



React Quickly

AZAT MARDAN



MANNING



**MEAP Edition
Manning Early Access Program
React Quickly
Version 5**

Copyright 2016 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP edition of *React Quickly*. This book is for developers and software engineers with two to three years' experience who want to start using React for their web or mobile development. The book will focus on React as a web library for user interfaces. At the very least the book is supposed to open readers' minds to some rather unusual concepts like JSX, unidirectional data flow and declarative programming.

If you are anything like me, I'm sure you've spent lots of painful, teary-eyed hours trying to track that pesky bug in a bunch of event listeners and bidirectional data flows between multiple models and views... and failed miserably. The biggest secret that Facebook and Instagram know is not really a secret. It's open sourced. It's ready. It's waiting for you. Liberate yourself from the MVC-madness with the high performance, scalability and developer-friendly experience of React for building web, mobile and other UIs.

In this book, we'll discuss what React is and isn't and spend some time with JSX, the preferred syntax of React. First, we'll work on making React interactive with states and events, and then move into scaling react code for re-use. Next, we'll build forms for user inputs and address persistence by dealing with React components and data flow. After that, we'll look at unit testing which is an important but often forgotten step. We'll also learn how to make React more powerful by utilizing isomorphic JavaScript and discuss alternative rendering such as React Native. Finally, we'll look at some issues related to routers and ES6.

Because your feedback is essential to creating the best book possible, I hope you'll be leaving comments in the Author Online forum. After all, I may already know how to do all this stuff, but I need to know if my explanations are working for you!

Thank you once again!

—Azat Mardan

brief contents

PART 1: CORE REACT

- 1 Meeting React*
- 2 Baby Steps with React*
- 3 Introduction to JSX*
- 4 Making React Interactive With States*
- 5 React Component Lifecycle Events*
- 6 Handling Events in React*
- 7 Working with Forms in React*
- 8 Scaling React Components*
- 9 Project: Menu*
- 10 Project: Tooltip*
- 11 Project: Timer*

PART 2: REACT & FRIENDS

- 12 The Webpack Build Tool*
- 13 React Routing*
- 14 Working with data using Redux and GraphQL*
- 15 Unit Testing React with Jest*
- 16 React on Node and Universal JavaScript*
- 17 Project: Nile Book Store with React Router*
- 18 Project Password with Jest*
- 19 Project: Autocomplete with Jest, Express and MongoDB*

APPENDIXES

- A Installation*
- B React Cheatsheet*
- C Express Cheatsheet*
- D MongoDB \ Mongoose Cheatsheet*
- E ES2015 Cheatsheet*

1

Meeting React

In this chapter, we'll cover the following topics:

- What is React and What Problems it Solves
- How Does React Fit Into My Web Application?
- Your First React Code: Hello World



Figure 1.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch01>

It's a fact that as the Internet has evolved, the complexity of a building websites went up dramatically. They have become web applications with complex user interfaces, business logic and data layers that require changes and updates over time, in many case in real-time. When I started with web development in early 2000, all you needed is some HTML and server-side language like Perl or PHP to generate it.

Nowadays, many JavaScript template libraries have been written to try and solve the problems of dealing with complex user interfaces (UIs), but they still require developers to adhere to the old separations of concerns which don't meet modern day needs any more.

React, by contrast, is a powerful UI library that offers an alternative that many big firms such as Facebook, Netflix, and Airbnb have adopted and see as the way forward.

Instead of defining a one off template for your UIs, *React allows you to create reusable UI components in JavaScript that you can use again and again in your sites*. Do you need a captcha control or date picker for your forms? Then use React to define a `<Captcha />` or `<DatePicker />` component which you can just add to your form; a simple drop-in component which has all the functionality and logic to communicate with the back-end. Do you need some form of an auto-complete box that asynchronously queries a database once the user has typed four or more letters? Define an `<Autocomplete charNum="4"/>` component that makes that asynchronous query. You can choose whether it has a textbox UI as well or whether it has no UI, but uses another custom form element instead: `<Autocomplete textbox="???" />` perhaps.

This approach isn't new. Creating *Composable UIs* has been around for a long time, but React is the first to use pure JavaScript without templates to make this possible. And this approach has proven easier to maintain, re-use and extend. However, React is only a UI library rather than a complete SPA framework or something similar.

Given these points, React is a great library for UI and it's a part of your front-end web toolkit rather than a complete solution to all of the front-end web development. In this chapter, we're going to look at the pros and cons of using React in your applications, and how you might fit it into your existing web development stack.

Assuming you want to learn more about React, Part 1 will focus on the main React concept and features while Part 2 of the book will look at working with libraries related to React to build more complex front-end apps (a.k.a. React Stack or React and Friends). Each part demonstrates both greenfield and [brownfield development](#) with React with the most popular libraries, so you can get the idea of how to approach working with it in the *real-world* scenarios.

TIP We all learn differently. Some prefer text, other videos, while others learn best with in-person instructions. Each chapter comes with short videos. They explain the gist in under 5 minutes. Watching them is totally *optional*. They allow to get a summary if you prefer a video format or need a refresher. After watching a video, you can decide if you need to read this chapter or you can skip to the next one.

The source code for the examples in this chapter is in [the ch01 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

1.1 What is React and What Problems It Solves

To introduce React.js properly the first thing we need to do is to define React.js. So what is React?

React is a User Interface (UI) component library. The UI components are created with React using JavaScript and not a special template language. This approach is called *creating composable UIs* and is fundamental to React's philosophy.

React UI components are highly self-contained concern-specific blocks of functionality. For example, there could be components for date picker, captcha, address or zip code elements. Such components have both: the visual representation and the dynamic logic. Some components can even talk to the server on their own, e.g., an autocomplete component which fetches the autocompletion list from the server.

User Interfaces

In a broad sense, a [user interface](#) is everything which facilitates communications between computers and humans. Think of a punchcard or a mouse. They are both user interfaces. When it comes to software, engineers talk about GUI or graphical user interface which was pioneered by early personal computers such as Mac and PC. GUI consists of menus, text, icons, pictures, borders and other elements. Web elements are a narrow subset of the GUI meaning web elements live inside of browsers but there are other elements for other desktop applications in Windows, OS X and other operating systems.

To clarify, every time I mention user interface in this book, I imply a web GUI.

As mentioned earlier, component-based architecture or CBA (don't confuse with Web Components which is just one of the most recent implementations of CBA) existed before React, and generally they tend to be easier to re-use, maintain and extend rather than monolithic UIs. What React bring to the table is the use of pure JavaScript (without templates) and a new way to look at composing components.

Then what problem React solves? Looking at the last few years in web development, React was mostly born out of the problem how to manage and build complex web UIs for front-end applications. Think large web apps like Facebook! One of the most painful tasks when developing such application is managing how the views change with the changes in data.

Let's refer to the official React's website for more hints at the problem React solves: "*We built React to solve one problem: building large applications with data that changes over time.*" Interesting! Moreover, we can look into the history of React for more info on the problem it was designer to tackle. In one of the [interviews](#), it was said that the creator of React Jordan Walke was solving a problem at Facebook of having *multiple data sources updating an autocomplete field*. The data was coming from a back-end asynchronously. It was becoming more and more complex to determine where to insert the new rows in order to re-use the DOM elements. He decided to just generate the field representation (DOM elements) each time anew. It was an elegant in its simplicity solution—UIs as functions. Call them with data and you get rendered views each time predictably.

Later, it turned out that generating elements in memory is extremely fast and that the actual bottleneck is the rendering in the DOM. However, React team came up with an algorithm which avoid unnecessary paining of the DOM. It made React very fast (cheap performance wise). React's splendid performance along with it's developer-friendly component-based architecture is a winning combination. These and other benefits of React are in the next section.

So that was the original problem React solved for Facebook, but many large firms agree with this approach. React adoption is solid and its popularity is growing every month. React emerged from Instagram and is now used, not only by Instagram, but by Facebook, PayPal, Uber, Sberbank, [Asana](#), [Khan Academy](#), [HipChat](#), [Flipboard](#), and [Atom](#) to name just [a few](#). Most of these applications originally used something else (typically template engines with Angular or Backbone), but switched to React and are extremely happy about it.

1.1.1 React Benefits

Every new library or framework claims to better its predecessors in some aspect or another. In the beginning, we had jQuery, and it was leaps and bounds better for writing cross-browser code in native JavaScript. If you remember, a single AJAX call taking many lines of code had to account for Internet Explorer and Webkit-like browsers. With jQuery, it takes only a single call; `$.ajax()`, for example. And, back in the day, jQuery used to be called a framework—but not anymore! Now framework is something bigger and more powerful.

Similarly with Backbone and then Angular, each new generation of JavaScript frameworks bring something new to the table. React is not unique in this. But what is new is that *React challenges some of the core concepts used by most popular front-end frameworks*. For example, one of them is that you need to have templates.

The following list highlights some of the benefits of React over some of the other libraries and frameworks:

- Build simpler apps: Component-based architecture with pure JavaScript, declarative style, and powerful developer-friendly DOM abstraction (and not only DOM, but iOS, Android and other)
- Build fast UIs: React has an outstanding performance thanks to its virtual DOM and the smart reconciliation algorithm which as a side benefit allows to perform testing without spinning a headless browser
- Write less code: With great community and **a vast ecosystem of components** developers have an active community which provide variety of libraries and components is very important when considering what framework to use for your development

There are many features that make React simpler to work with than most other front-end frameworks. Let's unpack these items one by one starting with simplicity.

SIMPLICITY

The concept of simplicity in computer science is a quality which is one of the most sought by developers and users. It doesn't equate to ease. Something simple can be hard to implement but in the end it will be more elegant and efficient. Furthermore, very often an easy thing will end up being complex. Simplicity is closely related to the [KISS principle \(keep it simple, stupid\)](#). The gist is that simpler systems work better.

React's approach allows for more simple solutions via a dramatically better web development experience for the software engineers. In fact when I just started working with React, it was a dramatic shift in a positive direction which reminded me of switching from plain no-framework JavaScript to using jQuery.

In React, this simplicity is achieved with the following features:

- Declarative over imperative style: React embraces declarative style over imperative by updating the views automatically
- Component-Based Architecture using pure JavaScript: React is not using any Domain-Specific Languages (DSL) for its components, just pure JavaScript; more over, there's no separation when working on the same functionality.
- Powerful abstractions: React has a powerful abstraction over DOM which allow to have normalized events and other interfaces which work similarly across browsers

Let's cover them one by one.

DECLARATIVE OVER IMPERATIVE STYLE

First, React embraces declarative style over imperative. Declarative style means developers write *how it should be*, not *what to do step-by-step(imperative)*, but why is declarative style is a better choice? The benefit is that declarative style is reduces complexity and makes it easier to read and understand code.

Consider this short JavaScript example to illustrate the difference between declarative and imperative programming. Let's say we need to create an array (`arr2`) whose elements are the results of doubling the elements of another array (`arr`). We can use a `for` loop to iterate over an array and tell the system what to do; that is, multiply by two and create a new element (`arr2[i]=`):

```
var arr = [1,2,3,4,5],
  arr2 = []
for (var i=0; i<arr.length; i++) {
  arr2[i]=arr[i]*2
}
console.log('a', arr2)
```

The result of this snippet, where each element was multiplied by 2, will be printed on the console as:

```
a [2, 4, 6, 8, 10]
```

This illustrates imperative programming and it works—until it's not working due to the complexity of code. It becomes too hard to understand what the end result is suppose to be when you have too many imperative statements. Gladly, we can rewrite the same logic in declarative style with `map()`:

```
var arr = [1,2,3,4,5],
  arr2 = arr.map(function(v, i){ return v*2 })
console.log('b', arr2)
```

The output is `b [2, 4, 6, 8, 10]` with the variable `arr2` being the same as in the previous example. Which code snippet is easier to read and understand? In my humble opinion, it's the declarative example.

Take a look at this imperative code (what to do) for getting a nested value of an object. The expression needs to return us a value based on a string such as `account` or `account.number` in such a manner that these statements print `true`:

```
var profile = {account: '47574416'}
var profileDeep = {account: { number: 47574416 }}
console.log(getNestedValueImperatively(profile, 'account') === '47574416')
console.log(getNestedValueImperatively(profileDeep, 'account.number') === 47574416)
```

This is the imperative style in which we literally telling the system what to do to get the results we need:

```
var getNestedValueImperatively = function getNestedValueImperatively(object, propertyName) {
  var currentObject = object
  var propertyNamesList = propertyName.split('.')
  var maxNestedLevel = propertyNamesList.length
  var currentNestedLevel

  for (currentNestedLevel = 0; currentNestedLevel < maxNestedLevel; currentNestedLevel++) {
    if (!currentObject || typeof currentObject === 'undefined') return undefined
    currentObject = currentObject[propertyNamesList[currentNestedLevel]]
  }

  return currentObject
}
```

Contrast it with declarative style (what is the result) which reduces the number of local variable. This simplifies the logic:

```
var getValue = function getValue(object, propertyName) {
  return typeof object === 'undefined' ? undefined : object[propertyName]
}

var getNestedValueDeclaratively = function getNestedValueDeclaratively(object, propertyName)
{
  return propertyName.split('.').reduce(getValue, object)
}
console.log(getNestedValueDeclaratively({bar: 'baz'}, 'bar') === 'baz')
console.log(getNestedValueDeclaratively({bar: { baz: 1 }}, 'bar.baz') === 1)
```

Most of us programmers have been trained to code imperatively, but in most of the cases the declarative code is simpler. In this example, having fewer variables and statements makes the declarative code easier to grasp at the first glance.

Now that was just some JavaScript code. What about React? It takes the same declarative approach when you compose UIs. First of all, React developers describe UI elements in a declarative style. Then, when there are changes to view generated by those UI elements, React takes care of the updates. Yay!

The convenience of React's declarative style comes to shine fully when there are changes that needs to be made to the view. Those changes are called changes of the *internal state*. When state changes, React will update the view accordingly.

NOTE : We'll cover how states work in chapter 5.

Under the hood, React uses a **virtual DOM** for finding differences (delta) between what is already in the browser and the new view. It's called DOM diffing or reconciliation of state and view (bring them back to similarity). *This allow developers not to worry about explicitly changing the view.* All they need to do is update the state and the view will be updated automatically as needed.

Conversely, with jQuery we would need to implement the updates imperatively. By manipulating the DOM, developers can programmatically modify the web page or certain parts of the web page (a more likely scenario) without re-rendering the entire page. DOM manipulation is what we do when we invoke jQuery methods.

Some frameworks like Angular can perform automatic view updates. In Angular it's called two-way data binding which basically means that views and models have a two-way communication/synching of data between them.

jQuery and Angular approaches are not great for two different reasons. Think about them as two extremes. On one end, the library (jQuery) is not doing anything and a developer (you!) needs to implement all the updates manually. On the other end, the framework (Angular) is doing everything.

The jQuery approach is prone to mistakes and take more work to implement. Also, this approach of direct manipulation of the regular DOM works fine with simple UIs, but it's very limiting when dealing with a lot of elements in the DOM tree. This happens because it's harder to see the results of imperative functions than declarative statements.

The Angular approach is hard to reason about because with its two-way binding things can spiral out of control quickly. You start putting more and more logic and now out of a sudden, different views are updating models and those models updating other views.

Yes, the Angular approach is somewhat more readable than imperative jQuery (and less manual coding!), but there's another issue. Angular relies on templates and a domain-specific language called ng-directives (e.g., `ng-if`). We'll discuss its drawback in the next section *Component-Based Architecture*.

COMPONENT-BASED ARCHITECTURE USING PURE JAVASCRIPT

[Component-based architecture](#) existed before React came to the scene. Separation of concerns, loose coupling and code reuse are at the heart of this approach because it provides a lot of benefits and software engineers including web developers love CBA. A building block of the component-based architecture in React is a component class. As with other CBAs, it has many benefits with code reuse being the biggest (write less code!).

What was lacking before React came to the scene is having a pure JavaScript implementation of this architecture. You see, when you're working with Angular or Backbone or Ember or most of the other MVC-like front-end frameworks, you have one file for JavaScript and another for the template. (Angular uses term directives for components.) There are a few issues with having two languages (and two or more files) for a single component.

The HTML and JavaScript separation worked well when we had to render HTML on the server and JavaScript was only to make your text blink. Now SPAs handle complex user input and perform rendering on the browser. This means functionally HTML and JavaScript are very closely coupled together. For developers, it makes more sense if they don't need to separate between HTML and JavaScript when working on a piece of a project (component).

Consider this Angular code which displays different links based on the value of `userSession`:

```
<a ng-if="user.session" href="/logout">Logout</a>
<a ng-if="!user.session" href="/login">Login</a>
```

You can still read it but you might have doubts what `ng-if` takes, a boolean or a string, and will it hide the element or not render it at all? In the Angular case, I'm not sure if the element will hidden on true or false unless I'm familiar with how this particular directive `ng-if` works.

Compare it with the React code shown in Listing 1-TK in which it's absolutely positively clear what the value of `user.session` must be and what element (logout or login) is rendered if the value is true. Why? Because it's just JavaScript!

Listing 1-TK: Using JavaScript if/else to implement conditional rendering in React

```
if (user.session) return React.createElement('a', {href: '/logout'}, 'Logout')
else return React.createElement('a', {href: '/login'}, 'Login')
```

The second example working with templates is when you need implement iterate over an array of data and print a property. We work with lists of data all the time! Let's take a look at a `for` loop in Angular.

In Angular, we need to use a domain-specific language called directives. The directive for a `for` loop is `ng-repeat`.

```
<div ng-repeat="account in accounts">
  {{account.name}}
</div>
```

So one of the problems with templates is that often developers have to learn yet another language. In React, we use pure JavaScript (e.g., Listing 1-) which means developers don't need to learn a new language!

Listing 1.1 Composing UI for a list of account names with pure JavaScript

```
accounts.map(function(account) {
  return React.createElement('div', null, account.name)
})
```

- 1 `Array.map()` is a regular JavaScript method which takes an iterator expression as a parameter
- 2 Iterator expression returning `<div>` with account name

Let's move on and imagine a situation when we are working and making some changes to the list of accounts. We need to display account number and other fields. How do you know what fields account has besides `name`?

You need to open corresponding JavaScript file. The file which calls and uses this template. Then you'll need to find `accounts` to see what properties are there. So the second problem with having templates is that the logic about the data and the description of how that data should be rendered are separated.

It's actually much better to have JavaScript and the markup in one place so we don't have to switch between file and languages. This is exactly how React works and you've already seen it with Hello World how React renders elements.

NOTE: Separation of concerns generally is a good pattern. In a nutshell, it means separation of different functions such as data service, view layer, etc. When we are working with template markup and corresponding JavaScript code, we are working on **one functionality**. That's why having to file `.js` and `.html` is not a separation of concerns.

Now, if we want to explicitly set the method by which to keep track of items (for example to ensure there's no duplicates) in the rendered list of items, we can use `track by` feature of Angular:

```
<div ng-repeat="account in accounts track by account._id">
  {{account.name}}
</div>
```

If we want to track by an index of the array, there's `$index`:

```
<div ng-repeat="account in accounts track by $index">
```

```
  {{account.name}}
</div>
```

But what isconcerting to me and many other developers is how and what is this magic \$index?

In React, we use an argument from `map()` for the value of the `key` attribute:

Listing 1.2 Rendering a list of items using `Array.map()`

```
accounts.map(function(account, index) {           ①
  return React.createElement('div', {key: index}, account.name)    ②
})
```

- ① Use array element `value` (`account`) and its index provided by `Array.map()` `
- ② Return a React element `'<div>'` with an attribute key with value index and inner text set to account.name`

I'm not picking on Angular. It's a great framework. However, the bottom line is that with framework that use a domain-specific language you need to learn its magic variables and methods. In React, we can just use pure JavaScript!

If you use React, then you can carry your knowledge to the next project even if it's not in React. On the other hand, if you use an X template engine (or an Y framework with a built-in DSL template engine), then you are "locked in" to that system and have to describe yourself as an X/Y developer. Your knowledge is not transferable to projects which don't use X/Y. :(

To summarize, **pure JavaScript component-based architecture** is about discrete, well-encapsulated and reusable components ensure better separation of concerns based on functionality without the need of DSL, templates or directives

Working with many developer teams I observed another factor related to simplicity. React has a better, shallower and more gradual learning curve compared to MVC frameworks (well, React is not an MVC, so I'll stop comparing them) and template engines that have special syntax, for example, angular directives or Jade/Pug. The reason is that instead of leveraging of power of JavaScript, most template engines build abstractions with their own language in a way reinventing things like an `if` condition or a `for` loop.

POWERFUL ABSTRACTIONS

React has a powerful abstraction of the Document Model. In other words, it hides the underlying interfaces and provides normalized/synthesized methods and properties. For example, when you create a `onClick` event in React, the event handler will receive not a native browser-specific Event object, but a Synthetic Event object which is a wrapper around native Event objects. We can expect the same behavior from Synthetic Events irrespective of the browser in which we run the code. React also has a set of synthetic events for touch events which are great for building web apps for mobile devices.

Another example of React's DOM abstraction is that you can render React elements on the server. This can become handy for better search engine optimization (SEO) and/or improving performance.

There are more options when it comes to rendering React components than just DOM or strings for server back-end. I cover them in the *React Rendering Targets* section of this chapter.

Speaking of the DOM, one of the most sought after benefits of React is its splendid performance.

SPEED AND TESTABILITY

In addition to the necessary DOM updates, your framework might perform unnecessary updates as well, which makes the performance of complex UIs even worse. This becomes especially noticeable and painful for users when you have a lot of dynamic UI elements on your web page.

On the contrary, React's virtual DOM exists only in the JavaScript memory. Every time there's a data change, React first compares the differences using its virtual DOM, and only when the library knows there has been a change in the rendering will it update the actual DOM. Take a look at the diagram Figure 1.1 to see a high-level overview of how React's virtual DOM works when there are data changes.

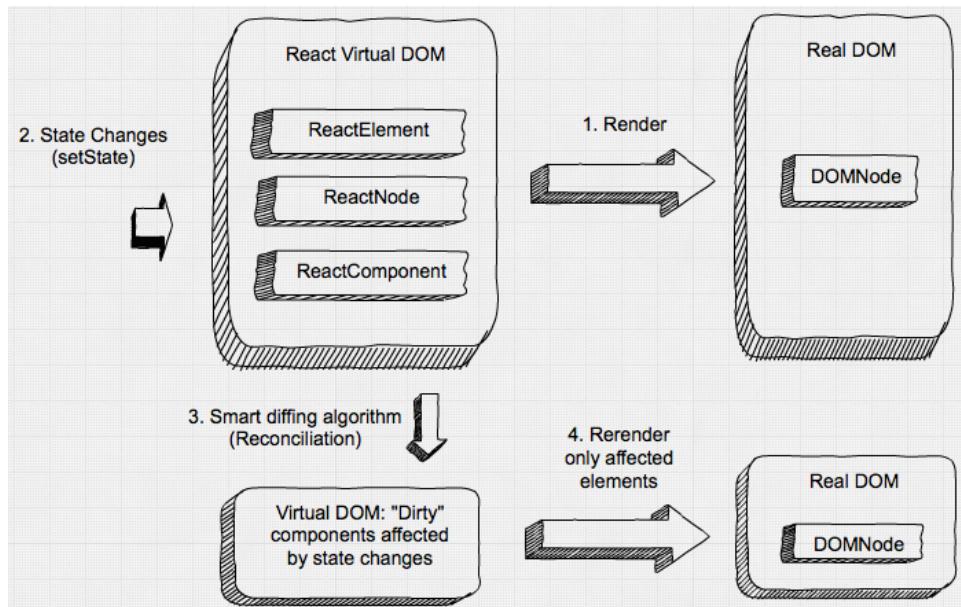


Figure 1.2 Once a component has been rendered, if its state changes it is compared to the in-memory

virtual DOM and re-rendered if necessary

Ultimately, React updates only those parts that are absolutely necessary so that the internal state (virtual DOM) and the view (real DOM) are the same. For example, if there's a `<p>` element and we augment the text via the state of the component, only the text will be updated (that is, `innerHTML`), not the element itself. This results in increased performance compared to re-rendering entire sets of elements or, even more so, entire pages (server-side rendering).

If you like to geek out on algorithms and Big Os, then these two articles do a great job at explaining how React team managed to turn $O(n^3)$ problem into a $O(n)$ one:

- [Reconciliation](#)
- [React's diff algorithm](#)

The added benefit of the virtual DOM is that you can do unit testing without headless browsers like [PhantomJS](#). There's a [Jasmin](#) layer called [Jest](#) that lets you test React components right on the command line!

GREAT ECOSYSTEM AND COMMUNITY

Last, but not least, React is supported by developers of a juggernaut web application called Facebook, as well as by their peers over at Instagram. As with Angular and some other libraries, having a big company behind the technology provides a sound testing ground (it's deployed to millions of browsers), a re-assurance in the future, and an increase in contribution velocity.

Take a look at these community resources:

- <http://react-components.com>: Searchable database of React components
- [Material-UI](#): Material design React components
- <http://react-toolbox.com>: Set of React components that implement Google Material Design specification
- <https://js.coach>: Opinionated catalog of open source JS (mostly React) packages
- <https://react.rocks>: Catalog of React components
- <https://khan.github.io/react-components>: Khan Academy React components
- <http://www.reactjsx.com>: Registry of React components

My personal anecdotal experience with open source taught me that the marketing of open source projects is as important to its wide adoption and success as is the code itself. By that I mean, if a project has a poor website, lacks documentation and examples, and has an ugly logo, most developers won't take it seriously, especially now, when there are so many JavaScript libraries! Developers are picky and they won't use the ugly duckling library.

My teacher used to say: "Don't judge a book by its cover." This might sound controversial, but sadly most people, including software engineers, are prone to such biases as good branding. Luckily, React has a great engineering reputation backing it. And, speaking of book covers, I hope you didn't buy this book just for its cover! :)

1.1.2 React Disadvantages

Of course, almost everything has its drawbacks. This is true with React, but the full list of cons really depends on whom you ask. Some of the differences like declarative vs. imperative are highly subjective. Therefore, they can be both pros and cons. Here's my list of React disadvantages (and as with any such list, it could be biased because it's based on opinions I've heard from other developers):

- React is not a full-blown, swiss army knife type of framework: Developers need to pair it with out libraries like Redux or React Router to achieve functionality comparable to Angular or Ember. This can also be an advantage if you need a minimalistic UI library to integrate with your existing stack.
- React is not as mature yet as other framework: React core's API still is changing albeit very little after the 0.14 release; the best practices for React (as well as the ecosystem of add-ons) are still developing.
- React uses somewhat new approach to web development: JSX and Flux (often used with React as the data library) can be intimidating to beginners. There is a lack of the best practices, good books, courses, and resources to master React.
- React has only a one-way binding: Although one-way binding is better for complex apps and removes a lot of complexity, some developers (especially Angular developers) who got used to a two-way binding will find themselves writing a little bit more code. I will explain how React's one-way binding works compared to Angular's two-way binding in chapter 14 where we cover working with data.
- React is not reactive (as in reactive programming and architecture which are more event-driven, resilient and responsive) out of the box: Developers need to use other tools such as [Reactive Extensions \(RxJS\)](#) to compose asynchronous data streams with Observables.

To continue with React introduction let's take a look at how it fits into the a web application.

1.2 How Does React Fit Into My Web Application?

In a way, just the React library by itself, without React Router or a data library, is less comparable to frameworks (like Backbone, Ember, or Angular), and more comparable to libraries for working with user interfaces like template engines (Handlebars, Blaze) and DOM manipulation libraries (jQuery, Zepto).

In fact, many teams have swapped traditional template engines like Underscore in Backbone or Blaze in Meteor with React with great success. For example, PayPal switched from Dust to Angular, and then finally to React as Jeff Harrel announced at a conference (Figure 1-8).

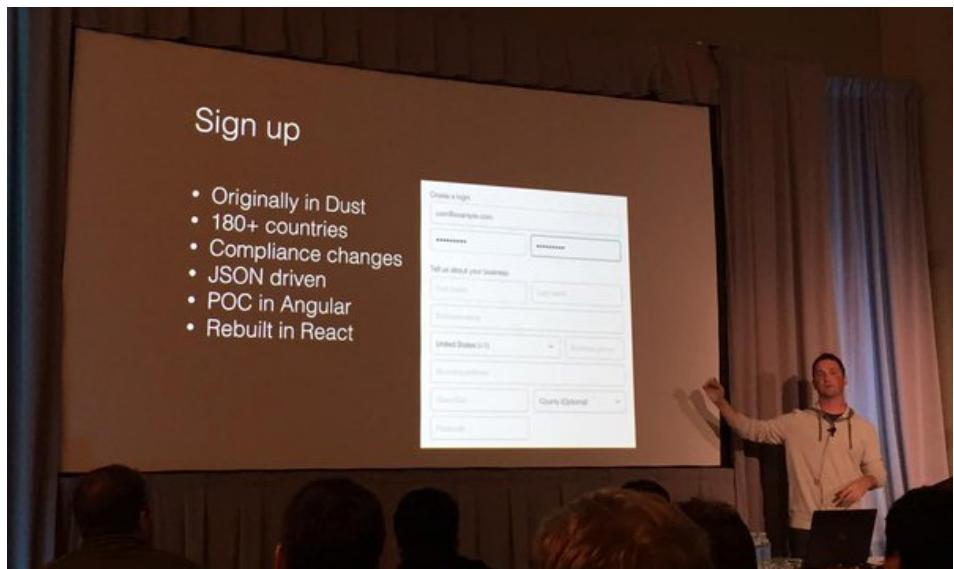


Figure 1.3 Jeff Harrell, Head of Payments and Engineering at PayPal on switching to React

Moreover, you can use React for just part of your UI. For example, let's say you have a load application form on a web page built with jQuery. You can gradually start to convert this front-end app to React by first converting the city and state fields to populate automatically based on the zip code.

You can keep the rest of the form using jQuery. Then if you want to proceed, you can convert the rest of the form elements from jQuery to React until your entire page is built on React. Taking a similar approach, many team successfully integrated React with Backbone, Angular or other existing front-end frameworks.

React is back-end agnostic for the purposes of the front-end development. In other words, you don't have to rely on a Node.js backend or MERN (MongoDB, Express.js, React.js and Node.js) to use React. It's perfectly fine to use React with any other backend technology like Java, Ruby, Go or Python. React is a UI library after all. You can integrate it with any backend and any front-end data library (Backbone, Angular, Meteor, etc.).

To summarize how React fits into a web app, it's most often used in these scenarios:

- As a UI library in a React-related-stack SPAs, e.g., React+React Router+Redux
- As a UI library (V in MVC) in a non-fully-React-related-stack SPAs, e.g.,

React+Backbone

- As a drop-in UI component in *any* front-end stack, e.g., a React autocomplete input component in some jQuery+server-side rendering stack
- As a server-side template library in a purely thick-server (traditional) web app or in a hybrid or isomorphic/universal web app, e.g., an Express server which uses `express-react-views`.
- As a UI library in mobile apps, e.g., React Native iOS app.
- As a UI description library for different rendering targets (next section)

So React works nicely with other front-end technologies, but we mostly use it as a part of single-page architecture just because SPA seems to be the most advantageous and popular approach to building the web apps. I'll cover how React fits into an SPA in *Single-Page Applications and React*.

In some extreme scenarios, you can even use React **only on the server** as a template engine of sorts. For example, there is an `express-react-views` library. What it does is renders view server-side from React components. This server-side rendering is possible because React allows to use different rendering targets.

1.2.1 React Libraries and Rendering Targets

In versions 0.14 and higher, React team split the library into two packages: React Core (`react` package on npm) and React DOM (`react-dom` package on npm). By doing it, maintainers of React made it clear that React is on a path to become not just a library for web, but a universal (or sometimes called isomorphic because it can be used in different environments) library for describing user interfaces which can be utilized in different environments.

For example in version 0.13, React had method `React.render()` to mount an element to a web page's DOM node. Now in versions 0.14 and higher, developers need to include `react-dom` and call `ReactDOM.render()` instead of `React.render()`.

Having multiple packages created by community to support various rendering targets made this approach of separating writing components and rendering logical. Some of these modules include:

- [`react-blessed`](#): Renderer for terminal interface [`blessed`](#)
- [`react-art`](#): Renderer for [`the ART library`](#)
- [`react-canvas`](#): Renderer for `<canvas>`
- [`react-three`](#): Renderer for [`3D`](#) library using [`three.js`](#)

In addition to the support of aforementioned libraries, the separation of React core and React DOM made it easier to share the code between React and React Native libraries (used for native mobile iOS and Android development).

In essence, when using React for web developer will need to include at least React core, and React DOM. More over, there are [React utility libraries called add-ons](#) which allow you to enhance functionality or perform testing. They are in an experimental stage and are likely to evolve going forward.

Last, React is almost often used with JSX—a tiny language which allows developers to write React UIs more eloquently. They can transpile JSX into regular JavaScript by using Babel or a similar tool.

As you can see, there are a lot of modularity meaning the functionality of React related things is split into different packages. This gives the power and the choice to developers which is a good thing. There's no monolith or an opinionated library that dictates you the only possible way to implement things. More on this in React Stack.

Generally speaking, if you are a web developer reading this book, most likely you use the single-page application (SPA) architecture. You either already have a web app built using and you want to re-engineer it with React (brownfield), or you starting a new project from scratch (greenfield). Therefore, next we'll zoom in on React's place in SPAs as the most popular approach to building web apps.

1.2.2 Single-Page Applications and React

The next diagram is very basic and shows a typical single-page application (SPA) architecture with a user, browser and server. Another name for SPA architecture is thick client, because browser being a client holds more logic and performs functions such as rendering of the HTML, validation, UI changes and others.

Let's start with a bird's-eye view and take a look at the SPA architecture diagram shown on Figure 1.4 which shows processes happening between user, browser and server.

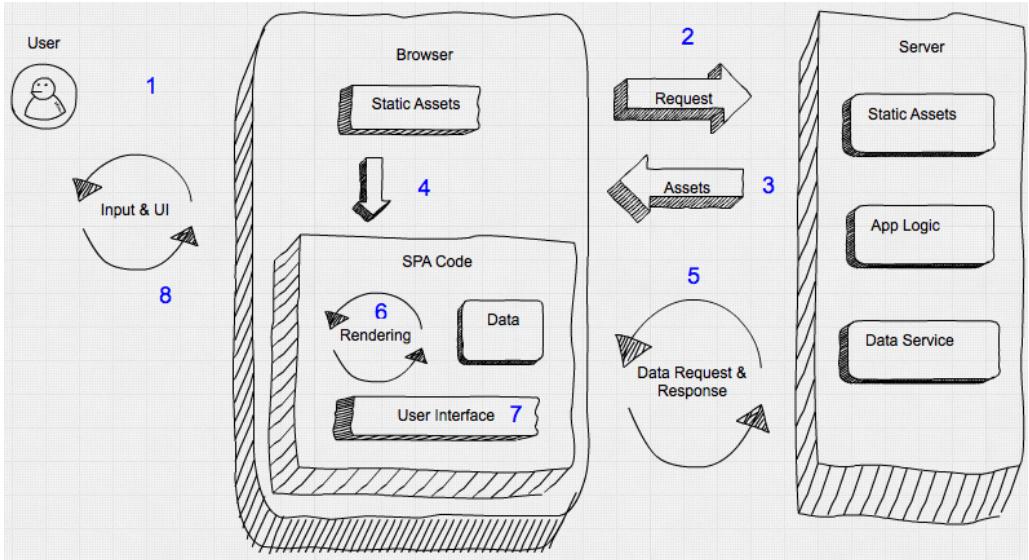


Figure 1.4 A typical single-page application architecture

The diagram on Figure 1.4 of a typical single-page application depicts a user making a request and inputs (actions like click a button, drag&drop, mouse hover, etc.).

1. User type a URL in the browser to open a new page
2. The browser sends a URL request to the server which in turn responds with static assets such as HTML, CSS and JavaScript.
3. In most cases the HTML is bare-bone, i.e., it has only a skeleton of the web page. In many cases there would be a message "*Loading...*" and the rotating spinner GIF.
4. The Static Assets will contain the JavaScript code for the SPA. When loaded, this code will make additional requests for data (AJAX/XHR request).
5. The data comes back in JSON, XML or any other format.
6. Once SPA receives the data, it can render missing HTML (User Interface block on the diagram). In other words, UI rendering happens on the browser by SPA hydrating templates with data.
7. Once the browser rendering is done, SPA will replace the "*Loading...*" message and the user will be able to work with the page.
8. The user sees a beautiful web page. The user may interact with the page (Input), triggering new requests by SPA to Server and the cycle of steps 2-6 continues. At this stage browser routing might happen if SPA implements it meaning that the navigation to a new URL will not actually trigger a new page reload from the server but a SPA re-render in the browser.

To summarize, in the SPA approach most of rendering for user interfaces happens on the browser. Only the data travels to and from browser. Contrast that with a think server approach where all of the rendering happens on the server. (Rendering as in generating HTML from templates or UI code, not as in actually rendering that HTML in the browser which sometimes called paining or drawing the DOM.)

Let's pause for a moment on the SPA code assuming we use an MVC-like framework. The MVC-like architecture is the most popular approach, but the only one. In fact, React is not requiring you to use an MVC-like architecture. For the sake of simplicity, let's assume that our SPA is using an MVC-like architecture. We can see what distinct parts it could have in Figure 1-2. There is navigator or a routing library which acts as a controller of sort in the model-view-controller paradigm. It dictates what data to fetch and what template to use.

Navigator/controller makes a request to get the data and then hydrates the templates (views) with this data to render user interface in the form of the HTML. User interface sends actions back to the SPA code, e.g., clicks, mouse hover, key strokes, etc.

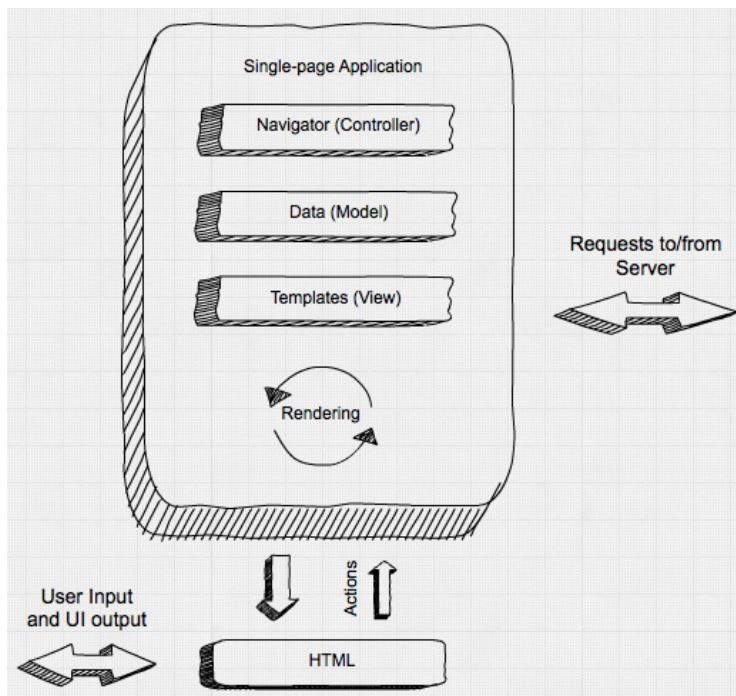


Figure 1.5 inside-spa

In SPA architecture, data is interpreted and processed in the browser (browser rendering) and is used by SPA to render additional HTML or to change existing HTML. This makes for nice

interactive web applications that rival the desktop ones. Angular.js, Backbone.js, and Ember.js are examples of front-end frameworks for building SPAs.

SIDENOTE: Different framework implement navigator, data and templates differently so the diagram 1-2 is not applicable to all frameworks, but rather illustrates the most wide-spread separation of concerns in a typical SPA.

When it comes to React, its place in the SPA diagram (figure 1-2) is in the template block. React is a view layer so you can use it to render HTML by providing it with data. Of course, React is doing much more than your typical template engine.

What is different between React and other template engines like Underscore, Handlebars or Mustache, is that in how you develop, update UIs, and manage their states.

We'll talk about states in chapter 5 in more details. For now, think about states as some data which change and which is related to user interface.

1.2.3 React Stack

React's not a full-blown front-end JavaScript framework. React is very minimalistic. It's not enforcing a particular way for doing things like data modeling, styling or routing (a.k.a. non-opinionated). Because of that, developers need to pair React with a routing and/or modeling library.

