React Design Patterns and Best Practices

Build easy to scale modular applications using the most powerful components and design patterns

Packt

www.packt.com

By Michele Bertoli

React Design Patterns and Best Practices

Build easy to scale modular applications using the most powerful components and design patterns

Michele Bertoli

Packt

BIRMINGHAM - MUMBAI

React Design Patterns and Best Practices

First published: January 2017

Production reference: 1100117

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B32PB, UK.

ISBN 978-1-78646-453-8

www.packtpub.com

Credits

Author	**Copy Editor**
Michele Bertoli	Safis Editing
Reviewer	**Project Coordinator**
Clay Diffrient	Ulhas Kambali
Commissioning Editor	**Proofreader**
Ashwin Nair	Safis Editing
Acquisition Editor	**Indexer**
Shweta Pant	Rekha Nair
Content Development Editor	**Graphics**
Onkar Wani	Abhinash Sahu
Technical Editor	**Production Coordinator**
Rashil Shah	Aparna Bhagat

About the Author

Michele Bertoli is a frontend engineer with a passion for beautiful UIs. Born in Italy, he moved to London with his family to look for new and exciting job opportunities. He has a degree in computer science and loves clean and well-tested code. Currently, he is working with React.js, crafting modern JavaScript applications. He is a big fan of open source and is always trying to learn something new.

I would like to thank my wife and my son for making my days better with their smiles. Dante, I hope the time I spent writing the book instead of playing with you will make sense when you are older, and you will be proud of me. I would also like to thank Packt Publishing for giving me this opportunity.

About the Reviewer

Clay Diffrient is a JavaScript enthusiast who is always looking to improve and do more. He currently works mostly in React, but has proficiency with other frameworks and libraries, such as Angular, Backbone, and jQuery. He is a maintainer of the popular react-modal library. He takes joy in creating software that is accessible for all people.

Clay currently works as a software engineer at Instructure, where they make software that makes people smarter. He currently works on Instructure's flagship product, Canvas, an open source learning management system.

Clay has previously reviewed *MEAN Web Development* (ISBN: 9781783983285), and enjoys being involved with the community.

I wish to thank my wife Rachael and my two sons, Roger and Beau, for their unending support of my continual learning.

Customer Feedback

Table of Contents

Preface 1

Chapter 1: Everything You Should Know About React 7

- **Declarative programming** 8
- **React elements** 10
- **Unlearning everything** 11
- **Common misconceptions** 14
- **Summary** 18

Chapter 2: Clean Up Your Code 19

- **JSX** 20
  - Babel 21
  - Hello, World! 22
  - DOM elements and React components 23
  - Props 23
  - Children 24
  - Differences with HTML 25
    - Attributes 25
    - Style 25
    - Root 26
    - Spaces 26
    - Boolean attributes 27
  - Spread attributes 28
  - JavaScript templating 28
  - Common patterns 29
    - Multi-line 29
    - Multi-properties 30
    - Conditionals 30
    - Loops 35
    - Control statements 35
    - Sub-rendering 37
- **ESLint** 38
  - Installation 39
  - Configuration 39
  - React plugin 42
  - Airbnb configuration 44
- **The basics of functional programming** 45
  - First-class objects 45

Purity 46
Immutability 47
Currying 47
Composition 48
FP and user interfaces 48
Summary 49

Chapter 3: Create Truly Reusable Components 50

Creating classes 51
The createClass factory 51
Extending React.Component 52
The main differences 52
Props 52
State 54
Autobinding 55
Stateless functional components 57
Props and context 58
The this keyword 59
State 59
Lifecycle 59
Refs and event handlers 59
No reference to component 60
Optimization 60
The state 61
External libraries 61
How it works 61
Asynchronous 62
React lumberjack 64
Using the state 64
Derivables 65
The render method 66
Prop types 67
React Docgen 70
Reusable components 72
Living style guides 76
Summary 80

Chapter 4: Compose All the Things 81

Communication between components 82
Children 83
Container and Presentational pattern 85
Mixins 91

Higher-order Components 94
Recompose 98
Context 101
Function as Child 103
Summary 105
Chapter 5: Proper Data Fetching 106
Data flow 107
Child-parent communication (callbacks) 109
Common parent 110
Data fetching 111
React-refetch 117
Summary 124
Chapter 6: Write Code for the Browser 125
Forms 126
Uncontrolled components 126
Controlled components 131
JSON schema 134
Events 136
Refs 139
Animations 143
React motion 145
Scalable Vector Graphics 147
Summary 149
Chapter 7: Make Your Components Look Beautiful 150
CSS in JS 151
Inline styles 153
Radium 158
CSS Modules 162
Webpack 162
Setting up a project 163
Locally scoped CSS 166
Atomic CSS Modules 172
React CSS Modules 173
Styled Components 175
Summary 178
Chapter 8: Server-Side Rendering for Fun and Profit 179
Universal applications 180

Reasons to implement Server-Side Rendering 181
SEO 181
A common code base 182
Better performance 183
Don't underestimate the complexity 183
A basic example 184
A data fetching example 191
Next.js 194
Summary 198

Chapter 9: Improve the Performance of Your Applications 199

Reconciliation and keys 200
Optimization techniques 205
Should component update 206
Stateless functional components 208
Common solutions 208
Why did you update? 209
Creating functions inside the render method 211
Constants props 215
Refactoring and good design 217
Tools and libraries 223
Immutability 223
Monitoring tools 224
Babel plugins 225
Summary 227

Chapter 10: About Testing and Debugging 228

The benefits of testing 229
Painless JavaScript testing with Jest 230
Mocha is a flexible testing framework 238
JavaScript testing utilities for React 242
A real-world testing example 245
React tree Snapshot Testing 253
Code coverage tools 255
Common testing solutions 257
Testing Higher-Order Components 258
The Page Object pattern 262
React Dev Tools 266
Error handling with React 267
Summary 269

Chapter 11: Anti-Patterns to Be Avoided 270
Initializing the state using props 270
Mutating the state 273
Using indexes as a key 276
Spreading props on DOM elements 279
Summary 281
Chapter 12: Next Steps 282
Contributing to React 283
Distributing your code 285
Publishing a npm package 288
Summary 290
Index 291

Preface

Taking a complete journey through the most valuable design patterns in React, this book demonstrates how to apply design patterns and best practices in real-life situations, whether it's for new or already existing projects. It will help you make your applications more flexible, perform better, and easier to maintain--giving your workflow a huge boost when it comes to speed, without reducing quality.

We'll begin by understanding the internals of React before gradually moving on to writing clean and maintainable code. We'll build components that are reusable across the application, structure applications, and create forms that actually work.

