
- Intercept request and response
- Transform request and response data
- Cancel requests
- Automatic transforms for JSON data
- Client side support for protecting against XSRF

Generally, Axios is more popular with JavaScript developers because it automatically converts JSON response into a valid JavaScript array or object and can be used both in the server and the browser (`fetch` is not native to Node.js)

Using Axios in React

First, you need to install the library using NPM:

```
npm install axios
```

Just like using Fetch, you need to put your Axios request inside the `componentDidMount()` function or a `useEffect` hook because the ideal place to fetch data with React.

Here's a sample code for class-based component:

```
componentDidMount() {  
  axios.get("https://api.github.com/users?per_page=3")  
    .then(  
      (response) => {  
        this.setState({  
          data: response.data  
        });  
      }  
    )  
    .catch(error => {  
      console.log(error)  
    })  
}
```

And here's a sample for function-based component:

```
useEffect(() => {  
  axios.get("https://api.github.com/users?per_page=3")  
    .then((response) => {  
      setData(response.data);  
    })  
    .catch((error) => {  
      console.log(error);  
    });  
}, []);
```

Requests example with Axios

Axios supports all HTTP request methods. you can specify the request method using the axios object configuration:

```
axios({
  method: 'get',
  url: 'https://api.github.com/users'
})
.then(function (response) {
  console.log(response)
});

// or

axios({
  method: 'post',
  url: 'https://api.github.com/users',
  data: {
    firstName: 'Fred',
    lastName: 'Flintstone'
  }
})
.then(function (response) {
  console.log(response)
});
```

Or using function aliases by calling the right method:

```
axios.get("https://api.github.com/users")
axios.post("https://api.github.com/users")
axios.put("https://api.github.com/users")
axios.patch("https://api.github.com/users")
axios.delete("https://api.github.com/users")
```

You can put the request `data` argument right after the `url` :


```
axios.post("https://api.github.com/users", {  
  firstName: 'Fred',  
  lastName: 'Flintstone'  
})
```

Executing multiple concurrent requests

You can use `Promise.all()` function with Axios to wait for multiple requests to return a response before your JavaScript app execute the next line of code:

```
function getUserAccount() {  
  return axios.get('/user/12345');  
}  
  
function getUserPermissions() {  
  return axios.post('/user/12345/permissions', {  
    firstName: 'Fred',  
    lastName: 'Flintstone'  
  });  
}  
  
Promise.all([getUserAccount(), getUserPermissions()])  
  .then(function (results) {  
    const acct = results[0];  
    const perm = results[1];  
  });
```

Axios will return the response as an array, following the order of the functions you put as the arguments in `Promise.all()` function.

For more information, you can read [Axios documentation](#)

React Router

React Router is a third party library created to solve the problem of routing in React app. It wraps around the browser history API and does the job of keeping your React application UI in sync with the browser's URL.

That means when you go to `/about` page, React Router will ensure that the `About` page will be displayed on the screen. Traditional web applications are build in the server, But React applications are build in the browser. React Router simply gives you the ability to navigate on the browser, without ever sending requests to a server.

There are two packages of React Router: `react-router-dom` for React and `react-router-native` for React Native. Since you're learning about making web application, you only need to install `react-router-dom` :

```
npm install react-router-dom
```

There are 3 basic React Router components commonly used in minimal navigation, they are `BrowserRouter` , `Route` and `Link` . Let's explore about `BrowserRouter` and `Route` first:

```
import { BrowserRouter as Router, Route } from 'react-router-dom'

class RouterNavigationSample extends React.Component {
  render() {
    return (
      <Router>
        <>
          <NavigationComponent />
          <Route exact path="/" component={Home} />
          <Route path="/about" component={About} />
        </>
      </Router>
    )
  }
}
```

The `BrowserRouter` , which is imported as `Router` , acts as the parent component that wraps all of your React component. It will intercept Browser request URL and match its path with the corresponding `Route` component. So if the browser URL is `localhost:3000/about` , the `Router` will take that information and then look for a `Route` component that has the `path` attribute of `/about` .

You will determine what will be rendered by adding the `component` attribute to `Route` .

In the sample above, an `exact` attribute is added to the default `Route` path (`/`), because without it, any route with `/` will also render the `Home` component, causing inconsistencies in the navigation.

The third component `Link` is used for navigation, replacing the regular `<a>` tag of HTML. This is because a regular HTML anchor tag will do

a full refresh of the browser on click, which is not suited for React application. A React app only needs to update the URL, browser history and component rendered without any browser refresh:

```
import { Link } from "react-router-dom";

class NavigationComponent extends React.Component {
  render() {
    return (
      <>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/about">About page</Link>
          </li>
        </ul>
        <hr />
      </>
    );
  }
}
```

You can try a working demo [here](#).

Note how you can use the previous and next button of the browser's navigation bar, and the url is updated with subsequent navigation, without the browser loading. This is the power of React Router in action.

Making dynamic routing

You've seen how to create simple navigation using React Router, yet most web application require more advanced function than that. You probably

need a dynamic routing, where you can put something like `/user/:id` , in which React needs to render something based on the value of `:id` .

Old links can also be dead and need to be redirected to new link.

Also, if the browser URL doesn't match any existing route, you need to display a 404 page.

That's why you need to learn about 2 more components, `Switch` and `Redirect` . `Switch` is a unique component that will render the first matching `Route` , then stop. To illustrate this example:

```
import { Route } from 'react-router'

<Route path="/about" component={About}/>
<Route path="/:user" component={User}/>
<Route component={NoMatch}/>
```

In the above code, a browser URL of `/about` will match all three routes, causing it all to be rendered and stacked below each other. Now by using the `Switch` component, React Router will render the `About` component route and then stop.

```
import {Switch, Route} from 'react-router';

<Switch>
  <Route path='/about' component={About} />
  <Route path='/:user' component={User} />
  <Route component={NoMatch} />
</Switch>;
```

The order of the `Route` component inside `Switch` is important, so make sure you declare all static route first before declaring routes with url parameter and 404 route.

Now for `Redirect`, the component is pretty simple. You only need to add `from` attribute that states the old URL and `to` attribute specifying the new URL to link to.

```
import {Redirect} from 'react-router';  
  
<Redirect from='/old-match' to='/will-match' />;
```

Nested route

In order to create nested route, you need to declare another `Route` inside the parent component. For example, let's say you have `/users` route that render to Users component.

Let's do a little exercise. First, create an array of objects that store user data, the following will do:

```
const users = [  
  {  
    id: '1',  
    name: 'Nathan',  
    role: 'Web Developer',  
  },  
  {  
    id: '2',  
    name: 'Johnson',  
    role: 'React Developer',  
  },  
];
```

```
    },  
    {  
      id: '3',  
      name: 'Alex',  
      role: 'Ruby Developer',  
    },  
  ],  
];
```

Now create a simple routing in the application:

```
class RouterNavigationSample extends React.Component {  
  render() {  
    return (  
      <Router>  
        <>  
          <NavigationComponent />  
          <Route exact path="/" component={Home} />  
          <Route path="/about" component={About} />  
          <Route path="/users" component={Users} />  
        </>  
      </Router>  
    );  
  }  
}
```

The NavigationComponent is where you write the `Link` component for navigating the application:

```
class NavigationComponent extends React.Component {  
  render() {  
    return (  
      <>  
        <ul>  
          <li>  
            <Link to="/">Home</Link>  
          </li>  
          <li>  
            <Link to="/about">About page</Link>  
          </li>  
        </ul>  
      </>  
    );  
  }  
}
```

```
        </li>
      <li>
        <Link to='/users'>Users page</Link>
      </li>
    </ul>
    <hr />
  </>
);
}
```

It's time to create components to render on specific routes. `Home` and `About` component will render a single div, while `Users` will have another `Link` and `Route` component.