For example, a project which already uses Backbone.js and the Underscore.js template engine can switch Underscore for React, and keep existing data models and routing from Backbone. Other times, developers opt to use the so called React stack consisting of data and routing libraries created to be specifically used with React:

- [RefluxJS](#), [Redux](#), [Meteor](#) or [Flux](#): Data model libraries or backends
- [React Router](#): Routing library
- [React Bootstrap](#): Collection of React components to consume Twitter Bootstrap library

The ecosystem of libraries for React is growing everyday. Also, React's ability to describe composable components (a self-contained chunks of UI) is very helpful in reusing of the code. There are many components packaged as npm modules. Just to illustrate you the point that having small composable components is good for code reuse, here are some of the popular React components:

- [react-datepicker](#): Datepicker component
- [react-forms](#): Set of tools to handle form rendering and validation
- [react-autocomplete](#): WAI-ARIA compliant autocomplete (combobox) component

Then there's JSX which is probably the most frequent argument for not using React. If you're familiar with Angular, then you've already had to write quite a lot of JavaScript in your template code. This is because in modern web development, plain HTML is too static and

hardly can be of any use by itself. My advice, have a benefit of the doubt and give JSX a fair run.

JSX, or JavaScript Extension, is a hybrid of JavaScript and XML/HTML. React plays nicely with JSX because developers can better implement and read code. Think about JSX as a mini-language that is compiled into native JavaScript (Listing 1-TK). So JSX is not run on the browser itself, but is used as the source code for compilation.

```
if (user.session)
  return <a href="/logout">Logout</a>
else
  return <a href="/login">Login</a>
```

Even if you load a JSX file in your browser with the run-time transformer library that compiles JSX into native JavaScript on the run, you still don't run the JSX, but you run JavaScript instead. In this sense, JSX is akin to CoffeeScript. We compile these languages into native JavaScript to get better syntax and features than regular JavaScript has.

I know for some of you it looks bizarre to have XML interspersed with JavaScript code. It took me a while to adjust because I was expecting an avalanche of syntax error messages. And yes, JSX is optional. For these two reasons, I'm not covering JSX until chapter 3, but trust me, it's powerful once you get hold of it.

By now you have an understanding of what React is, its stack and its place in the higher level single-page application architecture. It's time to get our hand dirty and write our first React code.

1.3 Your First React Code: Hello World

Let's explore our first React code, the quintessential example used for learning programming languages, the Hello World application. (If we don't do it, the God's of programming might punish us!) We won't be using JSX just yet, just plain JavaScript. The project will print "Hello World" heading (`<h1>`) on a web page. Figure 1-3 shows how it will look like when we're done.



Figure 1.6 Hello World

Although most React developers write in JSX, browsers will only run the standard JavaScript. That's why it's beneficial to be able to understand React code in pure JavaScript. Another reason that we start with plain JS is to show that JSX is optional, albeit it's de facto the standard language for React. Last, pre-processing JSX requires some tooling. We want to get started with React as soon as possible without spending too much time on the setup in this chapter. We'll perform all the necessary setup for JSX in chapter 2.

The folder structure of the project is rather very simple. It consists of two JavaScript files inside of the `js` folder, and one HTML file `index.html`:

```
/hello-world
  /js
    react-15.0.2.js
    react-dom-15.0.2.js
  index.html
```

The two files in the `js` folder are for the React library version 15.0.2: `react-dom-15.0.2.js` (web browser DOM renderer) and `react-15.0.2.js` (React core package).

First, we need to download the aforementioned React and React DOM libraries. There are many ways to do it. I recommend using the files provided in the source code for this book which you can find on [GitHub](#) and [Manning](#). This is the most reliable and easiest approach because it doesn't require dependency on any other service or tool. You can find more ways to download React in Appendix A.

NOTE Prior to version 0.14, these two libraries were bundled together. For example, for version 0.13.3, all you needed was `react.js`. We are using the version 15.0.2, so we'll need two files: `react.js` and `react-dom.js`.

After you placed the React files into the `js` folder, create `index.html` in the project folder `hello-world`. This HTML file will be the entry point of our Hello World application (meaning we'll need to open it in the browser).

The code for `index.html` is rather simple and starts with the inclusion of the libraries in `<head>`. In the `<body>` element, we create a `<div>` container with the ID `content` and a `<script>` element (that's where our app's code will go later) as shown in Listing 1.1.

Listing 1.3 Loading React libraries and code for Hello World (ch01/hello-world/index.html)

```
<!DOCTYPE html>
<html>
  <head>
    <script src="js/react-15.0.2.js"></script> ①
    <script src="js/react-dom-15.0.2.js"></script> ②
  </head>
```

```
<body>
  <div id="content"></div>
  <script type="text/javascript">
    ...
  </script>
</body>
</html>
```

3
4

- 1 Import React library
- 2 Import ReactDOM library
- 3 Define an empty `<div>` element to mount React UI
- 4 Start React code for the Hello World view

Why not render it directly in the `<body>` element? Because it can lead to conflict with other libraries and browser extensions which manipulate the document body. In fact, if you try attaching an element directly to the body, you'll get this warning:

Rendering components directly into document.body is discouraged...

This is another good thing about React. It has great warning and error messages!

React warning and error messages are not part of the production build for the reasons of reducing the noise, security and minimizing the distribution size. The production build is the minified file from the React core library, for example `react-15.0.2.min.js`. The development version with the warnings and error messages is the unminified version, for example `react-15.0.2.js`.

By including the libraries in the HTML file, we get access to the `React` and `ReactDOM` global objects: `window.React` and `window.ReactDOM`. We'll need two methods from those objects: one to create an element (`React`) and another to render it in the `<div>` container (`ReactDOM`) as shown in Listing 1.2.

To create a React element, all we need to do is call `React.createElement(elementName, data, child)` with three arguments which have the following meaning:

1. `elementName`: HTML as a string (for example '`'h1'`') or custom component class as an object (e.g., `HelloWorld`, see Creating Component Class later)
2. `data`: Data in the form of attributes and properties (we'll cover properties later); for example, `null` or `{name: 'Azat'}`
3. `child`: Child element or inner HTML/text content; for example, `Hello world!`.

Listing 1.4 Creating h1 React element with text and rendering it into the DOM

(ch01/hello-world/index.html)

```
var h1 = React.createElement('h1', null, 'Hello world!') ①
ReactDOM.render(
  h1,
  document.getElementById('content')
) ②
```

- 1 Create and save in a variable a React element of `<h1>` type using `React.createElement()`
- 2 Render the element into the DOM using `ReactDOM.render()`

- ② Render the element `h1` in the real DOM element with ID of `content` using `ReactDOM.render()`

In Listing 1.4, we are getting the React element of the `h1` type and storing the reference to this object into the `h1` variable. The `h1` variable is not an actual DOM node. Rather it's an instantiation of `React h1 component (element)`. You can name it anyway you want, `helloWorldHeading` for example. In other words, React provides abstraction over the DOM.

NOTE The `h1` variable name is totally arbitrary. You can name it anything you want (`banana`, for example) as long as you utilize the same variable in `ReactDOM.render()`.

Once the element is created and stored in `h1`, we can render it to the actual DOM node/element with ID `content` using the `ReactDOM.render()` method as shown in Listing 1.2.

If you prefer, you can move the `h1` variable to the `render` call. The result is the same except we don't use an extra variable:

```
ReactDOM.render(  
  React.createElement('h1', null, 'Hello world!'),  
  document.getElementById('content')  
)
```

Now open the `index.html` file served by a static HTTP web server in your favorite browser. I recommend using an up-to-date version of Chrome, Safari or Firefox. As a result, you should be able to see the "Hello world!" message on the webpage as shown in Figure 1.7.

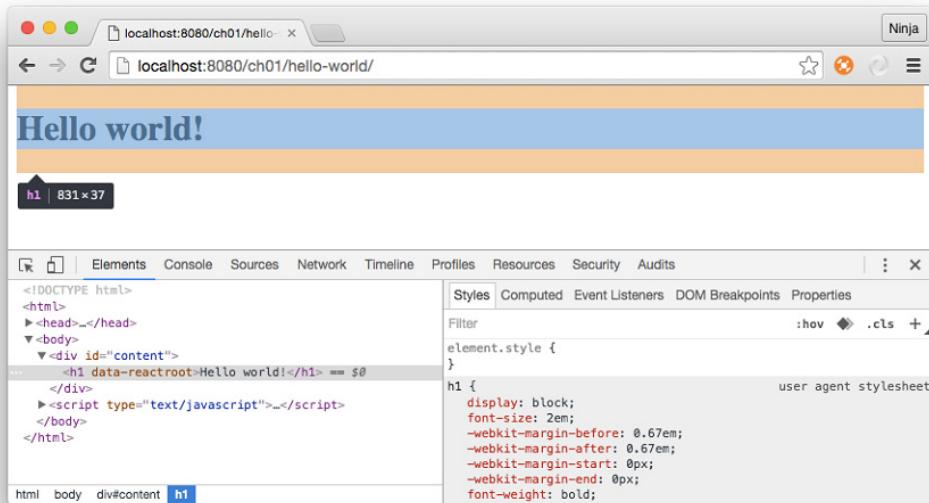


Figure 1.7 Inspecting Hello World rendered by React

Figure 1.7 illustrates the Elements tab in Chrome DevTools with the selection of the `<h1>` element. You can observe the `data-reactroot` attribute. This marks that this element was rendered by React DOM.

Local Dev Web Server

It's better to use a local web server instead of opening an `index.html` file in the browser directly, because with a web server our JavaScript apps will be able to make AJAX/XHR requests. You can tell whether it's a server or a file by looking at the URL address bar. If the address starts with `file` then it's a file and if it starts with `http` then it's a server. We'll need this feature later in the future projects. Typically, a local HTTP web server will listen to incoming requests on `127.0.0.1` or `localhost`.

You can get any of the open source web servers like Apache, MAMP or (my favorites because they are written in Node.js) [node-static](#) or [http-server](#).

To install `node-static` or `http-server`, you must have Node.js and npm installed. If you don't have them, you can find installation instructions for Node and npm in Appendix A or by going to <http://nodejs.org>.

So assuming you have Node.js and npm on your machine, run `npm i -g node-static` or `npm i -g http-server` in your Terminal / Command Prompt. Then, navigate to the folder with the source code and run `static` or `http-server`.

In my case, I'm launching `static` from the `react-quickly` folder, so I need to put the path to Hello World in my browser URL bar, i.e., <http://localhost:8080/ch01/hello-world/> (Figure 1-4).

One quick note, we can abstract our React code (Listing 1.2) into a separate file, instead of creating elements and rendering them with `ReactDOM.render()` all in the `index.html` file (Listing 1.3).

For example, we can create `script.js` and copy and paste the `h1` element and `ReactDOM.render()` call into that file. Then, in `index.html`, we need to include `script.js` after `<div>` with the ID `content` like this:

```
<div id="content"></div>
<script src="script.js"></script>
```

Congratulations! You've just implemented your first React code!

1.4 Quiz

1. Declarative style of programming does not allow for mutation of its stored values. It's "this is what I want" vs. the imperative's "this is how to do it." True or false?
2. React components are rendered into the DOM with this method (beware, it's a tricky question!): `ReactDOM.renderComponent`, `React.render`, `ReactDOM.append` or `ReactDOM.render?`
3. You have to use Node.js on the server to be able to use React in your SPA. True or false?
4. You must include `react-dom.js` in order to use render React elements on a web page? True or false?
5. The problem React solve is of updating view based on the data changes. True or false?

1.5 Summary

This is the end of the first chapter in which we covered a few important React concepts and even coded a few small React apps. Here's what you learned:

- React is declarative and it's only a view or user interface (UI) layer.
- React uses components that we bring into existence with `ReactDOM.render()`.
- React component classes are created with `React.createClass` and its mandatory `render()` method.
- React components are reusable and take immutable properties that are accessible via `this.props.NAME`.
- You use pure JavaScript to develop and compose UIs in React.
- You don't need to use JSX (an XML-like syntax for React objects); JSX is totally optional when developing with React!

These are the foundational concepts of React and to summarize the React's definition: *React for web consists of React core and React DOM. React core is a library geared towards building and sharing composable UI components using JavaScript and (optionally) JSX (an XML-like syntax) in a isomorphic/universal manner.* On the other hand, to work with React on the

browser, developers can use the React DOM library which has methods for the DOM rendering as well as for server-side rendering.

Now, you should be able to talk intelligently about React's benefits and create a Hello World app. In the next chapter, we'll cover React basics using plain JavaScript such as creation of component classes, passing properties, attributes and rendering them!

1.6 Quiz Answers

1. True, declarative is "what I want" style while the imperative is "this is how to do it".
2. ReactDOM.render
3. False, you can use any back-end technology.
4. True, you need React DOM library.
5. True, this is the main problem which React solves.

2

Baby Steps with React

In this chapter, we'll learn the following React basics

- Nesting Elements
- Creating Component Class
- Working with Properties



Figure 2.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch02>

These topics will teach us how to walk the baby steps and lay the foundation for the following chapters. This short chapter is paramount in understanding the React concepts such as element and component. In a nutshell, elements are instances of components (also called component classes). What is the use case and why we use either of them? Read on!

The source code for the examples in this chapter is in [the ch02 folder](#) of the GitHub repository [azat-co/react-quickly](https://github.com/azat-co/react-quickly). And some demos can be found at <http://reactquickly.co/demos>.

2.1 Nesting Elements

In the last chapter, we've learned how to create a React element. To remind you, the method is `React.createElement()`, for example you can create a link element like this:

```
let linkReactElement = React.createElement('a', {href: 'http://webapplog.com'}, 'Webapplog.co  
m')
```

But the problem is that most UI have more than one element (a link inside of a menu). Take a look at Figure 2.2. There are buttons in the section, video thumbnails and a YouTube player.

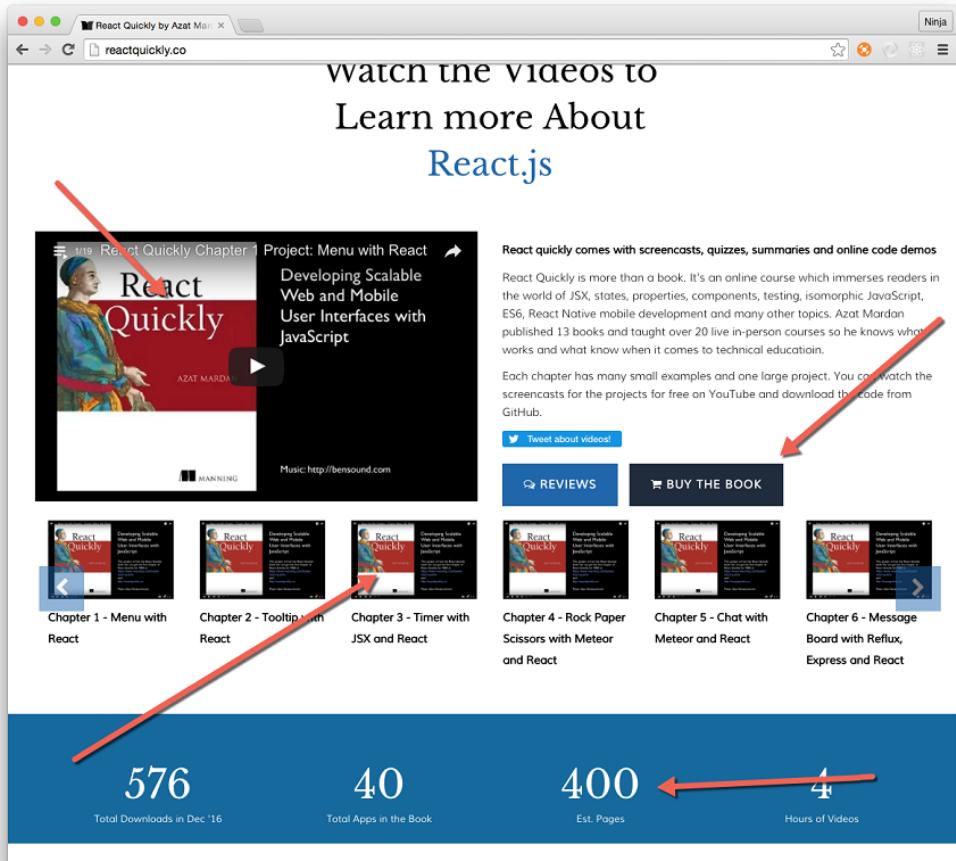


Figure 2.2 React Quickly website has many nested UI elements

The solution to creating more complex structures in a hierarchical manner is nesting elements.

In the previous chapter, you implemented your first React code by creating an `<h1>` React element and rendering it into the DOM with `ReactDOM.render()`:

```
let h1 = React.createElement('h1', null, 'Hello world!')
ReactDOM.render(
  h1,
  document.getElementById('content')
)
```

It's important to note that `ReactDOM.render()` takes only one element as an argument, which is `h1` in our example (view is shown in Figure 2.3).

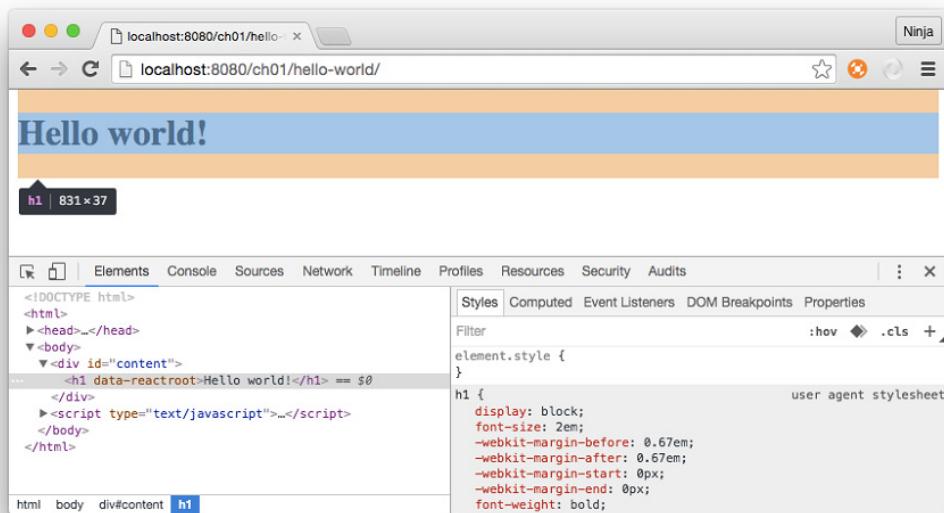


Figure 2.3 Rendering a single heading element

As mentioned in the beginning of this section, the problem arises when you need to render two same-level elements (for example, two `h1` elements). In this case, wrap the elements in a visually neutral element as depicted in the diagram on Figure 2-3. The `<div>` container is usually a good choice as well as ``.

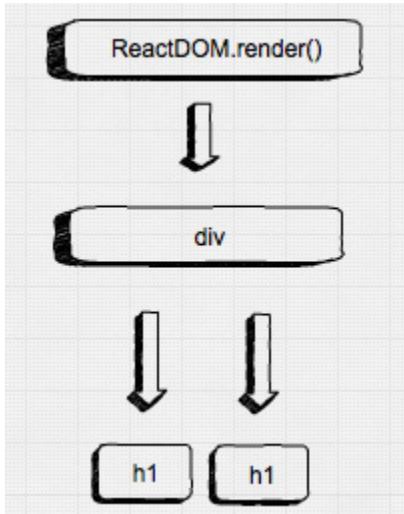


Figure 2.4 Structuring React render with the use of a wrapper div container to render sibling headings

Developers can pass unlimited number of parameters to `createElement()`. All of them after the second parameter will become children elements. Those children elements (`h1` in our case) will be siblings to one another, i.e., they'll be on the same level relative to each other as seen in Figure 2-4 when you open DevTools in Chrome.

REACT DEVTOOLS

In addition to the Elements tab, which is by default in DevTools, developers can install an additional extension (or a plugin) for React called React DevTools. You can see it as a last tab in Figure 2-4. React DevTools is available for Firefox as well. The benefits allow developers to inspect the results of React rendering closely with the component hierarchy, its name, props, states and more.

Here's the GitHub repository: <https://github.com/facebook/react-devtools>, or you can follow the links directly for [Chrome](#) and [Firefox](#).

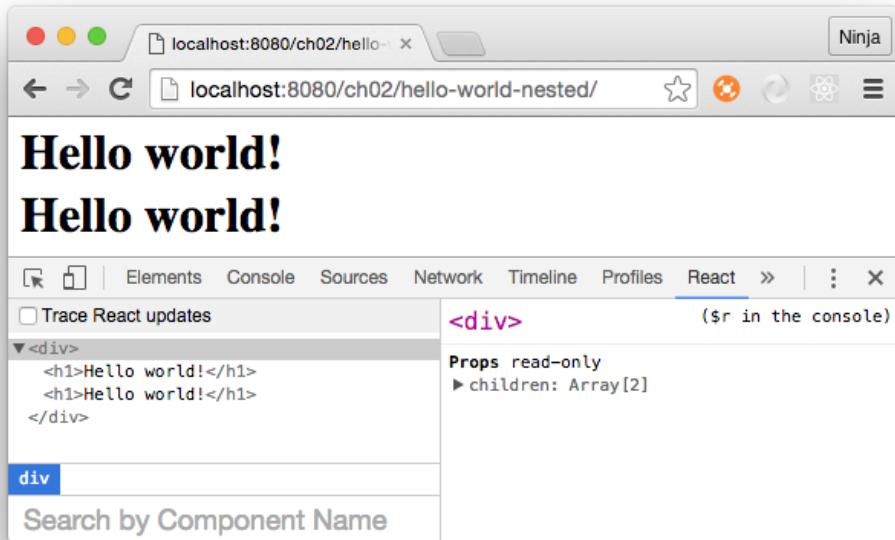


Figure 2.5 React DevTools shows a div wrapper for nested with sibling h1 elements

Knowing this, let's utilize `createElement()` to create the `<div>` element with two `<h1>` children elements as shown in Listing 2-1.

Listing 2.1 Creating `<div>` element with two `<h1>` children elements (ch02/hello-world-nested/index.html)

```
let h1 = React.createElement('h1', null, 'Hello world!') ①
ReactDOM.render(
  React.createElement('div', null, h1, h1), ②
  document.getElementById('content')
)
```

- ① If the third parameter of `createElement` is a string, it specifies the text value of the element being created
- ② If the third and subsequent parameters are not text, they specify the child elements of the element being created

The HTML code can stay the same as in the Hello World example (Listing 2-2), as long as we include the necessary React and ReactDOM libraries, and have the `content` node.

Listing 2.2 HTML for the nested elements example sans the React code (ch02/hello-world-nested/index.html)

```
<!DOCTYPE html>
<html>
  <head>
    <script src="js/react-15.0.2.js"></script>
    <script src="js/react-dom-15.0.2.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/javascript">
      ...
    </script>
  </body>
</html>
```

So far we've only provided string values as the first parameter of `createElement()`. However, the first parameter can have two types of input:

1. Standard HTML tag as a string, e.g., `'h1'`, `'div'`, `'p'`, (without the angle braces); name is lowercase.
2. React component classes as an object, e.g., `HelloWorld`; name is capitalized.

The first approach will render standard HTML elements. What happens is React goes through its list of standard HTML elements and when and if it finds a match, React will use it as a type for the React element. For example, when we pass `'p'`, React will find a match because `p` is a paragraph tag name. This will produce `<p>` in the actual DOM when/if we render this React element.

Now it's time to cover the second type of input where we creating and providing custom component classes.

2.2 Creating Component Class

After nesting elements with React, we stumble upon the next problem—soon there would be a lot of elements. We need to utilize the component-based architecture described in chapter 1. It will allow to reuse the code by separating the functionality into a loosely coupled parts. Meet component classes or just components as they often called for brevity (not to be confused with Web Components).

Think about standard HTML tags as building blocks. We can use them to compose our own React component classes which we use to create custom elements (instances of classes). By using custom elements we can encapsulate and abstract certain logic in portable classes (composable reusable components). This abstraction allows teams to reuse UIs in large and complex applications as well as in different projects. An example of such components: autocomplete, toolbox, menu, etc.

So if creating the *Hello world!* element with the HTML tag in the `createElement()` method was rather straightforward: `(createElement('h1', null, 'Hello World!'))`. What if we need to separate Hello World into its own class for the aforementioned purposes as shown in Figure 2.6. Let's say we need to reuse `HelloWorld` in 10 different projects! (Probably not, but a good autocomplete component will definitely be reused.)

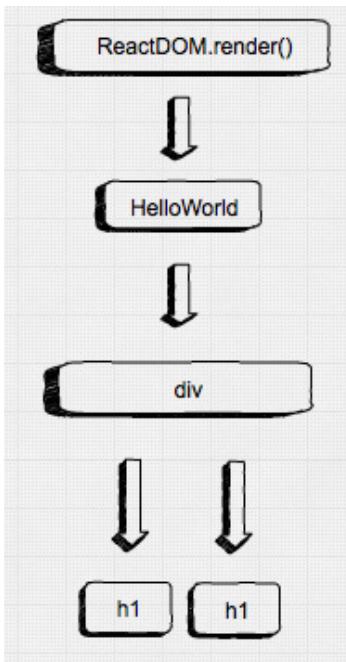


Figure 2.6 Rendering a `div` element created from a custom component class `Hello World` instead rendering it directly

The way developers create a React component class is interestingly enough by extending `React.Component` class with `class CHILD extends PARENT ES6 syntax`. Let's create a custom `HelloWorld` component class with `class HelloWorld extends React.Component`.

The one mandatory thing we must implement for this new class is the `render()` method. This method **must return a single** React element (`createElement()`) which is created from another custom component class or an HTML tag. Either of them can have nested elements.

Listing 2-3 shows how we can refactor our nested *Hello world!* example (Listing 2-1) into an app with a custom React component class `HelloWorld`. The benefit is that with a custom class we will be able to reuse this UI better.

The mandatory `render()` method of the `HelloWorld` component returns the same `<div>` element which we have in the previous examples (Listing 2-1). Once we have our custom class `HelloWorld` created, we can pass it as an object (not as a string) to `ReactDOM.render()` as shown in Listing 2-3.

Listing 2.3 Creating our own React component class and rendering it (ch02/hello-world-class/js/script.js)

```
let h1 = React.createElement('h1', null, 'Hello world!')
class HelloWorld extends React.Component {           ①
  render() {                                     ②
    return React.createElement('div', null, h1, h1)  ③
  }
}
ReactDOM.render(          ④
  React.createElement(HelloWorld, null),
  document.getElementById('content')
)
```

- ① Define a variable and save the result of `createClass` in it; name is capitalized to reflect that it's a custom React component
- ② Create a `render()` method as an expression (function returning a single element)
- ③ Implement a return statement with a single React element so that React class can invoke the `render()` and receive the element `div` with two `h1` elements
- ④ Use `HelloWorld` class to create an element by passing the object as the first argument instead of a string
- ⑤ Attach the React element to the real DOM element with ID `content`

By convention, variables containing React components have their names capitalized. Although, this is not required in regular JS (we can use lowercased `helloWorld` variable name), but because it will become necessary in JSX we apply this convention right now. (In JSX, React is using upper vs. lower case to determine if it's a custom component `<HelloWorld/>` or a regular HTML element such as `<h1/>`. However in regular JS it's differentiated by passing either a variable such as `HelloWorld` or a string such as `'h1'`.) More on the JSX is in the next chapter.

ES6+/ES2015+ AND REACT

In the component class example, I defined `render()` using ES6 style in which you omit colon and the word `function`. It's exactly the same as defining an attribute (a.k.a. key or object property) with a value which is a function, that is typing `render: function()`. My personal preference and a recommendation to you is to use the ES6 method style because it's shorter (the less we type, the fewer mistakes we make).

Historically, React had its own method to create component class—`React.createClass()`. There are slight differences between using the ES6 class to extend `React.Component` and using `React.createClass()`. You can find ES5 examples for some of the projects of this book in the GitHub repository prefixed with `-es5`.

Although you might still see `React.createClass()` method used by some teams, the general tendency in the React world is to move toward common standard, that is use the ES6 class approach. This book is forward thinking and using the most popular tools and approaches, so we are going to focus only on ES6.

As of Aug 2016, most modern browsers [support](#) these ES6 (and almost all others) features natively (without extra tools), thus the assumption is that my readers are familiar with it. If you're not familiar with it or if you need a refresher, or just for more information on ES6+/ES2015+ and its main features as they relate to React, take a look at Appendix E, [ES6 cheatsheet](#) or a comprehensive book such as *Exploring ES6* by Dr. Axel Rauschmayer ([free online version](#)).

Analogous with `ReactDOM.render()`, the `render` method in `createClass()` can **only return a single element**. If you need to return multiple same-level elements, wrap them in a `<div>` container or another unobtrusive element such as ``. You can run the code in your browser. The result is shown in Figure 2.7.

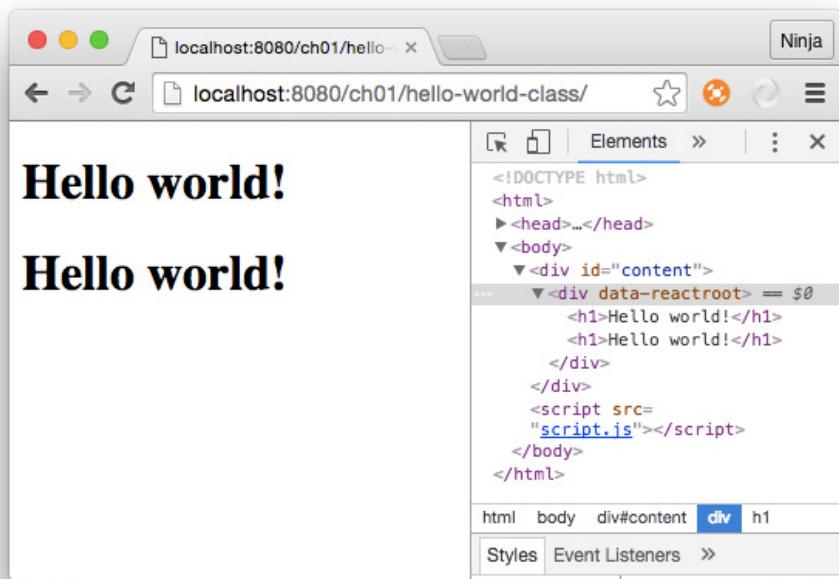


Figure 2.7 Rendering an element created from a custom `HelloWorld` component class

Have you noticed that the name of our reusable custom component starts with an uppercase `H`? When we are using plain JavaScript instead of JSX, the capitalization doesn't matter. It becomes important when we use JSX, because the letter case is how React distinguishes between custom components that developers implement and regular HTML tags

that React already knows about. So it's a good idea to start using capitalization convention for custom components now.

You might think that we didn't gain that much with the refactoring, but what if we need to print more Hello world statements? We can do it by reusing the HelloWorld components multiple times and wrapping them in a `<div>` container:

```
...
ReactDOM.render(
  React.createElement(
    'div',
    null,
    React.createElement(HelloWorld, null),
    React.createElement(HelloWorld, null),
    React.createElement(HelloWorld, null)
  ),
  document.getElementById('content')
)
```

This is the power of component reusability! It means faster development & fewer bugs. Components also have lifecycle events, states, DOM events and other features which allow to make them interactive and self contained. We'll cover them in the next chapters in this part of the book. Right now, our `HelloWorld` elements will all be the same. Is there a way to customize them? What if we can set element attributes and modify their content and/or behavior. Meet properties.

2.3 Working with Properties

Properties are a corner stone of the declarative style that React uses. *Think of properties as unchangeable values within an element.* They allow elements to have different variations if used in a view, e.g., changing a link URL by passing a new value of a property:

```
React.createElement('a', {href: 'http://node.university'})
```

One thing to remember is that **properties are immutable within their components**. A parent assigns properties to its children upon their creation. The child element is not suppose to modify its properties. (A child is an element which is nested inside of another element, e.g., `<h1/>` is a child of `<HelloWorld/>`.) For instance, we can pass a property `PROPERTY_NAME` with value `VALUE` like this:

```
<TAG_NAME PROPERTY_NAME=VALUE/>
```

Properties are closely related to HTML attributes. This is one of their purposes, but they carry another purpose as well. Developers can use properties of an element in their code as they wish. So the properties can be use as follows:

1. To render standard HTML attributes of an element, e.g., `href`, `title`, `style`, `class`, etc.

2. To use in JavaScript code of a React component class via the `this.props` values, e.g., `this.props.PROPERTY_NAME` (replace `PROPERTY_NAME` with your arbitrary name)

Under the hood, React will match the property name (`PROPERTY_NAME`) with the list of standard attributes. If there's a match, the property will be rendered as an attribute of an element (scenario 1). *The value of this attribute is also accessible* in `this.props.PROPERTY_NAME` in the component class code.

`Object.freeze()` and `Object.isFrozen()`

Internally, React uses the `Object.freeze()` ([MDN](#)) from the ES5 standard to make the `this.props` object immutable. To check if an object is frozen, we can use `Object.isFrozen()` method ([MDN](#)). For example, if this statement will return `true`:

```
class HelloWorld extends React.Component {
  render() {
    console.log(Object.isFrozen(this.props))
    return React.createElement('div', null, h1, h1)
  }
}
```

If you're interesting in more details like this, I encourage you to read changelog (e.g., [this](#)) and perform searches on React's GitHub repository (such as [this](#)).

If there's no match with any of the standard HTML attribute names (scenario 2), then the property name is not a standard attribute. It won't be rendered as an attribute of an element. However, the value will still be accessible in the `this.props` object, for example `this.props.PROPERTY_NAME`. It can be used in the code, or rendered explicitly in the `render()` method. This way developers can pass different data to different instances of the same class. This allows to reuse components because developers can change programmatically how elements are rendered by providing different properties.

Developers can even take this feature of properties a step further and completely modify the rendered elements based on the value of a property. For example, if `this.props.heading` is true then we render Hello as a heading. If it is false, we render Hello as a normal paragraph.

```
render() {
  if (this.props.heading) return <h1>Hello</h1>
  else return <p>Hello</p>
}
```

In other words, developers can use the same component, but by providing it with different properties, the elements rendered by the components can be different. Properties can be rendered by `render()`, used in component's code or used as HTML attributes.

To demonstrate properties of components, we'll slightly modify our Hello World with `props`. The goal is to reuse our `HelloWorld` component by do it in a way that each instance of this

class renders a different text and different HTML attributes. We are going to enhance `HelloWorld` headings (`<h1>` tag) with three properties (Figure 2.8):

- `id`: Matches standard attribute `id` and will be automatically rendered by React
- `frameworkName`: Doesn't match any standard attributes for `<h1>`, but will be explicitly printed in the text of headings
- `title`: Matches the standard attribute `title` and will be automatically rendered by React

The properties whose name matches a standard HTML attribute will be rendered as an attribute of `<h1>` element as shown in Figure 2.7 So the two properties `id` and `title` will be rendered as `<h1>` attributes, but not `frameworkName`.

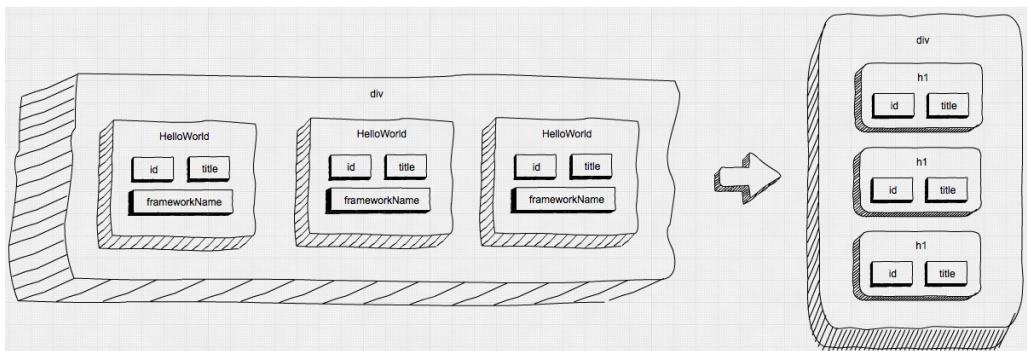


Figure 2.8 Component class `HelloWorld` renders properties which are standard HTML attributes but not `frameworkName`.

Let's zoom in on the `<div>` element implementation (Figure 2.9). Obviously, it needs to render three child elements of the `HelloWorld` class, but the text and attributes of the resulting headings (`<h1/>`) must be different. For example, we pass ID, `frameworkName` and `title` as shown in Figure 2.9. They'll be the part of `HelloWorld` class. Before we implement `<h1/>`, we need to pass the properties to `HelloWorld`. How do you do this?

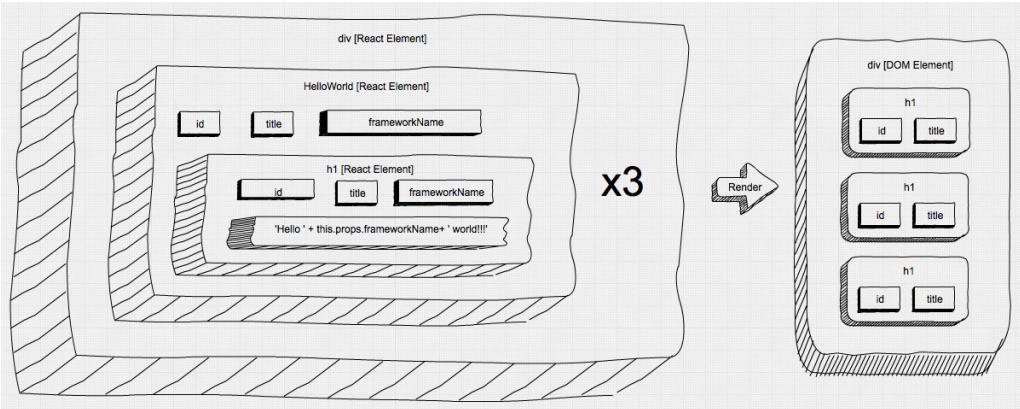


Figure 2.9 The HelloWorld class is used three times to generate three h1 elements which will have different attributes and innerHTML

We can achieve it by passing these properties in an object literal in the second argument to `createElement()` when we create HelloWorld elements in the `<div>` container.

```
ReactDOM.render(
  React.createElement(
    'div',
    null,
    React.createElement(HelloWorld, {
      id: 'ember',
      frameworkName: 'Ember.js',
      title: 'A framework for creating ambitious web applications.'}),
    React.createElement(HelloWorld, {
      id: 'backbone',
      frameworkName: 'Backbone.js',
      title: 'Backbone.js gives structure to web applications...'}),
    React.createElement(HelloWorld, {
      id: 'angular',
      frameworkName: 'Angular.js',
      title: 'Superheroic JavaScript MVW Framework'})
  ),
  document.getElementById('content')
)
```

Now we can take a look at the HelloWorld component implementation. The way React works is that the second parameter to `createElement()` (for example, `{id: 'ember'...}`) is an object whose properties are accessible via the `this.props` object inside of the component's `render()` method. Therefore, we can access the value of `frameworkName` like this:

Listing 2.4 Using property `frameworkName` in the `render()` method

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

```

    return React.createElement(
      'h1',
      null,
      'Hello ' + this.props.frameworkName + ' world!!!' ①
    )
  }
}

```

- ① Concatenate (combine) three strings: `Hello`, `this.props.name` and `world!`

The keys of the `this.props` object will be exactly the same as the keys of the object passed to `createElement()` as the second parameter. That is `this.props` will have `id`, `frameworkName` and `title` keys. The number of key value pairs you can pass in the second argument to `React.createElement()` is unlimited.

In addition, you might have already guessed that it's totally possible to pass all the properties of `HelloWorld` to its child `<h1/>`. This can be extremely useful when you don't know what properties are passed to a component, for example in `HelloWorld`, we want to leave it up to a developer instantiating `HelloWorld` what is the `style` attribute value. Therefore, we don't limit what attributes to render in `<h1/>` as shown in Listing 2-5.

Listing 2.5 Passing all the properties from HelloWorld to h1

```

class HelloWorld extends React.Component {
  render() {
    return React.createElement(
      'h1',
      this.props, ①
      'Hello ' + this.props.frameworkName + ' world!!!'
    )
  }
}

```

- ① Pass all the properties to the child heading element

And then, we render three `HelloWorld` elements into the `<div>` with the ID `content` as shown in Listing 2-6 and diagram in Figure 2.9.

Listing 2.6 Composing a component which uses properties passed during the creating of the element (`ch02/hello-javascript/js/script.js`).

```

class HelloWorld extends React.Component {
  render() {
    return React.createElement(
      'h1',
      this.props, ①
      'Hello ' + this.props.frameworkName + ' world!!!' ②
    )
  }
}

ReactDOM.render(
  React.createElement(

```

```

'div',
null,
React.createElement(HelloWorld, {
  id: 'ember',          ③
  frameworkName: 'Ember.js',
  title: 'A framework for creating ambitious web applications.'}),
  ③
React.createElement(HelloWorld, {
  id: 'backbone',
  frameworkName: 'Backbone.js',   ④
  title: 'Backbone.js gives structure to web applications...'),
  React.createElement(HelloWorld, {
    id: 'angular',
    frameworkName: 'Angular.js',
    title: 'Superheroic JavaScript MVW Framework')
  },
  document.getElementById('content')
)

```

- ① Any properties passed into HelloWorld when createElement is called below are passed into this <h1> element.
- ② Output the `frameworkName` property as a text of <h1>
- ③ id and title correspond to standard HTML attributes for <h1> and will be rendered as those attributes
- ④ frameworkName is not a standard HTML attribute for <h1> so it will not be rendered unless you do something with it.