Then, we'll style React components and optimize them to make applications faster and more responsive. Finally, we'll write tests effectively, and you'll learn how to contribute to React and its ecosystem.

By the end of the book, you'll be saved from a lot of trial and error and developmental headaches, and you will be on the road to becoming a React expert.

What this book covers

`Chapter 1`, *Everything You Should Know About React*, introduces you to the base concepts of React, seen from an advanced perspective.

`Chapter 2`, *Clean Up Your Code*, teaches that one of the most important aspects of writing maintainable code is to keep it clean and follow a coding style guide. To use React, it is also important to know the basics of functional programming.

`Chapter 3`, *Create Truly Reusable Components*, informs that building an application using components is a key factor, but creating truly reusable components is the most important thing to do in order to keep the codebase clean and maintainable.

`Chapter 4`, *Compose All the Things*, says that real applications are created using different components and it's important to make them communicate effectively, organizing and structuring the hierarchy in the right way.

`Chapter 5`, *Proper Data Fetching*, instructs that any client-side application has to deal with data at some point; it takes you through the different techniques and approaches that can be used to fetch data in the React-way.

`Chapter 6`, *Write Code for the Browser*, states that our applications live in the browser and we should know how to use it properly. This chapter will go through some advanced concepts, such as events, animations, and how to interact with the DOM.

`Chapter 7`, *Make Your Components Look Beautiful*, demonstrates that crafting beautiful UI components is a big part of the frontend engineering work. With React, there are different ways of doing it and each one tackles the problem from a different perspective. It's important to know which libraries are available and how they work in order to choose the right one.

`Chapter 8`, *Server-Side Rendering for Fun and Profit*, instructs that one of the greatest features of React is the server-side rendering. It works out of the box, but it's important to learn how to use it in the right way to get the most out of it.

`Chapter 9`, *Improve the Performance of Your Applications*, informs that on the Web, performance is one of the most important factors to engage users. React offers a set of tools and techniques to create lightning-fast applications, and this chapter goes through all of them.

`Chapter 10`, *About Testing and Debugging*, makes you realize that we all want our applications to be stable and handle all the edge cases: tests help with that. Writing a comprehensive set of tests is vital to creating rock-solid and maintainable code. On the other hand, bugs always happen and it's crucial to know how to debug and find an issue as early as possible.

`Chapter 11`, *Anti-Patterns to Be Avoided*, explains the fact that developers often try to use shortcuts and creative solutions, but in some cases these workarounds can be dangerous for their applications, especially with big teams and large codebases. This chapter takes you through the common anti-patterns to be avoided while using React.

`Chapter 12`, *Next Steps*, is the last chapter, and all the topics have been covered. I believe it's important to mention how to open source components (to give back to the community) and how to contribute to React and its ecosystem.

What you need for this book

We will need a computer with a terminal, a node.js/npm environment, and a browser.

Who this book is for

If you want to increase your understanding of React and apply it to real-life application development, this book is for you.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Inside the loop, there is some conditional logic to check if the `#first` and the `#link` properties exist, and depending on their values, a different piece of HTML is rendered. Variables are wrapped into curly braces."

A block of code is set as follows:

```
const toLowerCase = input => {
  const output = []
  for (let i = 0; i < input.length; i++) {
    output.push(input[i].toLowerCase())
  }
  return output
}
```

Any command-line input or output is written as follows:

```
npm install -g create-react-app
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Let's begin updating the tests, starting with the **renders with text** ones."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support`and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/React-Design-Patterns-and-Best-Practices`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support`and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Everything You Should Know About React

Hello, readers!

This book assumes that you already know what React is and what problems it solves for you. You may have written a small/medium application with React and you want to improve your skills and answer all your open questions.

You should know that React is maintained by developers at Facebook and hundreds of contributors within the JavaScript community.

React is one of the most popular libraries for creating user interfaces and it is well-known to be fast, thanks to its smart way of touching the DOM.

It comes with JSX, a new syntax to write markup in JavaScript, which requires you to change your mind regarding the separation of concerns. It has many cool features, such as the server-side rendering that gives you the power to write Universal applications.

To follow this book, you will need to know how to use the terminal to install and run `npm` packages in your `Node.js` environment.

All the examples are written in ES2015, which you should be able to read and understand.

In this first chapter, we will go through some basics concepts which are important to master to use React effectively, but are non-trivial to figure out for beginners:

- The difference between imperative and declarative programming
- React components and their instances, and how React uses elements to control the UI flow

- How React changes the way we build web applications, enforcing a different new concept of separation of concerns, and the reasons behind its unpopular design choice
- Why people feel the JavaScript Fatigue and what you can do to avoid the most common errors developers make when approaching the React ecosystem

Declarative programming

Reading the React documentation or blog posts about React, you have surely come across the term **declarative**.

In fact, one of the reasons why React is so powerful is because it enforces a declarative programming paradigm.

Consequently, to master React, it is important to understand what declarative programming means and what the main differences between imperative and declarative programming are.

The easiest way to approach the problem is to think about imperative programming as a way of describing how things work, and declarative programming as a way of describing what you want to achieve.

A real-life parallel in the imperative world would be entering a bar for a beer, and giving the following instructions to the bartender:

- Take a glass from the shelf
- Put the glass in front of the draft
- Pull down the handle until the glass is full
- Pass me the glass

In the declarative world, instead, you would just say: “Beer, please.”

The declarative approach of asking for a beer assumes that the bartender knows how to serve one, and that is an important aspect of the way declarative programming works.

Let's move into a JavaScript example, writing a simple function that, given an array of uppercase strings, returns an array with the same strings in lowercase:

```
toLowerCase(['FOO', 'BAR']) // ['foo', 'bar']
```

An imperative function to solve the problem would be implemented as follows:

```
const toLowerCase = input => {
  const output = []
  for (let i = 0; i < input.length; i++) {
    output.push(input[i].toLowerCase())
  }
  return output
}
```

First of all, an empty array to contain the result gets created. Then, the function loops through all the elements of the input array and pushes the lowercase values into the empty array. Finally, the output array gets returned.

A declarative solution would be as follows:

```
const toLowerCase = input => input.map(
  value => value.toLowerCase()
)
```

The items of the input array are passed to a map function, which returns a new array containing the lowercase values.

There are some important differences to note: the former example is less elegant and it requires more effort to be understood. The latter is terser and easier to read, which makes a huge difference in big code bases, where maintainability is crucial.

Another aspect worth mentioning is that in the declarative example, there is no need to use variables nor to keep their values updated during the execution. Declarative programming, in fact, tends to avoid creating and mutating a state.

As a final example, let's see what it means for React to be declarative.

The problem we will try to solve is a common task in web development: showing a map with a marker.

The JavaScript implementation (using the Google Maps SDK) is as follows:

```
const map = new google.maps.Map(document.getElementById('map'), {
  zoom: 4,
  center: myLatLng,
})

const marker = new google.maps.Marker({
  position: myLatLng,
  title: 'Hello World!',
})
```

```
    marker.setMap(map)
```

It is clearly imperative, because all the instructions needed to create the map, and create the marker and attach it to the map are described inside the code, one after the other.

A React component to show a map on a page would look like this instead:

```
    <Gmaps zoom={4} center={myLatLng}> '
      <Marker position={myLatLng} title="Hello world!" />
    </Gmaps>
```

In declarative programming, developers only describe what they want to achieve and there's no need to list all the steps to make it work.

The fact that React offers a declarative approach makes it easy to use, and consequently, the resulting code is simple, which often leads to fewer bugs and more maintainability.

React elements

This book assumes that you are familiar with components and their instances, but there is another object you should know if you want to use React effectively: the **Element**.

Whenever you call `createClass`, extend `Component`, or simply declare a stateless function, you are creating a component. React manages all the instances of your components at runtime, and there can be more than one instance of the same component in memory at a given point in time.

As mentioned previously, React follows a declarative paradigm, and there's no need to tell it how to interact with the DOM; you just declare what you want to see on the screen and React does the job for you.

As you might have already experienced, most other UI libraries work in the opposite way: they leave the responsibility of keeping the interface updated to the developer, who has to manage the creation and destruction of the DOM elements manually.

To control the UI flow, React uses a particular type of object, called **element**, which describes what has to be shown on the screen. These immutable objects are much simpler compared to the components and their instances, and contain only the information that is strictly needed to represent the interface.

The following is an example of an element:

```
{
  type: Title,
  props: {
    color: 'red',
    children: 'Hello, Title!'
  }
}
```

Elements have a type, which is the most important attribute, and some properties. There is also a special property, called **children**, which is optional and represents the direct descendant of the element.

The type is important because it tells React how to deal with the element itself. In fact, if the type is a string, the element represents a **DOM node**, while if the type is a function, the element is a **component**.

DOM elements and components can be nested with each other, to represent the render tree:

```
{
  type: Title,
  props: {
    color: 'red',
    children: {
      type: 'h1',
      props: {
        children: 'Hello, H1!'
      }
    }
  }
}
```

When the type of the element is a function, React calls it, passing the props to get back the underlying elements. It keeps on performing the same operation recursively on the result until it gets a tree of DOM nodes, which React can render on the screen. This process is called **reconciliation**, and it is used by both React DOM and React Native to create the user interfaces of their respective platforms.

Unlearning everything

Using React for the first time usually requires an open mind because it brings a new way of designing web and mobile applications. In fact, React tries to innovate the way we build user interfaces following a path that breaks most of the well-known best practices.

In the last two decades, we learned that the separation of concerns is important, and we used to think about it in terms of separating the logic from the templates. Our goal has always been to write the JavaScript and the HTML in different files.

Various templating solutions have been created to help developers achieve this.

The problem is that most of the time, that kind of separation is just an illusion and the truth is that the JavaScript and the HTML are tightly coupled, no matter where they live.

Let's see an example of a template:

```
{{#items}}
  {{#first}}
    <li><strong>{{name}}</strong></li>
  {{/first}}
  {{#link}}
    <li><a href="{{url}}">{{name}}</a></li>
  {{/link}}
{{/items}}
```

The preceding snippet is taken from the website of **Mustache**, one of the most popular templating systems.

The first row tells Mustache to loop through a collection of items. Inside the loop, there is some conditional logic to check if the `#first` and the `#link` properties exist, and depending on their values, a different piece of HTML is rendered. Variables are wrapped into curly braces.

If your application has only to display some variables, a templating library could represent a good solution, but when it comes to starting to work with complex data structures, things change.

In fact, templating systems and their Domain-Specific Language (DSL) offer a subset of features, and they try to provide the functionalities of a real programming language without reaching the same level of completeness.

As shown in the example, templates highly depend on the models they receive from the logic layer to display the information.

On the other hand, JavaScript interacts with the DOM elements rendered by the templates to update the UI, even if they are loaded from separate files.

The same problem applies to styles: they are defined in a different file, but they are referenced in the templates and the CSS selectors follow the structure of the markup, so it is almost impossible to change one without breaking the other, which is the definition of coupling.

That is why the classic separation of concerns ended up being more a separation of technologies, which is of course not a bad thing, but it does not solve any real problems.

React tries to move a step forward by putting the templates where they belong: next to the logic. The reason it does that is because React suggests you organize your applications by composing small bricks called **components**.

The framework should not tell you how to separate the concerns, because every application has its own, and only the developers should decide how to limit the boundaries of their apps.

The component-based approach drastically changes the way we write web applications, which is why the classic concept of separation of concerns is gradually being taken over by a much more modern structure.

The paradigm enforced by React is not new, and it was not invented by its creators, but React has contributed to making the concept mainstream and, most importantly, popularized it in such a way that is easier to understand for developers with different levels of expertise.

This is how the render method of a React component looks:

```
render() {
  return (
    <button style={{ color: 'red' }} onClick={this.handleClick}>
      Click me!
    </button>
  )
}
```

We all agree that it looks a bit weird in the beginning, but it is just because we are not used to that kind of syntax.

As soon as we learn it and we realize how powerful it is, we understand its potential.

Using JavaScript for both logic and templating not only helps us separate our concerns in a better way, but it also gives us more power and more expressivity, which is what we need to build complex user interfaces.

That is why, even if the idea of mixing JavaScript and HTML sounds weird in the beginning, it is important to give React five minutes.

The best way to get started with a new technology is to try it in a small side project and see how it goes. In general, the right approach is to always be ready to unlearn everything and change your mindset if the long-term benefits are worth it.

There is another concept, which is pretty controversial and hard to accept, and which the engineers behind React are trying to push to the community: moving the styling logic inside the component, too.

The end goal is to encapsulate every single technology used to create our components and separate the concerns according to their domain and functionalities.

Here is an example of a style object taken from the React documentation:

```
var divStyle = {
  color: 'white',
  backgroundImage: 'url(' + imgUrl + ')',
  WebkitTransition: 'all', // note the capital 'W' here
  msTransition: 'all' // 'ms' is the only lowercase vendor prefix
};