Inside the `Users` component, you will render a list of users, with a *nested route* to individual user by its ID, like `/users/:id` :

```
const Home = () => {
  return <div>This is the home page</div>;
};

const About = () => {
  return <div>This is the about page</div>;
};

const Users = () => {
  return (
    <>
      <ul>
        {users.map(({name, id}) => (
          <li key={id}>
            <Link to={`/users/${id}`}>{name}</Link>
          </li>
        ))}
      </ul>
      <Route path='/users/:id' component={User} />
      <hr />
    </>
  );
};
```



```
);  
};
```

There's nothing new with this code. So you can write the `User` component now:

```
const User = ({match}) => {  
  const user = users.find((user) => user.id === match.params.id);  
  
  return (  
    <div>  
      Hello! I'm {user.name} and I'm a {user.role}  
    </div>  
  );  
};
```

Now here is something new I haven't told you about. Every time a component is rendered in a specific route, the component receive route props from React Router. There are 3 route props being passed down into component: `match` , `location` , `history` .

You can look at the props by opening the React Developer Tools and highlight the matching component route:

```
{{<postimage "route-props.png" "Route props" >}}
```

(If you're opening from Codesandbox, you can open the demo in a new separate window to enable React DevTool)

Notice how you add `/:id` URL parameter in the `Users` component nested route. This id is passed down to the `User` component through

the `match.params.id` object property. If you passed the URL parameter as `/:userId`, it will be passed down as `match.params.userId`.

Now that you know about route props, let's refactor `Users` component a bit:

```
const Users = ({ match }) => {
  return (
    <>
      <ul>
        {users.map(({ name, id }) => (
          <li key={id}>
            <Link to={`/${match.url}/${id}`}>{name}</Link>
          </li>
        ))}
      </ul>
      <Route path={`/${match.url}/:id`} component={User} />
      <hr />
    </>
  );
}
```

As always, [here](#) is a working demo.

Passing props to Route component

You might think that passing props into Route component is the same as passing into regular component:

```
<Route path="/about" component={About} user='Jelly' />
```

Unfortunately, React Router doesn't forward the props entered into `Route` component into the `component` props, so you have to use another method.

Fortunately, React Router provides a `render` attribute that accepts a function to be called when the URL locations matches. This props also receives the same `route props` as the `component` props:

```
<Route
  path="/about"
  render={props => <About {...props} admin="Bean" />}
/>

// the component
const About = props => {
  return <div>This is the about page {props.admin}</div>;
};
```

First, you take the given `props` from React Router and pass it into the component, so that the component can use `match` , `location` or `history` props if necessary. Then you add your own extra props into it. The example above use arbitrary `admin` props as example.

You can see the [Code Sandbox demo here](#).

Now that you've learned about React Router features, let's learn about React Router hooks next.

React Router hooks

With the addition of hooks, React Router team has also developed its own hooks that you can use in any React function components. They will help you to write clean, neatly stacked navigation components for your application.

There are four hooks introduced by React Router:

- `useParams`
- `useLocations`
- `useHistory`
- `useRouteMatch`

`useParams` hook

The `useParams` hook will return an object of key/value pairs from your application URL that is set to be dynamic. In a complex application, it's common to have many navigation links that are dynamic.

For example, you may have a `/post/:id` URL that also initiates a fetch process to your application's backend. In this case, the most common React Router pattern would be to use a component prop.

```
export default function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/post/hello-world">First Post</Link>
            </li>
          </ul>
        </nav>
        <Switch>
          <Route path="/post/:slug" component={Post} />
          <Route path="/">
            <Home />
          </Route>
        </Switch>
      </div>
    </Router>
  );
}
```

By passing the `Post` component into the `/post/:number` `Route` component, you can extract the `params` object from the `match` prop that is passed by React Router into the `Post` component.

```
// Old way to fetch parameters
function Post({ match }) {
  let params = match.params;
  return (
    <div>
      In React Router v4, you get parameters from the props.
      Current parameter
      is <strong>{params.slug}</strong>
    </div>
  );
}
```

This method works just fine, but it's quite cumbersome if you have a big application with lots of dynamic routes. You have to keep track of which `Route` components need component props and which do not. Also, since the `match` object is passed from `Route` into the rendered component, you will need to pass them along to components further down the DOM tree.

This is where the `useParams` hook really shines. It's a neat helper function that gives you the parameters of the current route so you don't have to use the component props pattern.

```
<Switch>
  <Route path="/post/:slug" component={Post} />
  <Route path="/users/:id/:hash">
    <Users />
  </Route>
  <Route path="/">
    <Home />
  </Route>
</Switch>

function Users() {
  let params = useParams();
  return (
    <div>
      In React Router v5, You can use hooks to get parameters.
      <br />
      Current id parameter is <strong>{params.id}</strong>
      <br />
      Current hash parameter is <strong>{params.hash}</strong>
    </div>
  );
}
```

If a child component of `Users` need access to the parameters, you can

simply call `useParams()` there as well.

Here's an [Code Sandbox example](#) if you'd like to see it in action.

useLocation hook

In React Router version 4, just like fetching parameters, you had to use the component props pattern to gain access to a `location` object.

```
<Route path="/post/:number" component={Post} />

function Post(props) {
  return (
    <div>
      In React Router v4, you get the location object from props.
      <br />
      Current pathname: <strong>{props.location.pathname}</strong>
    </div>
  );
}
```

With version 5.1, you can call the `useLocation` hook to get the `location` object from React Router.

```
<Route path="/users/:id/:password">
  <Users />
</Route>

// new way to fetch location with hooks
function Users() {
  let location = useLocation();
  return (
    <div>
      In React Router v5, You can use hooks to get location object.
      <br />
    </div>
  );
}
```

```
    Current pathname: <strong>{location.pathname}</strong>
  </div>
);
}
```

Again, you can see the sample code on [Code Sandbox](#).

useHistory hook

That gives us one less reason to use component props. But don't you still need to use the component or render pattern to get the `history` object?

The `history` object is the last reason why you need to use component or render props pattern.

```
<Route path="/post/:slug" component={Post} />

// Old way to fetch history
function Post(props) {
  return (
    <div>
      In React Router v4, you get the history object from props.
      <br />
      <button type="button" onClick={() => props.history.goBack()}>
        Go back
      </button>
    </div>
  );
}
```

By using the `useHistory` hook, you can get the same history object without needing the `Route` component to pass it down:


```
<Route path="/users/:id/:hash">
  <Users />
</Route>

// new way to fetch history with hooks
function Users() {
  let history = useHistory();
  return (
    <div>
      In React Router v5, You can use hooks to get history object.
      <br />
      <button type="button" onClick={() => history.goBack()}>
        Go back
      </button>
    </div>
  );
}
```

See this [sample code](#) for a comparison between `useHistory` and regular `history` props.

Now you can have a clean routing component without any weird component compositions that use `component` or `render` props. You can simply put a `<Route>` component with `path` props and place the rendered `children` component inside it.

useRouteMatch hook

Sometimes, you need to use the `<Route>` component just to get access to the match object:

```
<Route path="/post">
  <Post />
</Route>

function Post() {
  return (
    <Route
      path="/post/:slug"
      render={({ match }) => {
        return (
          <div>
            Your current path: <strong>{match.path}</strong>
          </div>
        );
      }}
    />
  );
}
```

The example above shows one way to access the `match` object without using the `component` or `render` props. With the latest version of React Router, you can also use the `useRouteMatch` hook, which enables you to grab the `match` object and use it inside your component without using `<Route>` component:

```
<Route path="/users">
  <Users />
</Route>

// useRouteMatch to make your route above cleaner
function Users() {
  let match = useRouteMatch("/users/:id/:hash");
  return (
    <div>
      In React Router v5, You can use useRouteMatch to get match object.
      <br /> Current match path: <strong>{match.path}</strong>
    </div>
  );
}
```

Now you won't need to create a `<Route>` component just to grab a match object. Here's [the example](#) for you to mess around with.