As usual, you can run this code via a local HTTP web server. The result is to have three different headings (Figure 2.10) and do so by reusing the `HelloWorld` component class.

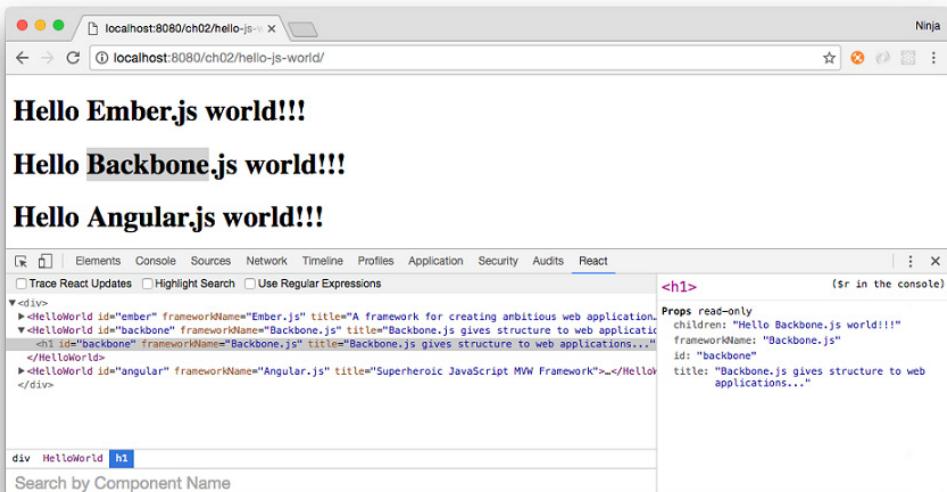


Figure 2.10 Result of reusing `HelloWorld` with different properties to render three different headings

We used `this.props` to render different text for the headings. We used properties to render different titles and IDs. Thus, we effectively reused most of the code which made us the masters of the React Hello World component classes. :-)

2.4 Quiz

1. A React component class can be created with which of the following: `createComponent()`, `createElement()`, `class NAME extends React.Component` or `class NAME extends React.Class?`
2. The only mandatory attribute or a method of a React component is which of the following: `function`, `return`, `name`, `render`, or `class`?
3. To access the `url` property of a component, use: `this.properties.url`, `this.data.url`, `this.props.url`, or `url`?
4. React properties are immutable in a context of a current component. True or false?
5. React component class allows developers to create re-usable UIs. True or false?

2.5 Summary

In this chapter, you've learned how to

- Nest React elements
- Create elements from custom component classes
- Modify the resulting elements with the use of properties
- Pass properties to child element(s)
- Use component-based architecture (one of the features of React) by creating components

We've covered a few permutations of Hello World. Yes, I know, it's still this boring good old Hello World. But by starting small, we are building a solid foundation for future more advanced topics! Believe me, there are a lot of great things which can be achieved with component classes.

It's super important to know how React works in regular JavaScript event if you are (like many React engineers) plan to use JSX. This is because in the end your browsers will still run regular JS and you'll need to understand the results of the JSX to JS transpilation from time to time. Knowing that, going forward we'll be using JSX which is covered in the next chapter.

2.6 Quiz Answers

1. class NAME extends React.Component
2. render()
3. this.props.url
4. True
5. True

3

Introduction to JSX

In this chapter:

- What is JSX and Its Benefits
- Understanding JSX
- Setting Up JSX Transpilers with Babel
- React and JSX Gotchas



Figure 3.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch03>

Welcome to JSX! One of the greatest things about React in my opinion and, probably one of the most controversial subject surrounding React in the mind of a few developers I spoke with (who, not surprisingly, haven't build anything large in React yet).

Thus far, we've covered how to create elements and how to create components so that we can create custom elements and organize our UIs better. And we use JavaScript to create React elements, instead of working with HTML. But there's a problem. Take a look at this code with three nested elements, and see what is happening there.

```
render() {  
  return React.createElement(  
    "div", {  
      style: {  
        border: "1px solid black",  
        padding: "10px",  
        width: "200px",  
        height: "100px"  
      }  
    },  
    "Hello World!"  
  );  
}
```

```

'div',
{ style: this.styles },
React.createElement(
  'p',
  null,
  React.createElement(
    reactRouter.Link,
    { to: this.props.returnTo },
    'Back'
  )
),
this.props.children
);
}

```

Did you understand that there are three elements and they are nested and we are using a component from React Router? How this code is readable comparing to the standard HTML. Do you think this code eloquent? React team agrees with you that reading (and typing for that matter) bunch of `React.createElement()` is not much fun. JSX is the solution to this problem.

The source code for the examples in this chapter is in [the ch03 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

3.1 What is JSX and Its Benefits?

JSX is a really great way of writing React components. The JSX benefits include:

- Improved developer experiences (DX): Better code reading due to more eloquent code because an XML-like syntax is better at representing nested declarative structures
- More productive team members: Casual developers (e.g., designers) can modify code easier, because JSX looks like HTML, something that's already familiar to them
- Fewer wrist injuries and syntax errors: Developers have to type less code (i.e., sugarcoating) which means they'll make fewer mistakes and will have smaller chances of developing repetitive stress injuries

While JSX is not required for React, it fits in very nicely and is highly recommended by me and React creators. The official [page](#) states "We recommend using it with React..." with it being JSX.

JSX stands for *JavaScript eXtension*, which is just syntactic sugar (or sugar-coating) for function calls and object construction in particular `React.createElement()`. To paraphrase, JSX might look like a template engine or HTML but in fact it's not. JSX produces React element while allowing developers to harness the full power of JavaScript.

To demonstrate the eloquence of JSX, this is JSX code to create `HelloWorld` and a link elements:

```
<div>
```

```
<HelloWorld/>
<br/>
<a href="http://webapplog.com">Great JS Resources</a>
</div>
```

The JSX example with `HelloWorld` and an `<a>` link will be converted to this JavaScript:

```
React.createElement(
  "div",
  null,
  React.createElement(HelloWorld, null),
  React.createElement("br", null),
  React.createElement(
    "a",
    { href: "http://webapplog.com" },
    "Great JS Resources"
  )
)
```

In essence, JSX is a small language with an XML-like syntax but it's changed the way people write UI components. You see, before developers wrote HTML, and JS code for controllers and views in an MVC-like manner jumping between various files. That stemmed from the early-days separation of concerns. It served the web well when it was static HTML, a bit CSS and tiny JS to make your text blink (anyone remembers the term DHTML?).

This is not the case anymore because developers build highly interactive UIs and JS and HTML are tightly coupled together to implement various pieces of functionality. React fixes the broken separation of concerns principle by bringing the description of the UI and the JS logic together, while JSX makes the code look like HTML and make it more writable and readable for developers. If for nothing else, I would have used React and JSX just for this new approach to writing UI. :-)

JSX is compiled by various transformers (that is, tools) into standard ECMAScript (Figure 3.2). Most of you know that JavaScript is ECMAScript too, but JSX is not part of the specification and it doesn't have any defined semantics.

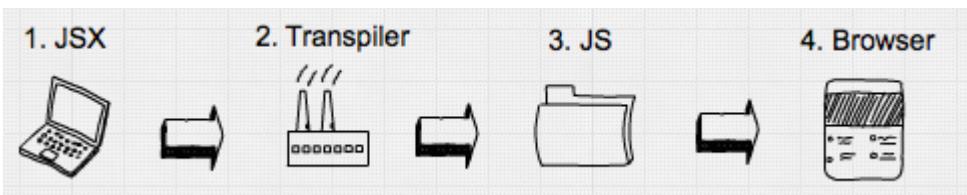


Figure 3.2 JSX is transpiled into regular JavaScript

A source-to-source compiler, transcompiler or transpiler is a type of compiler that takes the source code of a program written in one programming language as its input and produces the equivalent source code in another programming language.—[Source-to-source compiler on Wikipedia](#)

You might think, why should I bother with JSX? That's a great question. Considering how counter-intuitive JSX code looks to begin with (Listing 3-1 shows angle braces right in the JavaScript code), it's no surprise many developers are turned off by this amazing technology.

Listing 3.1 With JSX, having angle braces inside of your JavaScript code might look bizarre at first:

```
ReactDOM.render(<h1>Hello</h1>, document.getElementById('content'))
```

What makes it amazing are the shortcuts to `React.createElement(NAME, ...)`. Instead of writing that function call over and over, you can simply use `<NAME/>`. Yes, the less we type, the fewer mistakes we make. In essence, JSX is just a sugarcoating or a shortcut to create React objects but benefits are not dismal. DX is as important as User Experience.

The main reason to use JSX is that many people find code with angle brackets (`<>`) is easier to read. And, once you get into the habit of thinking about `<NAME/>` not as XML, but as an alias to JavaScript code, you'll get over the perceived weirdness of the JSX syntax. Knowing and using JSX can make a lot of difference in developing React components and, subsequently, React-powered applications.

ALTERNATIVE SHORTCUTS

To be fair, there are a few alternatives to JSX when it comes to avoiding typing verbose `React.createElement()` calls. One of them is to use an alias `React.DOM.*`. For example instead of an `<h1/>` element created with

```
React.createElement('h1', null, 'Hey')
```

This will suffice too while taking less space and time to implement:

```
React.DOM.h1(null, 'Hey')
```

Developers have access to all standard HTML elements in the `React.DOM` object which they can inspect as any other object:

```
console.log(React.DOM)
```

Please note that `React.DOM` and `ReactDOM` are two completely different objects and should not be confused or used one instead of the other.

One more alternative recommended by React official documentation for the situations when JSX is impractical (for example when there's no build process) is to use a short variable. For example, developers can create a variable `$` such as:

```
const E = React.createElement
E('h1', null, 'Hey')
```

As mentioned before, **JSX needs to be transpiled** (or compiled as it's often called for brevity) into regular JavaScript before browsers can execute its code. We'll explore various available methods how to do so at well at the recommended method in details in the section *Setting Up JSX Transpiler with Babel*.

3.2 Understanding JSX

Let's learn how to work with JSX. You can just read this section and keep it bookmarked for your reference, or (if you prefer to have some of the code examples running on your computer), you have the following options:

1. Set up JSX transpiler with Babel on your computer as shown in the next section of this chapter
2. Use online Babel REPL service which will transpile JSX into JavaScript in the browser: <https://babeljs.io/repl/>

The choice is up to you. I recommend reading about the main JSX concepts first and then do the proper Babel set up on your computer.

3.2.1 Creating Elements with JSX

Creating `ReactElement` objects with JSX is straightforward. For example, instead of writing JavaScript where name is a string (`h1`) or component class object (`HelloWorld`):

```
React.createElement(
  name,
  {key1: value1, key2: value2, ...},
  child1, child2, child3, ..., childN
)
```

You would write JSX:

```
<name key1=value1 key2=value2 ...>
  <child1/>
  <child2/>
  <child3/>
  ...
  <childN/>
</name>
```

In the JSX code, the attributes and their values (e.g., `key1=value1`) come from the second argument of `createElement()`. We'll focus on working with properties later in this chapter.

For now, let's look at an example of JSX element without properties. In Listing 3-2 is our good old friend Hello World in JavaScript:

Listing 3.2 Hello World in JavaScript (ch03/hello-world/index.html)

```
ReactDOM.render(
  React.createElement('h1', null, 'Hello world!'),
  document.getElementById('content')
)
```

The JSX version is much more compact as shown in Listing 3-3:

Listing 3.3 Hello World in JSX (ch03/hello-world-jsx/js/script.jsx)

```
ReactDOM.render(
  <h1>Hello world!</h1>,
  document.getElementById('content')
)
```

Developers can also store objects created with JSX syntax in variables, because JSX is just a syntactic improvement over `React.createElement()`. In this example, we store the reference to the element object in a variable.

```
let helloWorldReactElement = <h1>Hello world!</h1>
ReactDOM.render(
  helloWorldReactElement,
  document.getElementById('content')
)
```

3.2.2 Working with JSX in Components

In the previous example, we used `<h1>` JSX tag which is also a standard HTML tag name. When working with components, we apply the same syntax. The only difference is that the component class name must start with a capital letter `<HelloWorld/>`.

Here's a more advanced iteration of Hello World rewritten in JSX. In this example, we created a new component class and use JSX to create an element out of it: Listing 3-4.

Listing 3.4 Creating Hello World Class in JSX

```
class HelloWorld extends React.Component {
  render() {
    return (
      <div>
        <h1>1. Hello world!</h1>
        <h1>2. Hello world!</h1>
      </div>
    )
  }
}
ReactDOM.render(
  <HelloWorld/>,
  document.getElementById('content')
)
```

Can you read this code better than this JavaScript code?

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement('div',
      null,
      React.createElement('h1', null, '1. Hello world!'),
      React.createElement('h1', null, '2. Hello world!'))
  }
}
ReactDOM.render(
  React.createElement(HelloWorld, null),
  document.getElementById('content')
)
```

Seeing angle brackets (`<>`) in JavaScript code may be a bit weird for experienced JavaScript developers at first. My brain went bananaz when I saw it first, because for years I trained it to spot syntax errors in JavaScript! Yes. This brackets are the main controversy around JSX and one of the most frequent objections I hear! This is why we dive into JSX early in the book, so that you can get as much experience with it as possible.

3.2.3 Outputting Variables in JSX

When we composing components, we want them to be smart enough to change the view based on some code. For example, it would be useful if a current date time component would should a current date and time, not some hard-coded value.

When working with JavaScript-only React, we resort to concatenation `+` or if you're using ES6+/ES2015+, to string templates marked by tilde. For example, to use a property in text in a `DateTimeNow` component in regular JavaScript React, we would have written this line:

```
class DateTimeNow extends React.Component {
  render() {
    let dateTimeNow = new Date()
    return React.createElement(
      'span',
      null,
      `Current date and time is ${dateTimeNow}.`)
  }
}
```

Conversely in JSX, we can use `{ }` notation to output variables dynamically which reduces the code bloat substantially:

```
class DateTimeNow extends React.Component {
  render() {
    let dateTimeNow = new Date()
    return <span>Current date and time is {dateTimeNow}.</span>
  }
}
```

The variables can be properties not just locally defined variables.

```
<span>Hello {this.props.userName}, your current date and time is {dateTimeNow}.</span>
```

Moreover, we can execute JavaScript expressions or any JS code inside of {}. For example, we can format a date:

```
<p>Current time in your locale is {new Date(Date.now()).toLocaleTimeString()}</p>
```

Now, we can re-write our HelloWorld class in JSX using the dynamic data, that is JSX stores in a variable:

Listing 3.5 Developers can output variables in JSX even if the value is a React element written in JSX (ch03/hello-world-class-jsx)

```
let helloWorldReactElement = <h1>Hello world!</h1>
class HelloWorld extends React.Component {
  render() {
    return <div>
      {helloWorldReactElement}
      {helloWorldReactElement}
    </div>
  }
}
ReactDOM.render(
  <HelloWorld/>,
  document.getElementById('content')
)
```

Let's discuss working with properties in JSX.

3.2.4 Working with Properties in JSX

I touched on this topic previously when I was introducing JSX: element properties are defined using attributes' syntax. That is we use key1=value1 key2=value2... notation inside of the JSX tag to define both HTML attributes and React component props. This is similar to attribute syntax in HTML/XML.

In other words, *if you need to pass properties, write them in JSX as you would in normal HTML*. Also, we render standard HTML attributes by setting element properties (*chapter 2*). For example,

```
ReactDOM.render(
  <div>
    <a href="http://reactquickly.co">Time for React?</a> ①
    <DateTimeNow userName='Azat'/> ②
  </div>
),
document.getElementById('content')
```

① Render a standard HTML attribute href

② Set value for the userName property

Now, let's take it a step further and consider a component which needs to pass a dynamically generated value of a property. It's easy. We can use the curly braces notation ({}) inside of the angle braces (<>) to pass dynamic values of the properties to elements. For example, we have a dynamic component that renders a link <a> using the properties `url` and `label`. This is the `ProfileLink` class. But the values are defined on the `ProfileLink` creation, so it needs to take its properties and pass their values to <a> using {} .

```
class ProfileLink extends React.Component {
  render() {
    return <a href={this.props.url} title={this.props.label} target="_blank">Profile</a>
  }
}
```

The values for `url` and `label` are passed when a `ProfileLink` instance is created. The resulting <a> tag will get the values.

```
<ProfileLink url='/users/azat' label='Profile for Azat'/>
```

From the previous chapter, you should remember that when rendering standard element (e.g., <h>, <p>, <div>, <a>, etc.) React takes and renders all attributes from the HTML specification and *omits all other attributes that are not part of the specification*. This is not a JSX gotcha; it's React's behavior.

But sometimes developers want to add some custom data as an attribute. Let's say you have a list item and maybe there's some information essential to your app but not needed for users. A common pattern is to put this information in the DOM element as an attribute. For example, using attributes `react-is-awesome` and `id`:

```
<li react-is-awesome="true" id="320">React is awesome!</li>
```

Storing data in custom HTML attributes in DOM is generally considered an anti-pattern, because you don't want the DOM to be your database or a front-end data store. Getting data from DOM is slower than getting it from a virtual/in-memory store.

In cases when you *must* store some data as elements' attributes and you use JSX, you would need to use the `data-NAME` prefix. For example, to render the element with a value of `this.reactIsAwesome` in an attribute, they would write this:

```
<li data-react-is-awesome={this.reactIsAwesome}>React is awesome!</li>
```

Let's say `this.reactIsAwesome` is `true`, then the resulting HTML will be this:

```
<li data-react-is-awesome="true">React is awesome!</li>
```

However, if you attempt to pass a non standard HTML attribute to a standard HTML element, the attribute won't render (as covered in Working with Properties in chapter 2). For example, this code:

```
<li react-is-awesome={this.reactIsAwesome}>React is orange</li>
```

or this code:

```
<li reactIsAwesome={this.reactIsAwesome}>React is orange</li>
```

will produce only this code:

```
<li>React is orange</li>
```

Obviously, because custom elements (i.e., component classes) don't have built-in renderers and rely on standard HTML element or other custom elements, this issue of using `data-` is not important for them. They will get all attributes as props in `this.props`.

Speaking of component classes, this is the code from Hello JS World (*chapter 2*) written in regular JavaScript.

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement(
      'h1',
      this.props,
      'Hello ' + this.props.frameworkName + ' world!!!'
    )
  }
}
```

In the `HelloWorld` components, we pass the properties through to `<h1>` no matter what properties are there. How can we do it in JSX? We don't want to pass each property individually, because that's more code and when we need to change a property, we'll have a tight coupled code that we'll have to update too. Imagine having to pass each property manually and what is your have two or three levels of components to pass them through? Here's an anti-pattern. Don't do this:

```
class HelloWorld extends React.Component {
  render() {
    return <h1 title={this.props.title} id={this.props.id}>
      Hello {this.props.frameworkName} world!!!
    </h1>
  }
}
```

Don't pass the props individually when your intention is to pass ALL of them, because JSX offer a spread solution which looks like ellipses ... as you can see in Listing 3-6:

Listing 3.6 Working with properties, creating elements and components

(ch03/jsx/hello-js-world-jsx)

```
class HelloWorld extends React.Component {
  render() {
    return <h1 {...this.props}>
      Hello {this.props.frameworkName} world!!!
    </h1>
  }
}
```

```

ReactDOM.render(
  <div>
    <HelloWorld
      id='ember'
      frameworkName='Ember.js'
      title='A framework for creating ambitious web applications.'/>,
    <HelloWorld
      id='backbone'
      frameworkName= 'Backbone.js'
      title= 'Backbone.js gives structure to web applications...'/>
    <HelloWorld
      id= 'angular'
      frameworkName= 'Angular.js'
      title= 'Superheroic JavaScript MVW Framework'/>
  </div>,
  document.getElementById('content')
)

```

TL;DR: With `{...this.props}`, we can pass every property to the child. The rest of the code is just converted to JSX example from chapter 2.

Ellipses in ES6+/ES2015+: Rest, Spread and Destructuring

Speaking of ellipses, there are a similar looking operators in ES6+ called destructuring, spread and rest. This is one of the reasons React's JSX is using ellipses!

If you have ever used or written a JavaScript function with a variable or even unlimited number of arguments you know the `arguments` object. This object contains all parameters passed to the function. The problem is that `this.arguments` object is not a real array. You have to convert it to an array if you want to use functions like `sort()` or `map()` explicitly. For example, this `request` function converts `arguments` using `call()`:

```

function request(url, options, callback){
  var args = Array.prototype.slice.call(arguments, f.length)
  var url = args[0]
  var callback = args[2]
  // ...
}

```

So is there a better way in ES6 to access an indefinite number of arguments as an array? Yes! It's rest parameters syntax and it's defined with ellipses For example, this is ES6 function signature with the rest parameter `callbacks` which become an array with the rest of the parameters:

```

function(url, options, ...callbacks) {
  var callback1 = callbacks[0]
  var callback2 = callbacks[1]
  // ...
}

```

NOTE: In the rest array, the first parameter is the one that doesn't have a name, e.g., the callback is at index 0, not 2 as in ES5's `arguments`. Also, putting other named arguments after the rest parameter will cause

a **syntax error**. The rest parameter is a great sugarcoating addition to JavaScript since it replaces the `arguments` object, which wasn't a real array.

Rest parameters can be destructred meaning they can be extracted into separate variables:

```
function(url, options, ...[error, success]) {
  if (!url) return error(new Error('oops'))
  // ...
  success(data)
}
```

What about spread then? In brief, spread allows developers to expand arguments or variables in the following places:

- Function calls, e.g., `push()` `method: arr1.push(...arr2)`
- Array literals, e.g., `array2 = [...array1, x, y, z]`
- new function calls (constructors), e.g., `var d = new Date(...dates)`

In ES5, if you wanted to use an array as an argument to a function, you would have to use the `apply()` function:

```
function request(url, options, callback) {
  // ...
}
var requestArgs = ['http://azat.co', {...}, function(){...}]
request.apply(null, requestArgs)
```

Now in ES6, we can use the spread parameters which look similar to the rest parameters in syntax as they use ellipses ...:

```
function request(url, options, callback) {
  // ...
}
var requestArgs = ['http://azat.co', {...}, function(){...}]
request(...requestArgs)
```

The spread operator has a similar syntax to the rest parameters, but rest is used in the function definition/declaration and spread is used in the calls and literals. They save developers from typing extra lines of imperative code, so knowing and using them is a good skill.

3.2.5 Creating React Component Methods

As a developer, you're free to write any component methods for your applications because the React component is a class. For example, you can create a helper method, `getUrl`:

```
class Content extends React.Component {
  getUrl() {
    return 'http://webapplog.com'
  }
}
```

```

    render() {
      ...
    }
  
```

The method `getUrl` is not sophisticated, but you get the idea that developers can create their own arbitrary methods, not just `render()`. You can use the `getUrl()` method to abstract a URL to your API server. Helper methods can have reusable logic, and you can call them anywhere within other methods of the component, including `render()`.

If you want to output the return of the custom method in the JSX, use `{}`, just as you would do with variables (Listing 3-7). For this, the helper method is invoked in `render`, and the method's return values will be used in the view. Remember to invoke the method with `()`.

Listing 3.7 Invoking a component method from the JSX code to get a URL string
 (ch03/method/jsx/script.jsx)

```

class Content extends React.Component {
  getUrl() {
    return 'http://webapplog.com'
  }
  render() {
    return (
      <div>
        <p>Your REST API URL is:
          <a href={this.getUrl()}>    ①
            {this.getUrl()}
          </a>
        </p>
      </div>
    )
  }
...
  ① Invoke class method in the curly braces
}
  
```

Once again, it's possible to invoke component methods directly from `{}` and JSX. For example, using `{this.getUrl()}` in our helper method `getUrl`, when you use this component, you'll see <http://webapplog.com> as its returned value in the link in the paragraph `<p>`.

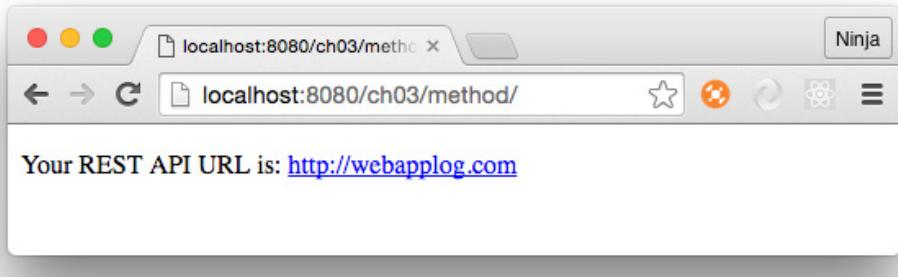


Figure 3.3 Results of rendering a link with the value from a method

I'm glad that you now understand component methods. And, my apologies for those readers who found this section too banal. Nevertheless, methods are important as a foundation for React event handlers.

3.2.6 If/Else in JSX

Akin to rendering dynamic variables, developers need the components to be able to change views based on the results of if/else conditions. We can start with a very simple example where we only rendering one element in a components class and that element depends on the condition. For example, a link text and URL is determined by the `user.session` value. This is how we can code it in plain JS:

```
...
render() {
  if (user.session)
    return React.createElement('a', {href: '/logout'}, 'Logout')
  else
    return React.createElement('a', {href: '/login'}, 'Login')
}
...
```

We can a similar approach and re-write this with JSX like so:

```
...
render() {
  if (this.props.user.session)
    return <a href="/logout">Logout</a>
  else
    return <a href="/login">Login</a>
}
...
```

Let's say there are other elements like a `<div>` wrapper. In this case in plain JS, we'd have to create a variable, use an expression or [a ternary operator](#), because we cannot simply use an if

condition inside of the div's createElement(). The idea is that we must get the value at a run time.

Ternary Operator

The ternary conditions works the way that if userAuth is true, then msg will be set to welcome. Otherwise, the value will be restricted.

```
let msg = (userAuth) ? 'welcome' : 'restricted'
```

The statement above is equivalent to this:

```
let session = ''
if (userAuth) {
  session = 'welcome'
} else {
  session = 'restricted'
}
```

So in some cases, ternary ? operator is a shorter version of if/else. However there's a big difference between them if we try to use ternary operator as an expression (it returns a value).

This code will be perfectly valid JS:

```
let msg = (userAuth) ? 'welcome' : 'restricted'
```

But if/else wouldn't work because it's not an expression, but a statement:

```
let msg = if (userAuth) {'welcome'} else {'restricted'} // Not valid
```

We use this quality of a ternary operator to get a value from it at a run-time in JSX.

To demonstrate three different styles (variable, expression and ternary operator) take a look at this regular JavaScript code before we convert it into JSX:

```
// Approach 1: Variable
render() {
  let link
  if (this.props.user.session)
    link = React.createElement('a', {href: '/logout'}, 'Logout')
  else
    link = React.createElement('a', {href: '/login'}, 'Login')
  return createElement('div', null, link) ①
}

// Approach 2: Expression
render() {
  let link = function(sessionFlag) { ②
    if (sessionFlag)
      return React.createElement('a', {href: '/logout'}, 'Logout')
    else
      return React.createElement('a', {href: '/login'}, 'Login')
  }
  return createElement('div', null, link(this.props.user.session))
}
```

```
// Approach 3: Ternary operator
render() {
  return createElement('div', null,
    (this.props.user.session) ? React.createElement('a', {href: '/logout'}, 'Logout') :
      ③ React.createElement('a', {href: '/login'}, 'Login')
  )
}
```

- ① Create a variable link
- ② Create an expression
- ③ Use a ternary operator

Not bad, but kind of clunky. Would you agree? With JSX, there's curly braces {} notation which can print variables and execute JS code. Let's use it for a better syntax!

```
// Approach 1: Variable
render() {
  let link
  if (this.props.user.session)
    link = <a href='/logout'>Logout</a>
  else
    link = <a href='/login'>Login</a>
  return <div>{link}</div>
}
// Approach 2: Expression
render() {
  let link = function(sessionFlag) {
    if (sessionFlag)
      return <a href='/logout'>Logout</a>
    else
      return <a href='/login'>Login</a>
  }
  return <div>{link(this.props.user.session)}</div>
}
// Approach 3: Ternary operator
render() {
  return <div>
    {(this.props.user.session) ? <a href='/logout'>Logout</a> : <a href='/login'>Login</a>}
  </div>
}
```

If we look closer at the example if the expression (Approach 2, i.e., a function outside of the JSX /before `return`), then we can come with an alternative. We can define the same function using an immediately-invoked function expression (IIFE) RIGHT inside of the JSX. This will allow us to execute the `if/else` at run-time:

```
render() {
  return <div>{
    function(sessionFlag) { ①
      if (sessionFlag)
        return <a href='/logout'>Logout</a>
      else
        return <a href='/login'>Login</a>
    }(this.props.user.session) ②
  </div>
}
```

- ➊ Define an IIFE
- ➋ Invoke an IIFE with a parameter

Furthermore, we can use the same principals not just for rendering entire elements (`<a>` in our examples), but for text and the values of properties. All we need to do is use one of the approaches above inside of curly braces: `{}`. For example, we can augment the url and text and not duplicate the code for the element creation. Personally, this is my favorite approach because I use just a single `<a>`.

```
render() {
  let sessionFlag = this.props.user.session
  return <div>
    <a href={(sessionFlag)?'/logout':'/login'}>
      {(sessionFlag)?'Logout':'Login'}
    </a>
  </div>
}
```

- ➊ Create a local variable to store the session boolean value for less code and better performance
- ➋ Use ternary operator to render different URLs based on the `sessionFlag` value
- ➌ Use ternary operator to render different text

As you can see, unlike template engines, there's not special syntax for these conditions in JSX—we just use JavaScript. Most often, developers use a ternary operator, because it's one of the most compact styles. To summarize, when it comes to implementing if/else logic in JSX, developers utilize these options:

- Variable defined outside of JSX, i.e., before `return`, and printed with `{}` inside of JSX
- Expression (function with returns a value) defined outside of JSX, i.e., before `return`, and invoked in `{}` inside of JSX
- Conditional ternary operator (also known as the Elvis operator)
- [Immediately-invoked function expression](#) (IIFE) inside your JSX

This is my rule of thumb when it comes to conditions and JSX: you can use if/else outside of JSX, i.e., before `return`, to generate a variable which you'll print in JSX with `{}`. Or you can skip the variable, and print results of Elvis operator `?` or expressions using `{}` right inside of JSX.

```
class MyReactComponent extends React.Component {
  render() {
    // Not JSX: Use a variable and if/else or ternary
    return (
      // JSX: Print result of ternary or expression with {}
    )
  }
}
```

We covered the conditions that are very important for building interactive UIs with React and JSX. Occasionally, we may want to narrate our beautiful and intelligent code so other people can quickly understand it. For that, we use comments.

3.2.7 Comments in JSX

Comments in JSX work similar to comments in regular JavaScript. To add JSX comments, you need to wrap standard JavaScript comments in {}, like this:

```
let content = (
  <div>
    {/* Just like a JS comment */}
  </div>
)
```

Or, you can use comments like this:

```
let content = (
  <div>
    <Post
      /* I
       am
       multi
       line */
      name={window.isLoggedIn ? window.name : ''} // We are inside of JSX
    />
  </div>
)
```

I hope now you have a taste of JSX and its benefits. The rest of the chapter is dedicated to the nuances of JSX. But before we can continue with JSX, for any JSX project to function properly, JSX needs to be compiled because browsers can't run JSX. Browsers can run only JavaScript, so we need to take that JSX and transpile it to normal JS (Figure 3-TK).

3.3 Setting Up JSX Transpiler with Babel

As mentioned earlier, in order for us to execute JSX, we need to convert it to regular JavaScript code. This process is called transpilation (from compilation and transformation) and there are various tool available to developers do this job.

There are several ways you can use JSX in your applications. Here are some of the recommended way to do so using:

- Babel command-line interface (CLI) tool: The `babel-cli` package will provide a command for transpilation - require less set up and the easiest to start
- Node.js or browser JavaScript script (API approach): A script can import `babel-core` package and transpile programmatically (`babel.transform`) - allows for low level control and removes abstractions and dependencies of the build tools and their plugins
- Build tool: A tool such as Grunt, Gulp or Webpack can utilize the Babel plugin - the most popular approach

All of them utilize Babel in one way or another. Babel is mostly a ES6+/ES2015+ compiler, but it can convert JSX to JavaScript with it too. In fact, React team stopped development on their own JSX transformer and recommends using Babel.

Can I use something other than Babel?

Although, there were various tools to transpile JSX, the most often used one, and the tool recommended by React team on the official React website as of Aug 2016, is Babel (former 5to6). Historically, React team maintained `react-tools` and `JSXTransformer` (transpilation in the browser), but since 0.13 they started recommending Babel and [stopped evolving](#) `react-tools` and `JSXTransformer` (early 2016).

Another option for the in-browser run-time transpilation is Babel version 5.x. It had `browser.js` which was a ready for usage distribution. You could just drop it in the browser as with `JSXTransformer`, and it will convert any `<script>` code into JS (use `type="text/babel"`). You can include the latest version which had `browser.js`, version 5.8.34, from [CDN directly](#).

Babel 6.x switched to not having default presets/configs (such at JSX) and removed `browser.js`. Developers are encouraged (by Babel team) to create their own distributions. There's also [a babel-standalone library](#), but you still have to tell it which presets/configs to use.

There's a tool called [Traceur](#) which can be used as a replacement for Babel.

Lastly, the [TypeScript](#) seems to [support](#) the JSX compilation via the `jsx-typescript`, but that's a whole new toolchain and the language (superset of regular JavaScript).

Most likely readers can still some of the aforementioned versions of these tools with React 15 (used in this book), because the changes are not that dramatic. However, do so at your own risk. I didn't test the code in this book to see if it works with it!

Clearly, by using Babel for React, you can get extra ES6 features to streamline your development just by adding an extra configuration and a module for ES6+/ES2015+. The sixth iteration of ECMAScript standard has a myriad of improvements, but it's not fully available as of this writing (July, 2016). To solve the lag in ES6 implementation by the browsers, Babel comes to rescue. Now Babel has support for next generations of JavaScript language (many languages... got the name hint?).

Next, we'll cover the recommended approach for the next few chapters—Babel CLI, because it requires minimal setup and doesn't require knowledge of Babel's API unlike the API approach.

To use [Babel](#) CLI, you'll need Node v6.2.0, npm v3.8.9, `babel-cli` 6.9.0 and `babel-preset-react` v6.5.0. Other versions are not guaranteed to work with this book's code due to the fast-changing nature of Node and React development.

If you need to install Node and npm, the easiest way to do so is to download the installer (just one for both Node and npm) from the official website <http://nodejs.org>. As for Babel, For more options and the detailed installation instructions from me, please see *Appendix B: Nod Yes*.

If you think you have these tools installed or you're not sure, then check the versions of Node and npm with these shell / terminal / command prompt commands:

```
node -v
npm -v
```

As for Babel CLI and React preset, you need to have them locally. Using Babel CLI globally (-g when installing with npm) is discouraged, because you might run into conflict when your projects rely on different versions of the tool.

Following the instructions in the *Appendix H: Tower of Babel*, you'll need to do these steps (given here in a short form for convenience):

1. Create a new folder ch03/test
2. Create package.json inside of your new folder, and fill package.json with an empty object {} or use npm init to generate the file
3. Optionally, fill package.json with information such as project name, license, GitHub repository, etc.
4. Install Babel CLI and React preset **locally** using npm i babel-cli@6.9.0 babel-preset-react@6.5.0 --save-dev to save these dependencies in devDependencies in package.json
5. Optionally, create an npm script with one of the Babel command described below

Babel ES6 Preset

In an unfortunate event when you have to support an older browser such as IE9, but you still want to write in ES6+/ES2015+ because that's the future standard, you can add babel-preset-es2015 transpiler. It will convert your ES6 into ES5 code. To do so, install the library:

```
npm i babel-preset-es2015 --save-dev
```

And add to presets configuration next to react:

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

I don't recommend using this ES2015 transpiler when you don't have to support older browser just in case, because first, you'll be running old ES5 code which is less optimized by browsers than ES6 code, second you're adding an additional dependency and complexity, third, if most people continue to run ES5 code in the browsers, they why did we—meaning browser teams and regular JavaScript developers—bothered with ES6? We could have used [TypeScript](#), [ClojureScript](#) or [CoffeeScript](#) which give you more bang for your money!

Then running this command (from your newly created project folder) to check the version, should work:

```
./node_modules/.bin/babel --version
```

After installation, you issue a command to process your `js/script.jsx` JSX into `js/script.js` JavaScript:

```
./node_modules/.bin/babel js/script.jsx -o js/script.js
```

This command is long because we're using a path to Babel. You can store this command into `package.json` to use use a shorter version `npm run build`. Just open it with your editor and add this line to `scripts`:

```
"build": "./node_modules/.bin/babel js/script.jsx -o js/script.js"
```

You can automate this command with the watch option (`-w` or `--watch`):

```
./node_modules/.bin/babel js/script.jsx -o js/script.js -w
```

The Babel command to watch for any changes in `script.jsx` and compile it to `script.js` when you save the updated JSX. When it happens, terminal/command prompt will display:

```
change js/script.jsx
```

When you start having more JSX files, simply use the command with `-d (--out-dir)` and folder names to compile each JSX source files (`source`) into many regular JS files (`build`):

```
./node_modules/.bin/babel source --d build
```

Often times, having just a single file to load is better for the performance of a front-end app than loading many files. This is because each request add a delay. We can compile all files in the source directory into a single regular JS file with `-o (--out-file)`:

```
./node_modules/.bin/babel src -o script-compiled.js
```

Depending on the path configuration on your computer, you might be able to run `babel` instead of `./node_modules/.bin/babel`. In both cases, we are executing locally. If you have an older `babel-cli` installed globally, please delete it with `npm rm -g babel-cli`.

If you're unable to run `babel` when you installed `babel-cli` locally in your project, then consider adding either one of this path statements into your shell profile (`~/.bash_profile`, `~/.bashrc` or `~/.zsh` depending on your shell if you're on Posix)

This statement will add a path to launch `babel` locally if there's `./node_modules/.bin`.

```
if [ -d "$PWD/node_modules/.bin" ]; then
  PATH="$PWD/node_modules/.bin"
fi
```

This statement will add the path `./node_modules/.bin` to your `PATH` environment variable:

```
export PATH=".node_modules/.bin:$PATH"
```

Added bonus: This setting will also allow you to run *any npm CLI tool* locally with just its name, not the path and the name.

For working examples of Babel `package.json` configuration, you can open projects in `ch03`. They follow the same approach which will be used in next chapters. In essence, there's a `package.json` in `ch03` which has `npm` build scripts for each project (subfolder) that needs compilation unless the project has `package.json` itself.

When you run a build script, for example `npm run build-hello-world`, it'll compile the JSX from `ch03/PROJECT_NAME/jsx` into regular JavaScript, and put that compiled file into `ch03/PROJECT_NAME/js`. Therefore, all you need is to install necessary dependencies with `npm i` (it will create a folder `ch03/node_modules`), check if a build script exists in `package.json`, and then run `npm run build-PROJECT_NAME`.

Thus far, you learned the easiest way to transpile JSX into regular JS in my humble opinion, but there are some tricky part when it comes to React and JSX which I want you to be aware off.

3.4 React and JSX gotchas

Now once we know some JSX basics and a few ways to compile JSX into native JavaScript, let's cover some edge cases. There are a few "gotchas" to be aware of when using JSX.

For instance, JSX requires a you to have a closing backslash / either in the closing tag or if you don't have any children and use a single tag in the end of that tag: `Azat, the master of callbacks`. Conversely, HTML is more fault tolerant. Most browsers will ignore the missing backslash and render the element just fine without it. There are other differences between HTML and JSX as well.