ReactDOM.render(
 <div style={divStyle}>Hello World!</div>,
 mountNode
);
```

This set of solutions, where developers use JavaScript to write their styles, is known as `#CSSinJS`, and we will talk about it extensively in `Chapter 7`, *Make Your Components Look Beautiful*.

Common misconceptions

There is a common opinion that React is a huge set of technologies and tools, and if you want to use it, you are forced to deal with package managers, transpilers, module bundlers, and an infinite list of different libraries.

This idea is so widespread and shared among people that it has been clearly defined, and has been given the name **JavaScript Fatigue**.

It is not hard to understand the reasons behind this. In fact, all the repositories and libraries in the React ecosystem are made using the shiny new technologies, the latest version of JavaScript, and the most advanced techniques and paradigms.

Moreover, there is a massive number of React boilerplates on GitHub, each one with tens of dependencies to offer solutions for any problems.

It is very easy to think that all these tools are required to start using React, but this is far from the truth.

Despite this common way of thinking, React is a pretty tiny library, and it can be used inside any page (or even inside a JSFiddle) in the same way everyone used to use jQuery or Backbone: just by including the script on the page before the closing body element.

To be fair, there are two scripts because React is split into two packages: `react`, which implements the core features of the library, and `react-dom`, which contains all the browser-related features. The reason behind that is because the core package is used to support different targets, such as React DOM in browsers and React Native on mobile devices.

Running a React application inside a single HTML page does not require any package manager or complex operation. You can just download the distribution bundle and host it yourself (or use `unpkg.com`), and you are ready to get started with React and its features in a few minutes.

Here are the URLs to be included in the HTML to start using React:

- `https://unpkg.com/react/dist/react.min.js`
- `https://unpkg.com/react-dom/dist/react-dom.min.js`

If we include the core React library only, we cannot use JSX because it is not a standard language supported by the browser; but, the whole point is to start with the bare minimum set of features and add more functionalities as soon as they are needed.

For a simple UI, we could just use `createElement` and, only when we start building something more complex, we can include a transpiler to enable JSX and convert it into JavaScript.

As soon as the app grows a bit more, we may need a router to handle different pages and views, and we can include that as well.

At some point, we may want to load data from some API endpoints, and if the application keeps growing, we will reach the point where we need some external dependencies to abstract complex operations. Only in that very moment, should we introduce a package manager.

Then the time will come to split our application into separate modules and organize our files in the right way. At that point, we should start thinking about using a module bundler.

Following this very simple approach, there's no fatigue.

Starting with a boilerplate that has one hundred dependencies and tens of `npm` packages of which we know nothing is the best way to get lost.

It is important to note that every programming-related job (and front end engineering in particular) requires continuous learning. It is the nature of the Web to evolve at a very fast pace and change according to the needs of both users and developers. This is the way our environment has worked since the beginning and what makes it very exciting.

As we gain experience working on the Web, we learn that we cannot master everything and we should find the right way to keep ourselves updated to avoid the fatigue. We become able to follow all the new trends without jumping into the new libraries for the sake of it, unless we have time for a side project.

It is astonishing how, in the JavaScript world, as soon as a specification is announced or drafted, someone in the community implements it as a transpiler plugin or a polyfill, letting everyone else play with it while the browser vendors agree and start supporting it.

This is something that makes JavaScript and the browser a completely different environment compared to any other language or platform.

The downside of it is that things change very quickly, but it is just a matter of finding the right balance between betting on new technologies versus staying safe.

In any case, Facebook developers care a lot about the DX (developer experience), and they listen carefully to the community. So, even if it is not true that to use React we are required to learn hundreds of different tools, they realized that people were feeling the fatigue and they released a CLI tool that makes it incredibly easy to scaffold and run a real React application.

The only requirement is to use a `node.js/npm` environment and install the CLI tool globally:

```
npm install -g create-react-app
```

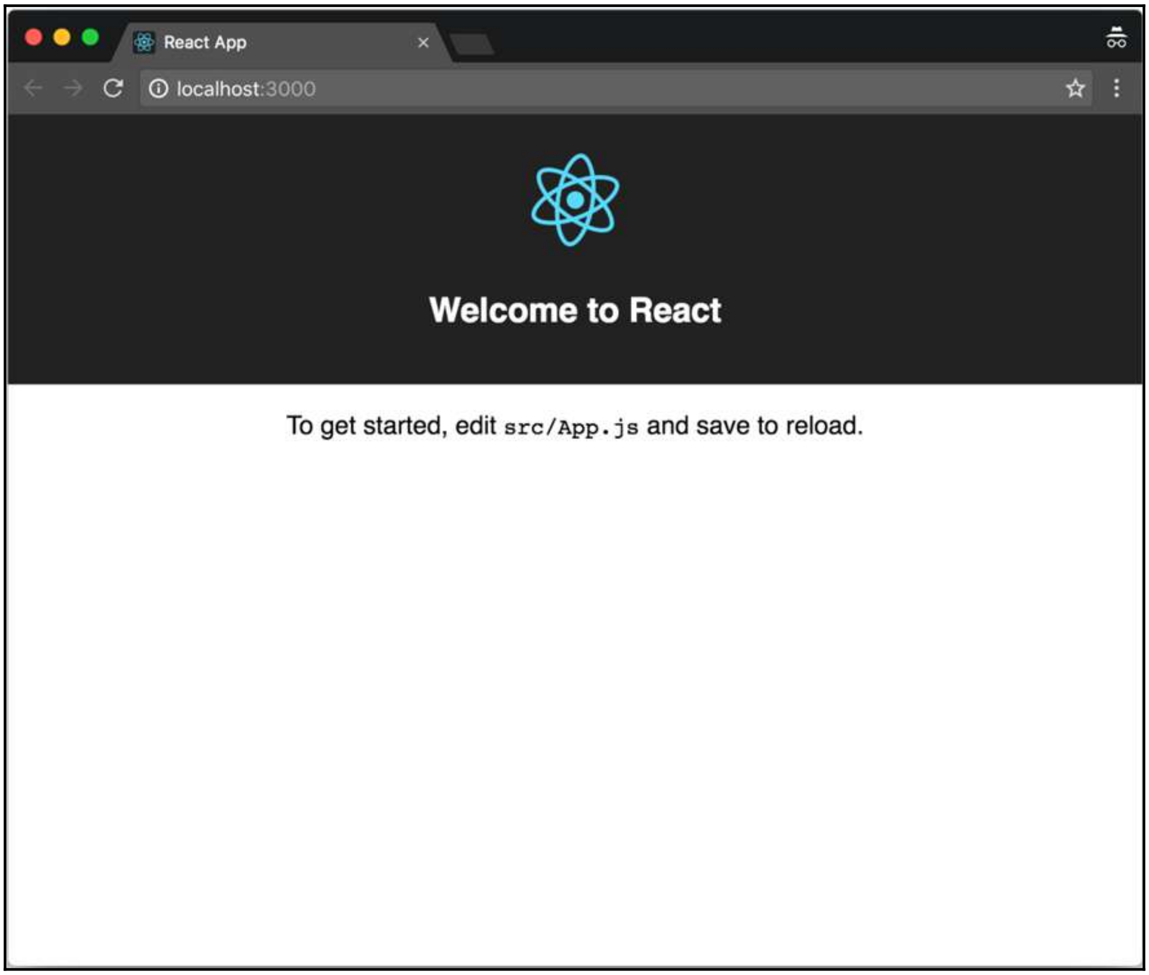
When the executable is installed, we can use it to create our application passing a folder name:

```
create-react-app hello-world
```

Finally, we move into the folder of our application with `cd hello-world` and we just run:

```
npm start
```

Magically, our application is running with a single dependency, but with all the features needed to build a complete React application using the most advanced techniques. The following screenshot shows the default page of an application created with create-react-app:



Welcome to React

To get started, edit `src/App.js` and save to reload.

We will use this tool throughout the book to run the examples for each chapter which are also available on GitHub at the following address:

`https://github.com/MicheleBertoli/react-design-patterns-and-best-practices`

Summary

In this first chapter, we have learned some basic concepts that are very important for following the rest of the book, and which are crucial to working with React daily.

We now know how to write declarative code and we have a clear understanding of the difference between the components we create and the elements React uses to display their instances on the screen.

We learned the reasons behind the choice of co-locating logic and templates together, and why that unpopular decision has been a big win for React.

We went through the reasons why it is common to feel fatigue in the JavaScript ecosystem, but we have also seen how to avoid those problems by following an iterative approach.

Finally, we have seen what the new `create-react-app` CLI is, and we are now ready to start writing some real code.

2
Clean Up Your Code

This chapter assumes that you already have experience with JSX and you want to improve your skills to use it effectively.

To use JSX without any problems or unexpected behaviors, it is important to understand how it works under the hood and the reasons why it is a useful tool for building UIs.

Our goal is to write clean and maintainable JSX code, and to achieve that, we have to know where it comes from, how it gets translated to JavaScript, and which features it provides.

In the first section, we will do a little step back, but please bear with me because it is crucial to master the basics in order to apply the best practices.

In this chapter, we will cover the following topics:

- What JSX is and why we should use it
- What Babel is and how we can use it to write modern JavaScript code
- The main features of JSX and the differences between HTML and JSX
- Best practices to write JSX in an elegant and maintainable way
- How linting, and ESLint in particular, can make our JavaScript code consistent across applications and teams
- The basics of functional programming and why following a functional paradigm will make us write better React components

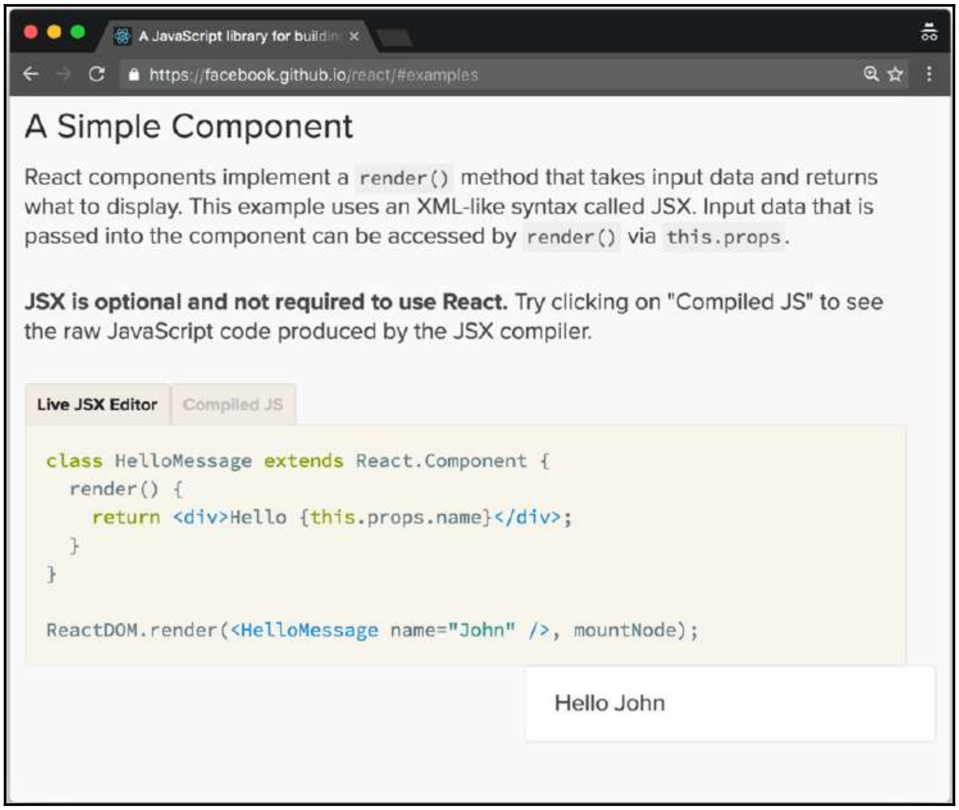
JSX

In the previous chapter, we saw how React changes the concept of separation of concerns, moving the boundaries inside components.

We also learned how React uses the elements returned by the components to display the UI on the screen.

Let's now see how we can declare our elements inside our components.

React provides two ways to define our elements. The first one is by using JavaScript functions, and the second one is by using JSX, an optional XML-like syntax. Here is the examples section of the official React.js website:


A JavaScript library for buildin
https://facebook.github.io/react/#examples

A Simple Component

React components implement a `render()` method that takes input data and returns what to display. This example uses an XML-like syntax called JSX. Input data that is passed into the component can be accessed by `render()` via `this.props`.

JSX is optional and not required to use React. Try clicking on "Compiled JS" to see the raw JavaScript code produced by the JSX compiler.

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