With the power of hooks, you're now able to share React Router logic across components without the need to pass it down from from the top of the tree.

Should you use spread attributes when passing props?

As you've probably learned, React library passes data around its components by using `props` and `state`. To pass data from one component to another, you just need to define the attributes after the component name:

```
function MainComponent(){  
  return <Hello name="Jack" />  
}
```

You can then retrieve the attributes from the `props` parameter, which always gets passed as the first argument in a function component:

```
function Hello(props){  
  return <div>Hello World! My name is {props.name}</div>  
}
```

Occasionally, you will need to pass multiple props into a component. For example:

```
function MainComponent(){  
  return <Hello firstName="Jack" lastName="Skeld" />  
}
```

When you find you need to pass multiple props, you can wrap them as an object and use [the spread operator](#) to pass the whole props object:

```
function MainComponent(){
  const props = { firstName: "Jack", lastName: "Skeld" }
  return <Hello {...props} />
}
```

This is a totally valid way to pass props in React, but should you do it?

Personally, I think the spread props pattern is a **bad way** to pass props because as you develop your React project, you will have a hard time tracking what props are actually being passed from one component to another.

What if the props you passed came from state that gets its values from an API call?

```
function App() {
  const [data, setData] = useState([]);

  useEffect(() => {
    fetch("https://api.github.com/users?per_page=3")
      .then((res) => res.json())
      .then(
        (data) => {
          setData(data);
        }
      );
  }, []);

  return (
    <Hello {...data} />
  );
}
```

You'll be confused as to what exactly is being passed from `data` state into `<Hello>` component. The spread props method is hard to maintain

when you have more than ten components around, as you need to look for what actually is being passed into the component.

It's always much better to **explicitly define** the attributes you're passing into the component rather than using the spread method.

Using propTypes and defaultProps

As you develop your React application, sometimes you might need a prop to be structured and defined to avoid bugs and errors. In the same way a `function` might require mandatory arguments, a React component might require a prop to be defined if it is to be rendered properly.

Consider the following example. Notice how the `Greeting` component use certain props in the `return` statement:

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

function App() {
  return <Greeting name="Nathan" />;
}

function Greeting(props) {
  return (
    <p>
      Hello! I'm {props.name},
      a {props.age} years old {props.occupation}.
      Pleased to meet you!
    </p>
  );
}

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

While `props.age` and `props.occupation` are undefined in the `Greeting` component above, React will simply ignore the expression to call on their value and render the rest of the text. It doesn't trigger any

error, but you know you can't let this kind of thing go unaddressed in your projects.

This is where propTypes can help you. PropTypes is a special component property that can be used to validate the props you have in a component. It's a separate, optional NPM package, so you need to install it first before using it:

```
npm install --save prop-types
```

Now let's make required props in the `Greeting` component:

```
import React from "react";
import ReactDOM from "react-dom";
import PropTypes from "prop-types";

function App() {
  return <Greeting name="Nathan" />;
}

function Greeting(props) {
  return (
    <p>
      Hello! I'm {props.name},
      a {props.age} years old {props.occupation}.
      Pleased to meet you!
    </p>
  );
}

Greeting.propTypes = {
  // name must be a string and defined
  name: PropTypes.string.isRequired,
  // age must be a number and defined
  age: PropTypes.number.isRequired,
  // occupation must be a string and defined
  occupation: PropTypes.string.isRequired
}
```



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

With the `propTypes` property declared, the `Greeting` component will throw a warning to the console when its props aren't passing `propTypes` validation.

You can also define default values for props in cases where props are not being passed into the component on call by using another special property called `defaultProps`. Just add it below the `propTypes` declaration:

```
Greeting.propTypes = {  
  name: PropTypes.string.isRequired,  
  age: PropTypes.number.isRequired,  
  occupation: PropTypes.string.isRequired  
};  
  
Greeting.defaultProps = {  
  name: "Nathan",  
  age: 27,  
  occupation: "Software Developer"  
};
```

And now the default values in `defaultProps` will be used when you call the component without props. Here's a [codesandbox example](#) you can play with.

Passing data from child components to parent components

A parent component is any component that calls other components in its code block, while a child component is simply a component that gets called by a parent component. A parent component passes data down to child components using props.

You might wonder, “how can I pass data up from a child component to a parent component?”

The answer is that it’s not possible at least not directly. But here’s the thing in React: you can also pass a `function` as props.

How is that relevant to the question? Notice how two props are being passed into the `Greeting` component below:

```
function App() {
  const [textSwitch, setTextSwitch] = useState(true);
  return (
    <div>
      <Greeting
        isTrue={textSwitch}
        handleClick={() => setTextSwitch(!textSwitch)} />
    </div>
  );
}

function Greeting(props) {
  let element = (
    <p>Howdy! I'm Jane.</p>
  );
  if (props.isTrue) {
```

```
    element = (  
      <p>  
        Hello! I'm Nathan.  
      </p>  
    );  
  }  
  return (  
    <>  
      {element}  
      <button onClick={() => props.handleClick()}>  
        Toggle Name  
      </button>  
    </>  
  )  
}
```

Here's a working [Code Sandbox](#) example.

In the example above, the `App` component is sending the `handleClick` prop, which has the function to change the state, into the `Greeting` component.

The `Greeting` component will call the `handleClick` function when the button is clicked, causing `App` to execute the function.

When the state in `App` is updated, React re-renders the view, and the new state value is then sent to `Greeting` through `isTrue` prop.

So yes, React can't send data up from a child component into its parent component, but the parent component can send a function to a child component. Knowing this, you can send a function that updates state into the child component, and once that function is called, the parent component will update the state.

You can't send data, but you can send a signal for change using the state update function.

Third project: Github User Finder


It's time to create your third React application, and this time you're going to use Github's open API to create an application that can search Github users based on their username.

This application will have two pages. The first page will display a single input to search for Github users and display the users found as a list of items:

Github Users Finder

Search users in GitHub using this simple app.

Click on the card to see more detail about individual user. The search default is nsebhashtian (me!)



Username : nsebhashtian

Url : <https://github.com/nsebhashtian>

Score : 1

Figure 18: Github app search page

When you click on one of the users, you'll be taken into the second page which displays more detail about a certain user, like this:

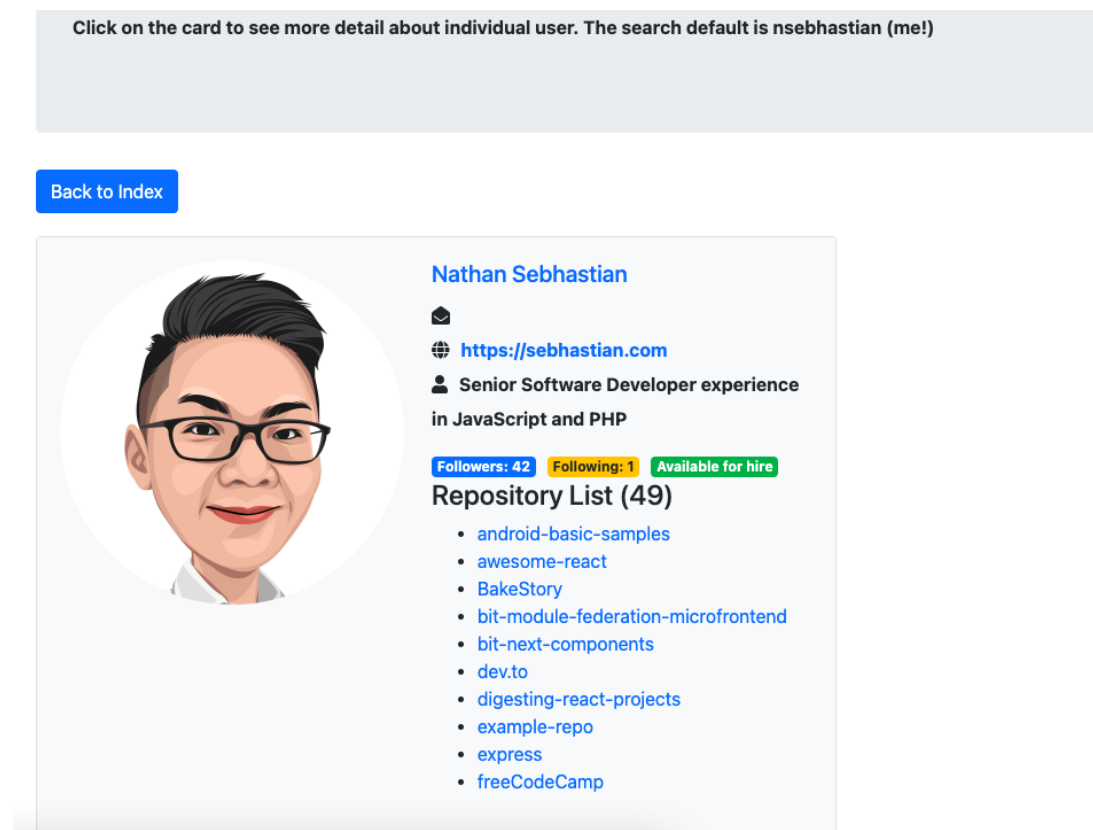


Figure 19: Github user profile page

This application will give you more practical experience with React as you interact with remote data source to fetch and display data. You're also going to use React Router to render the two pages.

You also need to use the following Github API route to fetch your data:

- Github [search users](#)
- Github [get a user](#)
- Github [list repositories for a user](#)

Here's the [demo of the complete app](#). Can you guest how many components you need to write to build this application?

Here are the libraries you need to install for this project:

- axios
- react-router-dom
- bootstrap
- @fortawesome/fontawesome-free

By this point, I'm sure you already understand how to install these dependencies. Without further ado, let's start composing the components.

The components composition

Based on the demo from the previous section, you'll need to write three components to build the application:

- One component for the big header
- One for the search page
- One for the profile page

These composition seems fine, but on further inspection, it seems that the search page can further be broken down between the search form and the results, which is a list of users. Those smaller components are similar to the wizard form steps component.

Writing the app structure

Just like with previous components, you're going to write a minimal application structure before actually writing your component. This way, you'll get the underlying mechanics of your app in place.

First, create the `App.js` file with the following content:

```
import React from "react";
import { BrowserRouter as Router, Route } from "react-router-dom";

import Header from "../components/Header";
import Search from "../components/Search";
import Profile from "../components/Profile";
```



```
export default function App() {  
  return (  
    <Router>  
      <div className="container">  
        <Header />  
        <Route exact path="/">  
          <Search />  
        </Route>  
        <Route path="/user/:username">  
          <Profile />  
        </Route>  
      </div>  
    </Router>  
  );  
}
```

Because the `Header` component will be shared between both routes, it's being called outside of React Router's `Route` component. The `Search` component and the `Profile` component will be rendered to the screen according to the current URL of the application.

Let's write the components inside the `directory` folder. You just need to return a one-liner `<h1>` element from the components:

```
import React from "react";  
  
export default function Header(){  
  return <h1>Header component</h1>  
}
```

Do the same for `Search` and `Profile` component and your application structure is now set.

Writing the Header component

Next, let's write the `Header` component. Open your code editor and create the `Header.js` file inside the `components` directory. Here's the content of the component:

```
import React from "react";

export default function Header() {
  return (
    <div className="jumbotron">
      <h1>Github Users Finder</h1>
      <h2>Search users in GitHub using this simple app.</h2>
      <p>
        Click on the card to see more detail about individual user.
        The search default is nsebbastian (me!)
      </p>
    </div>
  );
}
```

It's just a simple static component with no state or props.

Writing the CSS

Next, let's write the CSS needed for this application. Although you're using Bootstrap for this project, there is a custom CSS code included in order to prettify the interface:

```
.search-bar {  
  margin: 20px;  
  text-align: center;  
}  
  
.search-bar input {  
  width: 75%;  
}  
  
.jumbotron {  
  padding: 4rem 2rem;  
}  
  
img.user {  
  height: 85px;  
  border-radius: 50%;  
  margin-right: 10px;  
  vertical-align: middle;  
  float: left;  
}  
  
a:hover,  
a:active,  
a:focus {  
  text-decoration: none;  
}  
  
.input-group {  
  margin: 0 auto;  
}  
  
.bs-callout {  
  padding: 20px;  
  margin: 20px 0;  
  border: 1px solid #eee;  
  border-left-width: 5px;  
  border-radius: 3px;  
}  
  
.bs-callout h4 {  
  margin-top: 0;  
  margin-bottom: 5px;  
}  
  
.bs-callout h4,
```

```
p {
  font-weight: bold;
}

.bs-callout p:last-child {
  margin-bottom: 0;
}

.bs-callout-info {
  border-left-color: #5bc0de;
}

.bs-callout-info:hover {
  border-left-color: #5cb85c;
  color: #5cb85c !important;
}

.bs-callout-info:hover h4 {
  color: #5cb85c !important;
}

.bs-callout-info:hover p {
  color: #5cb85c !important;
}

.bs-callout-info h4 {
  color: #5bc0de;
}

.fas {
  margin-bottom: 10px;
  margin-right: 10px;
}

.container {
  padding-top: 30px;
}

small {
  display: block;
  line-height: 1.428571429;
  color: #999;
}
```

In the next section, you will start writing the search page components.

Writing the search page

It's time to replace the placeholder of your `Search` component. This component will need to do the following:

- A search form
- Calling Github API when search form is submitted
- Display the response data in a list
- Link to profile page for each returned user

Let's start by declaring the `API` constant, which will serve as the base URL when calling the Github API:

```
import React, { useState, useEffect } from "react";
import { Link } from "react-router-dom";
import axios from "axios";

const API = "https://api.github.com/";
```

Then write the state values you need to use for this component. You'll need two states:

- `users` for keeping the response returned by Github
- `keyword` for storing the input element value:

```
export default function Search() {
  const [users, setUsers] = useState(false);
  const [keyword, setKeyword] = useState("nsebastian");
```

Next, write the function to send request to Github. Let's name this function `fetchSearch` :

```
const fetchSearch = (keyword) => {  
  let url = `${API}search/users?q=${keyword}&per_page=10`;  
  
  axios  
    .get(url)  
    .then((response) => {  
      setUsers(response.data);  
    })  
    .catch((error) => {  
      console.log("Oops! Fetching failed:", error);  
      setUsers(false);  
    });  
};
```

To make your application immediately fetch users using the default `keyword` value, you need to write a `useEffect` hook:

```
useEffect(() => {  
  fetchSearch(keyword);  
}, []);
```

React will complain that the hook has a missing dependency, but if you include the `keyword` variable into the dependency array, your application will call the function each time the `keyword` value is changed. Since the app needs to fetch only when the user trigger a submit event, you can safely ignore this warning by disabling ESLint for that line:

```
useEffect(() => {  
  fetchSearch(keyword);  
  // eslint-disable-next-line  
}, []);
```

Next, let's write the `return` statement of the component:

```
return (  
  <>  
    <SearchForm  
      keyword={keyword}  
      setKeyword={setKeyword}  
      fetchSearch={fetchSearch}  
    />  
    <UsersList users={users} />  
  </>  
);
```

Let's write the `SearchForm` component next so the application can have input from users.