3.4.1 Special Characters

The HTML entities are special codes to display special characters such as copyright symbols, em-dashes, quotation marks, and so on. For example:

```
&copy;  
&mdash;  
&ldquo;
```

You can render those codes as any string in `span` or in the `string` attribute `input`. For example this is static JSX (text is defined in code without variables or properties):

```
<span>&copy;&mdash;&ldquo;</span>  
<input value="&copy;&mdash;&ldquo;" />
```

However, if you want to dynamically output HTML entities (from a variable or a property) with `span`, all you'll get is the direct output (`©—“`) and not the special characters. Thus, the code below won't work:

```
// Anti-pattern. Will NOT work!
var specialChars = '&copy;&mdash;&ldquo;'

<span>{specialChars}</span>
<input value={specialChars}/>
```

This is because React/JSX will auto-escape the dangerous HTML. This is convenient security-wise (security by default rocks!), and what we need to output special characters is to use one of these approaches:

- Escape with \u and use a unicode number ([search](#) if you don't remember it; who does?).
- Convert from a character code to a character number with `String.fromCharCode(charCodeNumber)`.
- Break into multiple strings by outputting an array. For example, `{ [©—“] }`.
- Copy the special character directly into your source code (make sure you use a UTF-8 character set).
- Use internal method `__html` to [dangerously set inner HTML](#)

To illustrate the last approach, take a look at this code:

```
var specialChars = {__html: '&copy;&mdash;&ldquo;'}

<span dangerouslySetInnerHTML={specialChars}/>
```

Obviously, React team has a sense of humor to name a property `dangerouslySetInnerHTML`. Sometimes React naming makes me giggle on the inside. :)

3.4.2 data- Attributes

We've touched on this before in the section on properties in JSX, but let's cover it one more time because this is an important topic. React will blissfully ignore any non-standard HTML attributes that you add to components. It doesn't matter whether you use JSX or native JavaScript--that's React's behavior.

Sometimes, we want to pass some additional data using DOM nodes. This is an "anti-pattern" because your DOM shouldn't be used as your database or as local storage. In case you still want to create some custom attributes and get them rendered, use the `data-` prefix.

For example, this is a valid custom `data-object-id` attribute which will be rendered in the view by React:

```
<li data-object-id="097F4E4F">...</li>
```

If your input is the following React/JSX element, React won't render the `object-id` attribute, because it's not a standard HTML attribute.

```
<li object-id="097F4E4F">...</li>
```

3.4.3 style Attribute

The `style` attribute in JSX works differently than that in plain HTML. With JSX, instead of a string, you need to pass a JavaScript object, and CSS properties need to be in camelCase. For example:

- `background-image` becomes `backgroundImage`
- `font-size` becomes `fontSize`
- `font-family` becomes `fontFamily`

You can save the JavaScript object in a variable or render it inline with double curly braces (`{}{...}{}{}`). The double set of braces is needed because one set is for JSX and the other is for the JavaScript object literal.

Consider that there's an object with this font size:

```
let smallFontSize = {fontSize: '10pt'}
```

Then, in your JSX, utilize the `smallFontSize` object:

```
<input style={smallFontSize} />
```

Or not, and settle for a larger font (30pt) by passing the values directly without an extra variable:

```
<input style={{fontSize: '30pt'}} />
```

Another example of passing styles directly. This time we're setting a red border on our ``:

```
<span style={{border: '1px red solid'}}>Hey</span>
```

The main reason why classes are not opaque strings but JavaScript objects is so React can work with them faster when it applies the changes to the views.

3.4.4 class and for

React and JSX accept any attribute that is a standard HTML attribute, except `class` and `for`. This is because those names are reserved words in JavaScript/ECMAScript and JSX is converted to regular JavaScript. Use `className` and `forHtml` instead. For example, if you have a class `hidden`, you can define it in a `div` this way:

```
<div className="hidden">...</div>
```

If you need to create a label for a form element, use `forHtml`:

```
<div>
  <input type="radio" name={this.props.name} id={this.props.id}>
  </input>
  <label htmlFor={this.props.id}>
    {this.props.label}
  </label>
```

```
</div>
```

3.4.5 Boolean Attribute Values

Last, but not least, there are some attributes (such as `disabled`, `required`, `checked`, and `readOnly`) specific only to form elements. The most important thing to remember here is that the attribute value MUST be set in the JavaScript expression (that is, inside {}) and not set in strings.

For example, use `{false}` to enable the input:

```
<input disabled={false} />
```

But not "false" value, because it'll pass the truthy check (a non-empty string is truthy in JavaScript—see sidenote) and render the input as disabled (`disabled` will be true):

```
<input disabled="false" />
```

In JavaScript/Node, a truthy value is a value which translates to true when evaluated as a Boolean, for example in an `if` statement. The value is truthy if it's not falsy. (Official definition... brilliant, right?) And there are only six falsy values

- `false`
- `0`
- `""`: Empty string
- `null`
- `undefined`
- `NaN`: Not a number

I hope now you can see that a string "false" will in fact be a non-empty string which is truthy and translates to `true`, hence we'll get `disabled=true` in HTML.

In case you omit the value, React will assume the value is true:

```
<input disabled />
```

In the subsequent chapters, we'll be using JSX exclusively. but knowing the underlying regular JavaScript which will actually be run by the browsers is a great skill to have in your toolbox.

3.5 Quiz

1. To output a JavaScript variable in JSX, we use: `=`, `<%= %>`, `{}`, or `<?= ?>`?
2. The `class` attribute is not allowed in JSX. True or false?
3. The default value for an attribute without a value is `false`. True or false?

4. The inline style attribute in JSX is a JavaScript object and not a string like other attributes. True or false?
5. If you need to have an if/else logic in JSX, you can simply use it inside of {}, for example `class={if (!this.props.admin) return 'hide'}`. True or false?

3.6 Summary

So far we've covered one of the hardest and also the weirdest looking part about the preferred React stack—JSX. I wrote *preferred* because JSX is optional. While you can use native JavaScript with React, the benefits of JSX far out weigh the learning curve.

Here's what you learned about JSX:

- JSX is just a sugar-coating for React's methods like `createElement`.
- You should use `className` and `forHtml` instead of the standard HTML `class` and `for` attributes.
- The `style` attribute takes a JavaScript object, not a string like normal HTML.
- Ternary operators and IIFE are the best ways to implement if/else statements.
- Outputting variables, comments, HTML entities, and compiling JSX code into native JavaScript is easy.
- There are a few choices to turn JSX into regular JavaScript: Compiling with Babel CLI requires minimal set up comparing to configuring build processing with tools like Gulp, or Webpack or writing Node/JavaScript scripts to use Babel API.

I hope this chapter cleared some of your concerns about JSX. It's good to have this amazing language in your toolbelt early on in the book because now we can use it for more complex projects. Other key React components that will let us build more interactive (and more useful, real-life examples) are state and events. Those topics are coming in the next chapters.

3.7 Quiz Answers

1. {}
2. true, class is a reserved or special JavaScript statement. For this reason, we use className in JSX.
3. false
4. true
5. true

4

Making React Interactive with States

In this chapter, we cover four very important topics that make React interactive:

- What is React Component State
- Working with States
- State vs. Properties
- Stateful vs. Stateless components



Figure 4.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch04>

Imagine, you're building an autocomplete input field (Figure 4.2). When you type in it, you want to make a request to the server to fetch the information about matches in order to show on the web page. So far we've worked with properties (or props for short) and you've learned that by changing props we can get different views. However, props can't change in a context of the current component, because they are passed on this component's creation.

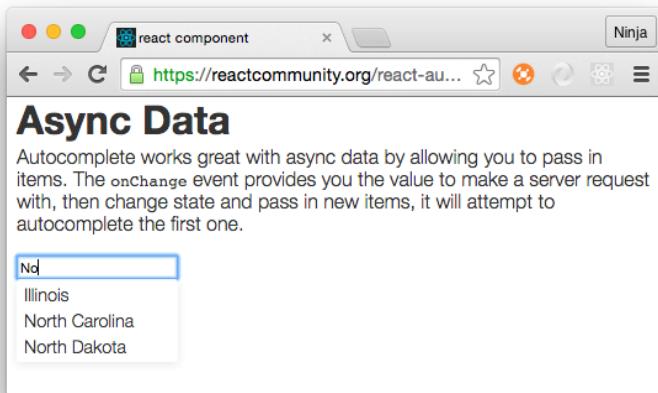


Figure 4.2 react-autocomplete component in action

To put it another way, props are immutable in a current component meaning we don't change props inside of the component unless we re-create this component by passing new values from a parent (Figure 4.3). However, we must store the information which we receive from the server somewhere, and then display the new list of matches in the view. How do we update the view if the properties are unchangeable?

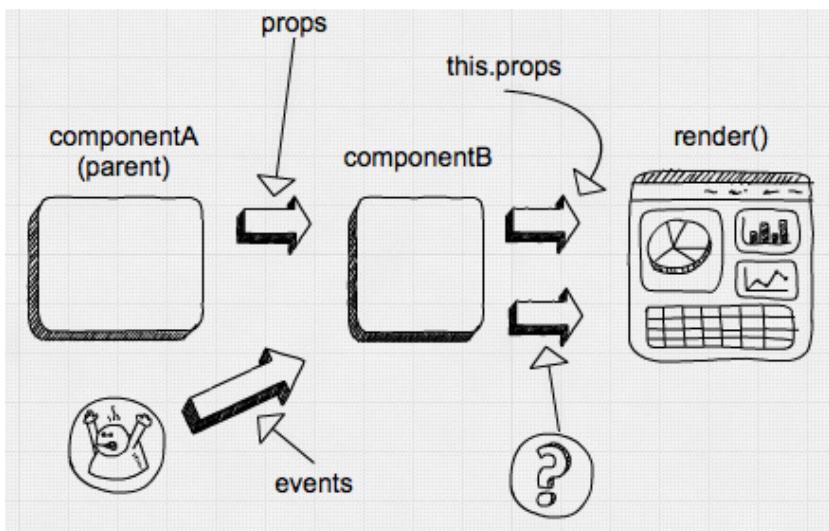


Figure 4.3 There's a need for another data type which is mutable inside of the component to make the view change

One solution is to render an element with new properties each time we get the new server response. However, then we'll have to have some logic outside of the component. The component stops being self-contained... Clearly, if we cannot change values of properties and our autocomplete needs to be self-contained, we cannot use properties. Thus the question is how do we update views in response to some events without re-creating a component (`createElement()` or `JSX <NAME/>`)? This is the problem that states solve.

Once the response from the server is ready, our callback code will augment the component state accordingly. We will have to write this code ourselves. Once the state is updated though, React will intelligently update the view for us (only in the places where it needs to be updates, that is where we use the state data).

With React component states, you'll be able to build some really meaningful and interactive React applications. States is the core concept that let you build React components which can store data in them and automatically augment views based on the data change. If you read only one chapter in the entire book, this is the chapter to read! Because without states, your React components are just glorified static templates. :-) I hope you're as excited as I am because understanding the concepts in this chapter will allow us to build much more interesting applications!

The source code for the examples in this chapter is in [the ch04 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

4.1 What is React Component State

A *React state* is a *mutable data store of component—self-contained functionality-centric blocks of UI and logic*. Mutable means state **values can change**. By using state in views (`render()`) and changing its values later, we can affect the representation.

Here's a metaphor: if you think of a component as a function which has props and state as its input, then the result of this function is the UI description (view). Or as React teams put it "Components are state machines". Props and state both augment view but they use for different purposes (see *States and Properties*).

To work with states, we access them by their names which is an attribute (a.k.a. as an object key or an object property—not a component property) of the `this.state` object, for example `this.state.autocompleMatches` or `this.state.inputFieldValue`.



Generally speaking, the phrase **word states** refers to the attributes of the `this.state` object in a component. Depending on a context, **word state** (singular) can refer to the `this.state` object or an individual attribute (e.g., `this.state.inputFieldValue`). Conversely, **word states** (plural) almost always refers to the multiple attributes of the `state` object in a single component.

State data is often used to display dynamic info in view to augment rendering of views. Going back to the earlier example with autocomplete, the state change in response to the XHR request to the server which is in turn triggered by a user typing in the field. React is taking care of keeping views up-to-date when the states used in views changes. In essence, when state changes, ONLY the corresponding parts of the views change (down to single elements or even just an attribute value of a single element).

Everything else in DOM remains intact. This is possible due to the virtual DOM (chapter 1) which React is using to determine the delta (reconciliation process). This is how we are able to write declaratively. React simply does all the magic for us. The view change steps and how it happens will be discussed in the next chapter *React Component Specifications and Lifecycle Events*.

Given these points, React developers use states to generate new UIs. Component properties (`this.props`), regular variables (`inputValue`) or class attributes (`this.inputValue`) won't do it, because they won't trigger a view change when you change their values (in the current component context). For instance, all these are anti-patterns:

Listing 4.1 An anti-pattern showing that, by changing value in anything except state, we won't get view updates

```
// Anti-pattern: Stay away from it!
let inputValue = 'Texas'
class Autocomplete extends React.Component {
  updateValues() {          ①
    this.props.inputValue = 'California'
    inputValue = 'California'
    this.inputValue = 'California'
  }
  render() {
    return (
      <div>
        {this.props.inputValue}
        {inputValue}
        {this.inputValue}
      </div>
    )
  }
}
```

① Trigger as a result of a user **action** (i.e., typing)

Next, we'll see how to work with React component states.



As mentioned before (repetition is the mother of skills), props will change the view if you pass a new value from a parent which in turn will create a new instance of a component you're currently working with. In the context of a given component, changing `props.this.props.inputValue = 'California'` won't cut it.

4.2 Working with States

To be able to work with state, we need to know how to access the values, update them and how to set the initial values. Let's start with accessing state in React components.

4.2.1 Accessing States

The `state` object is an attribute of a component and can be accessed with `this` reference, e.g., `this.state.name`. If you remember, we can access and print variables in JSX with curly braces `{}`. Similarly, we can render `this.state` (as any other variable or a custom component class attributes) inside of `render()`. For example, `{this.state.inputFieldValue}`. This syntax is similar to how we access properties with `this.props.name`

Let's use what we've learned so far and try to implement a clock (Figure 4.4). The goal is to have a self-containing component class which anyone can import and use in their application without having to jump over hoops. The clock must render current time.

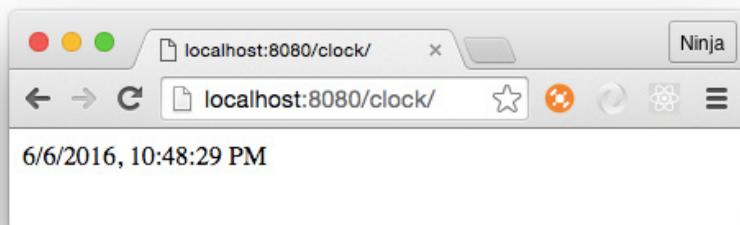


Figure 4.4 Clock component shows current time in digital format—updated every second

The structure of the Clock project is as follows:

```
/clock
  - index.html
/jsx
  - script.jsx
  - clock.jsx
/js
  - script.js
  - clock.js
  - react-15.0.2.js
  - react-dom-15.0.2.js
```

And I'm using Babel CLI with a watch `-w` and a directory flag `-d` to compile all source JSX files from `clock/jsx` to a destination folder `clock/js` and recompile on changes. Moreover, I have the command saved as an npm script in my `package.json` in a parent folder `ch04` in order to simply run `npm run build-clock` from `ch04`:

```
"scripts": {
  "build-clock": "./node_modules/.bin/babel clock/jsx -d clock/js -w"
},
```

Obviously, time is always changing (for good or for bad). Because of that, we'll need to update the view and we can do so by using state. We can give it a name `currentTime` and try to render this state as shown in Listing 4-2.

Listing 4.2 Rendering state in JSX

```
class Clock extends React.Component {
  render() {
    return <div>{this.state.currentTime}</div>
  }
}

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

We'll get `Uncaught TypeError: Cannot read property 'currentTime' of null`. It's good that React is so helpful with error messages. This one means we don't have any value for `currentTime`. Unlike `props`, `states` are not set on a parent. We can't `setState` in `render()` either because it'll create a circular (`setState->render->setState...`) loop and React will throw another error.

4.2.2 Setting the Initial State

Thus far readers saw that before use a state data in `render()`, we must initialize it. To set the initial state, use `this.state` in the constructor with your ES6 class `React.Component` syntax. Don't forget to invoke `super()` with `props` (short for properties), otherwise the logic in parent (`React.Component`) will not work.

```
class MyFancyComponent extends React.Component {
  constructor(props) {
    super(props)
    this.state = {...}
  }
  render() {
    ...
  }
}
```

Developers can also add other logic while they are setting the initial state. For example, we can set the value of `currentTime` using `new Date()`. We can even use `toLocaleString()` to get the proper date and time format in user location:

Listing 4-3: Clock component constructor (ch04/clock)

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

The value of `this.state` must be an object. We won't get into a lot of details about ES6 `constructor()`, because there is information in Appendix E and the [ES6 cheatsheet](#). The gist is that as with other OOP languages, `constructor()` will be invoked when an instance of this class is created. The constructor method name must be exactly `constructor`. Think about it as a ES6 convention. Futher more, **if you create `constructor()` method, you almost always will need to invoke `super()` inside of it**, otherwise the parent's constructor won't be executed. On the other hand, if you don't define `constructor()` method, then the call to `super()` will be assumed under the hood.

Class Attributes

Hopefully, the TC39 (people behind the ECMAScript standard) will add attributes to the class syntax in future versions of ECMAScript! This way, developers will be able to set `state` not just in `constructor`, but in the body of the class:

```
class Clock extends React.Component {
  state = {
    ...
  }
}
```

The proposal is called [class instance fields or class properties](#), but as of this writing (Jul, 2016) it's only available with transpilers: Babel, Traceur or TypeScript, which means not a single browser will run this feature natively. Checkout the current compatibility of class properties in [ECMAScript Compatibility Table](#).

Here, `currentTime` is just an arbitrary name, and we'll need to use the same name in later when accessing and updating this state. You can name your state anyway you want as long as you refer to it later using this name.

The state object can have nested objects or arrays. Look at this example, where I add an array of my books to the state:

```
class Content extends React.Component {
```

```

constructor(props) {
  super(props)
  this.state = {
    githubName: 'azat-co',
    books: [
      'pro express.js',
      'practical node.js',
      'rapid prototyping with js'
    ]
  }
  render() {
    ...
  }
}

```

The `constructor()` method will be called just once, when a React element is created from this class. This way, we can set state directly by using `this.state` just once, in the `constructor()` method. **Avoid setting and updating state directly** with `this.state = ... anywhere else`, because it might lead to unintended consequences.

With React's own `createClass()` method to define a component, you'll need to use `getInitialState()`. For more information on `createClass()` and example in ES5, take a look at chapter *Odds and Ends* of this book.

This will only get us the first value which will be out dated really soon, like in one second. What's the point of a clock which don't show current time? There's a way to update the state.

4.2.3 Updating States

We change the state with `this.setState(data, callback)` class method. When this method is invoked, React merges the data with current states and calls `render()`. After it, React calls `callback`.

Having the callback in `setState()` is important because the method work **asynchronously**. If you're relying on the new state, you can use `callback` to make sure that this new state is available. If you rely on a new state without waiting for `setState()` to finish its work, i.e., working synchronously with asynchronous operation, then you might have a bug when the state is still an old state.

So far we've rendered the time from a state, we also set the initial state, but we need to update the time every second, right? We can use a browser timer function `setInterval()` which will execute the state update every n number of milliseconds. The `setInterval()` method implemented in virtually all modern browsers as a global which means developers can use it without any libraries or prefixes.

```

setInterval(()=>{
  console.log('Updating time...')
  this.setState({
    ...
  })
})

```

```

        currentTime: (new Date()).toLocaleString()
    })
}, 1000)
}

```

To kick-start the clock, we need to invoke `setInterval()` once. We can create a method `launchClock()` to do just that. We'll call `launchClock()` in constructor. The final Clock might look like the one shown in Listing 4-4.

**Listing 4-4: Implementing clock with React state
and `setInterval()` (ch04/clock/jsx/clock.jsx).**

```

class Clock extends React.Component {
  constructor(props) {
    super(props)
    this.launchClock()           ①
    this.state = {
      currentTime: (new Date()).toLocaleString()       ②
    }
  }
  launchClock() {
    setInterval(function(){
      console.log('Updating time...')
      this.setState({
        currentTime: (new Date()).toLocaleString()     ③
      })
    }, 1000)          ④
  }
  render() {
    console.log('Rendering Clock...')
    return <div>{this.state.currentTime}</div>
  }
}

```

- ① Trigger `launchClock()`
- ② Set initial state to current time
- ③ Update state with current time every second
- ④ Bind context to reference the component instance
- ⑤ Render state

You can use `setState()` anywhere, not just in `launchClock()` (which is invoked by `constructor`), as shown in the example. Typically, `setState()` is called from the event handler or as a callback for incoming data or data updates.

Simply changing a state value in your code like this `this.state.name= 'new name'` won't do any good. It won't trigger a re-render and a possible real DOM update which we want. For the most part, **changing state directly without `setState` is an anti-pattern and should be avoided.**

It's important to note that `setState()` updates only the states that you pass to it (partial or merge, but not a complete replace). It's not replacing the entire state object each time. So, if you have three states and then change one, the other two will remain unchanged. In the example below, `userEmail` and `userId` will remain intact:

```

constructor(props) {
  super(props)
  this.state = {
    userName: 'Azat Mardan',
    userEmail: 'hi@azat.co',
    userId: 3967
  }
}
updateValues() {
  this.setState({userName: 'Azat'})
}

```

However, if your intention is to update all three states, then you need to do so explicitly by passing the new values for these states to `setState()`. Another method which you might still see in old React code but which is no longer working and was *deprecated* is `this.replaceState()` method. As you can guess from the name, it was replacing the entire state object with all its attributes.

Keep in mind that `setState()` will trigger `render()`. It works in most cases. In some very edge-case scenarios when the code depends on external data, you can trigger re-render with `this.forceUpdate()`. But this approach should be avoided as a bad practice because relying on external data and not state makes components more fragile and depended on external factors (tight coupling).

As mentioned before, you can access the state object with `this.state`. If you remember, we output values with curly braces `({})`; therefore, to declare a state property in the view (that is, `render`'s return statement), simply apply `this.state.NAME`.

React magic really happens when you utilize state data in the view (for example, to print, in `if/else`, as a value of an attribute, or as a child's prop value) and then give `setState()` new values. Boom! React updates the necessary HTML for you. You can observe it in your DevTools console. It should show cycles of `updating...` and then `rendering...`. And, the best part is that ONLY the absolute minimally required DOM elements will be affected.

Binding `this` in JavaScript

In JavaScript, `this` mutates (changes) its value depending on the place a function is called from. To ensure that `this` refers to our component class, we need to bind the function to the proper context (`this` value, that is our component class).

If you're using ES6+/ES2015+ as I do here in this book, then you can use fat arrow function syntax to create a function with an autobinding:

```

setInterval(()=>{
  this.setState({
    currentTime: (new Date()).toLocaleString()
  })
}, 1000)

```

Autobinding means that the function created with a fat arrows will get the current value of `this` which is in our case is `Clock`.

Manual approach is to use `bind(this)` method on the closure:

```
function() {...}.bind(this)
```

Or for our Clock:

```
setInterval(function(){
  this.setState({
    currentTime: (new Date()).toLocaleString()
  })
}, bind(this), 1000)
```

This behavior is not exclusive to React. The `this` keyword mutates inside of a function's closure, so we need do some sort of binding or we can also save context (`this`) value to be able to use it later.

Typically, we'll see variables like `self`, `that`, or `_this` used to save the value of the original `this`. Most of you probably saw statements like the following:

```
var that = this
var _this = this

var self = this
```

The idea is straightforward, you create a variable and use it in the closure instead of referring to `this`. The new variable won't be a copy but a reference to the original `this` value. Here's our `setInterval()`:

```
var _this = this
setInterval(function(){
  _this.setState({
    currentTime: (new Date()).toLocaleString()
  })
}, 1000)
```

So we have our clock and it's working (Figure 4.5). Tadaaa!

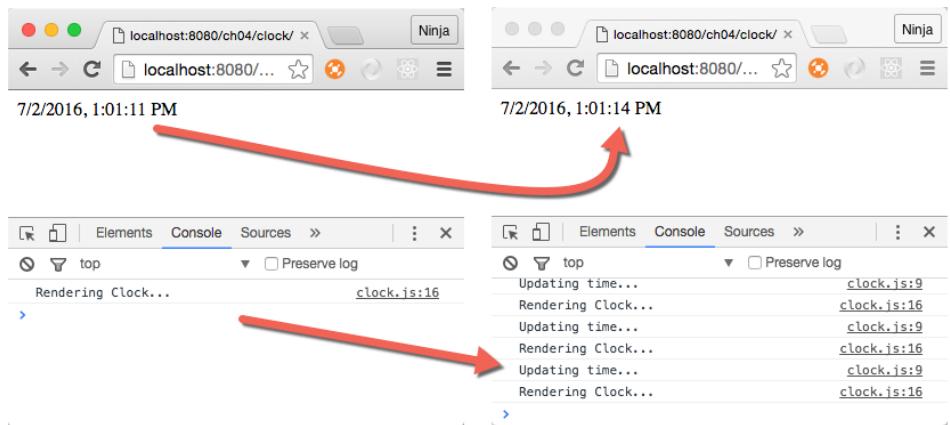


Figure 4.5 The Clock is ticking

One more quick thing before we move on. You can see how React is reusing the same DOM `<div>` element and only changes the text inside of it. Go ahead and use DevTools to modify CSS of this element. I added a style to make text blue: `color: blue` as shown in Figure 4-5. It created an inline style, not a class. The element and its new inline style stayed the same (blue) while the time kept on ticking.

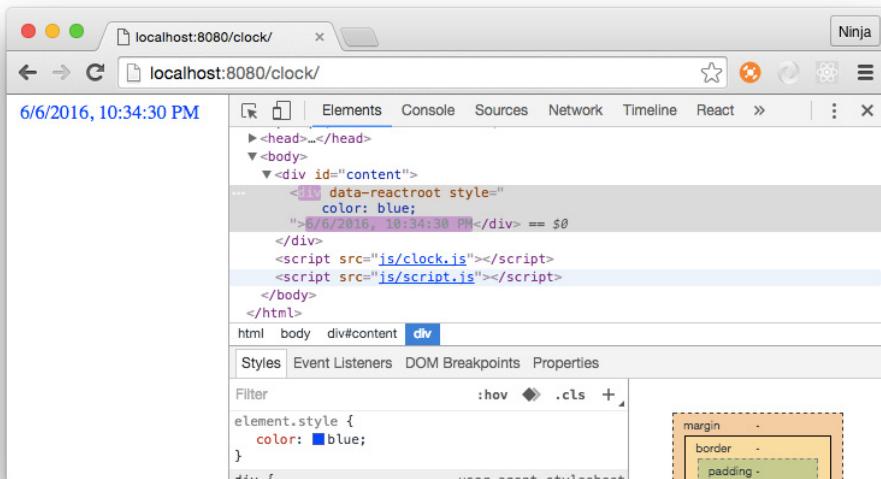


Figure 4.6 React is updating time as only text, not the element div (manually added `color: blue`)

React will only update the inner HTML (the content of the second `<div>` container); `<div>` itself, as well as **all other elements on this page, remain intact**. Neat. ;-)

4.3 States and Properties

States and properties are both attributes for a class meaning they are `this.state` and `this.props`. That's the only similarity! *One of the main differences between props and state is that former are immutable while the latter is mutable.*

Another difference between properties and states is that we pass properties (props) from parent components, while we define states in the component itself, not its parent. The philosophy here is that you can only change the value of a property from the parent, not the component itself. So properties determine the view upon creating and then they stay static (they don't change). The state, on the other hand, is set and updated by the object itself.

Props and states serve different purposes, but both are accessible as attributes of component class and both help developers to compose component with different representation (view). There are differences between props and states when it comes to component lifecycle (more on it in *React Component Lifecycle Events*). Think about props and states are inputs for a function which produces different outputs. Those outputs are views. So you can have different UIs (views) for each set of different props and states (Figure 4-6).

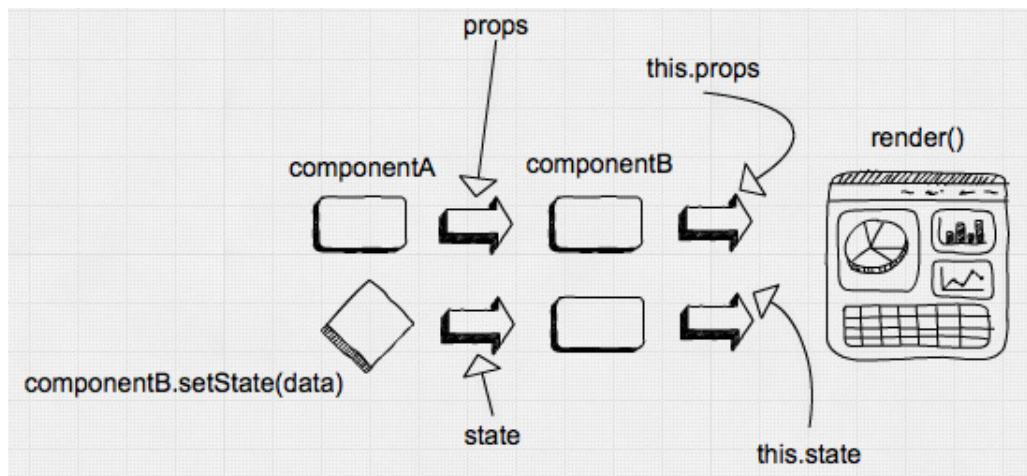


Figure 4.7 New values for props and states can change the UI, but for props the new values come from a parent and for state from the component itself

Not all components need to have state. In the next section, you'll see how to use properties with stateless components.

4.4 Stateless Components

The concept of a stateless component is a component which has no states or component or any other React lifecycle events/methods (chapter *React Component Specifications and Lifecycle Events*). The purpose of a stateless component is to just render the view. The only thing it can do is take props and do something with it—a simple function with an input (props) and the output (UI element).

The benefit of using stateless components is that they are very predictable because we have one input which determines the output. Predictability means they are easier to understand, maintain and debug. In fact, not having a state is the most desired React practice—the more stateless components you use and the less stateful one you use, the better.

We wrote a lot of stateless components in the first three chapters of this book . :-) For example, Hello World is a stateless component:

Listing 4.5 (ch03/hello-js-world-jsx/jsx/script.jsx)

```
class HelloWorld extends React.Component {
  render() {
    return <h1 {...this.props}>Hello {this.props.frameworkName} world!!!</h1>
  }
}
```

To have a smaller syntax for stateless components, React provides us with a function style. We simply create a function which takes props as an argument and returns the view. A stateless component will render like any other component. For example, the HelloWorld component can be re-write as a function which returns `<h1>`:

```
const HelloWorld = function(props){
  return <h1 {...props}>Hello {props.frameworkName} world!!!</h1>
}
```

Yes. You can use ES6+/ES2015+ arrow functions for stateless components:

```
const HelloWorld = (props)=>{
  return <h1 {...props}>Hello {props.frameworkName} world!!!</h1>
}
```

As can be seen, developers can also define functions as React components when there's no need for a state. In other words, to create a stateless component, define it as a function. Another example in which `Link` is a stateless component:

```
function Link (props) {
  return <a href={props.href} target="_blank" className="btn btn-primary">{props.text}</a>
}
ReactDOM.render(
  <Link text='Buy React Quickly' href='https://www.manning.com/books/react-quickly'/>,
  document.getElementById('content')
)
```

While there is no need for autobinding, we can use the fat arrows function syntax for brevity (when there's a single statement, the notation could be a one-liner):

```
const Link = props=> <a href={props.href} target="_blank" className="btn btn-primary">{props.text}</a>
```

In a stateless component, we can't have a state, but we can have two properties: `propTypes` and `defaultProps` (chapter—*Advancing React Components*). We set them on the object:

```
function Link (props) {
  return <a href={props.href} target="_blank" className="btn btn-primary">{props.text}</a>
}
Link.propTypes = {...}
Link.defaultProps = {...}
```

We also can't use references `refs` with stateless functions (about references in chapter *Working with Forms*). If you need to use `refs`, you can wrap a stateless component in a normal React component.

4.5 Stateful vs. Stateless Components

Ideally, developers should use as many stateless components as possible. However, as you've seen in the `Clock` example, it's not always possible and sometimes we have to resort to states.

Why use stateless components? They are more declarative and work better when all you need is to render some HTML without creating a backing instance or lifecycle components. Basically, stateless components reduce duplication, provide better syntax and more simplicity when all you need to do is mesh together some props and elements into HTML.

My suggested approach and the best practice according to React team is to use stateless components instead of normal components as often as possible. And then have a handful of stateful components on top of the hierarchy to handle the UI states, interactions, and other application logic (like loading data from a server).

Do not think that stateless components have to be static. By providing different properties to them, we can change their representation. Consider this refactoring and enhancing

of `Clock` into three components: stateful `Clock` which has the state and the logic to update it, and two stateless components `DigitalDisplay` and `AnalogDisplay` which are only outputting time (but doing it in different ways). The goal is to have something as shown in Figure 4-7. Pretty, right?

The structure of the project is as follows:

```
/clock-analog-digital
  /jsx
    - analog-display.jsx
    - clock.jsx
    - digital-display.jsx
    - script.jsx
  /js
    - analog-display.js
    - clock.js
    - digital-display.js
    - script.js
    - react-15.0.2.js
    - react-dom-15.0.2.js
  - index.html
```

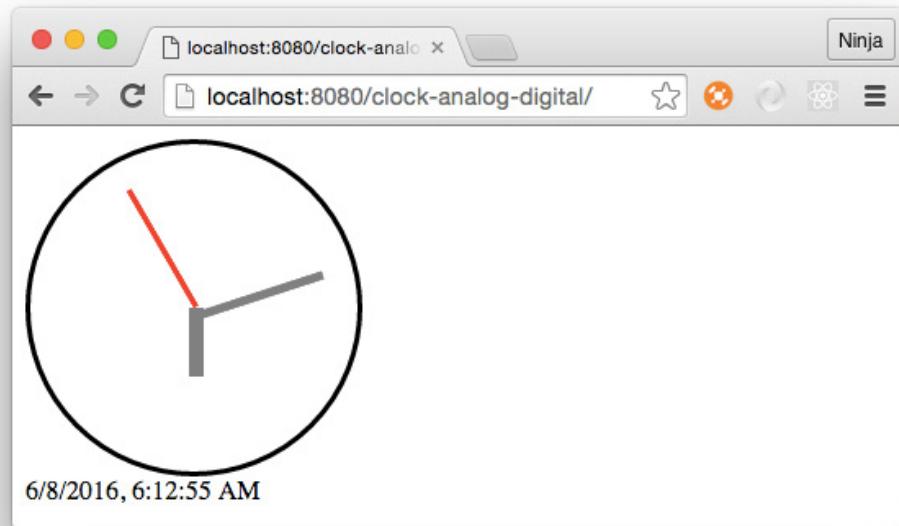


Figure 4.8 Clock with two ways to show time: analog and digital

The code of `Clock` will render the two child elements and pass the property `time` with value of the state `currentTime`. The state of a parent became a property of a child.

Listing 4.6 Passing state variable currentTime to children as a value of a child property time

```
...
render() {
  console.log('Rendering...')
  return <div>
    <AnalogDisplay time={this.state.currentTime}/>
    <DigitalDisplay time={this.state.currentTime}/>
  </div>
}
```

Now we need to create `DigitalDisplay` which is very simple. It's a function which take the properties and displays `time` from that property argument (`props.time`).

Listing 4.7 Stateless digital display component (ch04/clock-analog-digital/jsx/digital-display.jsx)

```
const DigitalDisplay = function(props) {
  return <div>{props.time}</div>
}
```

The `AnalogDisplay` will also be a function implementing a stateless component, but in its body, there would be some fancy animation to rotate the hands. The animation will work based on the property `time`, not based on any state.

The idea is to take the time as a string, convert it to a Date object, get minutes, hours and seconds and convert them to degrees. For example, to get seconds as angle degrees:

```
let date = new Date('1/9/2007, 9:46:15 AM')
console.log((date.getSeconds()/60)*360) // 90
```

Once we have the degrees, we can use them in CSS which is written as object literal. The difference is that in our React CSS the `style` properties are camelCased valid JavaScript name, while in regular CSS the dashes - make style properties invalid JavaScript. (As mentioned before, having objects for styles allows React to determine the difference between old element and the new element faster. Refer to chapter *Introduction to JSX*, section *React and JSX Gotchas* for more info on `style` and CSS in React.)

Listing 4.8 Stateless analog display component with CSS that uses values from property time (ch04/clock-analog-digital/jsx/analog-display.jsx)

```
const AnalogDisplay = function AnalogDisplay(props) {
  let date = new Date(props.time)           ①
  let dialStyle = {
    position: 'relative',
    top: 0,
    left: 0,
    width: 200,
    height: 200,
    borderRadius: 20000,                  ②
    borderStyle: 'solid',
```

```

        borderColor: 'black'
    }
    let secondHandStyle = {
        position: 'relative',
        top: 100,
        left: 100,
        border: '1px solid red',
        width: '40%',
        height: 1,
        transform: 'rotate(' + ((date.getSeconds()/60)*360 - 90 ).toString() + 'deg)',      3
        transformOrigin: '0% 0%',          4
        backgroundColor: 'red'
    }
    let minuteHandStyle = {
        position: 'relative',
        top: 100,
        left: 100,
        border: '1px solid grey',
        width: '40%',
        height: 3,
        transform: 'rotate(' + ((date.getMinutes()/60)*360 - 90 ).toString() + 'deg)',
        transformOrigin: '0% 0%',
        backgroundColor: 'grey'
    }
    let hourHandStyle = {
        position: 'relative',
        top: 92,
        left: 106,
        border: '1px solid grey',
        width: '20%',
        height: 7,
        transform: 'rotate(' + ((date.getHours()/12)*360 - 90 ).toString() + 'deg)',
        transformOrigin: '0% 0%',
        backgroundColor: 'grey'
    }
    return <div>
        <div style={dialStyle}>          5
            <div style={secondHandStyle}/>
            <div style={minuteHandStyle}/>
            <div style={hourHandStyle}/>
        </div>
    </div>
}

```

- 1 Convert string date into an object so we can parse it later
- 2 Use `borderRadius` (`border-radius` in regular CSS) on a `<div>` with high number relative to the width to make it a circle
- 3 Calculate the angle and use `transform` (`Ndeg`) to rotate the second hand with minus `90` to offset for the starting horizontal position of the hand
- 4 Use `transformOrigin` to offset the center of the rotation
- 5 Render the containers with applicable styles relative to the clock `dial` (large circle)

If you have React Developer Tools for Chrome or Firefox (available in [Chrome Web Store](#) and [Mozilla Add-ons](#)), then you can open the React pane in your DevTools (or an analog in Firefox). Mine is showing that `<Clock>` element is having two children (Figure 4-8).

The results along with React DevTools are shown in Figure 4-8. Notice that React is telling us the names of the components along with the state `currentTime`. What a great tool for debugging!

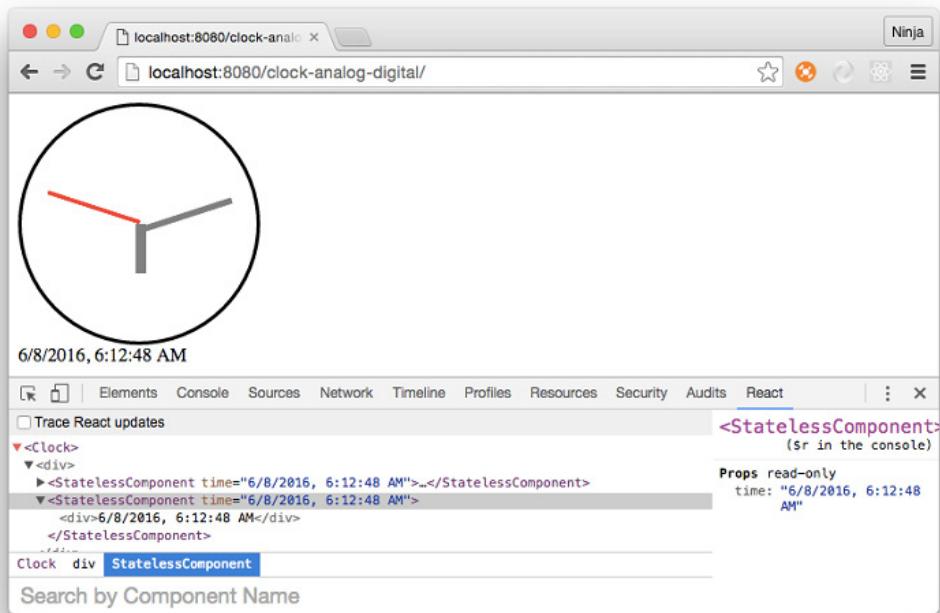


Figure 4.9 React DevTools v0.15.4 shows two components

I used anonymous expressions stored as `const` variables. Another approach would be to use the named function declarations syntax for example:

```
function AnalogDisplay(props) {...}
```

Or we can use the named function declaration referenced from a variable:

```
const AnalogDisplay = function AnalogDisplay(props) {...}
```

About Function Declaration in JavaScript

In JavaScript, there are several way to define a function.

Anonymous function expression used right away (typically as callback):

```
function() { return 'howdy'}
```

Or as anonymous immediately invoked function expression (IIFE):