```
Hello John
```

To begin with, JSX is one of the main reasons why people fail to approach React, because looking at the examples on the home page and seeing JavaScript mixed with HTML for the first time seems strange to most of us.

As soon as we get used to it, we realize that it is very convenient, precisely because it is similar to HTML and looks very familiar to anyone who has already created UIs on the web.

The opening and closing tags make it easier to represent nested trees of elements–something that would have been unreadable and hard to maintain using plain JavaScript.

Babel

In order to use JSX (and some features of ES2015) in our code, we have to install **Babel**.

First of all, it is important to clearly understand the problems it can solve for us and why we need to add a step to our process. The reason is that we want to use features of the language that have not yet been added in the browser, our target environment. Those advanced features make our code cleaner for developers, but the browser cannot understand and execute it.

The solution is to write our scripts in JSX and ES2015, and when we are ready to ship, we compile the sources into ES5, the standard specification implemented in major browsers today.

Babel is a popular JavaScript compiler widely adopted within the React community.

Babel can compile ES2015 code into ES5 JavaScript, as well as compile JSX into JavaScript functions. The process is called transpilation, because it compiles the source into a new source rather than into an executable.

Using it is pretty straightforward; we just install it:

```
npm install --global babel-cli
```

If you do not want to install it globally (developers usually tend to avoid this), you can install Babel locally to a project and run it through an `npm` script, but for the purposes of this chapter, a global instance is fine.

When the installation is complete, we can run the following command to compile any JavaScript file:

```
babel source.js -o output.js
```

One of the reasons Babel is so powerful is because it is highly configurable. Babel is just a tool to transpile a source file into an output file, but to apply some transformations, we need to configure it.

Luckily, there are some very useful presets of configurations, which we can easily install and use:

```
npm install --global babel-preset-es2015
babel-preset-react
```

Once the installation is complete, we create a configuration file called `.babelrc` in the root folder, and put the following lines into it to tell Babel to use those presets:

```
{
  "presets": [
    "es2015",
    "react"
  ]
}
```

From this point on, we can write ES2015 and JSX in our source files and execute the output files in the browser.

Hello, World!

Now that our environment has been set up to support JSX, we can dive into the most basic example: generating a `div` element.

This is how you would create a `div` with React's `createElement` function:

```
React.createElement('div')
```

And this is the JSX for creating a `div` element:

```
<div />
```

It looks similar to regular HTML.

The big difference is that we are writing the markup inside a `.js` file, but it is important to note that JSX is only syntactic sugar and it gets transpiled into JavaScript before being executed in the browser.

In fact, our `<div />` is translated into `React.createElement('div')` when we run Babel, which is something we should always keep in mind when we write our templates.

DOM elements and React components

With JSX, we can create both HTML elements and React components; the only difference is whether or not they start with a capital letter.

For example, to render an HTML button, we use `<button />`, while to render our `Button` components we use `<Button />`.

The first button is transpiled into the following:

```
React.createElement('button')
```

The second one is transpiled into the following:

```
React.createElement(Button)
```

The difference here is that in the first call we are passing the type of the DOM element as a string, while in the second call we are passing the component itself, which means that it should exist in the scope to work.

As you may have noticed, JSX supports self-closing tags, which are pretty good for keeping the code terse and do not require us to repeat unnecessary tags.

Props

JSX is very convenient when your DOM elements or React components have props. In fact, using XML is pretty easy to set attributes on elements:

```
<imgsrc="https://facebook.github.io/react/img/logo.svg"
alt="React.js" />
```

The equivalent in JavaScript would be as follows:

```
React.createElement("img", {
  src: "https://facebook.github.io/react/img/logo.svg",
  alt: "React.js"
});
```

This is far less readable, and even with only a couple of attributes it is harder to read without a bit of reasoning.

Children

JSX allows you to define children to describe the tree of elements and compose complex UIs.

A basic example is a link with text inside it, as follows:

```
<a href="https://facebook.github.io/react/">Click me!</a>
```

This would be transpiled into the following:

```
React.createElement(
  "a",
  { href: "https://facebook.github.io/react/" },
  "Click me!"
);
```

Our link can be enclosed inside a `div` for some layout requirements, and the JSX snippet to achieve that is as follows:

```
<div>
  <a href="https://facebook.github.io/react/">Click me!</a>
</div>
```

The JavaScript equivalent is as follows:

```
React.createElement(
  "div",
  null,
  React.createElement(
    "a",
    { href: "https://facebook.github.io/react/" },
    "Click me!"
  )
);
```

It should now be clear how the XML-like syntax of JSX makes everything more readable and maintainable, but it is always important to know the JavaScript parallel of our JSX in order to have control over the creation of elements.

The good part is that we are not limited to having elements as children of elements, but we can use JavaScript expressions such as functions or variables.

To do this, we just have to enclose expression within curly braces:

```
<div>
  Hello, {variable}.
  I'm a {function()}.
</div>
```

The same applies to non-string attributes:

```
<a href={this.makeHref()}>Click me!</a>
```

Differences with HTML

So far, we have looked at the similarities between JSX and HTML. Let's now look at the little differences between them and the reasons they exist.

Attributes

We must always keep in mind that JSX is not a standard language and that it gets transpiled into JavaScript. Because of this, some attributes cannot be used.

For example, instead of `class`, we have to use `className`, and instead of `for`, we have to use `htmlFor`:

```
<label className="awesome-label" htmlFor="name" />
```

The reason for this is that `class` and `for` are reserved words in JavaScript.

Style

A pretty significant difference is the way the style attribute works. We will look at how to use it in more detail in `Chapter 7`, *Make Your Components Look Beautiful*, but now we will focus on the way it works.

The style attribute does not accept a CSS string as the HTML parallel does, but it expects a JS object where the style names are **camelCased**:

```
<div style={{ backgroundColor: 'red' }} />
```

Root

One important difference with HTML worth mentioning is that since JSX elements get translated into JavaScript functions and you cannot return two functions in JavaScript, whenever you have multiple elements at the same level, you are forced to wrap them into a parent.

Let's look at a simple example:

```
<div />
<div />
```

This gives us the following error:

```
Adjacent JSX elements must be wrapped in an enclosing tag
```

On the other hand, the following works:

```
<div>
  <div />
  <div />
</div>
```

It is pretty annoying to have to add unnecessary `div` tags just to make JSX work, but React developers are trying to find a solution (at the time of writing):

`https://github.com/reactjs/core-notes/blob/master/2016-07/july-07.md`

Spaces

There's one thing that could be a little bit tricky in the beginning, and again it concerns the fact that we should always keep in mind that JSX is not HTML, even if it has an XML-like syntax.

JSX, in fact, handles the spaces between text and elements differently from HTML, in a way that's counter-intuitive.

Consider the following snippet:

```
<div>
  <span>foo</span>
  bar
  <span>baz</span>
</div>
```

In the browser, which interprets HTML, this code would give you `foo bar baz`, which is exactly what we expect.

In JSX, instead, the same code would be rendered as `foobarbaz`, which is because the three nested lines get transpiled as individual children of the `div` element, without taking the spaces into account. A common solution to get the same output, is putting a space explicitly between the elements:

```
<div>
  <span>foo</span>
  {' '}
  bar
  {' '}
  <span>baz</span>
</div>
```

As you may have noticed, we are using an empty string wrapped inside a JavaScript expression to force the compiler to apply the space between the elements.

Boolean attributes

A couple more things worth mentioning before starting for real regarding the way you define Boolean attributes in JSX. If you set an attribute without a value, JSX assumes that its value is true, following the same behavior of the HTML `disabled` attribute, for example.

This means that if we want to set an attribute to false, we have to declare it explicitly as false:

```
<button disabled />
React.createElement("button", { disabled: true });
```

The following is another example:

```
<button disabled={false} />
React.createElement("button", { disabled: false });
```

This can be confusing in the beginning because we may think that omitting an attribute would mean false, but it is not like that. With React, we should always be explicit to avoid confusion.

Spread attributes

An important feature is the **spread attributes** operator, which comes from the Rest/Spread Properties for ECMAScript proposal, (`https://github.com/sebmarkbage/ecmascript-rest-spread`) and is very convenient whenever we want to pass all the attributes of a JavaScript object to an element.

A common practice that leads to fewer bugs is not to pass entire JavaScript objects down to children by reference, but to use their primitive values, which can be easily validated, making components more robust and error-proof.

Let's see how it works:

```
const foo = { id: 'bar' }
return <div {...foo} />
```

The preceding code gets transpiled into the following:

```
var foo = { id: 'bar' };
return React.createElement('div', foo);
```

JavaScript templating

Finally, we started with the assumption that one of the advantages of moving the templates inside our components instead of using an external template library is that we can use the full power of JavaScript, so let's start looking at what that means.

The spread attributes is an example of that, and another common example is that JavaScript expressions can be used as attributes values by enclosing them within curly braces:

```
<button disabled={errors.length} />
```

Common patterns

Now that we know how JSX works and can master it, we are ready to see how to use it in the right way following some useful conventions and techniques.

Multi-line

Let's start with a very simple one. As stated previously, one of the main reasons we should prefer JSX over React's `createElement` is because of its XML-like syntax and because balanced opening and closing tags are perfect to represent a tree of nodes.

Therefore, we should try to use it in the right way and get the most out of it.

One example is as follows; whenever we have nested elements, we should always go multiline:

```
<div>
  <Header />
  <div>
    <Main content={...} />
  </div>
</div>
```

This is preferable to the following:

```
<div><Header /><div><Main content={...} /></div></div>
```

The exception is if the children are not elements, such as text or variables. In that case, it makes sense to remain on the same line and avoid adding noise to the markup, as follows:

```
<div>
  <Alert>{message}</Alert>
  <Button>Close</Button>
</div>
```

Always remember to wrap your elements inside parentheses when you write them in multiple lines. In fact, JSX always gets replaced by functions, and functions written on a new line can give you an unexpected result because of automatic semicolon insertion. Suppose, for example, you are returning JSX from your render method, which is how you create UIs in React.

The following example works fine, because the `div` is on the same line as the return:

```
return <div />
```

The following, however, is not right:

```
return
  <div />
```

The reason for this is because you would have the following:

```
return;
React.createElement("div", null);
```

This is why you have to wrap the statement in parentheses:

```
return (
  <div />
)
```

Multi-properties

A common problem in writing JSX comes when an element has multiples attributes. One solution is to write all the attributes on the same line, but this would lead to very long lines, which we do not want in our code (see the following section for how to enforce coding style guides).

A common solution is to write each attribute on a new line, with one level of indentation, and then align the closing bracket with the opening tag:

```
<button
  foo="bar"
  veryLongPropertyName="baz"
  onSomething={this.handleSomething}
/>
```

Conditionals

Things get more interesting when we start working with **conditionals**, for example, if we want to render some components only when certain conditions are matched. The fact that we can use JavaScript in our conditions is a big plus, but there are many different ways to express conditions in JSX and it is important to understand the benefits and problems of each one of these in order to write code that is both readable and maintainable.

Suppose we want to show a logout button only if the user is currently logged into our application.

A simple snippet to start with is as follows:

```
let button
if (isLoggedIn) {
  button = <LogoutButton />
}
return <div>{button}</div>
```

This works, but it is not very readable, especially if there are multiple components and multiple conditions.

In JSX, we can use an inline condition:

```
<div>
  {isLoggedIn && <LoginButton />}
</div>
```

This works because if the condition is false, nothing gets rendered, but if the condition is true, the `createElement` function of the `LoginButton` gets called and the element is returned to compose the resulting tree.

If the condition has an alternative, (the classic `if...else` statement), and we want, for example, to show a logout button if the user is logged in and a login button otherwise, we can use JavaScript's `if...else`, as follows:

```
let button
if (isLoggedIn) {
  button = <LogoutButton />
} else {
  button = <LoginButton />
}
return <div>{button}</div>
```

Alternatively, and better, we can use a ternary condition, which makes the code more compact:

```
<div>
  {isLoggedIn ? <LogoutButton /> : <LoginButton />}
</div>
```

You can find the ternary condition used in popular repositories such as the Redux realworld example (`https://github.com/reactjs/redux/blob/master/examples/real-world/src/components/List.js#L25`), where the ternary is used to show a loading label if the component is fetching the data, or *load more* inside a button depending on the value of the `isFetching` variable:

```
<button [...]>
```

```
    {isFetching ? 'Loading...' : 'Load More'}
  </button>
```

Let's now look at the best solution for when things get more complicated and, for example, we have to check more than one variable to determine whether to render a component or not:

```
  <div>
   {dataIsReady && (isAdmin || userHasPermissions) &&
     <SecretData />
   }
  </div>
```

In this case, it is clear that using the inline condition is a good solution, but the readability is strongly impacted. Instead, we can create a helper function inside our component and use it in JSX to verify the condition:

```
  canShowSecretData() {
    const { dataIsReady, isAdmin, userHasPermissions } = this.props
    return dataIsReady && (isAdmin || userHasPermissions)
  }

  <div>
    {this.canShowSecretData() && <SecretData />}
  </div>
```

As you can see, this change makes the code more readable and the condition more explicit. If you look at this code in six months' time, you will still find it clear just by reading the name of the function.

If you do not like using functions, you can use an object's getters, which make the code more elegant.

For example, instead of declaring a function, we define a getter:

```
  get canShowSecretData() {
    const { dataIsReady, isAdmin, userHasPermissions } = this.props
    return dataIsReady && (isAdmin || userHasPermissions)
  }

  <div>
    {this.canShowSecretData && <SecretData />}
  </div>
```

The same applies to computed properties. Suppose you have two single properties, for currency and value. Instead of creating the price string inside your render method, you can create a class function:

```
getPrice() {
  return `${this.props.currency}${this.props.value}`
}

<div>{this.getPrice()}</div>
```

This is better, because it is isolated and you can easily test it in case it contains logic.

Alternatively, you can go a step further and, as we have just seen, use getters:

```
get price() {
  return `${this.props.currency}${this.props.value}`
}

<div>{this.price}</div>
```

Going back to conditional statements, there are other solutions that require using external dependencies. A good practice is to avoid external dependencies as much as we can to keep our bundle smaller, but it may be worth it in this particular case because improving the readability of our templates is a big win.

The first solution is `render-if`, which we can install with the following:

```
npm install --save render-if
```

We can then easily use it in our projects, as follows:

```
const { dataIsReady, isAdmin, userHasPermissions } = this.props
const canShowSecretData = renderIf(
  dataIsReady && (isAdmin || userHasPermissions)
)

<div>
  {canShowSecretData(<SecretData />)}
</div>
```

We wrap our conditions inside the `renderIf` function.

The utility function that gets returned can be used as a function, which receives the JSX markup to be shown when the condition is true.

One goal we should always keep in mind is to never add too much logic inside our components. Some of them will require a bit of it, but we should try to keep them as simple and dumb as possible so that we can easily spot and fix errors.

We should at least try to keep the `renderIf` method as clean as possible, and to do that, we can use another utility library called `react-only-if`, which lets us write our components as if the condition is always true by setting the conditional function using a Higher-Order Component.

We will talk about Higher-Order Components extensively in `Chapter 4`, *Compose All the Things*, but for now you just need to know that they are functions that receive a component and return an enhanced one by adding some properties or modifying its behavior.

To use the library, we just need to install it as follows:

```
npm install --save react-only-if
```

Once it is installed, we can use it in our apps in the following way:

```
const SecretDataOnlyIf = onlyIf(
  ({ dataIsReady, isAdmin, userHasPermissions }) => {
    return dataIsReady && (isAdmin || userHasPermissions)
  }
)(SecretData)

<div>
  <SecretDataOnlyIf
    dataIsReady={...}
    isAdmin={...}
    userHasPermissions={...}
  />
</div>
```

As you can see here, there is no logic at all inside the component itself.

We pass the condition as the first parameter of the `onlyIf` function and when the condition is matched, the component is rendered.

The function used to validate the condition receives the props, state, and context of the component.

In this way, we avoid polluting our component with conditionals so that it is easier to understand and reason about.

Loops

A very common operation in UI development is to display lists of items. When it comes to showing lists, using JavaScript as a template language is a very good idea.

If we write a function that returns an array inside our JSX template, each element of the array gets compiled into an element.

As we have seen before, we can use any JavaScript expressions inside curly braces and the most common way to generate an array of elements, given an array of objects, is to use `map`.

Let's dive into a real-world example. Suppose you have a list of users, each one with a name property attached to it.

To create an unordered list to show the users, you can do the following:

```
<ul>
  {users.map(user =><li>{user.name}</li>)}
</ul>
```

This snippet is incredibly simple and incredibly powerful at the same time, where the power of the HTML and JavaScript converge.

Control statements

Conditionals and loops are very common operations in UI templates and you may feel wrong using the JavaScript ternary or the map function to perform them. JSX has been built in such a way that it only abstracts the creation of the elements, leaving the logic parts to real JavaScript, which is great except that sometimes, the code becomes less clear.

In general, we aim to remove all the logic from our components, and especially from our render methods, but sometimes we have to show and hide elements according to the state of the application, and very often we have to loop through collections and arrays.

If you feel that using JSX for that kind of operation will make your code more readable, there is a Babel plugin available to do just that: `jsx-control-statements`.

This follows the same philosophy as JSX, and it does not add any real functionality to the language; it is just syntactic sugar that gets compiled into JavaScript.

Let's see how it works.

First of all, we have to install it:

```
npm install --save jsx-control-statements
```

Once it is installed, we have to add it to the list of our babel plugins in our `.babelrc` file:

```
"plugins": ["jsx-control-statements"]
```

From now on we can use the syntax provided by the plugin and Babel will transpile it together with the common JSX syntax.

A conditional statement written using the plugin looks like the following snippet:

```
<If condition={this.canShowSecretData}>
  <SecretData />
</If>
```

This gets transpiled into a ternary expression as follows:

```
{canShowSecretData ? <SecretData /> : null}
```

The `If` component is great, but if for some reason you have nested conditions in your render method, it can easily become messy and hard to follow. Here is where the `Choose` component comes in handy:

```
<Choose>
  <When condition={...}>
    <span>if</span>
  </When>
  <When condition={...}>
    <span>else if</span>
  </When>
  <Otherwise>
    <span>else</span>
  </Otherwise>
</Choose>
```

Please note that the preceding code gets transpiled into multiple ternaries.

Finally, there is a *component* (always remember that we are not talking about real components but just syntactic sugar) to manage the loops which is also very convenient:

```
<ul>
  <For each="user" of={this.props.users}>
    <li>{user.name}</li>
  </For>
</ul>
```

The preceding code gets transpiled into a map function–no magic there.

If you are used to using **linters**, you might wonder why the linter is not complaining about that code. In fact, the variable `user` does not exist before the transpilation, nor is it wrapped into a function. To avoid those linting errors, there is another plugin to install: `eslint-plugin-jsx-control-statements`.

If you did not understand the previous sentence, don't worry; we will talk about linting in the following section.

Sub-rendering

It is worth stressing that we always want to keep our components very small and our render methods very clean and simple.

However, that is not an easy goal, especially when you are creating an application iteratively and in the first iteration you are not sure exactly how to split the components into smaller ones.

So, what should we be doing when the render method becomes too big to maintain? One solution is to split it into smaller functions in a way that lets us keep all the logic in the same component.

Let's look at an example:

```
renderUserMenu() {
  // JSX for user menu
}

renderAdminMenu() {
  // JSX for admin menu
}

render() {
  return (
    <div>
      <h1>Welcome back!</h1>
      {this.userExists && this.renderUserMenu()}
      {this.userIsAdmin && this.renderAdminMenu()}
    </div>
  )
}
```