Writing the SearchForm component

This component is simply a one input form, so you can write it like this:

```
function SearchForm({ keyword, setKeyword, fetchSearch }) {  
  
  const handleSubmit = (e) => {  
    e.preventDefault();  
    fetchSearch(keyword);  
  };  
  
  return (  
    <div className="search-bar">  
      <form className="input-group"
```



```
    onSubmit={handleSubmit}>
    <input
      className="form-control"
      placeholder="Type keyword and press Enter"
      value={keyword}
      onChange={(e) => setKeyword(e.target.value)}
    />
    <span className="input-group-btn">
      <button
        type="submit"
        className="btn btn-primary">
        Submit
      </button>
    </span>
  </form>
</div>
);
}
```

The `classNames` are simply assigned for styling the component, but it's just a simple form that calls on `fetchSearch` function on submit.

Writing the UsersList component

This component will render conditionally based on the value of `users` prop being passed into it. First, it will return “fetching data . . .” when `users` value is false:

```
function UsersList({ users }) {
  if (users) {
    // will be filled later ...
  } else {
    return <div>Fetching data . . .</div>;
  }
}
```

Next, if Github returns “Not Found” message, the app will ask for another keyword input:

```
function UsersList({ users }) {
  if (users) {
    if (users.message === "Not Found")
      return (
        <div className="notfound">
          <h2>Oops !!</h2>
          <p>
            The API couldn't find any user.
            Try again with a different keyword
          </p>
        </div>
      );
    } else {
      return <div>Fetching data . . .</div>;
    }
  }
}
```

Finally, when Github returns any user data, the app will display those users as a list of items. You need to use the `map` function to loop over the `users` data. Notice how the `<Link>` component from `react-router-dom` is also used to navigate to the `user/:username` route:

```
function UsersList({ users }) {
  if (users) {
    if (users.message === "Not Found"){
      // code omitted ...
    }
    else {
      let userList = users.items.map(function (user) {
        return (
          <Link key={user.id} to={"user/" + user.login}>
            <div className="bs-callout bs-callout-info">
```

```
        <img
          className="user"
          alt="User profile"
          src={user.avatar_url} />
        <h4>Username : {user.login}</h4>
        <p> Url : {user.html_url}</p>
        <p> Score : {user.score} </p>
      </div>
    </Link>
  );
});
return <div>{userList}</div>;
}
} else {
  // code omitted ...
}
}
```

And with that, you've finished the `Search` component! Here's a [Code Sandbox link](#) for you to inspect and compare with your code.

Let's write the profile component next.

Writing the Profile component

With the `Search` component done, you just need to write the `Profile` component, which gets rendered when a user clicks on one of the search results.

Just like the previous component, you're going to start with writing the `useState` hooks. In addition to the states, you also need to write the `useParams` hook from `react-router-dom` to fetch the `username` passed from the `:username` route parameters:

```
import React, { useState, useEffect } from "react";
import { Link, useParams } from "react-router-dom";
import axios from "axios";

const API = "https://api.github.com/";

export default function Profile() {
  const params = useParams();
  const [user, setUser] = useState("");
  const [repos, setRepos] = useState([]);
```

Next, write a function that will fetch the user detail:

```
const fetchUser = (username) => {
  let url = `${API}users/${username}`;
  axios
    .get(url)
    .then((response) => {
      setUser(response.data);
    })
    .catch((error) => {
      console.log(error);
      setUser(false);
    });
}
```

```
    });  
  };  
};
```

And another one for the repositories owned by that user:

```
const fetchRepo = (username) => {  
  let url = `${API}users/${username}/repos?per_page=10`;  
  axios  
    .get(url)  
    .then((response) => {  
      setRepos(response.data);  
    })  
    .catch((error) => {  
      console.log(error);  
      setUser(false);  
    });  
};
```

Now you need to write another `useEffect` hook that calls both functions:

```
useEffect(() => {  
  fetchUser(params.username);  
  fetchRepo(params.username);  
}, [params]);
```

In reality, the hook will run only once on render because the `params` variable won't change at all. But let's put the `params` inside dependency array anyway to stop the missing dependency warning.

It's time to write the `return` statement. Just like the `Search` component, you need to check the value of `user` prop and conditionally renders a different element:

```
if (user) {  
  
  // will be filled after this ...  
  
} else {  
  return <div>Please wait . . .</div>;  
}
```

Once the API returns any user, you're going to map over the `repos` state value and render each repository name into a list. If the `repos` value is still empty, then the component will return nothing. You don't need to check on the value of `repos` because it's okay for the component to render the rest of the profile first:

```
if (user) {  
  const repoList = repos.map(function (repo) {  
    return (  
      <li key={repo.id}>  
        <a href={repo.html_url}>{repo.name}</a>  
      </li>  
    );  
  });  
}
```

Once the `repos` state return any data, React will re-render the component automatically. All you need to do now is to write the `return` statement. You can write the UI as detailed or as simple as you want:

```
return (  
  <div className="card w-75 mb-2">  
    <div className="card-body d-flex flex-row bg-light">  
      <div>  
        <img
```

```
        className="mr-4"
        width="300"
        alt="User"
        src={user.avatar_url}
      />
    </div>
    <div>
      <h5 className="card-title">
        <a href={user.html_url}>{user.name}</a>
      </h5>
      <h4>Repository List ({user.public_repos}) </h4>
      <ul>{repoList}</ul>
    </div>
  </div>
</div>
);
```

And with that, you've finished the application! You can see the [online demo at Code Sandbox](#). If you see some icons missing, it may be because of Font Awesome encoding, so please download the local copy from the [Github repository](#) for this app.

Part 4: Context API

Congratulations for coming this far! You're almost finished with this book. You've learned the fundamentals of React and how to build applications using components, state, props, and third party React libraries.

In this part, you're going to learn about React's Context API and how it can help you to handle complex application by removing the need to pass props around.

After that, we will wrap up with another project that helps you to get a practical experience with the Context API.

Let's begin.

The Prop drilling pattern

Prop drilling (or also called “threading”) is the code pattern you create when you need to get data from one component into another by passing props multiple times through other components.

For example, imagine you have the following components in your React application:

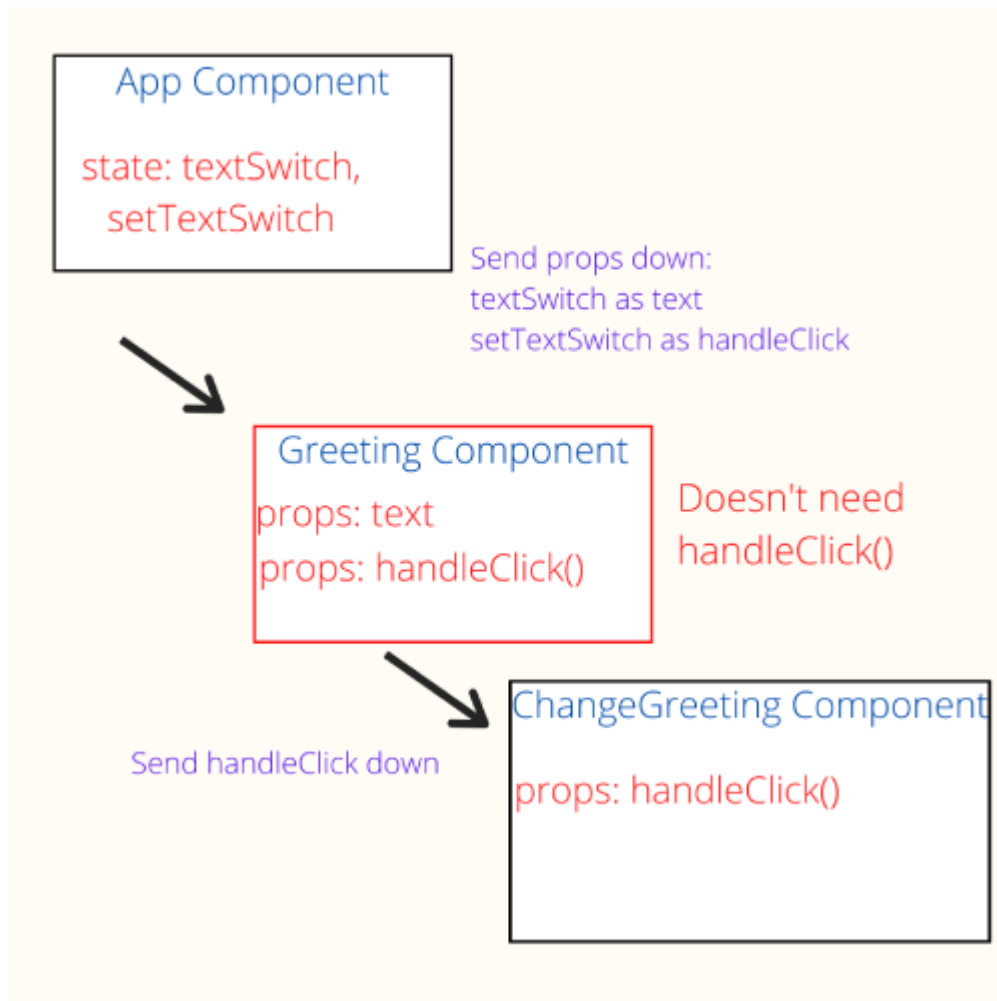


Figure 20: Prop drilling on paper

Here's an example you can play with in [Code Sandbox](#).

The pattern where you send setTextSwitch from `App` to `ChangeGreeting` is called prop drilling. The component `Greeting` doesn't need the prop, but you send it anyway because you need to get it into `ChangeGreeting`.

.

To be fair, prop drilling isn't all bad because the pattern provides you with a very specific way of passing data using a top-down approach.

Imagine if you can just declare a global variable in any component and then retrieve it from any component. Without a clear pattern and one-way data flow, you'd have a very confusing data model and have a hard time tracking where some data is initialized, updated and used.

But React does acknowledge that it's no fun to pass props down multiple components from the initial component. This is why React provides a way to simulate the nature of global variable. Enter the Context API.

The Context API

React Context API is a feature that provides a way to pass data down through the component tree without needing to pass props manually between components. The value of React Context **provided** by one component can be accessed by other components by **consuming** the context.

The data you shared through Context API can be considered as “global” for a tree of React components. The Context API can be used to share the currently authenticated user, selected theme or preferred language.

Creating context is as simple as calling the `React.createContext()` function and assign it into a variable. Just like `useState` hook, you can put a default data into as the first argument of the function:

```
import React from 'react';  
  
// defaults to 'en'  
const LanguageContext = React.createContext('en')
```

Now we have the `LanguageContext` available for use. This also gives you the `LanguageContext.Provider` and `LanguageContext.Consumer` component. Let's learn how to provide data value into the context first.

Providing context

The value provided into context is usually obtained from state, so that when the state change, the context value also change. You need to wrap your React component with the `Provider` component. Here's an example:

```
import React from 'react'

const LanguageContext = React.createContext('en')

function App() {
  const language = 'fr'

  return (
    <LanguageContext.Provider value={language}>
      <Hello />
    </LanguageContext.Provider>
  )
}
```

Now both `Hello` and all components under it can access the value of `LanguageContext`. There are many ways to extract the value of a context depending on your component type (function or class). Let's learn how to extract content value in function components first.

Consuming context in function components

Function components can grab the value of context by using the `useContext` hook. The hook accepts a context object — which is

returned by `React.createContext()` — and returns the value of that context.

Here's the example:

```
function Hello() {
  const language = useContext(LanguageContext)

  if (language === "fr") {
    return <h1>Bonjour!</h1>
  }
  return <h1>Hello!</h1>
}
```

In most application requirements, you will need the context value to change over time. You can use the state to change the value of context like this:

```
import React, { useState, useContext } from "react"

const LanguageContext = React.createContext()

function App() {
  const [language, setLanguage] = useState("en");

  const changeLanguage = () => {
    if (language === "en") {
      setLanguage("fr")
    } else {
      setLanguage("en")
    }
  }
};

return (
  <LanguageContext.Provider value={language}>
    <Hello />
    <button onClick={changeLanguage}>Change Language</button>
  </LanguageContext.Provider>
)
```

```
)  
}
```

Here's a working [Code Sandbox example](#). Instead of passing language down to `Hello` component as `prop`, you simply grab the context value and use it to conditionally render the output.

Consuming context in class components

In class components, you can get the value of context by using the `Context.Consumer` component. The component requires a function as its child, because it will pass the current context value into it. You then need to return a React element to be rendered on the screen:

```
class Hello extends React.Component {  
  render() {  
    return (  
      <LanguageContext.Consumer>  
        {(language) => {  
          if (language === "fr") {  
            return <h1>Bonjour!</h1>;  
          }  
          return <h1>Hello!</h1>;  
        }}  
      </LanguageContext.Consumer>  
    );  
  }  
}
```

I don't know about you, but the `Consumer` function pattern looks confusing to me! Fortunately, React team also feel the same way so they cre-

ated another way to consume context in class components by using the

`contextType` property:

```
Hello.contextType = LanguageContext
```

By assigning `contextType` to the context object, you can use

`this.context` to retrieve the context value. You can also assign the

`contextType` inside the class component as a static field. Here's the full code:

```
class Hello extends React.Component {
  static contextType = LanguageContext;
  render() {
    const language = this.context;
    if (language === "fr") {
      return <h1>Bonjour!</h1>;
    }
    return <h1>Hello!</h1>;
  }
}
```

And here's another [Code Sandbox example](#) for you to inspect.

Separating context from components

Because of the global nature of context API, you will pass its value into many components inside your application. It's very common to have one provider and many consumers of the Context API as your project grows.

As you've learned from previous sections, getting the value out of Context API requires you to reference the context object both in function and class components.

You need to organize and separate Context from component to make it more maintainable. Here's an [example project structure](#) that you can follow when you need to use context API.

Fourth project: E-commerce App

It's time to create your fourth and final project. The next application you're going to build is a part of e-commerce application UI, complete with the functionality to add new products and add those products to the cart.

Creating an e-commerce is one of the most complex project you can build, that's why to keep this project down to a level that you can finish in a few hours, You will not implement the full features of an e-commerce app.

The point of this project is to give you practical experience with React's Context API that you've learned in the previous section.