```
(function() {
  return('howdy')
})()
```

Anonymous function expression referenced in a variable:

```
let sayHelloInMandarin = function() { return 'nǐ hǎo'}
```

Named or hoisted function expression:

```
function sayHelloInTatar() { return 'säläm'}
```

Named or hoisted function expression referenced in a variable:

```
let sayHelloInSpanish = function digaHolaEnEspanol() { return 'hola'}
```

Immediately invoked named function expression:

```
(function() {
  return('howdy')
})()
```

There's no fat arrow syntax for named/hoisted function.

As can be seen, the `AnalogDisplay` and `DigitalDisplay` components are stateless. They don't have any states. They also don't have any methods except for the body of the function which is not unlike `render()` method in a normal React class definition. All of the logic and states of the app is in `Clock`.

In contrast, the only logic we put into one of the stateless components is the animation, but that's closely related to the analog display. Clearly, it would have been a bad design to have analog animation in `Clock`. Now, we have two components and we can render either of them from `Clock` or both of them. Given these points, using stateless components properly with a handful of stateful components will allow for more flexible, simpler and better design.

Most of the times when React developers say stateless, they mean a component created with a function or a fat arrow syntax. It's possible to have a stateless component created with a class but it's not recommended because then it will be too easy for someone else (or you in six months) to add a state.

For now you might be wondering: can a stateless component have methods? Obviously, if you use class then yes, they can have methods but as mentioned before most developers use functions. While you can attach methods to functions because they are also object in JavaScript, the code is not very elegant because we cannot use `this` inside of the function (the value is not the component but `window`):

```
// Anti-pattern: Don't do this.
const DigitalDisplay = function(props) {
  return <div>{DigitalDisplay.locale(props.time)}</div>
}
```

```
DigitalDisplay.locale = (time)=>{
  return (new Date(time)).toLocaleString('EU')
}
```

So if you need to perform some logic related to the view, then create a new function right inside of the stateless component:

```
// Good pattern
const DigitalDisplay = function(props) {
  const locale = time => (new Date(time)).toLocaleString('EU')
  return <div>{locale(props.time)}</div>
}
```

Let's keep stateless components simple. No states and no methods. Especially no calls to external methods or functions because their result might break predictability (and violate the concept of purity).

4.6 Quiz

1. You can set state in a component method (not a constructor) with this syntax: `this.setState(a)`, `this.state = a` or `this.a = a`.
2. If you want to update the render process, it's normal practice to change properties in components like this `this.props.a=100`. True or false?
3. States are mutable and properties are immutable. True or false?
4. Stateless components can be implemented as function. True or false?
5. How do you define the first state variables when element is created? `setState()`, `initialState()`, `this.state = ...` in constructor, or `setInitialState()`?

4.7 Summary

This is the end of the fourth chapter in which we covered a few more very important React concepts and even coded a few non-trivial interactive applications (CSS animation rocks!). Here's what you learned:

- States are mutable; properties are immutable.
- `getInitialState` allows components to have an initial state object.
- `this.setState` updates only the properties that you pass to it and not all state object properties.
- `{}` is a way to render variables and execute JavaScript in JSX code.
- `this.state.NAME` is the way to access state variables.
- Stateless components are the preferred way of working with React.

We're done with the fundamentals. In the next chapters, we'll shift gears and cover how React is working with events such as click, mouse hover, etc. This will allow us to make UIs which interact with user input.

4.8 Quiz Answers

1. `this.setState(a)`
2. False. Changing a property in the component itself won't trigger a re-render.
3. True
4. True
5. `this.state =...` in constructor or `getInitialState()` if you're using `createClass()`

5

React Component Lifecycle Events

In this chapter,

- A Bird's-eye View of React Component Lifecycle Events
- Categories of events
- Defining an event
- All events together
- Mounting, updating and unmounting events



Figure 5.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch05>

Chapter 2 provided information on how to create components, but there are certain situations when we need a more granular control over a component. For instance, you're building a custom radio button component which can change in size depending on a screen width. Another example is you're building a menu which needs to get information from the server by sending an XHR request.

One approach would be to implement the necessary logic before instantiating a component and then simply re-create it by providing different props. Unfortunately, this won't create a

self-contained component and thus we'll lose React's benefit of providing component-based architecture.

The best approach is to use component lifecycle events. Developers can use one of the mounting events to inject necessary logic in their component. More over, other events can be used to make components smarter by providing specific logic on when to re-render their views or not (overwrite of the React's default algorithm).

Onward to learning about them!

The source code for the examples in this chapter is in [the ch05 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

5.1 A Bird's-eye View of React Component Lifecycle Events

React provides a way for developers to control and customize component's behavior based on its lifecycle events (think of [the hooking term](#) in computer programming). They belong to the following categories:

1. Mounting Events: Happens when React element (instance of a component class) is attached to a DOM node
2. Updating Events: Happens when React element is updating either as a result of new values of its properties or state
3. Unmounting Events: Happens when React element is detached from DOM

Each and every React component has *lifecycle events* which are triggered at certain moments depending on what a component has done or will do. Some of them execute just once while other events can be executed continuously.

They allow developers to implement custom logic which will enhance what components can do. Another usage is to modify the behavior of components, for example decide when to re-render and when not. This will allow to enhance performance because unnecessary operations will be eliminated. Another usage is to fetch data from the back-end or integrate with DOM event or other front-end libraries. Let's see closer how categories of events operate, what events they possess and in what sequence those events are executed.

5.2 Categories of Events

React defines several component events within three categories (also shown in Figure 5-1 and Table 5-1). Each category can fire event various number of times:

1. Mounting: React invokes events only once
2. Updating: React can invoke events many times
3. Unmounting: React invokes event only once

In short, the categories are shown in Figure 5-1 going from left to right in terms of the component lifecycle.

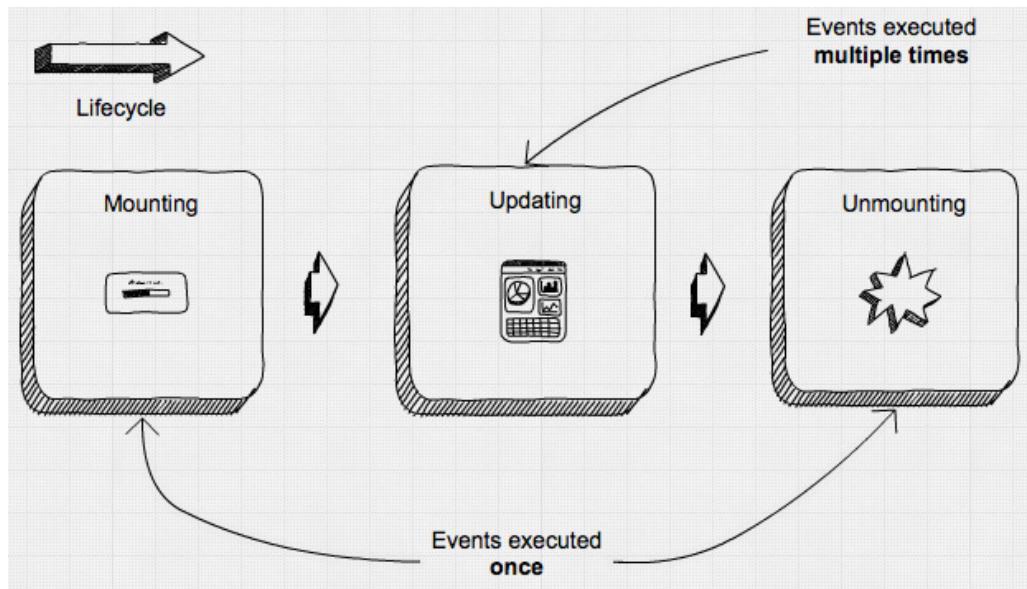


Figure 5.2 Categories of lifecycle events as component goes in its lifecycle and how many times events in a category can be called

In addition to lifecycle events, I'll include `constructor` to illustrate the order of execution in the order from start to end of the lifecycle (updating can happen multiple times):

1. `constructor()`: Happens when element is created and allows to set the default properties (chapter 2) and the initial state (chapter 4)

Mounting

1. `componentWillMount()`: Happens before mounting to the DOM
2. `componentDidMount()`: Happens after mounting and rendering

Updating

1. `componentWillReceiveProps(nextProps)`: Happens when component is about to receive properties
2. `shouldComponentUpdate(nextProps, nextState) -> bool`: Allows to optimize component's re-rendering by determining when to update and when to not
3. `componentWillUpdate(nextProps, nextState)`: Happens right before component will update

4. `componentDidUpdate(prevProps, prevState)`: Happens right after component updated

Unmounting

1. `componentWillUnmount` function(): Allows to unbind and detach any event listeners or do other clean up work before component is unmounted

Most of the time, it's clear when the event is triggered by its name. For example, `componentDidUpdate()` will be fired when the component is updated. In other cases, there are subtle differences. Here's a table which shows the sequence of lifecycle events (from top to bottom), and how some of them are dependent on changes of properties or state (columns "Component Properties" and "Component State").

There's one more case in which component might be re-rendered. It's when `this.forceUpdate()` is called. As you can guess from the name, it's a forced update. Developers resort to using it when for one reason or the other when updating state or properties won't trigger a desired re-render. As an example, this might happen when developers uses some data in `render()` which is not part of state or props and that data changes. Hence the need to manually trigger an update. Generally speaking (and according to React core team), the `this.forceUpdate()` method should be avoided, because it makes component impure.

Pure Functions

In computer science in general not just React, a pure function is a function which:

1. Given the same input will **ALWAYS** return the same input
2. Has no side effects (altering of external states)
3. Does not rely on any external state.

For example, here's a pure function which doubles the value of the input `f(x) = 2x` or in JavaScript/Node: `let f=(n)=>2*n` is **pure**. In action:

```
let f = (n)=>2*n
console.log(f(7))
```

An impure function to double numbers will look like this in action (by putting curly braces we remove the implicit return of the one-liner-fat-arrow function):

```
let sharedStateNumber = 7
let double
let f = ()=> {double =2*sharedStateNumber}
f()
console.log(double)
```

Pure functions are the corner stone of functional programming (FP) which minimizes state as much as possible. The main reason why developers (especially functional programmers) prefer pure functions because their usage mitigates

shared states which in turns simplifies development and de-couples different pieces of logic. In addition, it makes testing easier. When it comes to React, we already know that having more stateless components and less dependencies is better that's why the best practice is to create pure functions.

In some way FP contradicts OOP (or is it OOP contradicts FP?) with FP fans saying that Fortran and Java was a dead end of programming and Lisp (and nowadays Clojure, Elm) is the way to go. It's a fascinating debate to follow (and spend time on). Personally, I am slightly biased toward functional approach.

There's a great deal of good books on written on FP because the concept has been around for decades. For this reason, we won't get into much details here... but I highly recommend learning more about FP because it will make you better programmer even if you never plan to use on your job.

Table 5.1

Mounting	Updating			Unmounting
	Component Properties	Component State	Using forceUpdate()	
<code>constructor()</code>				
<code>componentWillMount</code> ()				
	<code>componentWillReceiveProps()</code> ()			
		<code>shouldComponentUpdate()</code>		
			<code>componentWillUpdate()</code>	
			<code>render()</code>	
			<code>componentDidUpdate()</code>	
<code>componentDidMount</code> ()				<code>componentWillUnmount</code> ()

Let's define one of the events to see it in action.

5.3 Implementing an Event

The way we implement lifecycle events is we define them on a class as methods (React Component Methods in chapter 3)— it's just a convention that React expects developers to follow. React will check if there's a method with an event name, in case it found a method, React will call the method. Otherwise, React will continue its normal flow.

To put it differently, under the hood, React will call certain methods during a component lifecycle if they are defined (for example, if you define `componentDidMount()`, then React will call this method when an element of this component class is mounted).

Obviously, event names are case sensitive as any name in JavaScript.

An example is `componentDidMount` which belongs to mounting category. It will be called just once per each instance of this component class.

```
class Clock extends React.Component {
  componentDidMount() {
  }
  ...
}
```

If there's no method `componentDidMount()` defined, React will not execute any code for this event. Thus, name of the method must match the name of the event and going forward for lifecycle event, I'll be using terms events, event handler and methods interchangeably in this chapter.

As you might guess from its name, the `componentDidMount` method is invoked right when a component is inserted into the DOM. This method is a recommended place to put code to integrate with other front-end frameworks and libraries as well as send XHR requests to a server, because at this point in the lifecycle the component's element is in the real DOM and you get the access to all its elements including children.

Going back to the aforementioned problems: resizing and fetching data from a server. For the first, we can create event listener in `componentDidMount()` which will listen for `window.resize` events. For the second, we can make an XHR call in `componentDidMount()` and update the state when we have the response from the server.

Equally important, `componentDidMount` comes in handy in isomorphic/universal code (i.e., same components being used on the server and in the browser). With this event, you can put some browser-only logic inside of it and rest assured that it'll only be called for the browser rendering and not on the server-side. There's more on isomorphic JavaScript with React in chapter 16.

We learn best by examples. For this reason, let's consider the following trivial snippet, which uses `componentDidMount()` to print the DOM information to the console. This is feasible because this event is fired after all the rendering has happened; thus we have access to the DOM elements.

The way to create event listeners for the component lifecycle events is straightforward—you define a method on the component/class. For the fun of it, let's add `componentWillMount()` to contrast the absence of the real DOM for this element at this stage.

The DOM node information is obtained via React DOM's utility function `ReactDOM.findDOMNode()`, to which we pass the class. Please note that DOM is not camelCase, but rather in all caps

```
class Content extends React.Component {
  componentWillMount() {
    console.log(ReactDOM.findDOMNode(this))      ①
  }
  componentDidMount() {
    console.dir(ReactDOM.findDOMNode(this))       ②
  }
  render() {
    return (
      <div/>
    )
  }
}
```

- ① Expect DOM node to be null
- ② Expect DOM node to be <div>

The result is this output in the developer console which reassures us that `componentDidMount()` will happen when we have real DOM elements (Figure 5-2):

```
null
div
```

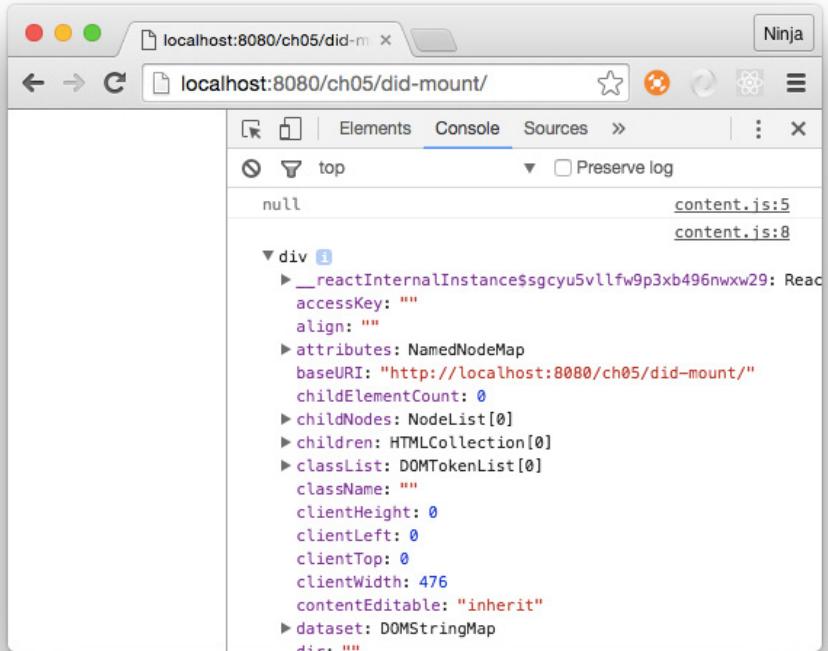


Figure 5.3 Second log shows DOM node because `componentDidMount()` was fired when the element is rendered and mounted to the real DOM, thus we have the node

In view of the diagram of event categories shown on Table 5-1, `componentDidMount()` belongs to the mounting category. Let's cover each category, starting with mounting.

5.4 All Events Together

To see all the events in action at once. For now all you need to know is that they are like classes in the sense that they allow developers to reuse code. This logger mixin can be useful for debugging. It has all the events and props and states when the component is about to be re-rendered or after it has been re-rendered.

Listing 5-1: Rendering and updating Display component 3 times
 (ch05/logger/jsx/content.jsx):

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.launchClock()
```

```

    this.state = {
      counter: 0,
      currentTime: (new Date()).toLocaleString()
    }
  }
  launchClock() {
    setInterval(()=>{
      this.setState({
        counter: ++this.state.counter,
        currentTime: (new Date()).toLocaleString()
      })
    }, 1000)
  }
  render() {
    if (this.state.counter > 2) return <div/>
    return <Logger time={this.state.currentTime}></Logger>
  }
}

```

Listing 5-2: Observing different component lifecycle events, when and how many times they are invoked (ch05/logger/jsx/logger.jsx).

```

class Logger extends React.Component {
  constructor(props) {
    super(props)
    console.log('constructor')
  }
  componentWillMount() {
    console.log('componentWillMount is triggered')
  }
  componentDidMount(e) {
    console.log('componentDidMount is triggered')
    console.log('DOM node: ', ReactDOM.findDOMNode(this))
  }
  componentWillReceiveProps(newProps) {
    console.log('componentWillReceiveProps is triggered')
    console.log('new props: ', newProps)
  }
  shouldComponentUpdate(nextProps, nextState) {
    console.log('shouldComponentUpdate is triggered')
    console.log('new props: ', nextProps)
    console.log('new state: ', nextState)
    return true
  }
  componentWillUpdate(nextProps, nextState) {
    console.log('componentWillUpdate is triggered')
    console.log('new props: ', nextProps)
    console.log('new state: ', nextState)
  }
  componentDidUpdate(oldProps, oldState) {
    console.log('componentDidUpdate is triggered')
    console.log('new props: ', oldProps)
    console.log('old props: ', oldState)
  }
  componentWillUnmount() {
    console.log('componentWillUnmount')
  }
  render() {

```

```
// console.log('rendering... Display')
return (
  <div>{this.props.time}</div>
)
}
```

The functions and lifecycle events from the `Display` component will give us console logs when run this web page. Don't forget to open your browser console, because all the logging happens there (Figure 5.4)! ;-)

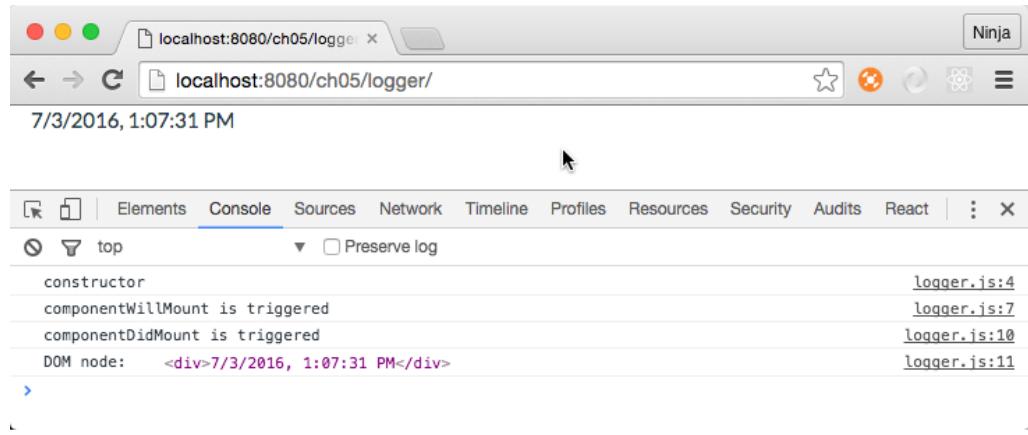


Figure 5-3: Logger has been mounted

As noted in the text and illustrated above, mounting event will fire only once. You can clearly see it in the logs. After the counter in `Context` reaches 3, the `render` function won't use `Display` anymore and it will be unmounted as shown in Figure 5.5.

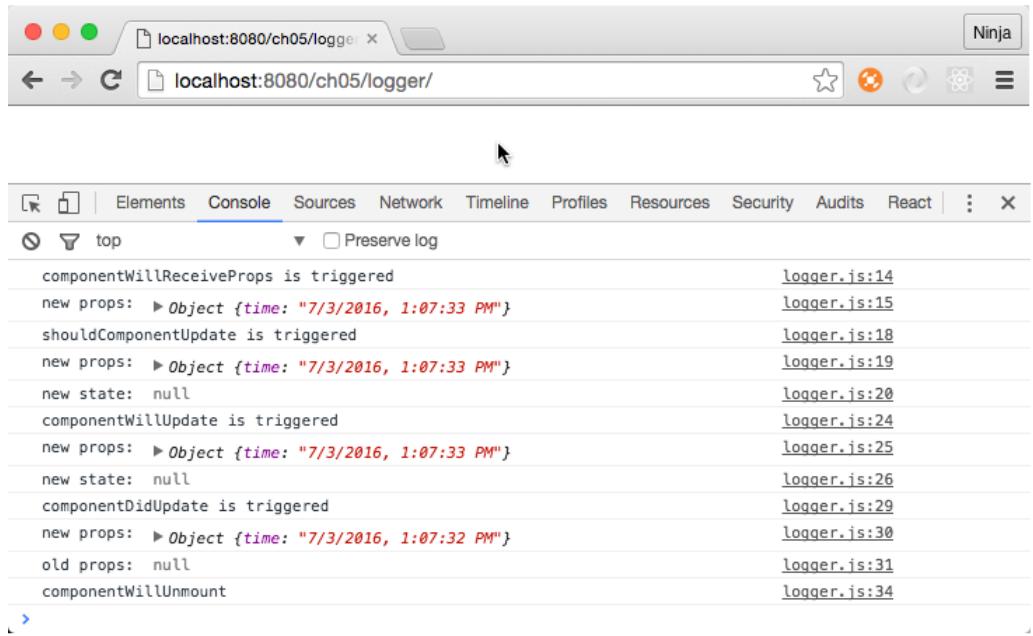


Figure 5.5 Content removed Logger after 2 seconds, hence the `componentWillUnmount()` log

Now that we've learned about component lifecycle events, we can use them when we need to implement some logic for the components, such as fetching the data.

5.5 Mounting Events

Mounting category of events is all about a component is being attached to the real DOM. Think about mounting as a way for a React element to see itself in the DOM. This typically happens when you use a component in a `ReactDOM.render()` or in `render()` of another higher order component which will be rendered to the DOM. The mounting events are as follows:

1. `componentWillMount()`: React knows that this element will be in the real
2. `componentDidMount()`: React mounted the element

The `constructor()` will happen prior to `componentWillMount()`. Also, React first renders then mounts elements. (Rendering in this context means calling `render()` of a class, not painting the DOM). Refer to the Table 5-1 for events in between `componentWillMount()` and `componentDidMount()`.

5.5.1 componentWillMount()

It's worth mentioning that `componentWillMount()` will be invoked only once in the component's lifecycle. The timing of the execution is right *before the initial rendering*.

The lifecycle event `componentWillMount()` will be executed when you render a React element on the browser by calling `ReactDOM.render()`. Think about it as attaching (or mounting) a React element to a real DOM node. This happens in the browser, that's front-end.

If you render a React component on a server (that's back-end using isomorphic/universal JavaScript, see chapter *React on The Server*) which is basically just getting an HTML string, then—even there's no DOM on the server or mounting for that case—this event will also be invoked!

We saw in chapter *Making React Interactive with States* how to update the `currentTime` state using `Date` and `setInterval()`. We triggered the series of updates in `constructor()` by calling `launchClock()`. We can do so in `componentWillMount()` as well.

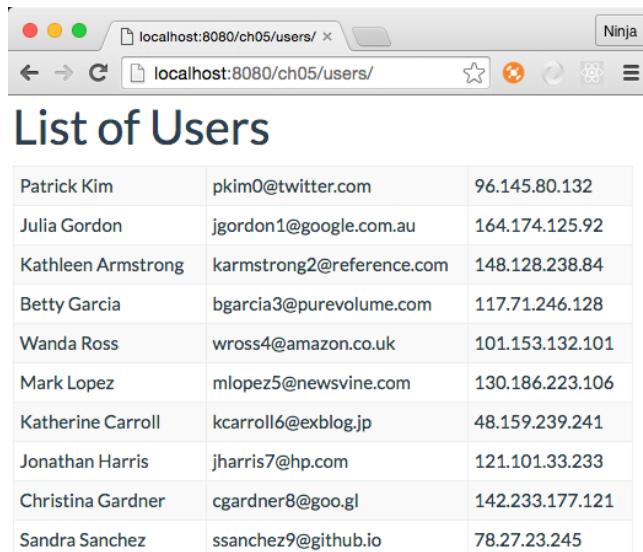
Typically, a state change triggers a re-render, right? At the same time, if you update the state with `setState()` inside of the `componentWillMount()` method or trigger updates as we did with `Clock`, then `render()` will get the updated state. The best thing is that even if the new state is different, there will be *no re-rendering* because `render()` will get the new state. To put it another way, we can invoke `setState()` in `componentWillMount()`, the `render()` will get the new values if any and there would be no extra re-rendering because of that.

5.5.2 componentDidMount()

In contrast, `componentDidMount()` is invoked right *after the initial rendering*. It's **executed only once and only in the browser, not on the server**. This comes in handy when developers need to implement code which runs only for browsers such as XHR requests.

In this lifecycle event, you can access any references to your children (e.g., to access the corresponding DOM representation). Note that the `componentDidMount()` method of child components is invoked before that of parent components.

As mentioned before, the `componentDidMount` event is the best place to integrate with other JavaScript libraries. We can fetch a JSON payload which has list of users with their info. Then, we can print that info using Twitter Bootstrap table to get the page shown in Figure 5-5.



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/ch05/users/'. The title of the page is 'List of Users'. Below the title is a table with 10 rows, each representing a user with three columns: Name, Email, and IP Address.

Patrick Kim	pkim0@twitter.com	96.145.80.132
Julia Gordon	jgordon1@google.com.au	164.174.125.92
Kathleen Armstrong	karmstrong2@reference.com	148.128.238.84
Betty Garcia	bgarcia3@purevolume.com	117.71.246.128
Wanda Ross	wross4@amazon.co.uk	101.153.132.101
Mark Lopez	mlopez5@newsvine.com	130.186.223.106
Katherine Carroll	kcarroll6@exblog.jp	48.159.239.241
Jonathan Harris	jharris7@hp.com	121.101.33.233
Christina Gardner	cgardner8@goo.gl	142.233.177.121
Sandra Sanchez	ssanchez9@github.io	78.27.23.245

Figure 5-5: Showing a list of users (fetched from a data store) styled with Twitter Bootstrap

The structure of the project is as follows:

```
/users
  /css
    - bootstrap.css
  /js
    - react-15.0.2.js
    - react-dom-15.0.2.js
    - script.js
    - users.js
  /jsx
    - script.jsx
    - users.jsx
  - index.html
  - real-user-data.json
```

You have the DOM element in the event, and you can send XHR/AJAX requests to fetch the data with the new `fetch()` API:

```
fetch(this.props['data-url'])
  .then((response)=>response.json())
  .then((users)=>this.setState({users: users}))
```

Fetch API

[Fetch API](#) allows to make XHR request using promises in a unifying manner. It's available in most modern browsers, but [refer to the specs](#) and [the standard](#) to find out if the browsers you need to support for your apps implement it. The usage is straightforward, pass the URL and define as many promise `then` statements as needed:

```
fetch('http://node.university/api/credit_cards/')
  .then(function(response) {
    return response.blob()
  })
  .then(function(blob) {
    // Process blob
  })
  .catch(function(error) {
    console.log('There has been a problem with your fetch operation: ' + error.message)
  })
```

In case the browser you develop for is not supporting `fetch()` yet, you can shim it, or utilize any other HTTP agent library such as [superagent](#), [request](#), [axios](#)... or even [jQuery's](#) `$.ajax()` or `$.get()`.

We can put our XHR fetch request in `componentDidMount()`. You might think that by putting the code in `componentWillMount()` developers can optimize loading, but there are two issues: if you get data from the server faster than your rendering finishes, you might get trigger re-render on an unmounted element which might lead to unintended consequences; also, if you're planning on using a component on the server, then `componentWillMount()` will fire there as well.

So now we can take a look at the entire component with the fetching happening in the `componentDidMount()` as shown in Listing 5-3.

Listing 5-3 Fetching data from a server in `componentDidMount()` to display it in a table (ch05/users/jsx/users.jsx)

```
class Users extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      users: []           ①
    }
  }
  componentDidMount() {
    fetch(this.props['data-url'])
      .then((response)=>response.json()) ②
      .then((users)=>this.setState({users: users})) ③
  }
  render() {
    return <div className="container">
      <h1>List of Users</h1>
      <table className="table-striped table-condensed table table-bordered table-hover">
```

```

<tbody>{this.state.users.map((user)=>          ④
  <tr key={user.id}>
    <td>{user.first_name} {user.last_name}</td>
    <td> {user.email}</td>
    <td> {user.ip_address}</td>
  </tr>)
</tbody>
</table>
</div>
}
}

```

- ➊ Initialize `users` state with an empty array
- ➋ Perform GET XHR request using URL from the property to fetch user data
- ➌ Retrieve user info from response and assign to state
- ➍ Iterate over users state to create table rows

Notice that the `users` is set to an empty array `[]` in the constructor. This will avoid the need to check for existence later in `render()`. Setting your initial values will avoid lots of pain later! In other words, this is an anti-pattern:

```

// Anti-pattern: Don't try this at home!
class Users extends React.Component {
  constructor(props) {           ➊
    super(props)
  }
  ...
  render() {
    return <div className="container">
      <h1>List of Users</h1>
      <table className="table-striped table-condensed table-bordered table-hover">
        <tbody>{this.state.users && this.state.users.length>0) ?           ➋
          this.state.users.map((user)=>
            <tr key={user.id}>
              <td>{user.first_name} {user.last_name}</td>
              <td> {user.email}</td>
              <td> {user.ip_address}</td>
            </tr>) : ''
          </tbody>
        </table>
      </div>
    }
}

```

- ➊ Avoid not setting the empty value initially
- ➋ Avoid excessive complexity of existence checks

5.6 Updating Events

As noted before, the previous category of mounting events is often used for integrating React with outside world be it other frameworks, libraries or data stores. This category of updating events deals with the events associated with updating the component. The events are as follows shown in the order from beginning to the end of the lifecycle (see Table 5-2 for the just the updating lifecycle events and Table 5-1 for all events including other categories).

1. componentWillReceiveProps(nextProps)
2. shouldComponentUpdate
3. componentWillUpdate
4. componentDidUpdate

Updating		
Component Properties	Component State	Using forceUpdate()
componentWillReceiveProps()		
shouldComponentUpdate()		
	componentWillUpdate()	
	render()	
		componentDidUpdate()

5.6.1 componentWillReceiveProps ()

`componentWillReceiveProps()` is triggered when component receives new properties (props). This stage called an incoming prop transition. This event allows developers to intercept component at the stage between getting new properties and before `render()` for the reasons of putting there some logic.

The `componentWillReceiveProps(nextProps)` method takes the new prop(s) as an argument. It is not invoked on the initial render of the component. This method is useful if you would like to capture the new prop and set the state accordingly before the re-render. The old prop value is in the `this.props` object. For example, here we set the state `opacity` which is used in CSS to 0 or 1 according to the boolean prop `isVisible`:

```
componentWillReceiveProps(newProps) {
  this.setState({
    opacity: (newProps.isVisible) ? 1 : 0
  })
}
```

Generally speaking, the `setState` method within `componentWillReceiveProps` will not trigger extra re-rendering.

In spite of receiving new properties, these properties might not necessarily have new property values (meaning values different from current properties), because React has no way of knowing if the property values have been changed. Therefore, `componentWillReceiveProps()` is invoked each time there's a re-rendering (of a parent structure or a call), irrespective of the property value changes. Thus, developers cannot assume that `nextProps` has always different values from the current properties.

At the same time, re-rendering (invoking `render()`) doesn't necessarily means change in the real DOM. The actual decision on whether to update and what to update in the real DOM is

delegated to `shouldComponentUpdate()` and the reconciliation process. For more reasons why React can't perform smarter check before calling `componentWillReceiveProps()`, read this extensive article ([A => B](#)) \Rightarrow ([B => A](#)).

5.6.2 `shouldComponentUpdate()`

Then we have the `shouldComponentUpdate()` event, which is invoked right before the rendering. The rendering is preceded by new props or state being received. The `shouldComponentUpdate()` event is not triggered for the initial render or for `forceUpdate()`—see Table 5-1.

Developers can implement the `shouldComponentUpdate()` event with return false to prohibit React from re-rendering. This is useful when you're checking that there are no changes and you want to avoid an unnecessary performance hit (when dealing with hundreds of components). For example, here I'm using the `+ binary operator` to convert the Boolean `isVisible` into a number and compare that to the `opacity` value:

```
shouldComponentUpdate(nextProps, nextState) {
  return this.state.opacity !== +newProps.isVisible
}
```

When `isVisible` is `false` and `this.state.opacity` is `0`, the entire `render()` will be skipped, and `componentWillUpdate()` and `componentDidUpdate()` won't be called either. In essence, developers can control whether component is re-rendered.

5.6.3 `componentWillUpdate()`

Speaking of `componentWillUpdate()`, this event is called right before rendering preceded by receiving new props or state. This method is not called for the initial render. Use the `componentWillUpdate()` method as an opportunity to perform preparations before an update occurs, and avoid using `this.setState()` in this method! Why? Well, can you imagine trying to trigger a new update while the component is being updated? It sounds like a bad idea to me. :)

If `shouldComponentUpdate()` returns `false`, then `componentWillUpdate()` will not be invoked.

5.6.4 `componentDidUpdate()`

The `componentDidUpdate` event is triggered right after the component's updates are reflected in the DOM. Again, this method is not called for the initial render. The `componentDidUpdate()` event is useful to put code to work with DOM after the component has been updated, because at this stage we'll get all the updates rendered in the DOM.

Every time we have something mounted or updated, there should be a way to unmount it. The next event will provide a place for developers to put some logic for unmounting.

5.7 Unmounting Event

In React, unmounting means detachment or removal of an element from DOM. There's just a single event in this category and this is the last category in the component lifecycle.

5.7.1 componentWillMount()

The last category of lifecycle events has a single event in it: `componentWillUnmount()`. It's called right before a component is unmounted from the DOM. You can add any necessary cleanup to this method: for example, invalidating timers or cleaning up any DOM elements or detaching events that were created in `componentDidMount`.

5.8 A Simple Example

For instance, you tasked with creating a Note web app (for saving some text online). You have the component implemented but first feedback came from users is that they lose their progress if they close the window (or a tab) unintentionally. Let's implement a confirmation dialog (Figure 5-6).

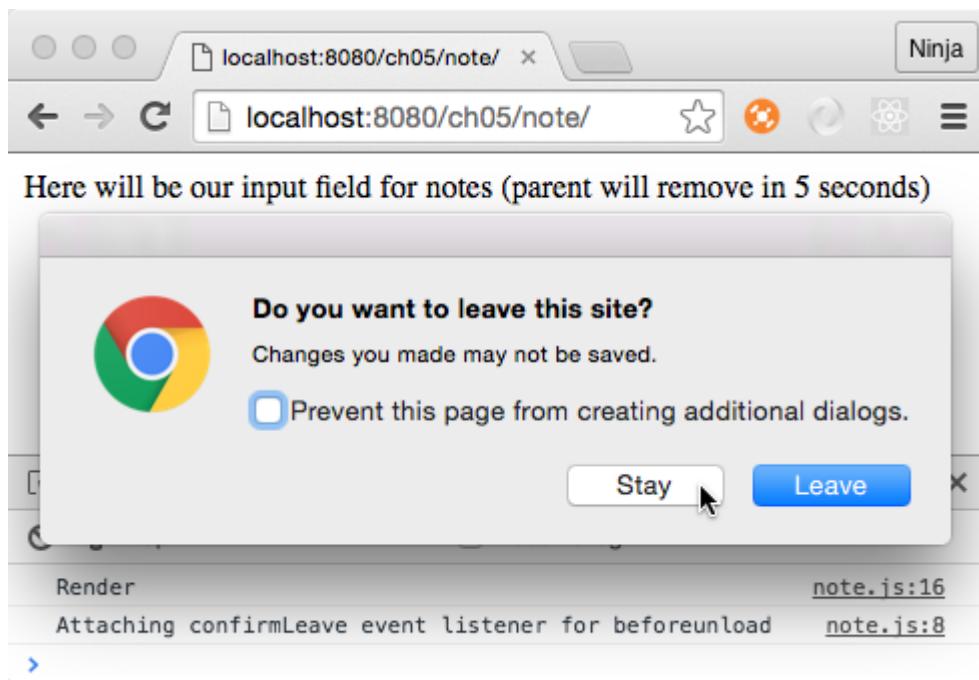


Figure 5.7 A dialog confirmation on when user tries to leave the page

The structure of the project is as follows:

```
/note
  /jsx
    - note.jsx
    - script.jsx
  /js
    - note.jsx
    - react-15.0.2.js
    - react-dom-15.0.2.js
    - script.js
  - index.html
```

The `window.onbeforeunload` native browser event (some additional code is for cross-browser support) is straightforward:

```
window.addEventListener('beforeunload',function () {
  let confirmationMessage = 'Do you really want to close?'
  e.returnValue = confirmationMessage           // Gecko, Trident, Chrome 34+
  return confirmationMessage                   // Gecko, WebKit, Chrome <34
})
```

The following approach will work too:

```
window.onbeforeunload = function () {
  ...
  return confirmationMessage
}
```

Let's put this code in an event listener in `componentDidMount()` and remove the event listener in `componentWillUnmount()`:

Listing 5-4: Adding and removing window close event listener according to component's lifecycle (ch05/note/jsx/note.jsx)

```
class Note extends React.Component {
  confirmLeave(e) {
    let confirmationMessage = 'Do you really want to close?'
    e.returnValue = confirmationMessage           // Gecko, Trident, Chrome 34+
    return confirmationMessage                   // Gecko, WebKit, Chrome <34
  }
  componentDidMount() {
    console.log('Attaching confirmLeave event listener for beforeunload')
    window.addEventListener('beforeunload', this.confirmLeave)
  }
  componentWillUnmount() {
    console.log('Removing confirmLeave event listener for beforeunload')
    window.removeEventListener('beforeunload', this.confirmLeave)
  }
  render() {
    console.log('Render')
    return <div>Here will be our input field for notes (parent will remove in {this.props.secondsLeft} seconds)</div>
  }
}
```

We would like to remove this `Note` element so that it is dismounted, and we'll use timer for that (`setInterval()` all the way!) as you can observe from Listing 5-5 and Figure 5.8.

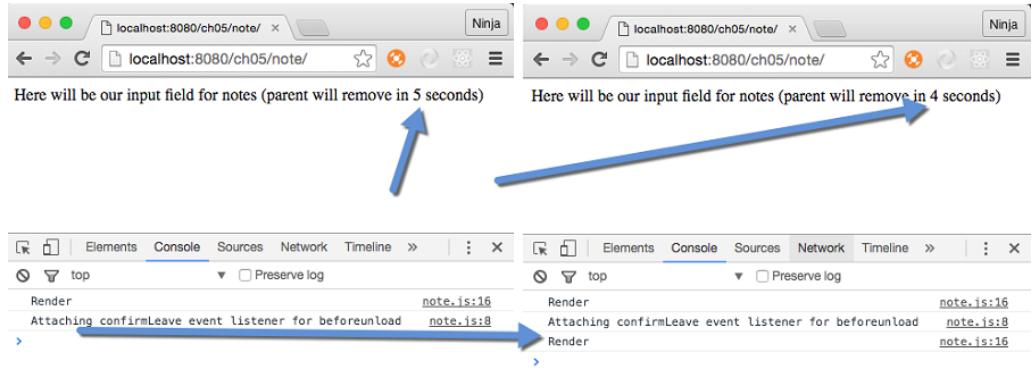


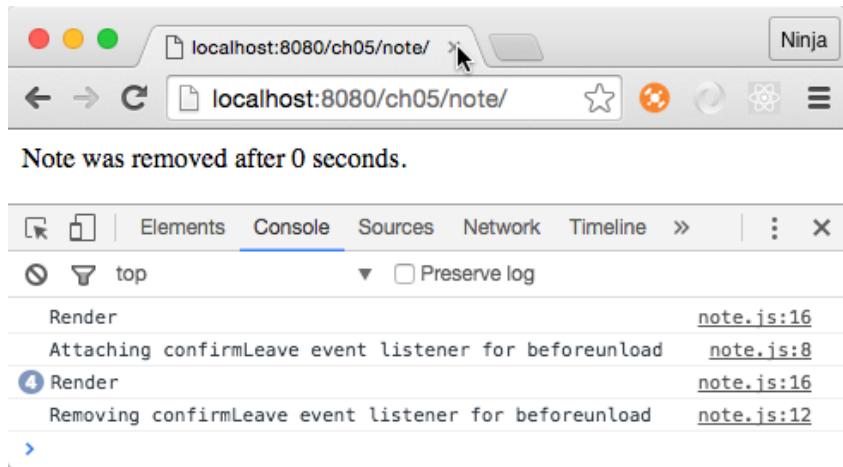
Figure 5.8 Note will be replaced by another element in 5, 4, ... seconds

Listing 5-5: Rendering Note a few times before removing it from DOM (ch05/note/jsx/script.jsx)

```
let secondsLeft = 5

let interval = setInterval(()=>{
  if (secondsLeft == 0) {
    ReactDOM.render(
      <div>
        Note was removed after {secondsLeft} seconds.
      </div>,
      document.getElementById('content')
    )
    clearInterval(interval)
  } else {
    ReactDOM.render(
      <div>
        <Note secondsLeft={secondsLeft}/>
      </div>,
      document.getElementById('content')
    )
  }
  secondsLeft--
}, 1000)
```

Here's how the result (with console logs) will look like: Render, attach event listener, render 4 more times, remove event listener:



If we don't remove this event listener in `componentWillUnmount()` (you can simply comment out this method), this page will still have this pesky dialog even that the `Note` element is long gone as shown in Figure 5-9. This is not a good user experience and might lead to bugs. Use this lifecycle event to clean up after a component.