First, the application will have the Products screen, displaying a list of available products to buy:

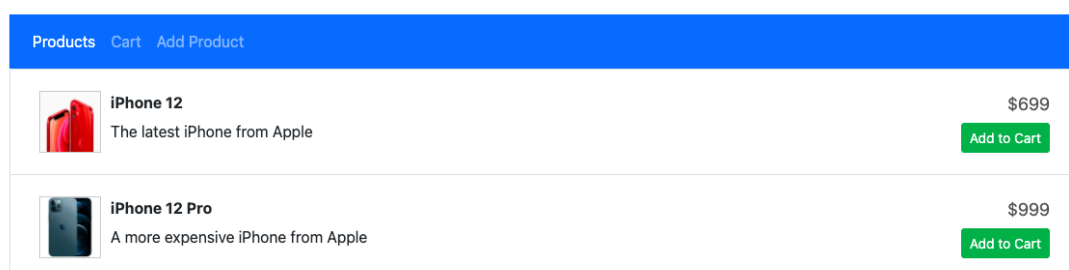


Figure 21: Products page

When “Add to cart” is clicked, React will render the Cart page:

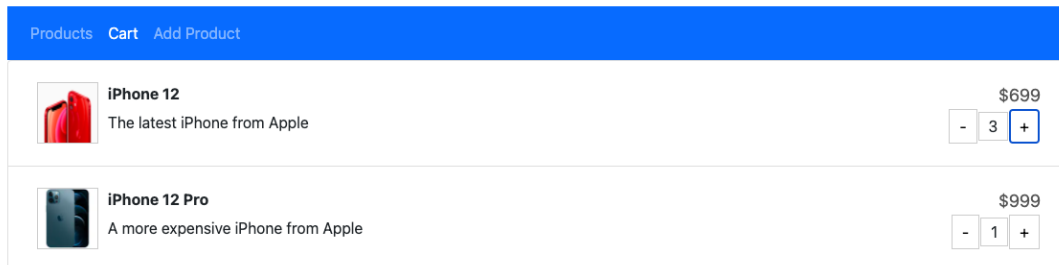


Figure 22: Cart page

Finally, the app also has the “Add Product” page to add new product into the `products` list:



Products Cart Add Product

Add New Product

Name

Price

Description

Image URL

Submit

Figure 23: Add product page

Here's the [complete application demo](#).

This application will also have the `products` and `cart` persisted.

Which means they won't disappear when you refresh the browser.

You need to install both Bootstrap and `react-router-dom` for this project, so let's install it first:

```
npm install bootstrap react-router-dom
```

In the next section, you'll start creating the application structure.

Creating the layout for the application

Based on the demo, you can split this application into four components beside the wrapper `App` component:

- One for the navigation `Header`

One for each page:

- `Products`
- `Cart`
- `AddProduct`

Let's start by writing the placeholder components for each page:

```
import React from "react"

function Products(){
  return <h1> Products Component </h1>
}
```

Since the `Header` will be used for navigation, you can write the navigation link with React Router immediately. Here's the `Header` code:

```
import React from "react";
import { NavLink } from "react-router-dom";

export default function Header() {
  return (
    <nav className="navbar navbar-dark navbar-expand bg-primary">
      <ul className="navbar-nav mr-auto">
        <li className="nav-item">
```

```
        <NavLink
          exact to="/"
          className="nav-link"
          activeClassName="active">
          Products
        </NavLink>
      </li>
      <li className="nav-item">
        <NavLink
          to="/cart"
          className="nav-link"
          activeClassName="active">
          Cart
        </NavLink>
      </li>
      <li className="nav-item">
        <NavLink
          to="/add-product"
          className="nav-link"
          activeClassName="active"
        >
          Add Product
        </NavLink>
      </li>
    </ul>
  </nav>
);
}
```

Once done, head over to the `App.js` file and flesh out your app structure using React Router. The `Header` component will be rendered outside of any `Route` component, while the rest of the components will render accordingly:

```
import React from "react";
import {
  BrowserRouter as Router,
  Route
} from "react-router-dom";
```

```
import "bootstrap/dist/css/bootstrap.css";
import "./styles.css";

import Header from "./components/Header";
import AddProduct from "./components/AddProduct";
import Products from "./components/Products";
import Cart from "./components/Cart";

export default function App() {
  return (
    <Router>
      <div className="container">
        <Header />
        <Route exact path="/">
          <Products />
        </Route>
        <Route path="/add-product">
          <AddProduct />
        </Route>
        <Route path="/cart">
          <Cart />
        </Route>
      </div>
    </Router>
  );
}
```

Here's the complete [application structure](#).

In the next section, you're going to write the Context API.

Creating the Context API

You'll need to create two Context for this application: one for `products` and one for `cart`. Create a new directory called `context` and write the `ProductsContext.js` file content as follows:

```
import React from "react";

const ProductsContext = React.createContext();

export const ProductsProvider = ProductsContext.Provider;
export const ProductsConsumer = ProductsContext.Consumer;

export default ProductsContext;
```

And another one for the cart:

```
import React from "react";

const CartContext = React.createContext();

export const CartProvider = CartContext.Provider;
export const CartConsumer = CartContext.Consumer;

export default CartContext;
```

Because you won't write any class component, you can skip exporting the `Consumer` component entirely. The `useContext` hook only requires you to pass the context object, which is assigned as the default export in the code above.

With the contexts setup, you can import them into your `App.js` file and wrap your application using the providers:


```
import { CartProvider } from "../context/CartContext";
import { ProductsProvider } from "../context/ProductsContext";

//...

return (
  <Router>
    <CartProvider value={cart}>
      <ProductsProvider value={products}>
        <div className="container">
          {/* the rest of the code... */}
        </div>
      </ProductsProvider>
    </CartProvider>
  </Router>
)
```

The value for the providers will come from your `App` state, so let's create it as well:

```
export default function App() {
  const [cart, setCart] = useState([]);
  const [products, setProducts] = useState([]);
```

Now the contexts are setup and will be synced to your `App` states.

You're going to write the "Add Products" page next.

Writing AddProduct component

First, let's write the `addProduct` function to add new product into the `products` array:

```
const addProduct = (product) => {  
  product.id = products.length + 1;  
  setProducts((currentProducts) => [...currentProducts, product]);  
};
```

You can't add a new product directly into the array, because you need to assign the `id` property into each object.

Let's pass the function into the `AddProduct` component:

```
<Route path="/add-product">  
  <AddProduct addProduct={addProduct} />  
</Route>
```

Now you need to create a form in the `AddProduct` component with the following elements:

Alright, let's write the states for the form:

```
export default function AddProduct({ addProduct }) {  
  const [name, setName] = useState("");  
  const [price, setPrice] = useState(0);  
  const [description, setDescription] = useState("");  
  const [url, setUrl] = useState("");  
}
```

Then, write the `handleSubmit` function like this:

```
const handleSubmit = (e) => {
  e.preventDefault();
  const product = {
    name,
    price,
    description,
    url
  };
  addProduct(product);
  alert(`${name} is added to products list`);
};
```

When a new product is added to the state, you can redirect the user immediately to the products page by using React Router's `useHistory` hook. Import the hook from the library:

```
import { useHistory } from "react-router-dom";
```

Then assign the function call into a variable. You can place it below the `useState` assignments:

```
const history = useHistory()
```

Then call on `history.push("/")` function below the `alert()` :

```
addProduct(product);
alert(`${name} is added to products list`);
history.push("/");
```

Finally, write your form inside the `return` statement:

```
return (  
  <form onSubmit={handleSubmit}>  
    <h1>Add New Product</h1>  
    <div className="form-group">  
      <label>Name</label>  
      <input  
        type="text"  
        className="form-control"  
        value={name}  
        onChange={(e) => setName(e.target.value)}  
      />  
    </div>  
    <div className="form-group">  
      <label>Price</label>  
      <input  
        type="number"  
        className="form-control"  
        value={price}  
        onChange={(e) => setPrice(e.target.value)}  
      />  
    </div>  
    <div className="form-group">  
      <label>Description</label>  
      <input  
        type="text"  
        className="form-control"  
        value={description}  
        onChange={(e) => setDescription(e.target.value)}  
      />  
    </div>  
    <div className="form-group">  
      <label>Image URL</label>  
      <input  
        type="text"  
        className="form-control"  
        value={url}  
        onChange={(e) => setUrl(e.target.value)}  
      />  
    </div>  
    <button  
      type="submit"  
      className="btn btn-primary">  
      Submit  
    </button>  
  </form>  
)
```

```
);
```

You're `AddProduct` component is now finished. Here's the [Sandbox demo](#) up until this point.