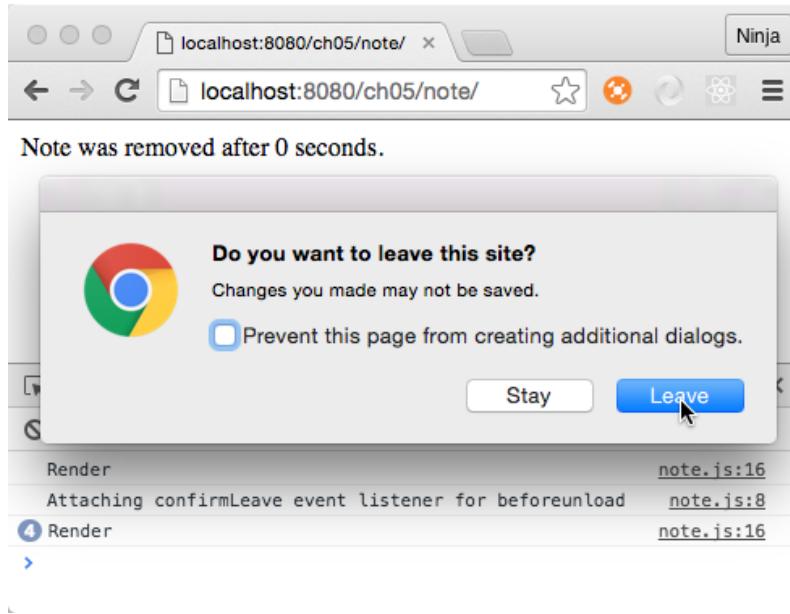


Figure 5.9 Note is replaced by a div and there would be no dialog confirmation when user tries to leave the page

React is evolving fast, and there might be changes or additions to the lifecycle events in the future. If you need to refer to the official documentation, here it is: <https://facebook.github.io/react/docs/component-specs.html#lifecycle-methods>. In the next section, we'll see all events at once in a single component.

5.9 Quiz

1. `componentWillMount()` will be rendered on the server. True or false?
2. What event will fire first, `componentWillMount()` or `componentDidMount()`?
3. What is a good place to put an AJAX call to server to get some data for our component: `componentWillUnmount()`, `componentHasMounted()`, `componentDidMount()`, `componentWillReceiveProps()` or `componentWillMount()`?
4. `componentWillReceiveProps()` means there was a re-rendering of this element (from a parent structure), and we know for sure that we have new values of the props. True or false?
5. Mounting event happen multiple time on each re-rendering. True or false?

5.10 Summary

- `componentWillMount()` is invoked on both the server and the client, while `componentDidMount` is invoked only on the client.
- Mounting events typically used for integrating React with other libraries and for getting data from stores or servers.
- Developers use `shouldComponentUpdate()` to optimize rendering
- Developers use `componentWillReceiveProps()` to perform state change with new properties
- Unmounting events typically used for a clean up
- Updating events provide place to put logic which relies on new properties or state as well as for more granular control over when to update a view.

5.11 Quiz Answers

1. True
2. `componentWillMount`
3. `componentDidMount()`
4. False, because we cannot guarantee new values
5. False

6

Handling Events in React

In this chapter, we'll learn the following:

- Working with DOM Events in React
- Responding to DOM Events Not Supported by React
- Integrating React with Other Libraries: jQuery UI Events



Figure 6.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch06>

So far, you've learned how to render UIs which had zero user interaction. In other words, we were just displaying data. For example, we've built a clock which wasn't accepting any user inputs like setting a time zone.

Most of the time, having static UIs is not the case, because we need to build elements smart enough to respond to user actions. So how do you respond to user actions such as click of a mouse or drag of a mouse?

This chapter will provide the solution on how to handle events in React, and in the next chapter, we'll apply this knowledge of events to working with web forms and their elements.

More over, I've mentioned that React supports only certain events. In this chapter, I'll show you how to work with events not supported by React.

The source code for the examples in this chapter is in [the ch06 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

6.1 Working with DOM Events in React

Let's look how we can make React elements respond to user actions by defining event handlers for those actions. The way we do this is to define the event handler (function definition) as the value of an element attribute in JSX and as an element property in plain JavaScript (when `createElement()` is called directly without JSX). For attributes which are event names, we use standard W3C DOM event names in camelCase such as `onClick`, `onMouseOver`, etc.

```
onClick={function() {...}}
```

or

```
onClick={()=>{}}
```

For example in React, we can define an event listener which will be triggered when user clicks on this button. In the event listener, we are logging context `this` and an event object is an enhanced version of a native DOM event object (it's called `SyntheticEvent`):

```
<button onClick={(function(event) {
  console.log(this, event)
}).bind(this)}>
  Save
</button>
```

The `bind()` is needed so that in the event handler function we get the reference to the instance of the class (that is React element). If you don't bind, `this` will be `null`. You wouldn't bind context to the class using `bind(this)` in cases:

1. When you don't need to refer to this class by using `this`
2. When you're using older style of `React.createClass()` instead of newer ES6+ class style because `createClass()` autobinds it for you
3. When you're using fat arrows `()=>{}`

We can use also make things a bit neater using a class method as event handler for this `onClick` event. Consider a `SaveButton` component which when clicked prints the value of `this` and `event` as shown in Figure 6-1 and Listing 6-1.

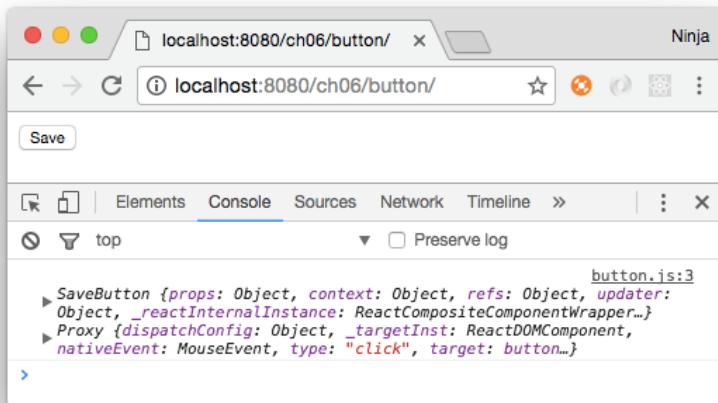


Figure 6.2 Clicking on the button prints the value of `this` which is SaveButton

**Listing 6-1: Declaring event and event handler as a class method
(ch06/button/jsx/button.jsx)**

```

class SaveButton extends React.Component {
  handleSave(event) {
    console.log(this, event)
  }
  render() {
    return <button onClick={this.handleSave.bind(this)}>
      Save
    </button>
  }
}

```

①

- ① Pass the function definition returned by `bind()` to the `onClick`

Moreover, developers can bind an event handler to the class in the class constructor. Functionally, there's no difference, but if you're using the same method more than once in your `render()`, then you can reduce the duplication by using the constructor binding. Here's the same button but with constructor binding for the event handler:

```

class SaveButton extends React.Component {
  constructor(props) {
    super(props)
    this.handleSave = this.handleSave.bind(this) ①
  }
  handleSave(event) {
    console.log(this, event)
  }
  render() {

```

①

```

    return <button onClick={this.handleSave}>
      Save
    </button>
  }
}

```

- ➊ Bind context `this` to the class to use `this` in the event handler to refer to this class
- ➋ Pass the function definition to onClick

Binding event handlers is my favorite and recommended approach, because it eliminates duplication and puts all binding neatly in one place.

Table 6-1 lists the current event types supported by React v15 (15.0.2 and maybe 15.x). Notice the use of the camelCase in the event names to be consistent with the rest of the attribute names in React.

Table 6.1 DOM Events Supported by React v15

Event Group	Events Supported by React
Mouse events	onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave onMouseMove onMouseOut onMouseOver onMouseUp
Keyboard events	onKeyDown onKeyPress onKeyUp
Clipboard events	onCopy onCut onPaste
Form events	onChange onInput onSubmit
Focus events	onFocus onBlur
Touch events	onTouchCancel onTouchEnd onTouchMove onTouchStart
UI events	onScroll
Wheel events	onWheel
Selection events	onSelect
Image events	onLoad onError
Animation Events	onAnimationStart onAnimationEnd onAnimationIteration
Transition Events	onTransitionEnd

As you can see, React supports several types of normalized events. If you contrast this with the list of standard events at [MDN](#) you'll see that React's support is very extensive and you can be sure that team React will be adding more events in the future! For more information and events names, visit the documentation page: <http://facebook.github.io/react/docs/events.html>.

6.1.1 Capture and Bubbling Phases

As has been noted, React is declarative, not imperative. So we don't attach events to our code like we would do with jQuery (thank God!). Instead, we declare an event in the JSX as an attribute. If we're declaring mouse events, the attribute name can be any one of the supported events from table 6-1. The value of the attribute is our event handler.

For example, if we want to define a mouse hover event we can use `onMouseOver` as shown in Listing 6-1. The hover on this element will print "mouse is over" in your DevTools or Firebug console when you move your cursor over the div's red border.

```
<div
  style={{border: '1px solid red'}}
  onMouseOver={()=>{console.log('mouse is over')}} >
  Open DevTools and move your mouse cursor over here
</div>
```

The events shown prior such as `onMouseOver` are triggered by an event in the bubbling phase (a.k.a. bubble up). As you know, there's also a capture phase (a.k.a. trickle down) which precedes bubbling and target phases. First goes capture phase from window down to the target element, then the target phase, and only then the bubbling phase when event is traveling up the tree back to window as shown on Figure 6-2.

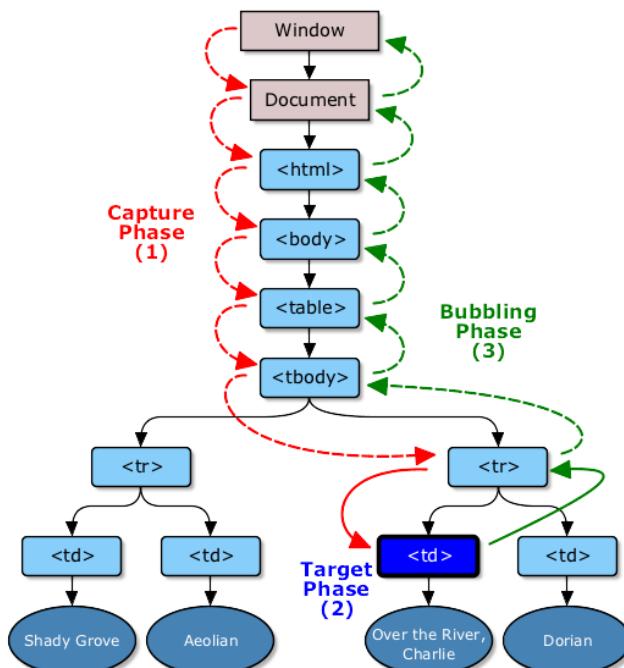


Figure 6-3 Capture, target and bubbling phases

The distinction between phases becomes important when you have the same event on an element and its ancestor(s). In the bubbling mode, the event is first captured and handled by the innermost element (target) and then propagated to outer elements (ancestors starting with the target's parent). In the capturing mode, the event is first captured by the outermost element and propagated to the inner elements.

To register an event listener for the capture phase, append `Capture` to an event name. For example, instead of using `onMouseOver`, you would use `onMouseOverCapture` to handle the mouse over event in the capture phase. This applies to all the event names listed above.

To illustrate, we can have a `<div>` with a regular (bubbling) event and a capture event. Those events are defined with `onMouseOver` and `onMouseOverCapture` respectively.

Listing 6.2 Between two events the capture stage event `onMouseOverCapture` will happen first (ch06/mouse-capture/jsx/mouse.jsx).

```
class Mouse extends React.Component {
  render() {
    return <div>
      <div
        style={{border: '1px solid red'}}
        onMouseOverCapture={({event})=>{
          console.log('mouse over on capture event')
          console.dir(event, this)}.bind(this)}
        onMouseOver={({event})=>{
          console.log('mouse over on bubbling event')
          console.dir(event, this)}.bind(this)} >
        Open DevTools and move your mouse cursor over here
      </div>
    </div>
  }
}
```

The container will have a red border with 1px width and some text inside of it as shown in Figure 6-3 so we know where to hover the cursor. Each mouseover event will log what type of event that is as well as the event object (hidden under the Proxy in DevTools on Figure 6-3 due to the usage of `console.dir()`).

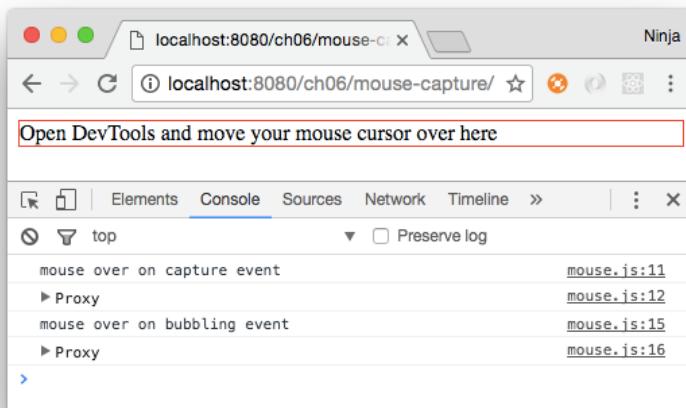


Figure 6.4 Capture event is happening first before the regular event

Not surprisingly, the capture event will log first. Developers can use this behavior to stop propagation and set priorities between events.

It's important to understand how React is implementing events because events are the cornerstone of UIs. The next chapter will dive even deeper into React events.

6.1.2 React Events Under The Hood

The way events work in React is different from jQuery or plain JavaScript which typically will put the event listener directly on the DOM node. When developers put events directly on nodes, there might be problems with removing and adding the events during the UI lifecycle. For example, you have a list of accounts and each has an edit button which brings a popup.

```
<ul id="account-list">
  <li id="account-1">Account #1</li>
  <li id="account-2">Account #2</li>
  <li id="account-3">Account #3</li>
  <li id="account-4">Account #4</li>
  <li id="account-5">Account #5</li>
  <li id="account-6">Account #6</li>
</ul>
```

If accounts are removed or added frequently to the list, then managing events become difficult. A better approach is to have one event listener on a parent (`account-list`), and listen for bubbled up events (an event will bubble up higher up the DOM tree if nothing caught it on a lower level). Internally, React keeps track of the events attached to the higher element

and the target elements in as a mapping. This allow React to trace back the target from the parent (`document`) as shown in Figure 6.5.

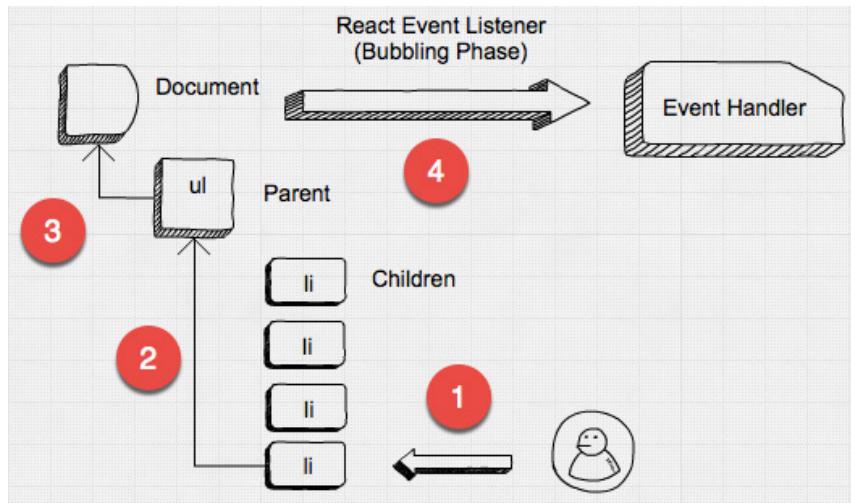


Figure 6.5 Bubbling of a DOM event (1) to its ancestors (2-3) where it's captured by a regular (bubbling stage) React event listener (4), because in React events are captured at the root (Document)

Let's see how this event delegation to the parent looks in action on the example of the Mouse component (Listing 6-2). If you remember we had a `<div>` element with the mouse over React event. If you open Chrome DevTools or Firebug and point to the `data-reactroot` element in the (Inspect Element in the context menu) Elements or Inspector, then you can get the `<div>` in the console by typing `$0` and enter. It's a nice little trick that DevTools has. See Figure 6-5 for visuals.

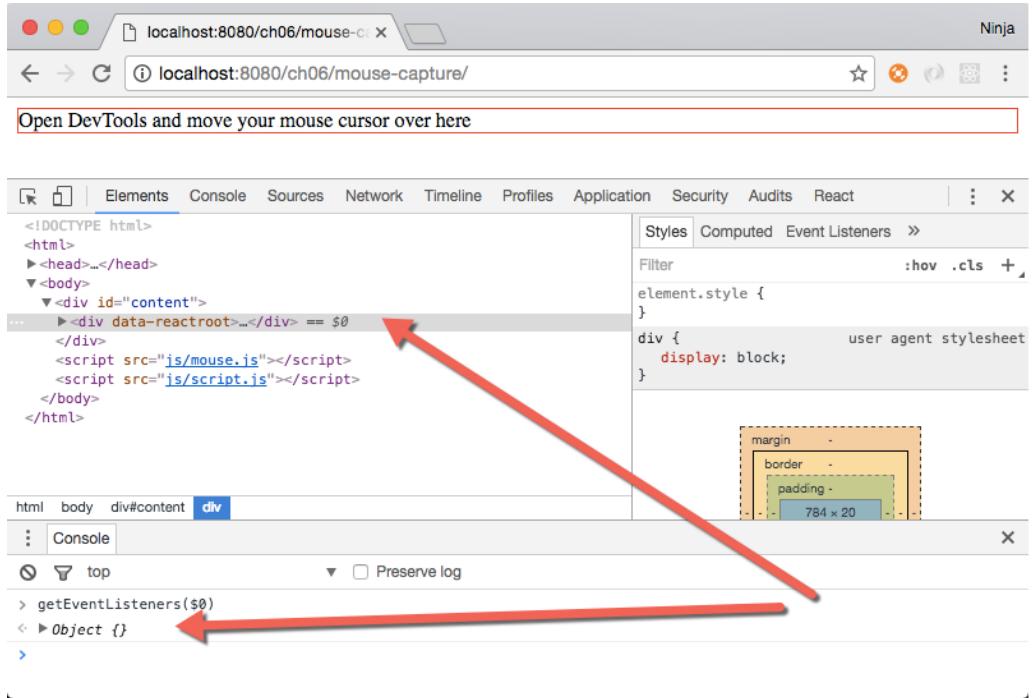


Figure 6.6 .Inspecting events on the div element of which there are none]

Interestingly enough, this DOM node **won't have any event listeners**. The `$0` is `reactroot` element. Therefore, we can check what events attached to this particular element (DOM node) by using global `getEventListeners()` method in the console of DevTools (as shown in Figure 6.6):

```
getEventListeners($0)
```

The result will be an empty `object {}`. So React didn't attach event listeners to the `reactroot` node `<div>`. Where did it go because we are clearly see that the event is being captured!?

Feel free to repeat the procedure with `<div id="content">` or maybe with the red border `<div>` element (child of `reactroot`). For each currently selected element in the Elements tab, the `$0` will be the selected element, so you simply select a new element and repeat `getEventListeners($0)`. Still nothing?

Okay. Let's examine events on `document` by calling this code from the console:

```
getEventListeners(document)
```

Boom! We have our event: `Object {mouseover: Array[1]}` as shown in Figure 6-6. Now, we know that React attached the event listener to the *ultimate parent* a.k.a. `document`, not to the individual node like `<div>` or even an element with the `data-reactroot` attribute.

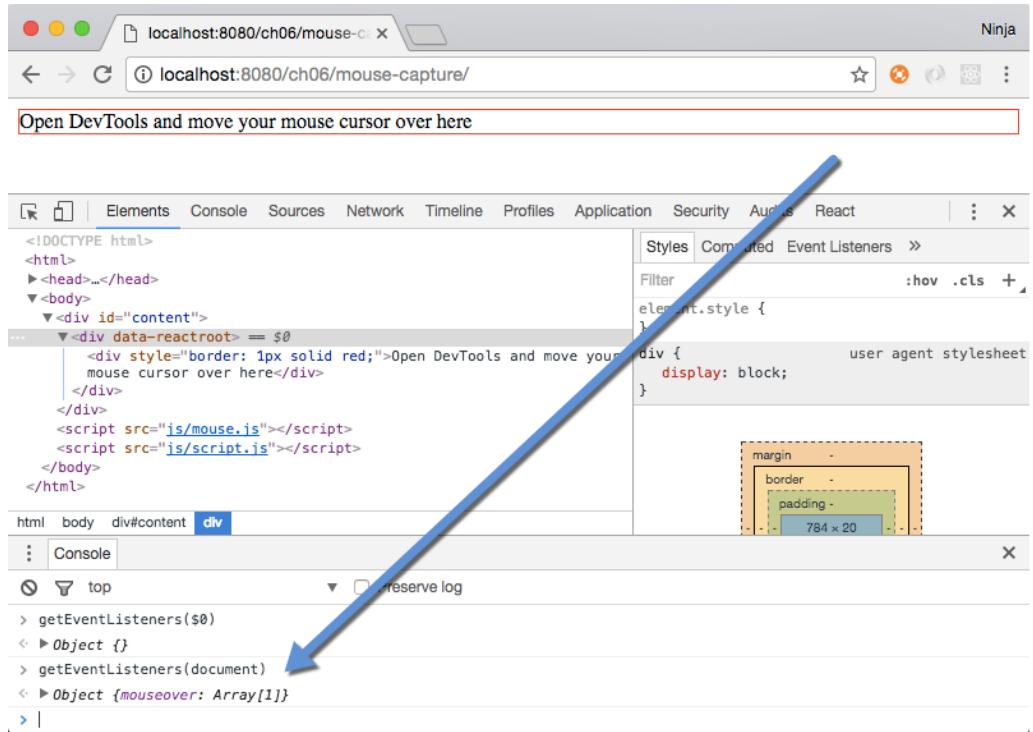


Figure 6.7 Inspecting events on the `div` element of which there are none

Next, we can remove this event by invoking the next line in the console:

```
getEventListeners(document).mouseover[0].remove()
```

Now, the message "mouse is over" won't appear when we move the cursor. The event listener which was attached to `document` is gone. This illustrates that React attaches events to the `document`, and not each element. It allows React to be faster especially when working with lists. This is contrary to how jQuery works, because with that library events are attached to individual elements. So kudos to React for thinking about performance.

If you have other elements with the same type of event, e.g., two `mouseover`s, then they would be attached at one event and handled by React's internal mapping to the right child (target element):

```

> getEventListeners(document)
< ▼ Object {click: Array[1], mouseover: Array[1]} ⓘ
  ▼ click: Array[1]
    ► 0: Object
      length: 1
    ► __proto__: Array[0]
  ▼ mouseover: Array[1]
    ► 0: Object
      length: 1
    ► __proto__: Array[0]
>

```

Figure 6.8 React will "reuse" the event listeners on the root so you'll see only one of each type even when you have 1+ elements with mouseover

Speaking of target elements, we can get information about the target node (where the event originated) from the event object.

6.1.3 Working with React Event Object SyntheticEvent

Browsers can differ in their implementations of [the W3C specification](#). When working with DOM event, the event object passed to the event handler might have different properties and methods. This might lead to cross-browser issues when writing event handling code. For example to get the target element in IE versions 8, you would need to access `event.srcElement` while in Chrome, Safari and Firefox: `event.target`.

```
var target = event.target || event.srcElement
console.log(target.value)
```

Of course, things are better in terms of cross-browser issues in 2016 than in 2006, but still. Do you want to spend time reading specs and debugging issues due to obscure discrepancies between browser implementations? I don't.

Cross-browser issues are not good for developers, because they require more manual if/else code and more testing in different browsers. React has a solution which is a wrapper around browsers' native events. This makes the event consistent with the W3C specification irrespective of the browser you run your pages on.

Under the hood, React is using its own special class for events called synthetic event (`SyntheticEvent`). The instances of this `SyntheticEvent` class will be passed to the event handler. For example to get access to this synthetic event object, we can add an argument `event` to the event handler function as shown in Listing 6-3.

Listing 6.3 Event handlers receive a Synthetic Event (ch06/mouse/jsx/mouse.jsx)

```
class Mouse extends React.Component {
  render() {
    return <div>
      <div style={{border: '1px solid red'}}
        onMouseOver={((event)=>{           ①
          console.log('mouse is over with event')
          console.dir(event)}))} >          ②
          Open DevTools and move your mouse cursor over here
        </div>
      </div>
    }
}
```

- ① Define an `event` argument
 ② Access `SyntheticEvent` object to log interactively (`dir`)

This way, we will get the event object outputted in the console as depicted in Figure 6-8.

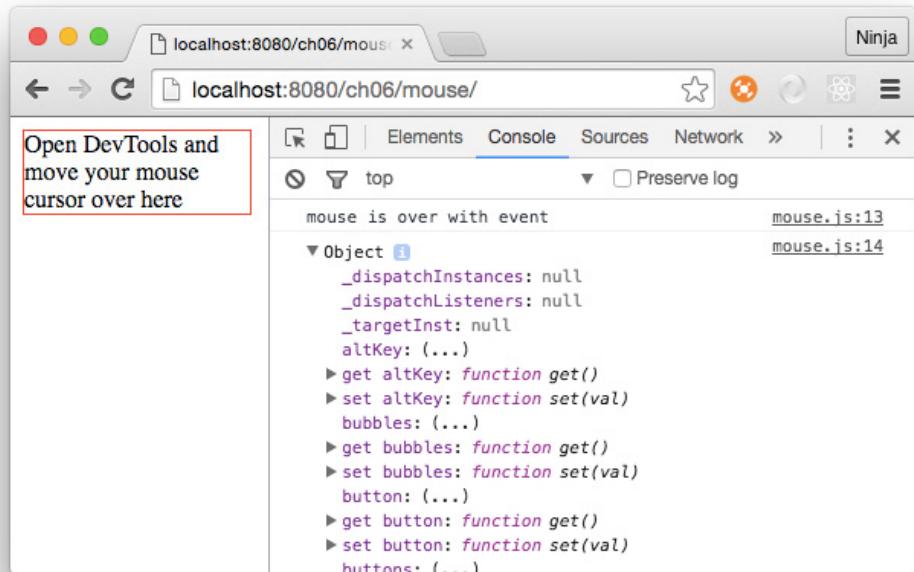


Figure 6.9 Hovering mouse over the box will print the event object in the DevTools console

As we've seen before, we can move the event handler code into a component method or a stand alone function. For example, we can create a method `handleMouseOver` using

ES6+/ES2015+ syntax and refer to it from the return of `render()` with `{this.handleMouseOver}`. The name `handleMouseOver` is totally arbitrary (unlike names of lifecycle events which we covered in the previous chapter), and doesn't have to follow any convention as long as you and your team understands it. Most of the times, React developers prefix an event handler with `handle` to distinguish it from a regular class method as well as include either an event name (`mouseOver`) or a name of the operation (`save`).

```
class Mouse extends React.Component {
  handleMouseOver(event) {
    console.log('mouse is over with event')
    console.dir(event.target)
  }
  render(){
    return <div>
      <div
        style={{border: '1px solid red'}}
        onMouseOver={this.handleMouseOver.bind(this)} >
        Open DevTools and move your mouse cursor over here
      </div>
    </div>
  }
}
```

The event will have the same properties and methods as most native browser events, for example `stopPropagation()`, `preventDefault()`, `target` or `currentTarget`. If you can't find a native property or a method (a full list of React v15.0.2 `SyntheticEvent` interface is shown in Listing 6-4), you can access a native browser event with `nativeEvent`. For example,

```
event.nativeEvent
```

Table 6.2 Some of the attributes and methods of React's SyntheticEvent

- `currentTarget`: `DOMEVENTARGET` of the element which is capturing the event (could be a parent of target or a target itself)
- `target`: `DOMEVENTTARGET`, the element where the event was triggered
- `nativeEvent`: `DOMEVENT`, native browser event object
- `preventDefault()`: prevents default behavior, e.g., link or a form submit button
- `isDefaultPrevented()`: boolean, true if default behavior was prevented
- `stopPropagation()`: stop propagation of the event
- `isPropagationStopped()`: boolean, true if propagation was stopped
- `type`: string, tag name
- `persist()`: removes the synthetic event from the pool and allows references to the event to be retained by user code
- `isPersistent`: boolean, true if `SyntheticEvent` was taken out of the pool

The aforementioned `target` property of the event object will have the DOM Node of the object on which the event happened and not where it was captured as with `currentTarget`

([Event.target on MDN](#)). Most often when we build UIs, in addition to capturing we need to get the text of an input field. You can get it from `event.target.value`.

The synthetic event will be nullified (meaning it will become unavailable) once the event handler is done. So you cannot simply use the same event reference in a variable to access it later or to access it asynchronously (in the future) in some callback function. For example, if I try to save the reference of the event object in a global `e` as shown in Listing 6-5.

Listing 6.4 Without `event.persist()` the Synthetic event will be nullified and reused
(`ch06/mouse-event/jsx/mouse.jsx`).

```
class Mouse extends React.Component {
  handleMouseOver(event) {
    console.log('mouse is over with event')
    window.e = event // Anti-pattern
    console.dir(event.target)
    setTimeout(()=>{
      console.table(event.target)
      console.table(window.e.target)
    }, 2345)
  }
  render() {
    return <div>
      <div
        style={{border: '1px solid red'}}
        onMouseOver={this.handleMouseOver.bind(this)}>
        Open DevTools and move your mouse cursor over here
      </div>
    </div>
  }
}
```

- 1 Use the `event` object and its attributes inside of the method
- 2 By default, you cannot use `event` in an asynchronous callback or by calling `window.e`

I'll get a warning saying that React is reusing synthetic event for performance reasons:

This synthetic event is reused for performance reasons. If you're seeing this, you're accessing the property 'target' on a released/nullified synthetic event. This is set to null.

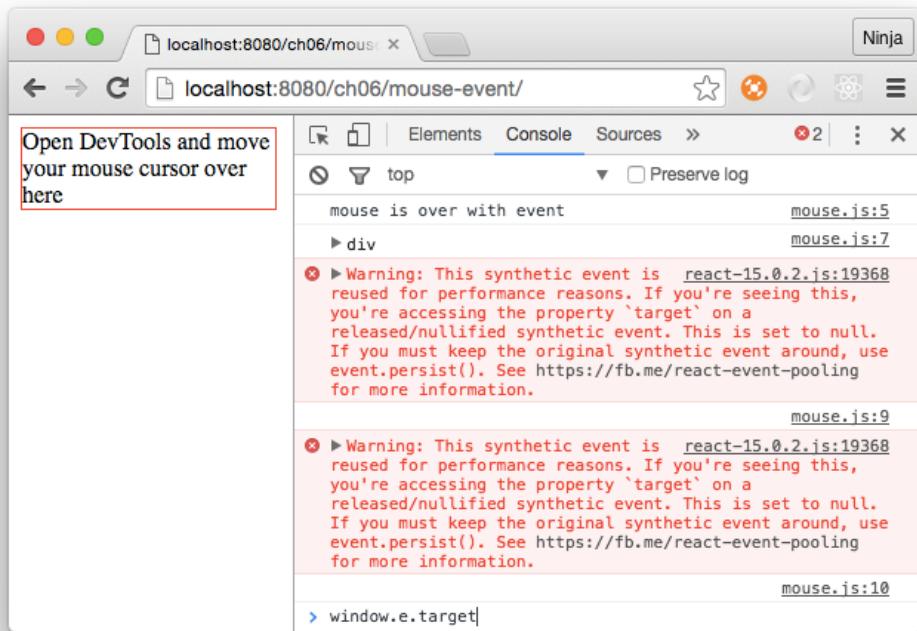


Figure 6.10 Saving synthetic event object for later use won't be possible by default hence the warning

If we need to keep the synthetic event after the event handler is over, use the `event.persist()` method. When you apply it, the event object won't be reused and nullified.

We've seen that React will even synthesize (or normalize) them for you, meaning that React will create a cross-browser wrapper around the native event objects. The benefit of this is that events work identically in virtually all browsers.

In most cases, you have all the native methods on the React event, including `event.stopPropagation()` and `event.preventDefault()`, and if you still need to access a native event, it's in the `event.nativeEvent` property of the synthetic event object. Obviously, developers who work with native events directly, we'll need to know and work with any cross-browser differences they might encounter.

6.1.4 Using Events and State

Using states with events, or to put it differently being able to change a component's state in response to an event, will give us interactive UIs which respond to user actions. This is

getting *really fun* because basically we'll be able to capture any events and change views based on these events and our app logic. This will make our components self-contained because they won't need any external code or representation.

For example, we can implement a button with a label that has a counter (number 0). With each click on the button, we'll increment the number which is show on a button (1,2,3, etc.).

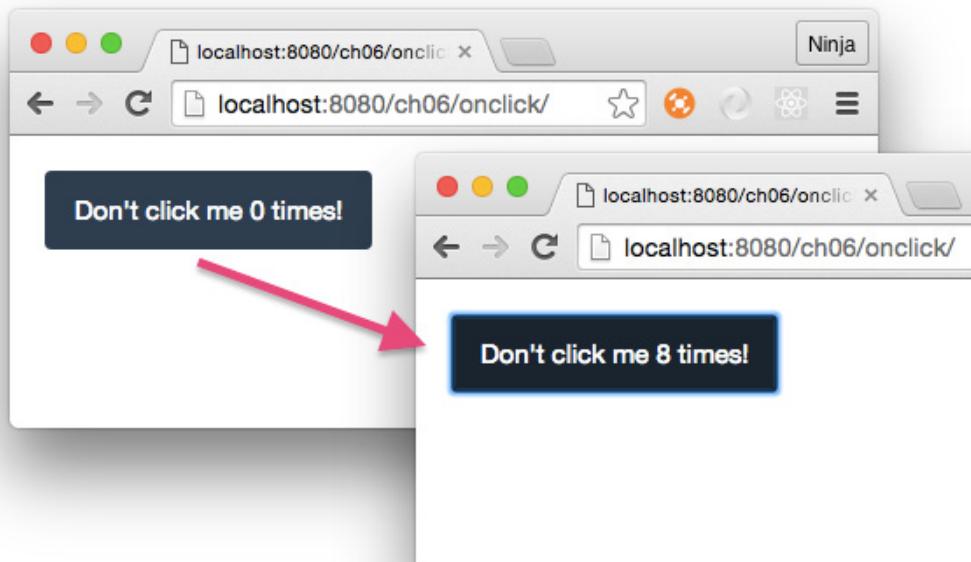


Figure 6.11 Clicking on a button increments the counter with initial value 0

We start by implementing:

1. `constructor(): this.state equals to {counter: 0}` because we must set the counter to 0 before we can use it in the view.
2. `handleClick(): Event handler which increments the counter`
3. `render(): Render methods which returns the button JSX`

The `click` method is not unlike any other React component method. Remember `getUrl` in chapter 3 or `handleMouseOver` in this chapter? This component method is declared similarly except that we have to manually bind the `this` context. The `handleClick` method will set the state `counter` to the current value of `counter` incremented by one.

Listing 6.5 Updating state as a result of user click action

(ch06/onclick/jsx/content.jsx)

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.state = {counter: 0}          ①
  }
  handleClick(event) {
    this.setState({counter: ++this.state.counter}) ②
  }
  render() {
    return (
      <div>
        <button
          onClick={this.handleClick.bind(this)}           ③
          className="btn btn-primary">
          Dont click me {this.state.counter} times! ④
        </button>
      </div>
    )
  }
}
```

- ① Set the initial state counter to 0
- ② Increase counter value by one
- ③ Attach event listener onClick to trigger handleClick
- ④ Display value of counter state

Invocation vs. Definition

Just to remind: did you notice that although `this.handleClick` is a method in Listing 6.6, we didn't invoke it in JSX when we assigned it to `<button onClick=!` In other words, there are no parentheses () after `this.handleClick` inside of the curly braces. That's because we need to pass a function definition, not invoke it. The functions are first-class citizens in JavaScript, and in this case, we pass the function definition as a value to the `onClick`attribute.

The `bind()` is invoked because it allows to use the proper value of `this` but `bind()` also return a function definition.

More over as has been noted before, the `onClick` is not a real HTML attribute, but syntactically it looks just like any other JSX declaration (for example, `className={btnClassName}` or `href={this.props.url}`).

When you click on the button, you'll see the counter increment with each click. Figure 6-10 shows that I made eight clicks and the counter is now on 8 but initially was on 0. Brilliant, isn't it?

Analogous to `onClick` or `onMouseOver`, you can use any DOM events supported by React. In essence, we defined the view and an event handler which changes the state. We didn't modify imperatively the representation. This is the power of declarative style!

Proceed to the next section, where I'll teach you how to pass event handlers and other objects to children elements.

6.1.5 Passing Event Handlers as Properties

Consider this scenario: there's a button which is a stateless component. All it has is some styling. How do you attach an event listener so this button can trigger some code?

Let's go back to properties for a moment. Properties are immutable, which means they don't change. They are passed by parent components to their children. Because functions are first-class citizens in JavaScript, we can have a property in a child element that is a function and use it as an event handler.

The solution to the problem outlined earlier where we need to trigger an event from a stateless component is to pass the event handler as a property to this stateless component and use it in it.

For example, let's break down the functionality of the previous example in two components: `ClickCounterButton` and `Content`. The first will be dumb (stateless) and the second smart (statefull).

Presentational/Dumb vs. Container/Smart Components

Dumb or smart components sometimes called presentational and container components respectively. These dichotomy is related to stateless and statefull but not exactly the same 100% of the time.

Most of the time, presentational components won't have states and be stateless or function components. That's not always the case, because developers might need to have some state which relates to the presentation.

Presentational/dumb components often use `this.props.children` and render DOM elements.

On the other hand, container/smart components describe things how things work without the DOM elements, have states and typically use higher-order component pattern and connect to data sources.

Using a combination of dumb and smart component is the best practice because it keeps things clean and allows for better separation of concerns.

When we run the code, we'll see the counter increasing with each click. Visually, nothing has changed from the previous example with the button and the counter, but internally, we moved the logic from the render component into its parent.

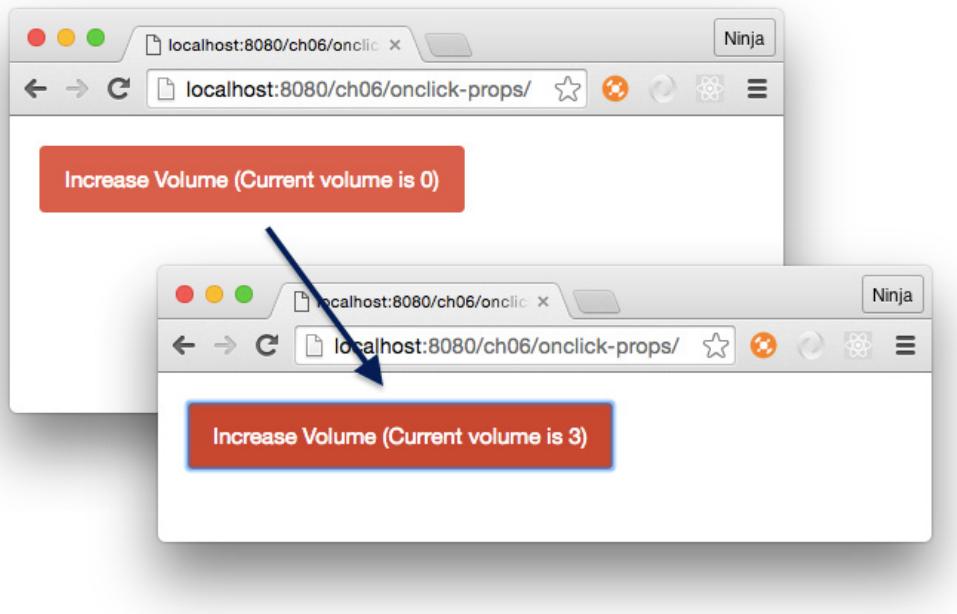


Figure 6.12 Passing event handler as a property to button (presentational component) will enable the increments of counter in the button label which is also a property

The `ClickCounterButton` doesn't have its own `onClick` event handler (that is, no `this.handler` or `this.handleClick`). It uses the handler passed down to it by its parent in a property `this.props.handler`. Generally speaking, using this approach is beneficial to just handling events in a button, because with this approach the button is a stateless presentational/dumb component. Developers can reuse this button in other UIs. Now, take a look at the code of this presentational component which will render the button (the parent `Content` which renders this element will be shown later):

Listing 6.6 Creating stateless component which uses an event handler from Content
`(ch06/onclick-props/jsx/click-counter-button.jsx)`

```
class ClickCounterButton extends React.Component {
  render() {
    return <button
      onClick={this.props.handler}
      className="btn btn-danger">
      Increase Volume (Current volume is {this.props.counter})
    </button>
  }
}
```

In this case, the counter is also a property that is rendered with `{this.props.counter}`. Supplying properties to children like `clickCounterButton` is straightforward if you remember examples from chapter 2. You use the standard attribute syntax: `name=VALUE`.

For example, to provide `counter` and `handler` properties to the `ClickCounterButton` component, specify the attributes in the JSX declaration of the parent's `render` parameter (the parent here is `Content`):

```
<div>
  <ClickCounterButton
    counter={this.state.counter}
    handler={this.handleClick}/>
</div>
```

To make sure we're on the same page, `counter` in `ClickCounterButton` is a property and thus *immutable*; but in the parent `Content`, it's a state and thus *mutable*. (For a refresher on props vs. state, refer to chapter 4.) Obviously, the names can differ. You don't have to keep the names the same when you pass props to children. However, for me personally keeping the same name helps to understand that the data is related between different components.

So what is happening? The initial `counter` (the state) is set in the parent `Content` to 0 (zero). The event handler is defined in the parent as well. Therefore, the child (`ClickCounterButton`) triggers the event on a parent. The entire code of the parent component `Content` with `constructor()` and `handleClick()` looks like this:

Listing 6.7 Passing event handler as a property to a stateless component ClickCounterButton (ch06/onclick-props/jsx/content.jsx)

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)      ①
    this.state = {counter: 0}
  }
  handleClick(event) {
    this.setState({counter: ++this.state.counter})
  }
  render() {
    return (
      <div>
        <ClickCounterButton
          counter={this.state.counter}
          handler={this.handleClick}/>
      </div>
    )
  }
}
```

① Bind context in `constructor` so we can use `this.setState()` which refers to the instance of this class `'Content'`

In JavaScript, functions are first-class citizens, and we can pass them as variables or properties. Therefore, fundamentally there should be no big surprises here. Now the question arises, where to put the logic such as event handlers (in a child or parent)?