Writing the Products component

This component will be responsible for rendering the `products` state and adding products to `cart`. Let's handle the rendering first, then tackle the add to cart function after that.

Open your `Products` component and grab the value of `ProductsContext` with `useContext()` :

```
import React, { useContext } from "react";
import { useHistory } from "react-router";
import { Link } from "react-router-dom";

import ProductsContext from "../context/ProductsContext";

function Products({ addToCart }) {
  const history = useHistory();
  const products = useContext(ProductsContext);
```

if the `products` array is empty, write a simple element telling the user to add a product:

```
if (!products.length) {
  return (
    <h2>
      Your have no products.
      Please <Link to="/add-product">add new product</Link> first
    </h2>
  );
}
```

If there's any product, you can call `map()` over the array as follows:

```
return (
  <ul className="list-group">
    {products.map((product) => (
      <li key={product.id} className="list-group-item">
        <div className="product">
          <div className="product-left">
            <img
              className="product-image"
              src={product.url}
              alt={product.name} />
            <div className="product-title">
              {product.name}
            </div>
            <div className="product-description">
              {product.description}
            </div>
          </div>
          <div className="product-right">
            <div className="product-price">
              ${product.price}
            </div>
            <button
              className="btn btn-sm btn-success"
            >
              Add to Cart
            </button>
          </div>
        </div>
      </li>
    ))}
  </ul>
);
```

Now you have the products on the screen, it's time to handle the “Add to Cart” button. Create a new function in `App` component called `addToCart()` .

You need to first look into the `cart` array and see if there's any product inside it. If there is no product, simply assign the product quantity as

one and add it into the cart:

```
const addToCart = (product) => {  
  if (cart.length) {  
    // will be filled later...  
  } else {  
    product.qty = 1;  
    const newCart = [product];  
    setCart(newCart);  
  }  
};
```

When the `cart` has any product, check if the product you want to add is already present in the `cart` by using `findIndex()` :

```
if(cart.length) {  
  const newCart = [...cart];  
  var foundIndex = newCart.findIndex((item) => {  
    return item.id === product.id  
  });  
}
```

If `findIndex()` return `-1` , it means the product is not present in the cart, so you can `push` the product into the `cart` . If the index is found, you increment the `qty` value by one:

```
if (foundIndex >= 0) {  
  newCart[foundIndex]["qty"] += 1;  
  setCart(newCart);  
} else {  
  product.qty = 1;  
  newCart.push(product);  
  setCart(newCart);  
}
```


Pass the function into the `Products` :

```
<Products addToCart={addToCart} />
```

Then back on the `Products` component, pass `addToCart()` into the `onClick` prop of the button:

```
<button
  className="btn btn-sm btn-success"
  onClick={() => {
    addToCart(product);
    alert(`${product.name} added to cart`);
    history.push("/cart");
  }}
>
  Add to Cart
</button>
```

Now your `Products` page is finished. Try add some products to the cart. You're going to render the `Cart` page in the next section.

Writing the Cart component

This component is very similar to the `Products` component, except that you need to create two small buttons to update the product quantity instead of an “Add to Cart” button.

First, grab the value of `CartContext` into the component. If the `cart` is empty, render a simple instruction to add some products into it:

```
import React, { useContext } from "react";
import { Link } from "react-router-dom";
import CartContext from "../context/CartContext";

export default function Cart({
  addToCart,
  removeFromCart
}) {
  const cart = useContext(CartContext);

  if (!cart.length) {
    return (
      <h2>
        Your cart is empty. Add some <Link to="/">products</Link> first.
      </h2>
    );
  }
}
```

When the `cart` has any product, render its value into the screen. Notice how the elements on the right side is different from the `Products` component:

```
return (
  <ul className="list-group">
    {cart.map((item) => (
      <li
        key={item.id}
        className="list-group-item">
        <div className="product">
          <div className="product-left">
            <img
              className="product-image"
              src={item.url}
              alt={item.name} />
            <div className="product-title">
              {item.name}
            </div>
            <div className="product-description">
              {item.description}
            </div>
          </div>
          <div className="product-right">
            <div className="product-price">
              ${item.price}
            </div>
            <div className="cart-controls">
              <button
                className="cart-btn-rm"
                onClick={() => removeFromCart(item)}
              >
                -
              </button>
              <span className="cart-qty">
                {item.qty}
              </span>
              <button
                className="cart-btn-add"
                onClick={() => addToCart(item)}
              >
                +
              </button>
            </div>
          </div>
        </div>
      </li>
    ))}
  </ul>
```

```
);
```

Finally, you need to create the `removeFromCart` function. First, you need to find the index of the item that you want to delete from the `cart`. Once found, reduce the `qty` value by one. If the value of the `qty` becomes zero, remove the object by using the `Array.splice` function.

Write this function back in your `App.js` file:

```
const removeFromCart = (product) => {  
  const newCart = [...cart];  
  var foundIndex = newCart.findIndex((item) => item.id === product.id);  
  newCart[foundIndex]["qty"] -= 1;  
  if (newCart[foundIndex]["qty"] === 0) {  
    newCart.splice(foundIndex, 1);  
  }  
  setCart(newCart);  
};
```

```
<Route path="/cart">  
  <Cart  
    addToCart={addToCart}  
    removeFromCart={removeFromCart} />  
</Route>
```

With that, the `Cart` component is now finished.

Persisting state into the Local Storage

Although your application is now finished, you'll find that the application data is reset every time you refresh the browser. You can persist the data by saving it to the local storage. If you're unfamiliar with local storage, you can read [my introduction here](#).

React community has created a hook to make use of local storage in the form of `useLocalStorage`. This hook will sync state data to local storage so that it persists through a page refresh. Usage is similar to `useState` except you need to pass two arguments: a local storage key and the default value.

Create a file named `util.js` and write the `useLocalStorage` hook following the recipe:

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

export function useLocalStorage(key, initialValue) {
  const [storedValue, setStoredValue] = useState(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      console.log(error);
      return initialValue;
    }
  });
  const setValue = value => {
    try {
      const valueToStore =
        value instanceof Function ? value(storedValue) : value;
      setStoredValue(valueToStore);
    } catch (error) {
      console.log(error);
    }
  };
  return [storedValue, setValue];
}
```

```
        window.localStorage.setItem(key, JSON.stringify(valueToStore));
    } catch (error) {
        console.log(error);
    }
};
return [storedValue, setValue];
}
```

Then, replace `useState` in your `App` component with `useLocalStorage` :

```
import { useLocalStorage } from "../util";

export default function App() {
    const [cart, setCart] = useLocalStorage("cart", []);
    const [products, setProducts] = useLocalStorage("products", []);
```

Now you will have the `products` and `cart` data saved into the local storage.

Here's the [completed application demo](#) again.

Part 5: Closing

Afterword

Congratulations for finishing this book! You have learned all of React's core concepts and write several applications using that knowledge. I hope you have fun learning React as I did writing it.

Though the book ends here, your journey to master React is not over yet. It's time for you to find some inspirations to start developing with it.

Once again, thanks for buying and reading this book. If there are some things you wish to tell me about this book, feel free to email me at nathan@sebbastian.com. I'm very open to feedbacks and eager to improve my book so that it can be a great resource for people learning to code.

I wish you luck on your software developer career onward, and I'll see you again around the web :)

Until next time!