6.1.6 Exchanging Data Between Components

In the previous example, the click event handler was in the parent element. We can put the event handler on the child itself, but using the parent allows us to exchange information among children components.

Let's take a button, but this time we remove from `render()` the counter value (numbers 1,2,3, etc.). The components are single-minded granular pieces of representation (remember?), so the counter will be in another component `Counter`. Thus, we'll have three components overall: `ClickCounterButton`, `Content` and `Counter`.

As you can observe in Figure 6.13 , now there are two components: button and the text below it. Each of them have properties that are states in the parent `Content`. The challenge in contrast to the previous example (Figure 6.12) and this one is that we need to communicate between the button and the text to keep the score on clicks. In other words, `ClickCounterButton` and `Counter` need to talk with each other. They'll do it via `Content`, *not directly* (directly would be a bad pattern because it will create tight coupling).

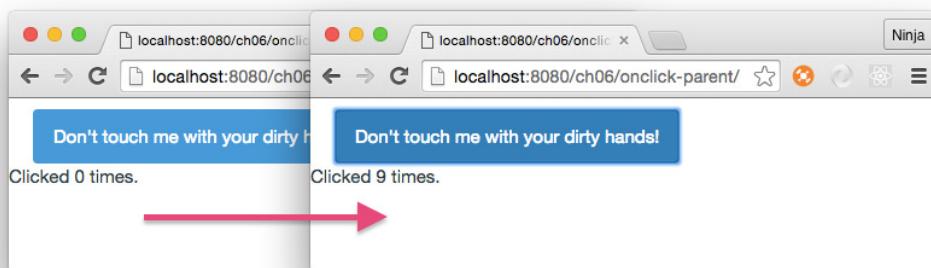


Figure 6.13 Splitting state and working with two stateless child components (by allowing them to exchange data via parent): one for counter (text) and another for the button

`ClickCounterButton` would look like stateless just as most of React components should look like—no thrills, just props and JSX:

```
class ClickCounterButton extends React.Component {
  render() {
    return <button
      onClick={this.props.handler}
      className="btn btn-info">
      Dont touch me with your dirty hands!
    
```

```

        </button>
    }
}

```

The following shows a new component, `Counter`, which displays the `value` property that is our counter:

```

class Counter extends React.Component {
  render() {
    return <span>Clicked {this.props.value} times.</span>
  }
}

```

Finally, we get to the parent component that provides the properties--one of which is the event handler and the other is a counter. We need to update the `render` parameter accordingly, but the rest of the code remains intact:

Listing 6.8 Passing event handler and state to two different components (ch06/onclick-parent/jsx/content.jsx)

```

class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
    this.state = {counter: 0}
  }
  handleClick(event) {
    this.setState({counter: ++this.state.counter})
  }
  render() {
    return (
      <div>
        <ClickCounterButton handler={this.handleClick}/>
        <br/>
        <Counter value={this.state.counter}/>
      </div>
    )
  }
}

```

To answer the initial question on where to put the event-handling logic, the rule of thumb is to put it in the parent or wrapper component if you need the interaction between children components. If the event concerns only the child components, there's no need to pollute the components higher up the parent chain with event-handling methods.

6.2 Responding to DOM Events Not Supported by React

In table 6-1, you saw the list of event supported by React. You might eventually wonder, but what about DOM events not supported by React? For example, you're tasked with creating scalable UI which needs to become bigger or smaller depending on a window size (`resize` event)... But this event is not supported! There's a way to capture `resize` and any other event. And you already know the React feature to implement it—lifecycle events.

We'll implement radio buttons. As some of you know, standard HTML radio button element scale (become larger or smaller) badly and not consistently across browsers. For this reason, back when I worked at DocuSign, I implemented [scalable CSS radio buttons](#) to replace standard HTML radio inputs. That was done in jQuery. These CSS buttons can be scaled via jQuery by manipulating their CSS. Let's see how we can create scalable radio UI in React. We will make the same CSS buttons scale with React when we resize the screen as shown in Figure 6.14

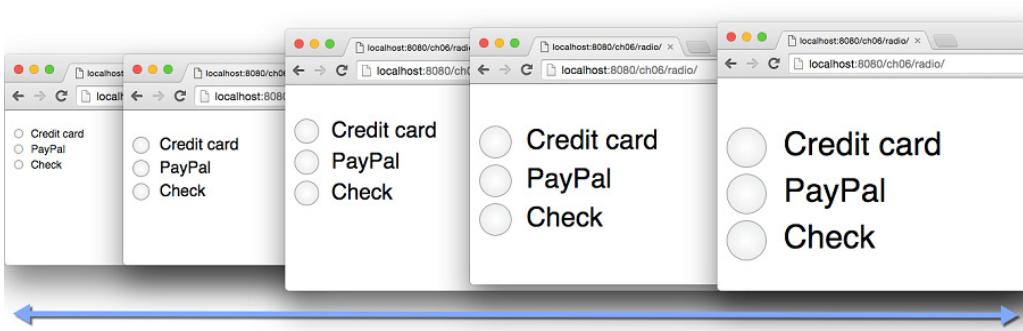


Figure 6.14Scalable CSS buttons managed by React which is listening to window resize. As the window size increases, so too does the font

The `resize` event is not supported by React; adding it to the element as shown below won't work.

```
...
  render() {
    return <div>
      <div onResize={this.handleResize}>
        ...
      </div>
    </div>
  }
}
```

There is a simple way to attach not supported events like `resize` and pretty much any custom elements developers need to support. The solution is to use React component lifecycle event! Examine the following example which adds `resize` event listeners to `window` in `componentDidMount()`, then remove the same event listener in `componentDidUnmount()` to make sure there's nothing left after this component is gone from the DOM.

The helper function `getStyle()` is there to abstract some of the styling because there's repetition in the CSS such as `top`, `bottom`, `left`, and `right` but with different values which will depend on the width of the window. Hence, `getStyle` take the value and the multiplier `m`, and returns pixels. (Numbers in React's CSS become pixels.)

Listing 6.9 By using component lifecycle event, developers can listen to any DOM events
(ch06/radio/jsx/radio.jsx)

```
class Radio extends React.Component {
  constructor(props) {
    super(props)
    this.handleResize = this.handleResize.bind(this)
    let order = props.order
    let i = 1
    this.state = {  
      outerStyle: this.getStyle(4, i),  
      innerStyle: this.getStyle(1, i),  
      selectedStyle: this.getStyle(2, i),  
      taggerStyle: {top: order*20, width: 25, height: 25}  
    }
  }
  getStyle(i, m) {  
    let value = i*m  
    return {  
      top: value,  
      bottom: value,  
      left: value,  
      right: value,  
    }
  }
  componentDidMount() {  
    window.addEventListener('resize', this.handleResize)  
  }  
  componentWillUnmount() {  
    window.removeEventListener('resize', this.handleResize)  
  }
  handleResize(event) {  
    let w = 1+ Math.round(window.innerWidth / 300)  
    this.setState({  
      taggerStyle: {top: this.props.order*w*10, width: w*10, height: w*10},  
      textStyle: {left: w*13, fontSize: 7*w}  
    })
  }
  ...
}
```

- ① Save styles in the state
- ② Use a function to create various styles from `width` (will change later) and a multiplier
- ③ Attach a non supported event listener to window
- ④ Remove the non supported event listener from window
- ⑤ Implement magic function to handle radio button resize according to the new screen size

The rest of the code is easy. All we need to do is implement `render()` method which uses the states and properties to render four `<div>` elements each with a special styles which we defined before in `constructor()`.

```
...
render() {
  return <div>
    <div className="radio-tagger" style={this.state.taggerStyle}>
      <input type="radio" name={this.props.name} id={this.props.id}>
    </input>
  </div>
}
```

```

        <label htmlFor={this.props.id}>
          <div className="radio-text" style={this.state.textStyle}>{this.props.label}</div>
          <div className="radio-outer" style={this.state.outerStyle}>
            <div className="radio-inner" style={this.state.innerStyle}>
              <div className="radio-selected" style={this.state.selectedStyle}>
                </div>
              </div>
            </div>
          </label>
        </div>
      </div>
    }
}

```

Therefore, by using Lifecycle Events in your components, you can create custom event listeners. The way we do it is by using `window`. This is similar how React's event listeners work (Figure 6-TK) because React attaches them to `window` as well. Don't forget to remove the custom event listener on the unmount event though.¹

6.3 Integrating React with Other Libraries: jQuery UI Events

As shown above, React provides standard DOM events, but what if you need to integrate with another library which uses (triggers or listens to) non-standard events? For example, you have jQuery components and they are using `slide` (as in the slider control element). You want to integrate a React widget with into your jQuery app. This is how you can attach any DOM events not provided by React. We'll be using component lifecycle events `componentDidMount` and `componentWillUnmount`.

As you might guess from the choice of the lifecycle events, we'll be attaching an event listener when component is mounted and detaching when it's unmounted, so it's not causing conflicts or performance issues by handing in there as an orphan. (Orphaned event handlers are ones without DOM nodes and they can cause memory leaks.)

For example, you are working at a music streaming company and you are tasked with implementing a volume controls on the new version of the web player (think Spotify or iTunes). You need to add a label and buttons in addition to the legacy jQuery slider ([jQuery UI Slider](#)).

So you would like to implement a label with a numeric value, and two buttons to decrease value and increase it by one. The idea is to make these pieces work as one: when users slide the pin (square looking peg on a slider) left and right, the numeric value and the values on the buttons should change accordingly. In the same fashion, users should be able press either of

¹ If you interested in the scalable buttons and their non-React implementation (jQuery for example), I wrote [a separate blog post](#) and created [an online demo](#). Of course, you can find the React implementation in the source code of the book. This brings us to the topic of integrating React with other UI libraries like jQuery.

the buttons and the slider pin should move left or right correspondingly. In essence, users want to have not just a slider, but a widget which looks like the one shown on Figure 6-14.

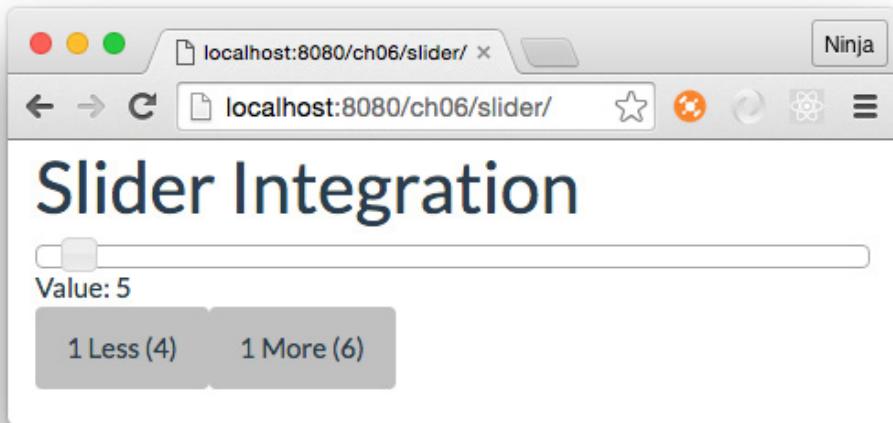


Figure 6.15 React (buttons and text "Value...") can be integrated with other libraries like jQuery Slider to respond to each other

6.3.1 Integrating Buttons

You have at least two options when it comes to integration. One would be to attach an event listener on the jQuery slider itself and trigger a method on a React component (`handleSlide`) when there's a `slide` event on the jQuery slider (meaning there's a change in values). With every `slide` and `change` in values, we'll update the state (`sliderValue`). `SliderButtons` implements this approach as shown in Listing 6-11.

This approach is tightly coupling. The other more loosely coupled option will be discussed after we implement this approach.

Listing 6.10 Integrating with jQuery plugin via its events (ch06/slider/jsx/slider-buttons.jsx)

```
class SliderButtons extends React.Component {
  constructor(props) {
    super(props)
    this.state = {sliderValue: 0} ①
  }
  handleSlide(event, ui) {
    this.setState({sliderValue: ui.value}) ②
  }
}
```

```

    }
    handleChange(value) {
      return ()=> {
        $('#slider').slider('value', this.state.sliderValue + value)
        this.setState({sliderValue: this.state.sliderValue + value}) ⑤
      }
    }
    componentDidMount() {
      $('#slider').on('slide', this.handleSlide) ⑥
    }
    componentWillUnmount() {
      $('#slider').off('slide', this.handleSlide) ⑦
    }
    ...
  })
}

① Set initial value to 0
② jQuery will pass two arguments: jQuery event and the ui object with the current value which we use to update the state
③ Define method to update slider when button is pressed
④ Use Factory Function pattern for -1 and +1 buttons
⑤ Use jQuery method to set the new value
⑥ Update state to a new value
⑦ Remove the event listener on unmount

```

The render method of `SliderButtons` will have two buttons with `onClick` events, dynamic disabled attribute so we don't set values below 0 (Figure 6-15) and above 100, and Twitter Bootstrap classes for buttons.

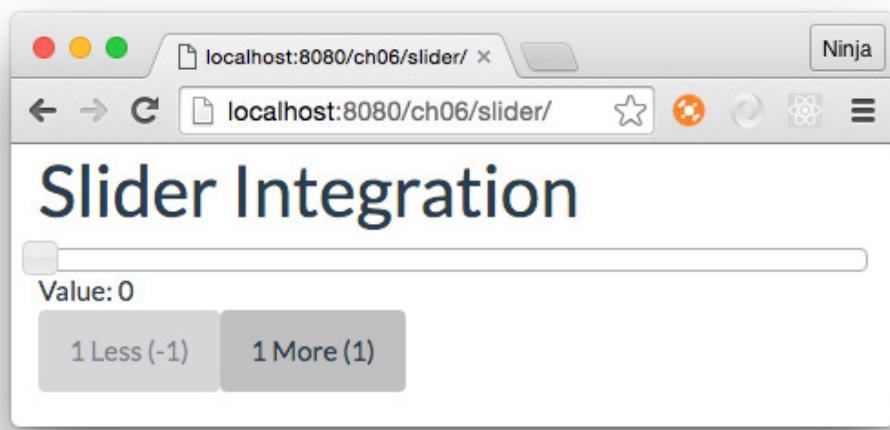


Figure 6.16 Programmatically disabled 1 Less button to prevent negative values

Now the code for `render()` of the slider buttons!

Listing 6.11 Rendering Slider buttons (ch06/slider/jsx/slider-buttons.jsx)

```
...
  render() {
    return <div>
      <button disabled={({this.state.sliderValue<1)?true:false}
        className="btn default-btn"
        onClick={this.handleChange(-1)}> ①
        1 Less ({this.state.sliderValue-1})
      </button>
      <button disabled={({this.state.sliderValue>99) ? true : false} ②
        className="btn default-btn" ③
        onClick={this.handleChange(1)}>
        1 More ({this.state.sliderValue+1}) ④
      </button>
    </div>
  }
}
```

- ① Invoke `this.handleChange` with `-1` to get a function from the function factory
- ② Use ternary operator to disable buttons when less than 1 and more than 99
- ③ Apply Twitter Bootstrap classes using `className`
- ④ Render the next value for the slider as buttons labels

6.3.2 Integrating Label

At the same time, we can decouple jQuery and React by using another object to catch events. This is loosely coupled patterns and often more preferred, because it helps to avoid extra dependencies. To put it differently, different components don't need to know about the details of each other implementation. That is React component `SliderValue` won't know about how to call jQuery Slider. This is good because later you can easier change Slider to Slider 2.0 with a different interface.

The way we can implement this is by dispatching events to `window` in jQuery events, and defining event listeners for `window` in React component lifecycle methods. Examine the following code for `SliderValue`.

Listing 6.12 Integrating with jQuery plugin via window (ch06/slider/jsx/slider-value.jsx)

```
class SliderValue extends React.Component {
  constructor(props) {
    super(props)
    this.handleSlide = this.handleSlide.bind(this)
    this.state = {sliderValue: 0}
  }
  handleSlide(event) {
    this.setState({sliderValue: event.detail.ui.value})
  }
  componentDidMount() {
    window.addEventListener('slide', this.handleSlide) ①
  }
}
```

```

    }
    componentWillUnmount() {
      window.removeEventListener('slide', this.handleSlide) ②
    }
    render() {
      return <div className="" >
        Value: {this.state.sliderValue}
      </div>
    }
  }
}

```

- ① Use `window` and `slide` event on it
- ② Remove `slide` from `window` to avoid orphan event handlers

In addition to that, we also need to dispatch a custom event. In the first approach (`SliderButtons`) we didn't need to do, it because we leveraged existing plugin events. In this implementation, we have to create and even and dispatch it to `window` with data. You can implement the dispatchers of the `slide`custom event along side the code which create the jQuery slider object which is a script tag in `index.html`.

Listing 6.13 Setting up event listeners on a jQuery UI plugin (ch06/slider/index.html)

```

let handleChange = (e, ui)=>{ ①
  var slideEvent = new CustomEvent('slide', { ②
    detail: {ui: ui, jQueryEvent: e} ③
  })
  window.dispatchEvent(slideEvent) ④
}
$('#slider').slider({ ⑤
  'change': handleChange,
  'slide': handleChange
}) ⑥

```

- ① Create an event handler for jQuery Slider which will dispatch custom events
- ② Create custom event
- ③ Pass jQuery data which has current slider value
- ④ Dispatch event to `window`
- ⑤ Create slider using container with ID `slider`
- ⑥ Attach event listeners on `change` (programmatic) and `slide` (UI)

Once you run the code, both buttons and the value label will work seamlessly. We used two approaches, one is loosely coupled and another tightly coupled. The later is shorter to implement but the former is better because it will allow to modify code more easily in the future.

As can be observed from this integration, React can work nicely with other libraries by listening to events in its lifecycle method `componentDidMount`.

6.4 Quiz

1. Select the right syntax for the event declaration: `onClick=this.doStuff`, `onClick={this.doStuff}`, `onClick="this.doStuff"`, `onClick={this.doStuff} or onClick={this.doStuff()}`.
2. `componentDidMount()` won't be triggered during server-side rendering of the React component on which it's declared. True or false?
3. One of the ways to exchange information among children components is to move the object to the parent of the children. True or false?
4. Developers can use `event.target` asynchronously and outside of the event handler by default. True or false?
5. Integration with third-party libraries and events not supported by React is done by setting up event listeners in the component lifecycle events. True or false?

6.5 Summary

User input by events and being able to respond to them is one of the most important aspect of building User Interfaces. Without it, our UIs will be static. React allows to bundle up the logic of processing events together with the UI view representation (React component classes) — component-based architecture!

- `onClick` is for capturing mouse and trackpad clicks
- The JSX syntax for event listeners is ``
- Bind event handlers with `bind()` in `constructor()` or in JSX if you want to use `this` in the event handler as the value of the component class instance
- `componentDidMount` is triggered only on the browser, and `componentWillMount` is triggered both on the browser and server.
- React supports most of the standard HTML DOM events by providing and using Synthetic Event objects.
- `componentDidMount()` and `componentWillUnmount()` can be used to integrate React with other frameworks and events not supported by React

In the next chapter, we will continue to work with events and states, and see how we might want to utilize them when it comes to web forms and its elements—one of the most important elements to capture user input.

6.6 Quiz Answers

1. `onClick={this.doStuff}` is correct because only the function definition must be passed to `onClick`, not the invocation (or the result of the invocation to be precise)
2. True, `componentDidMount()` is only executed for browser React. That's why developers use it for AJAX/XHR requests. Refer to chapter 5 for a refresher on component lifecycle events
3. True, moving data up the tree hierarchy of components will allow you to pass it to different children components
4. False, this object is re-used so it will not be possible to use it in an asynchronous operation
5. True, component lifecycle events are one the best places to do it, because they allow to do the prep work before component is "active" and before it will be removed.

7

Working with Forms in React

In this chapter,

- Defining with forms and its elements
- Capturing data changes
- Using references to access data
- Alternative approaches to capturing user input data from form elements
- Setting default values for form elements



Figure 7.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch07>

Thus far, we've learned about events, states, component composition and other very important topics, features and concepts of React. However, aside from capturing user events we didn't cover how to capture text input and input via other form elements like `input`, `textarea`, and `option`. Working with them is paramount to web development because they allow our applications to receive data (e.g., text) and actions (e.g., clicks) from users.

This chapter refers to pretty much everything you've covered so far. Now you'll start to see how everything is put together.

The source code for the examples in this chapter is in [the ch07 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

7.1 Recommended Way of Working With Forms in React

In regular HTML, when we are working with an input element, the page's DOM maintains that element's value in its state. It's possible to access the value via methods like: `document.getElementById('email').value` or by using jQuery methods. In essence, the DOM is our storage.

In React, when working with forms or any other user input fields such as standalone text fields or buttons, developers have an interesting problem to solve. From React's documentation: "React components must represent the state of the view at any point in time and not only at initialization time." React is all about keeping things simple by using declarative style to describe UIs. React describes the UI, it's end stage, how it should look like.

Can you spot a conflict? In the traditional HTML form elements, the states of the elements will change with the user input. However, React is using declarative approach to describe the UIs. The input needs to be dynamic to reflect the state properly.

Thus, if developers opt NOT to maintain the component state (in JavaScript) and not sync it with the view, then it adds problems: there might be a situation when internal state and view are different. React won't know about changed state. This can lead to all sort of troubles, bugs and mitigates simple philosophy of React. The best practice is to keep React's `render()` as close to the real DOM as possible and that includes the data in the form elements.

Consider this example of text input field. React must include the new value in its `render()` for that component. Consequently, we need to set the value for our element to new value using `value`. But if we implement an `<input>` field as we always did it in HTML, React will always keep the `render()` in sync with the real DOM. React won't allow users to change the value. Try it yourself. It drives me nuts a bit but that's the appropriate behavior for React!

```
render() {
  return <input type="text" name="title" value="Mr." />
}
```

The code above represents the view at any state, so the value will *always* be `Mr..` Whereas, with input fields they must change in response to the user clicks or typings. Given these points, let's make the value dynamic. This is a better implementation, because it'll be updated from the state:

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

However, what is the value of state? React cannot know about users typing in the form elements. Developers need to implement an event handler to capture changes with `onChange`.

```

handleChange(event) {
  this.setState({title: event.target.value})
}
render() {
  return <input type="text" name="title" value={this.state.title} onChange={this.handleChange}
    .bind(this) />
}

```

Given these points, the best practice is for developers to implement these things in order to sync the internal state with the view (Figure 7-1):

1. Define elements in `render()` with using values from state
2. Capture changes of a form element using `onChange()` as it happens
3. Update the internal state in event handler
4. New values will be saved in state and then the view will be updated by a new `render()`

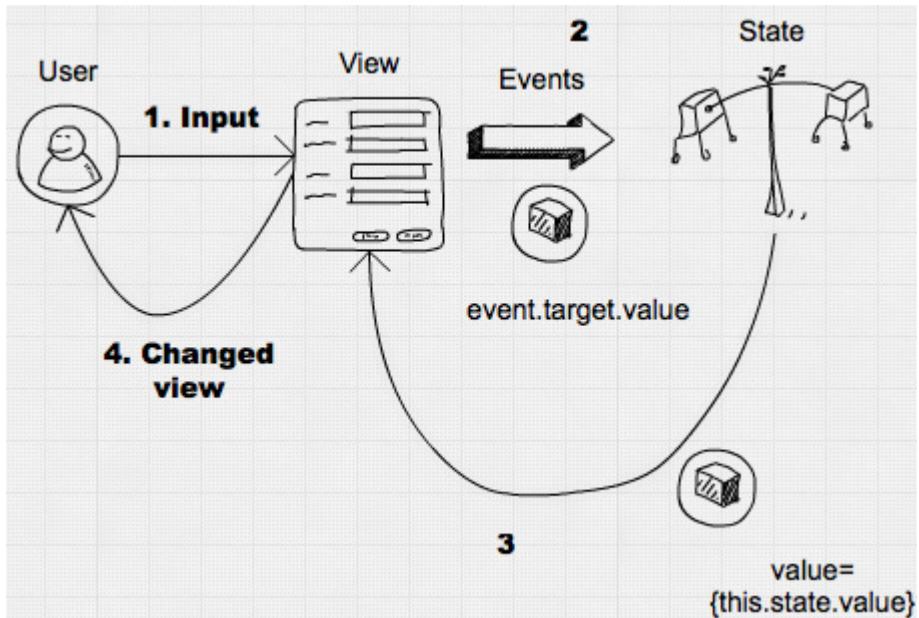


Figure 7.2 The RIGHT way of working with form elements: From users input to events, then to state and to view

It might seem like a lot of work at the first glance, but I hope once you start using React more, you'll appreciate this approach. *It's called is a one-way binding because state changes views and that's it.* There's no trip back, only a one way trip from state to view. With one-way binding a library won't update state (or model) automatically. One of the main benefits of one-way binding is that it removes complexity when working with large apps where many views implicitly can update many states (data models) and vice versa—Figure 7-2.

Simple does NOT always mean write less code. Sometimes like in this case, developers will have to write some extra code to manually set the data from event handlers to the state (which is rendered to view), but this approach tends to be superior when it comes to complex user interfaces and single-page applications with myriads of views and states. To put it shortly: simple is not always easy.

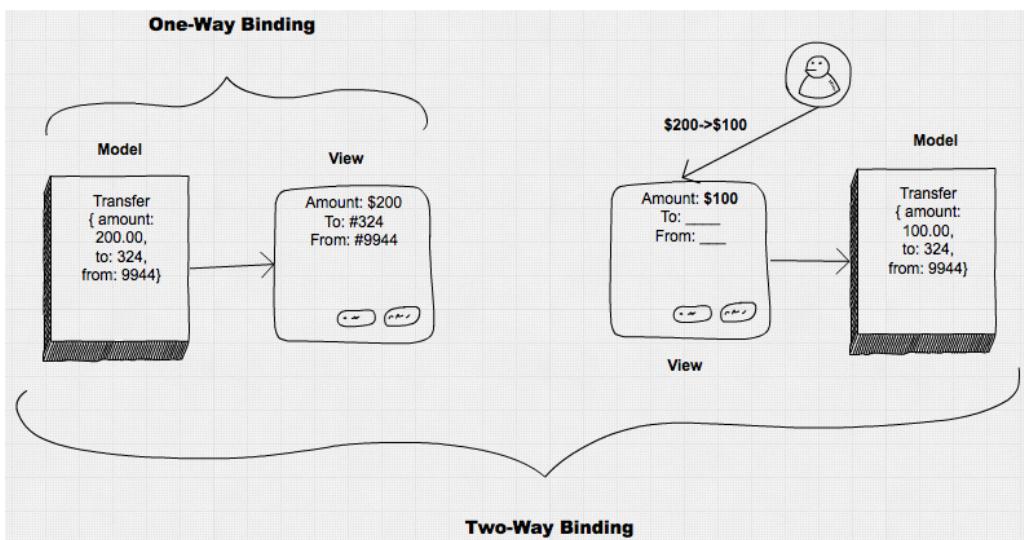


Figure 7.3 One-way binding is responsible for model->view transition while two-way binding also handles changes from view->model.

Conversely, a two-way binding would make it possible for views to change states automatically without developers explicitly implementing it. The two-way binding is how Angular 1 works. Interestingly, Angular 2 borrowed the concept of one-way binding from React and made it default (you can still have two-way binding explicitly).

For this reason, we will cover the best recommended approach of working with forms first. It's called controlled component and it will ensure that the internal component state is always in sync with the view. The alternative approach is uncontrolled component which we will also cover but later.

All in all, we learned the best practice of working with input fields in React which is to capture the change and apply it to state as depicted in Figure 7-1 (input to changed view). Next, let's take a look how we define form and its elements.

7.1.1 Defining Form and Its Events in React

We'll start with the `<form>` element. Typically, we don't want our input elements just hanging randomly in DOM. This can turn bad if we have many functionally different sets of inputs. Instead, we wrap input elements that share a common purpose in a `<form></form>` element.

Having `<form>` wrapper is not necessary. It's totally fine to just use form elements by themselves in simple user interfaces. In more complex UIs where developers might have multiple groups of elements on single page. In this case, it's wise to use `<form>` for each of such group. React's `<form>` is rendered at an HTML `<form>`, so whatever rules we have there will apply to React's `<form>` element too. According to [HTML5 spec](#), developers should NOT nest forms (it says content is flow content but with no `<form>` element descendants).

The form element itself can have events. React supports three events for forms in addition to standard React DOM events (as outlined fully in Table 6-1):

- `onChange`: Fires when there is a change in any of the form's input elements.
- `onInput`: Fires for each change in `<textarea>` `<input>` elements values. React team doesn't recommend using it (see sidebar).
- `onSubmit`: Fires when the form is submitted, usually by pressing enter.

onChange vs. onInput

React's `onChange` fires on every change in contrast to [the DOM's change event](#) which might not fire on each value change but fires on lost focus. For example, for `<input type="text">` a user can be typing with no `onChange` and only after user presses tab or click with his/her mouse away to another element (lost focus), only then the `onChange` will be fired in HTML (regular browser event). As mentioned earlier, in React `onChange` fires on each keystroke, not just on lost focus.

On the other hand, `onInput` in React is a wrapper for the DOM's `onInput` which fires on each change. React team recommends using `onChange` over `onInput`.

The bottom line is that React's `onChange` works differently than `onChange` in HTML in that it's more consistent (and more like HTML's `onInput`). `onChange` is triggered on every change and not on the lose of focus.

And the recommended approach in React is to use `onChange` in React, and `onInput` only when you need to access native behavior for the `onInput` event. The reason being is that the React's `onChange` wrapper behavior will provide consistency and thus sanity.

In addition to the three events listed above, the `<form>` can have standard React events such as `onKeyUp` or `onClick`. Using form events might come in handy when we need to capture a specific event for the entire form (that is, a group of input elements).

For example, it's a good UX to allow users to submit the data on enter (assuming you're not in the `textarea` field in which case enter should create a new line). We can listen to the form submit event by creating an event listener which triggers `this.handleSubmit()`.

```

handleSubmit(event) {
  ...
}
render() {
  <form onSubmit={this.handleSubmit}>
    <input type="text" name="email" />
  </form>
}

```

PLEASE NOTE: We'll need to implement the `handleSubmit` function outside of `render()`, just as we would do with any other event. There's no naming convention which React requires, so you can name the event handler however you wish as long as it's understandable and somewhat consistent. In this book, we'll stick with the most popular convention to prefix event handler with the word "handle" to distinguish them from regular class methods.

And, as a reminder, don't invoke a method (don't put parenthesis), and don't use double quotes around the curly braces (right way: `EVENT={this.METHOD}`) when setting the event handler. I know, for some readers it's basic JavaScript and straightforward but you would not believe how many times I saw errors related to these two misunderstandings in React code: we pass definition of the function, not its result and we use curly braces as values of the JSX attributes.

Another way how we can implement the form submission on enter is by manually listening to key up event (`onKeyUp`) and checking for the key code (13 for enter).

```

handleKeyUp(event) {
  if (event.keyCode == 13) return this.sendData()
}
render() {
  return <form onKeyUp={this.handleKeyUp}>
    ...
  </form>
}

```

Please note that the `sendData()` method is implemented somewhere else in the code. Also, we'll need to bind(`this`) event handler in `constructor()`.

To summarize, we can have events on the form element, not just on individual elements in the form. Next, we'll cover how to define form elements themselves.

7.1.2 Defining Form Elements

We implement almost all input fields in HTML with just these four elements: `<input>`, `<textarea>`, `<select>` and `<option>` (see sidenote for info on more elements). Do you remember that in React properties are immutable? Well, form elements are special because users need to interact with the elements and change these properties. For all other elements, this is impossible.

React made these elements special by giving them the mutable properties `value`, `checked`, and `selected`. Another word for these special mutable properties is *interactive properties*.

Here's the list of the interactive properties/fields (the one which can change) which we can read from events like `onChange` attached to `<form>` or the elements themselves (covered later).

- `value`: Applies to `<input>`, `<textarea>` and `<select>`
- `checked`: Applies to `<input>` with `type="checkbox"` and `type="radio"`
- `selected`: Applies to `<option>` (used with `<select>`)

React DOM also supports other elements related to building forms such as `<keygen>`, `<datalist>`, `<fieldset>`, `<label>`. Those elements don't possess any super powers like mutable `value` attribute/property. These elements will be simply rendered as corresponding HTML tags. For this reason, in this book we will focus only on the three main elements with super powers.

We can read the values and change it by working with these interactive (mutable) properties. Let's cover some examples how to define each of the elements.

INPUT

The `<input>` element renders multiple fields by using different values for its `type` attribute:

- Text: Plain text input field
- Password: Text input field with masked display (for privacy)
- Radio: Radio button (use the same name to create a group of radio buttons)
- Checkbox: Check box element (use the same name to create a group)
- Button: Button form element

The main use case for all `<input>` types elements—except for checkboxes and radio buttons—use `value` as its interactive/changeable property. For example, an email input field can utilize `email` state and `onChange` event handler:

```
<input
  type="text"
  name="email"
  value={this.state.email}
  onChange={this.handleEmailChange}/>
```

The two exceptions which don't use `value` as their primary mutable attributes are inputs with the types of `checked` and `radio`. They use `checked` because these two types have one value per HTML element, and thus the `value` doesn't change but the state of checked/selected changes. For example, we can define three radio buttons in one group (`radioGroup`) by defining these three elements.

Angular

React

Polymer

Figure 7.4 Radio button group

As mentioned before, the values (`value`) are hardcoded because we don't need to change them, but what actually changes with the user actions is the elements `checked` attribute.

Listing 7-1: Rendering radio buttons form elements and handling their changes

(ch07/elements/jsx/content.jsx)

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleRadio = this.handleRadio.bind(this)
    ...
    this.state = {
      ...
      radioGroup: {
        angular: false,
        react: true,          ①
        polymer: false
      }
    }
  }
  handleRadio(event) {
    let obj = {} // erase other radios
    obj[event.target.value] = event.target.checked // true ②
    this.setState({radioGroup: obj})
  }
  ...
  render() {
    return <form>
      <input type="radio"
        name="radioGroup"
        value='angular'
        checked={this.state.radioGroup['angular']}> ③
      <input type="radio"
        name="radioGroup"
        value='react'
        checked={this.state.radioGroup['react']}> ④
      <input type="radio"
        name="radioGroup"
        value='polymer'
        checked={this.state.radioGroup['polymer']}>
      ...
    </form>
  }
}
```

- 1 Set the default checked radio button in the state
- 2 Use target.checked attribute to get boolean of whether this radio button selected or not
- 3 Use attribute from the state object or just any state attribute
- 4 Use the same onChange event handler because we can get the radio button value from target.value

For checkboxes, we follow a similar approach as with radio buttons by using `checked` attribute and boolean values for states. Those booleans can be stored in a state `checkboxGroup` such so:

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleCheckbox = this.handleCheckbox.bind(this)
    // ...
    this.state = {
      // ...
      checkboxGroup: {
        node: false,
        react: true,
        express: false,
        mongodb: false
      }
    }
}
```

Then the event handler (which we binded in the constructor) will grab the current values, add true or false from `event.target.value` and set the state:

```
handleCheckbox(event) {
  let obj = Object.assign(this.state.checkboxGroup)
  obj[event.target.value] = event.target.checked // true or false
  this.setState({checkboxGroup: obj})
}
```

There was no need for the assignment from the state in radio because radio buttons can have only one selected value. Thus we used an empty object. This is not the case with checkboxes. They can have multiple values selected so we need a merge, not replace.

In JavaScript, objects are passed and assigned by references. So in a statement `obj = this.state.checkboxGroup`, `obj` is really a state. As you remember, developers are not supposed to change the state directly. To avoid any potential conflicts, it's better to assign by value with `Object.assign()`. Another name for this is cloning. Another less effective and more hacky way to assign by value is to use JSON such as:

```
clonedData = JSON.parse(JSON.stringify(originalData))
```

When dealing with state arrays instead of object and there's a need to assign by value, use `clonedArray = Array.from(originArray)` or `clonedArray = originArray.slice()`.

- Node
- React
- Express
- MongoDB

Figure 7.5 Rendering checkboxes with React as the pre-selected option

We can use `handleCheckbox()` event handler to get the value from `event.target.value`. This is the code for the `render()` which will use the state values for 4 checkboxes captured in Figure 7.5:

Listing 7.2 Defining checkboxes (ch07/elements/jsx/content.jsx)

```
<input type="checkbox"
      name="checkboxGroup"
      value='node'
      checked={this.state.checkboxGroup['node']}          ①
      onChange={this.handleCheckbox}/>
<input type="checkbox"
      name="checkboxGroup"
      value='react'
      checked={this.state.checkboxGroup['react']}          ②
      onChange={this.handleCheckbox}/>
<input type="checkbox"
      name="checkboxGroup"
      value='express'
      checked={this.state.checkboxGroup.express}           ③
      onChange={this.handleCheckbox}/>
<input type="checkbox"
      name="checkboxGroup"
      value='mongodb'
      checked={this.state.checkboxGroup['mongodb']}        ④
      onChange={this.handleCheckbox}/>
```

- ① Use state as a value; it can be an attribute of an object or just a state attribute.
- ② Use `onChange` to capture actions
- ③ Use the dot notation when keys are valid JS names
- ④ No need to bind in the element due to binding in constructor (true for all for checkboxes)

In essence, when using checkboxes or radio buttons, we can hard code the value in each individual element and use `checked` as our mutable attribute. Let's see how to work with other input elements.

TEXTAREA

`<textarea>` elements are for capturing and displaying long text inputs such as notes, blog posts, code snippets, etc. In regular HTML, `<textarea>` uses inner HTML (that is, children) for the value. For example,

```
<textarea>
  With the right pattern, applications...
</textarea>
```

You can see an example in Figure 7.6.

With the right pattern, applications will be more scalable and easier to maintain.

If you aspire one day to become a Node.js architect (or maybe you're already one and want to extend your knowledge), this presentation is for you.

Figure 7.6 Defining and rendering the textarea element

In contrast, React uses the `value` **attribute**. Knowing this, setting value as inner HTML/text is an anti-pattern. React will convert any children (if you use them) of `<textarea>` to the default value (more on default values later).

```
<!-- Anti-pattern: AVOID doing this! -->
<textarea name="description">{this.state.description}</textarea>
```

Instead, it's recommended that you use the `value` attribute (or property) for `<textarea>`.

```
render() {
  return <textarea name="description" value={this.state.description}/>
}
```

To listen for the changes, use `onChange` as you would for the `<input>` elements.

SELECT AND OPTION

Select and option fields are great UX-wise for allowing users to select just a single value or a few multiple values from a pre-populated list of values. The list of values is compactly hidden behind the element until users expand it (in the case of a single select) as shown in Figure 7-6.



Figure 7.7 Rendering and preselecting the value of a dropdown

`<select>` is another element whose behavior is different in React compared to regular HTML. For instance, in regular HTML we might use `selectDOMNode.selectedIndex` to get the index of the selected element or `selectDOMNode.selectedOptions`. In React, we use `value` for `<select>` as in this example:

Listing 7.3 Rendering form elements (ch07/elements/jsx/content.jsx)

```

...
constructor(props) {
  super(props)
  this.state = {selectedValue: 'node'}
}
handleSelectChange(event) {
  this.setState({selectedValue: event.target.value})
}
...
render() {
  return <form>
    <select
      value={this.state.selectedValue}
      onChange={this.handleSelectChange}>
      <option value="ruby">Ruby</option>
      <option value="node">Node</option>
      <option value="python">Python</option>
    </select>
  </form>
}
...

```

The code renders a drop-down menu and preselects the `node` value (must be set in `constructor()`) as shown in Figure 7-6. Yay for Node!

Sometimes, we need to use a multi-select elements. We can do so in JSX/React by providing the `multiple` attribute without any value (React defaults to `true`), or with value `{true}`.

Remember that for consistency and to avoid confusion, I recommend wrapping all boolean values in curly braces {} and not ""? Sure, the "true", and `{true}` will produce the same result. In contrast, "false" will also produce true. This is because a string "false" will be treated as true in JavaScript (truthy).

To preselect multiple items, we can pass an array of options to `<select>` via its `value` attribute. For example, in this code, we preselect Meteor and React:

```

<select multiple={true} value={['meteor', 'react']}>
  <option value="meteor">Meteor</option>
  <option value="react">React</option>
  <option value="jQuery">jQuery</option>
</select>

```

The `multiple={true}` will render the multi-select element, and the Meteor and React values will be preselected as shown in figure 7-7:

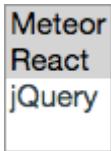


Figure 7.8 Rendering and preselecting multi-select element

Overall, defining form elements in React is not much different, than doing it in regular HTML except that we use `value` more often and I personally like this consistency. However, defining is half of the work. Another half is capturing the values. We did that a little bit in the previous examples. Let's zoom in on the event captures a bit more.

7.1.3 Capturing Form Changes

As mentioned before, to capture changes from a form element, we set up an `onChange` event listener. This event supersedes the normal DOM's `onInput`. In other words, if you need the regular HTML DOM behavior of `onInput`, then you can use React's `onInput`. On the other hand, React's `onChange` is not exactly the same as regular DOM's `onChange`. Regular DOM `onChange` might be fired only when element loses focus while the React's `onChange` will fire on all new input. What will trigger `onChange` varies for each element; here's the mapping:

- `<input>, <textarea> and <select>`: Change in `value`
- `<input>` with type checkbox and radio: Change in `checked`

Based on the mapping shown above, the approach to reading the value will change. As an argument of the event handler, we are getting Synthetic Event. It has `target` property which has `value`, `checked` or `selected` depending on the element.

To listen for changes, simply define the event handler somewhere in your component (you can define it inline too meaning right in the JSX's `{}`) and create the `onChange` attribute pointing to your event handler. For example, to capture changes from an email field:

Listing 7.4 Rendering form elements and capturing changes

(ch07/elements/jsx/content.jsx)

```
handleChange(event) {
  console.log(event.target.value)
}
render() {
  return <input
    type="text"
    onChange={this.handleChange}
    defaultValue="hi@azat.co"/>
}
```

Interestingly enough, if you don't define `onChange` but provide `value`, then React will issue a warning and make the element read-only. If that's our intention (have a read-only field), it's

better to define it explicitly by providing `readOnly` (set to `{true}` or just by itself because React by default will add the value of true to the attribute) and remove the warning.

Once you capture changes in the elements, you can store them in state of your component:

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

Sooner or later, you'll need to send this information to a server or another component. In this case, you'll have the values neatly organized in your state.

For example, you can have a loan application form with name, address, telephone and social security number. Each input field will be handling their own changes. Then at the bottom of this form, you will put a Submit button which will send the state to the server. The code below shows the name field with `onChange` which is keeping all the input in the state.

Listing 7.5 Rendering form elements (ch07/elements/jsx/content.jsx)

```
constructor(props) {
  super(props)
  this.handleInput = this.handleInput.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
  ...
}
handleFirstNameChange(event) {
  this.setState({firstName: event.target.value}) ①
}
...
handleSubmit() {
  fetch(this.props['data-url'], {method: 'POST', body: JSON.stringify(this.state)}) ②
    .then((response)=>{return response.json()})
    .then((data)=>{console.log('Submitted: ', data)})
}
render() {
  return <form>
    <input name="firstName"
      onChange={this.handleFirstNameChange}
      type="text"/>
    ...
    <input
      type="button"
      onClick={this.handleSubmit}
      value="Submit"/>
  </form>
}
```

① Capture changes of first name field by saving them to the state

② Send data to a URL from a prop `data-url` with `fetch` promises-based browser API (experimental as of this writing, but supported by most modern browsers)

③ Define event handler to handle submit button

Fetch is an experimental native browser method to perform promises-based AJAX/XHR requests. You can read about its usage and support (supported by most modern browsers as of this writing) at [Fetch API on Mozilla Developer Network](#).

We've learned how to define elements and capture the changes. The next section will provide an example.

7.1.4 Account Field Example

We've learned how to define elements, capture changes with events and update the state (which we use to display values). Continuing with our loan application scenario, once the loan is approved, users will need to be able to type in the account number where they want their loan money to be transferred to. Let's implement an account field component using our new skills. This would be a controlled element which is the best practice when it comes to working with forms in React.

In this account field component, we'll have an account number input field that needs to accept only numbers. To limit the input to a number (0-9), we can use a controlled component to weed out all non-numeric values. The event handler will set state only after filtering the input.

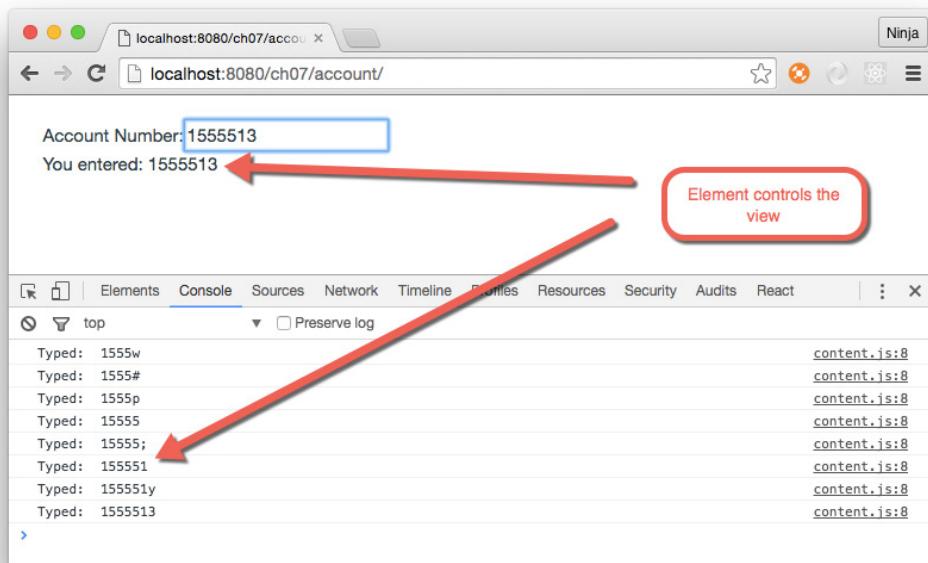


Figure 7.9 You can type anything you want as shown in the console, but only digits will be allowed as the value and in the view because this element is controlled

I'm using [a regular expression](#) `/[^0-9]/ig`, and the string function `replace` to remove all non-digits. The `replace(/[^0-9]/ig, '')` is just an uncomplicated regular expression function that replaces anything but numbers with an empty space. The `ig` stands for case insensitive and global (in other words, find all matches).

Listing 7.6 Implementing controlled component account

(ch07/account/jsx/content.jsx)

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleChange = this.handleChange.bind(this)
    this.state = {accountNumber: ''}          ①
  }
  handleChange(event) {
    console.log('Typed: ', event.target.value)  ②
    this.setState({accountNumber: event.target.value.replace(/[^0-9]/ig, '')})      ③
  }
  render() {
    return <div>
      Account Number:
      <input
        type="text"
        onChange={this.handleChange}          ④
        placeholder="123456"
        value={this.state.accountNumber}/>      ⑤
      <br/>
      <span>{this.state.accountNumber.length > 0 ? 'You entered: ' + this.state.accountNumber
        : ''}</span>                         ⑥
    </div>
  }
}
```

- ① Set initial value of the account number to an empty string
- ② Output the unfiltered value as it was typed
- ③ Filter the value and update the state
- ④ Capture changes
- ⑤ Control the element by assigning value to state
- ⑥ Print the account number if it's not empty (`length` is a string property which returns the number of characters). If the value is empty, print nothing.

The `render()` has the `input` field which is a controlled component because `value={this.state.accountNumber}`. When you try this example, you'll be able to type in only numbers because React will set the new state to the filtered number-only value (Figure 7-9).

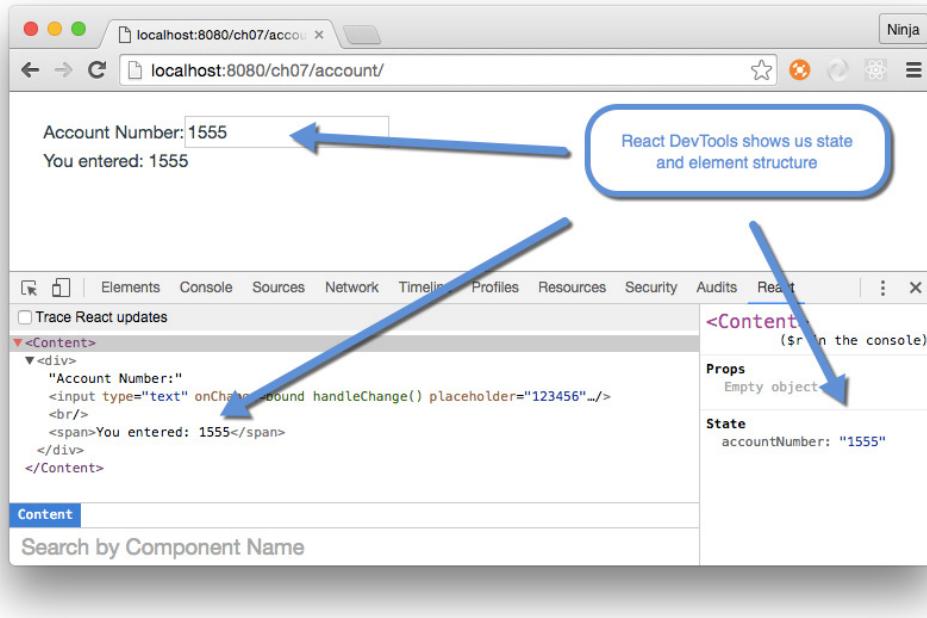


Figure 7.10 Controlled element is filtering the input by setting state only to digits

By following React's best practice for working with input elements and forms developers can implement validation and enforce that the representation is what the app wants it to be.

Obviously, in the account component we implemented a front-end validation which won't prevent a hacker from inputting some malicious data into your XHR request sent to the server. Therefore, make sure to have the proper validation on the back-end/server and/or business layer (like [ORM/ODM](#)).

So far we've learned about the best practice of working with forms which is to create controlled components. Let's cover some alternatives.

7.2 Alternative Ways of Working with Forms

So far we've worked with controlled form elements. They are called this way because React controls or sets the values. This is the best practice. However, as you've seen it requires some additional work from developers because we need to manually capture the changes and update the states. In essence, if developers define the value of attributes `value`, `checked` or `selected` using strings, props or states, then an element is controlled (by React).

At the same time, form elements can be uncontrolled when the value attributes are not set (neither to state nor to the static value). Even though this is discouraged for the reasons listed in the beginning of this chapter (you view DOM state might be different with internal state of React), uncontrolled elements be useful when all you're building is a simple form which will be submitted to the server. To put it differently, consider uncontrolled pattern you're not building a complex UI element with a lot of mutations and user actions. Think of an uncontrolled component pattern as a hack which *most of the times* should be avoided.

WARNING React is still relatively new and the best practices are still being formed through real life experiences of not just writing but maintaining apps. Recommendations might change based on a few years of maintaining a large React app. The topic of uncontrolled belongs to one of these grey areas on which is no clear consensus. You might hear that this is anti-pattern and should be avoided completely. I don't take sides but present you with enough information so you can make your own judgement. I do it because I truly believe that readers in the end should have the knowledge, and are smart enough to act upon it. The bottom line, **consider the rest of the chapter as an optional reading, a tool to have in your tool belt which you might or might not use.**

To illustrate when uncontrolled components can be utilized for faster development, consider the same application form which we discussed earlier. There are a few input fields and a submit button. There are no additional UIs or forms. There's no state.

Oh, and React team in their official documentation uses terms controlled and uncontrolled components which in this context means the same as elements.

Typically, to use uncontrolled components, define a form submit event which is typically `onClick` on a button and/or `onSubmit` on a form. Once you have this event handler, we have two options:

1. Capture changes as we did with controlled elements and use state for submission but not for values (it's uncontrolled approach after all!)
2. Don't capture changes

The first approach is straightforward. It's about having the same event listeners and updating the states. That's too much coding if we are using the state only at the final stage (for form submission).

7.2.1 Uncontrolled Elements with Capturing Changes

To refresh: in React, an *uncontrolled component simply means that the value property is not set by the React library*. When this happens, the component's internal value (or state) might differ from the value in the component's representation (or view). Basically, there's a dissonance between internal state and representation. The component state can have some logic (like validation), and with an uncontrolled component pattern, your view will accept any user input in a form element, thus creating the disparity between view and state.

For example, this text input field is uncontrolled because React doesn't set the value:

```
render() {
  return <input type="text" />
}
```

Any user input will be immediately rendered in the view. Is it good or bad? Please bear with me; I'll walk you through this scenario.

To capture changes in an uncontrolled component, use `onChange`. For example, this input field has an `onChange` event handler (`this.handleChange`), a reference (`textbook`), and a placeholder, which yield a grey text box when the field is empty (Figure 7-10).

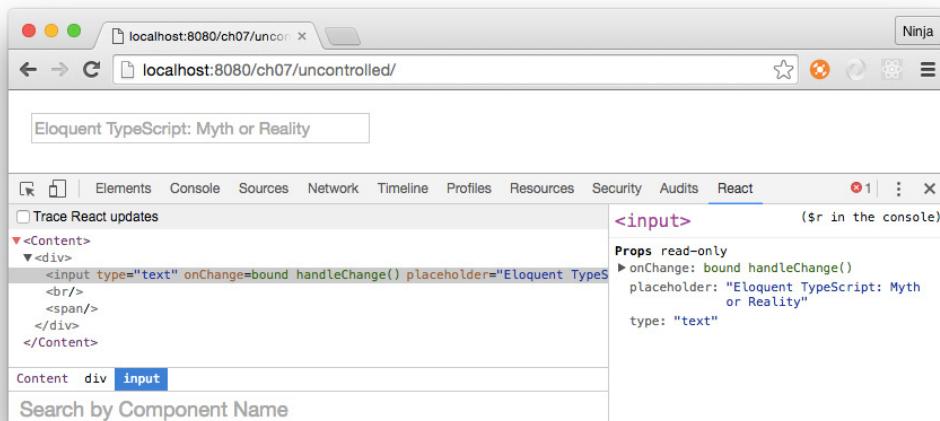


Figure 7.11 Uncontrolled component has no value set by the application

This is the `change` method that prints the values in the console and updates the state using `event.target.value`:

Listing 7.7 Rendering an uncontrolled element which captures changes (ch07/uncontrolled/jsx/content.jsx)

```
class Content extends React.Component {
  constructor(props){
    super(props)
    this.state = {textbook: ''}
  }
  handleChange(event) {
    console.log(event.target.value)
    this.setState({textbook: event.target.value}) ②
  }
  render() {
```

```

    return <div>
      <input
        type="text"
        onChange={this.handleChange}          ③
        placeholder="Eloquent TypeScript: Myth or Reality" />
      <br/>
      <span>{this.state.textbook}</span>       ④
    </div>
  }
}

```

- 1 Set initial value to an empty string
- 2 Update the state on each change in the input field
- 3 Dont set the value for input, only event listener
- 4 Use to output the state variable which we will set in the `change` method

The idea is that **users can enter whatever they want because React has no control over the value of the input field**. All React is doing is capturing new values (onChange) and setting the state. The change in state will, in turn, update (Figure 7.12).

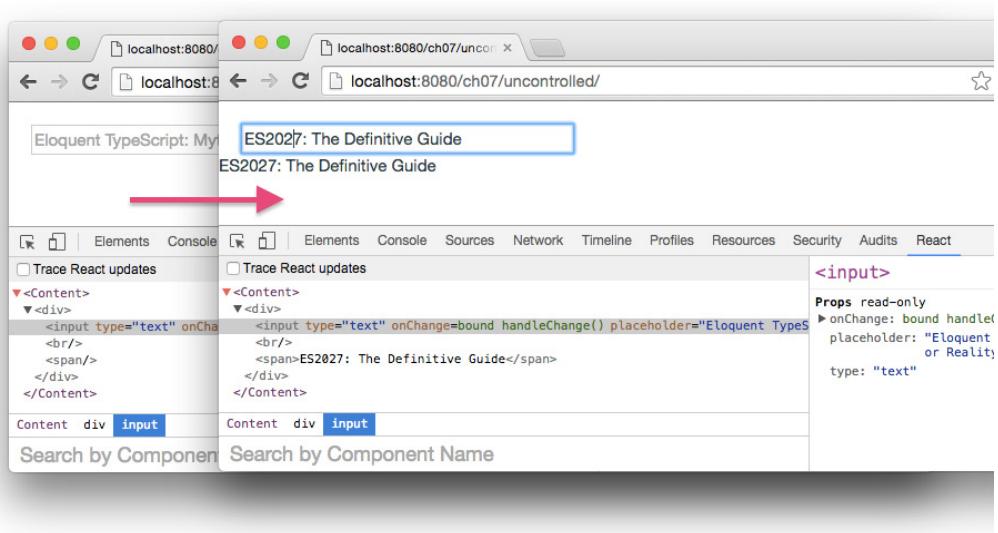


Figure 7.12 Typing will update the state due to capturing changes, but the value of the DOM input text element will not be controlled

In this approach we had to implement an event handler for the input field. Can we skip capturing events completely?

7.2.2 Uncontrolled Elements Without Capturing Changes

Let's take a look at a second approach. There's a problem of having all the values ready when we want to use them (form submit for example). You see, in the approach of capturing changes we had all the data in states. When we opt to not capture changes with uncontrolled elements, the data is still in the DOM. To get the data into JavaScript object, the solution is to use references as depicted in Figure 7.13.

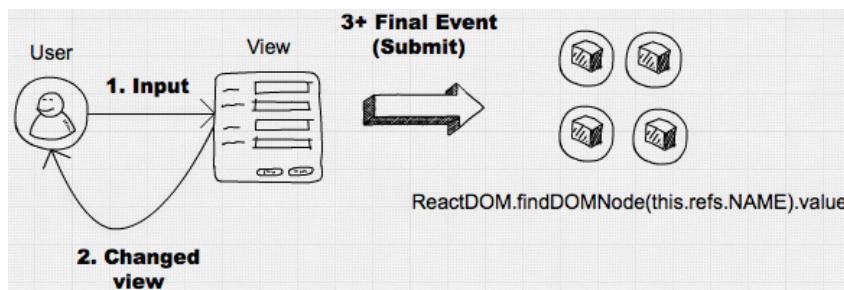


Figure 7.13 Using an uncontrolled element without capturing changes but with accessing values via refs

Contrast that with Figure 7-1 which shows how controlled elements function. By the way, when working with controlled components or with uncontrolled which capture the date, the data is in state all the time. This is not the case with this approach.

To sum up, for uncontrolled elements without capturing changes to work, developers need a way to access other elements to get that data from them.

7.2.3 Using References to Access Values

Developers using references to access value when working with uncontrolled components which don't capture events such as `onChange`, but the references are not exclusive to this particular pattern. React developers can use references in any other scenario they see fit albeit using references is frowned upon as an anti-pattern. The reason for that is that when React elements are defined properly with each element using internal state in sync with the view's state (DOM), then the need for references is almost non-existent. However, we need to cover references and know about them.

So what are the references? With references we can get the Document Object Model (DOM) element (or a node) of a React.js component. This comes in handy when you need to get form elements values but we don't capture changes in the elements.

To utilize a reference, you'll need to do two things:

- Make sure the element in the render's return has the `ref` attribute with some camelCase name (for example, `email: <input ref="userEmail" />`).

- Access the DOM instance with the named reference in some other method (for example, in the event handler `this.refs.NAME` becomes `this.refs.userEmail`).

`this.refs.NAME` will give us an instance of a React component, but how do we get the values? It's more useful to have the DOM node! You can access the component's DOM node by calling `ReactDOM.findDOMNode(this.refs.NAME)`. For example:

```
let emailNode = ReactDOM.findDOMNode(this.refs.email)
let email = emailNode.value
```

I find this method a bit clunky to write (too lengthy), with this in mind I use an alias:

```
let fD = ReactDOM.findDOMNode
let email = fD(this.refs.email).value
```

Consider an example in which we capture user email addresses and comments, as shown in Figure 7.14. The values are outputted to the browser console.

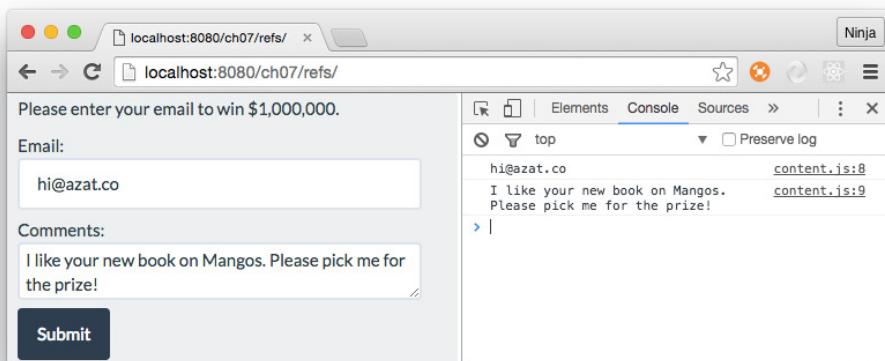


Figure 7.14 Uncontrolled form getting data from two fields and prints it in logs

The project structure is very different from other project structures and it looks like this:

```
/email
/css
  bootstrap.css
/js
  content.js      ①
  react-15.0.2.js
  react-dom-15.0.2.js
  script.js
/jsx
  content.jsx
  script.jsx      ②
index.html
```

- 1 Compiled script with the main component
- 2 `ReactDOM.render()` statement in JSX

When the Submit button is clicked, we can access our `emailAddress` and `comments` references and output the values to two logs:

Listing 7-8: Beginning of the email form with the attribute and the method (ch07/email/jsx/content.jsx)

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.submit = this.submit.bind(this)
    this.prompt = 'Please enter your email to win $1,000,000.' ①
  }
  submit(event) {
    let emailAddress = this.refs.emailAddress
    let comments = this.refs.comments
    console.log(ReactDOM.findDOMNode(emailAddress).value) ②
    console.log(ReactDOM.findDOMNode(comments).value)
  }
}
```

- 1 Define a class attribute
- 2 Access and print the value for the email address using reference

Next, we have the mandatory `render` function in which I use the Twitter Bootstrap classes to style our intake form. Remember to use `className` for the `class` attribute!

Listing 7.9 Render method of the email form (ch07/email/jsx/content.jsx)

```
render: function() {
  return (
    <div className="well">
      <p>{this.prompt}</p>
      <div className="form-group">
        Email: <input ref="emailAddress" className="form-control" type="text" placeholder="hi@azat.co"/> ①
      </div>
      <div className="form-group">
        Comments: <textarea ref="comments" className="form-control" placeholder="I like your website!"/>
      </div>
      <div className="form-group">
        <a className="btn btn-primary" value="Submit" onClick={this.submit}>Submit</a>
      </div>
    </div>
  )
}
```

- 1 Print the value of the `prompt` attribute of our component `Content`
- 2 Implement the input fields for the email which have a placeholder element attribute. A placeholder property is a visual aid to show an example of what to enter. Use the `className` and `ref` element attributes.
- 3 Code the submit button with the `onClick` event that calls `this.submit`

A regular HTML DOM node for `<textarea>` uses `innerHTML` as its value. As mentioned in the section on defining `<textarea>`, in React we can use `value` for this element:

```
ReactDOM.findDOMNode(comments).value
```

This is because React implements the `value` property. It's just one of this nice things when we get a more consistent API for form elements. At the same time, because the `ReactDOM.findDOMNode()` method returns a DOM node, we'll have access to other regular HTML attributes (like `innerHTML`) and methods (like `getAttribute()`).

Now, we know how to access elements and their values from pretty much any component method not just an event handler for that particular element. Again, references just for the rare cases when we use uncontrolled elements. The overuse of references is frowned upon as a bad practice. Most of the time, you won't need to use references with controlled elements, because you can use component states instead.

I hope you are clear on uncontrolled components. They require less coding (state updates and capturing changes are optional), but now we have another problem when using uncontrolled elements. We cannot set values to states or hardcoded values because then we'll have controlled elements (can't use `value={this.state.email}`). How to set the initial value? Let's say the loan application was pre-filled and saved and now the user just resumes the session. We need to show the information which was filled before but we cannot use `value` attribute. Let me show you how to set default values.

7.2.4 Default Values

In some cases, we want the application to pre-populate some fields with existing data. This React feature is most often used with uncontrolled components but, as with references, default values can be used with controlled components or any other scenarios. The reason why developers don't need default values as much in controlled components is because they can define those values in the state in constructor, e.g., `this.state = { defaultName: 'Abe Lincoln' }`.

In normal HTML, we define form field with `value` and users can modify the element on a page. However, React uses `value`, `checked` and `selected` to maintain consistency between the view and the internal state of the elements. In React if we hard code the `value` like this:

```
<input type="text" name="new-book-title" value="Node: The Best Parts"/>
```

It'll be a read-only input field. Not what we need (in most cases). Therefore, in React we have a special attribute `defaultValue` which will set the value and let our users to actually modify the form elements!

For example, the form was saved before, and we want to fill in the `<input>` field for the user. In this case, we'll need to use the property `defaultValue` for our form elements. We can set the initial value of the input field like this:

```
<input type="text" name="new-book-title" defaultValue="Node: The Best Parts"/>
```

If we use the `value` attribute (`value="JSX"`) instead of `defaultValue`, this element becomes read-only. Not only will it be controlled, the value won't change when the user types in the `input` shown in figure 7-14.

Figure 7.15 Value of an input element will appear on a web page as frozen (can't change it) when you set value to a string

This is because the value is hard-coded, and React will maintain that value. Probably not what you want. Obviously, in real life applications developers get values programmatically which in React be using properties (`this.props.name`):

```
<input type="text" name="new-book-title" defaultValue={this.props.title}/>
```

Or states:

```
<input type="text" name="new-book-title" defaultValue={this.state.title}/>
```

You can use `defaultValue` with controlled components too, but in most cases developers use `defaultValue` for uncontrolled components. This is because we can set the initial state value with `constructor()` when we using controlled components. This conclude usage of uncontrolled components.

7.3 Quiz

1. The uncontrolled component sets a value, while the controlled component doesn't. True or false?
2. The right syntax for default values is `default-value`, `defaultValue`, or `defVal`?
3. React team recommend using `onChange` over `onInput`. True or false?
4. You set a value for the text area with children, inner HTML, or value?
5. In a form, `selected` applies to `input`, `textarea`, or `option`?
6. What is the best way to extract the DOM node by reference: `React.findDOMNode(this.refs.email)`, `this.refs.email`, `this.refs.email.getDOMNode`, `ReactDOM.findDOMNode(this.refs.email)`, or `this.refs.email.getDomNode`?

7.4 Summary

To end this chapter, here are the common usages of React patterns when it comes to working with forms:

- Controlled components with event listeners capturing and storing data in state - most

preferred approach

- Uncontrolled components with or without capturing changes - a hack and should be avoided
- References and default values - can be used with any elements but mostly not needed when components are controlled

Also, in this chapter, we've covered the most important features of React with regards to scaling your code. You've learned about:

- React's `<textarea>` uses `value` attribute, not inner content
- `this.refs.NAME` is a way to access class references
- `defaultValue` allows to set the initial view (DOM) value for an element
- `ref="NAME"` is how developers define references

Actually, most of our user interface (UI) work is in those handy form elements. We need to make them beautiful, yet easy to understand and use. Then, we must have user-friendly error messages, front-end validation, and other non-trivial things like tooltips, scalable radio buttons, default values and placeholders. Building a UI can get complex and spiral out of control, very fast! Gladly, React makes our job easier. It lets us use a cross-browser application programming interface (API) for form elements.

7.5 Quiz Answers

1. False, the definition of controlled component/element is that it sets the value
2. `defaultValue`
3. True; in regular HTML `onChange` might not fire on every change while in React it always does
4. `value`
5. `option`
6. `ReactDOM.findDOMNode(reference)`

8

Scaling React Components

In this chapter, we cover these topics:

- Default Properties in Components
- React Property Types and Validation
- Rendering Any Children
- Creating Higher-Order Components for Code Re-Use
- Best Practices: Presentational vs. Container Components



Figure 8.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch08>

Thus far, we've covered how to create components, make them interactive and work with user input (events and input elements). Using the knowledge you have now, will get you a long way building your sites with React components but you will start to notice that certain annoyances keep appearing as you do. This is especially true for large projects when you rely on other components.

For example, when you consume a component someone else wrote how do you know if you're providing the right properties to it? Also, if you would like to use an existing components but only with a lightly added functionality (which is also applied to other components), how would

you go about it? These are *developmental scalability issues: how can you scale your code* (meaning how you work with your code when the code based grows larger and larger). There are certain features and patterns in React which can help with that!

These topics are important if you would like to learn how to effectively build a complex React application. For example, higher-order component will allow you to enhance functionality of a component, while property types provide the security of type checking and no small measure of sanity.

At the end of this chapter, you'll be familiar with most of the features of React. You'll become adept at making your code more developer friendly (by property types) and your work more efficient (component names, higher-order components). Your team mates might even marvel at your elegant solutions. Because these features will allow you to use React effectively, let's dive right in, without further ado.

The source code for the examples in this chapter is in [the ch08 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

8.1 Default Properties in Components

Imagine, you're building a component `<Datepicker/>` and it take a few required properties such as number of rows, locale and current date:

```
<Datepicker currentDate={Date()} locale="US" rows={4}/>
```

Now, what will happen if a new member of a team tries to use your components, but forgets to pass an essential property `currentDate`? Then another co-worker, passes "4" string instead of a 4 number. Your component does nothing (values `undefined`) or maybe even worse: crashes and they blame you (`ReferenceError` anyone?). Ooops.

Sadly, this is not uncommon situation in web development, because in JavaScript is loosely typed language. Fortunately, React provides a feature to set default values of properties which we'll look at now. We'll return to flagging issues with property types in the next section.

The key benefit of `defaultProps` is that if a property is missing, a default value is rendered. To set the default properties values on the component class, define the `defaultProps` static attribute. For example, in aforementioned `Datepicker` component definition, we'll add a static class attribute (not instance attribute because it won't work!—Instance attributes are set in `constructor()`):

```
class Datepicker extends React.Component {
  ...
}
Datepicker.defaultProps = {
  currentDate: Date(),
```

```

    rows: 4,
    locale: 'US'
}

```

To illustrate `defaultProps` further in the context of the previous concepts, let's say you have a component that renders a button. Typically, buttons have labels, but those labels need to be customizable. In case the custom value is omitted, it's good to have a default value.

The project structure for this example which covers this section and the next is as follows:

```

/default-props
/css
  bootstrap.css
  style.css
/js
  - button.js
  - content.js
  - react-15.0.2.js
  - react-dom-15.0.2.js
  - script.js
/jsx
  - button.jsx
  - content.jsx
  - script.jsx
- index.html

```

The hierarchy of component is `script.jsx` renders `Content (content.jsx)` which renders multiple buttons (`button.jsx`).

For JSX compilation and bundling, we are using the same setup with Webpack and npm script:

```

/ch08
/node_modules
  - package.json

```

The npm script from `package.json` is **build-default-props**:

```

...
"build-default-props": "./node_modules/.bin/babel default-props/jsx -d default-props/js -
w",
...

```

The button's label is the `buttonLabel` property, which we use in the render's return attribute. We want this property to always include `Submit`, even if the value is not set from above. To do this, we implement the `defaultProps` static class attribute, which is an object containing the property `buttonLabel` with a default value:

```
class Button extends React.Component {
  render() {
    return <button className="btn" >{this.props.buttonLabel}</button>
  }
}
Button.defaultProps = {buttonLabel: 'Submit'}
```

The parent component `Content` renders four buttons. However, three of these four button components are missing properties:

```
class Content extends React.Component {
  render() {
    return (
      <div>
        <Button buttonLabel="Start"/>
        <Button />
        <Button />
        <Button />
      </div>
    )
  }
}
```

Can you guess the results? The first button will have the label `Start`, while the rest of the buttons will have the label `Submit` (shown in Figure 8-1).

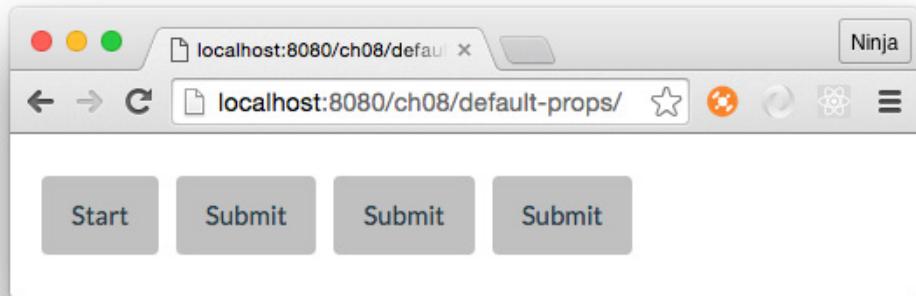


Figure 8.2 First button has a label set on creation while other elements don't and thus fall back to default property value

Setting default property values is almost always a good idea because it makes your components more fault tolerant. In other words, they become a bit smarter because they'd have some baseline look and behavior even when there's nothing supplied.

Looking at it another way, having a default value also means you can skip declaring the same old value over and over again. In essence, if you use just a single property value most of the time but still would like to provide a way to modify this value (override the default), the `defaultValue` feature is a way to go! Oh and by the way, don't worry about overriding a default value as a nonstarter, that's it won't cause any issues as you've seen with the first button element.

8.2 React Property Types and Validation

Going back to our earlier example with `Datepicker` and co-workers not being aware of property type ("5" vs 5), you can set property types to use with React.js component classes. This is done via `propTypes` static attribute. This feature of property types don't enforce data types on property values, but instead give you a warning. That is, if you're in development mode and a type doesn't match, you'll get a warning message in the console and in production, nothing will be done to prevent the wrong type from being used. In essence, **React.js suppresses this warning in production mode**. Thus, `propTypes` is mostly a convenience feature for developers to warn them about mismatches in data types at a developmental stage.

Production vs Development React

The React.js team defines development mode as when you're using the unminified (uncompressed) version and production mode as when you're using the minified version. From the mouth of React authors:

We provide two versions of React: an uncompressed version for development and a minified version for production. The development version includes extra warnings about common mistakes, whereas the production version includes extra performance optimizations and strips all error messages.

Here's a basic example of defining a static `propTypes` attribute on a `Datepicker` class with types of string, number and enumerator:

```
class Datepicker extends React.Component {
  ...
}
Datepicker.propTypes = {
  currentDate: React.PropTypes.string,
  rows: React.PropTypes.number,
  locale: React.PropTypes.oneOf(['US', 'CA', 'MX', 'EU'])
}
```

So the property types feature is used mostly for development to alert you when there's a type mismatch.

WARNING: Never rely on front-end user input validation because it can be easily bypassed. Use it only for better user experience and check everything on the server-side.

To validate property types, we use the property `propTypes` with the object containing the properties as keys and types as values. React.js types are in the `React.PropTypes` object. For example:

- `React.PropTypes.string`
- `React.PropTypes.number`
- `React.PropTypes.bool`
- `React.PropTypes.object`
- `React.PropTypes.array`
- `React.PropTypes.func`
- `React.PropTypes.shape`
- `React.PropTypes.any.isRequired`
- `React.PropTypes.objectOf(React.PropTypes.number)`
- `React.PropTypes.arrayOf(React.PropTypes.number)`
- `React.PropTypes.oneOfType([React.PropTypes.number, ...])`
- `React.PropTypes.instanceOf(Message)`
- `React.PropTypes.element`
- `React.PropTypes.node`

To demonstrate, let's enhance the `default-props` example by adding some prop types in addition to default prop values. The structure of this project is similar to `default-props` (previous example): `content.jsx`, `button.jsx` and `script.jsx`. The new project is in a folder `ch08/prop-types` for your reference.

In the `prop-types` project, there is a `Button` class with an optional `title` with a `string` type. Thus, to implement it we define a static class attribute (that is a property of that class) `propTypes` with key `title` and `React.PropTypes.string` as a value of that key:

```
Button.propTypes = {
  title: React.PropTypes.string
}
```

We can also require properties. To do so, we add `isRequired` to the type. For example, this button requires a `handler` property, which is a function:

```
Button.propTypes = {
  handler: React.PropTypes.func.isRequired
}
```

What's also nice is that you can define your own **custom validation**. To implement custom validation, all you need is to create an expression that returns an instance of `Error`. Then, you use that expression in the `propTypes: {...}` as the value of the property.

For example, this code validates email with the variable `emailRegularExpression`:

```
...
propTypes = {
  email: function(props, propName, componentName) {
    var emailRegularExpression =
      /^[^\w-]+(?:\.\[^\w-]+\.)*(?:[\w-]+\.)*\w[\w-]{0,66})\.(.[a-z]{2,6}(?:\.[a-z]{2})?)$/i
    if (!emailRegularExpression.test(props[propName])) {
      return new Error('Email validation failed!')
    }
  }
}
...
```

Now we can put everything together for an example in which the `Button` component will be called with and without a property title (string) and a handler (required function):

Listing 8-1: Using property types to ensure that handler is a function, title is a string and email adheres to the provided Regular Expression (ch08/prop-types)

```
class Button extends React.Component {
  render() {
    return <button className="btn">{this.props.buttonLabel}</button>
  }
}

Button.defaultProps = {buttonLabel: 'Submit'}

Button.propTypes = {
  handler: React.PropTypes.func.isRequired, ①
  title: React.PropTypes.string, ②
  email: (props, propName, componentName) {
    let emailRegularExpression =
      /^[^\w-]+(?:\.\[^\w-]+\.)*(?:[\w-]+\.)*\w[\w-]{0,66})\.(.[a-z]{2,6}(?:\.[a-z]{2})?)$/i
    if (!emailRegularExpression.test(props[propName])) {
      return new Error('Email validation failed!')
    }
  } ③
}
```

- ① Require handler with a function value
- ② Define optional title prop with a string value
- ③ Define an email validation with a regular expression

Now let's implement the parent component `Content` which will render six buttons:

Listing 8-2: Rendering six buttons to test the warning messages produced from prop types (ch08/prop-types/jsx/content.jsx)

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

```

let number = 1
return (
  <div>
    <Button buttonLabel="Start"/>
    <Button /> ①
    <Button title={number}/> ②
    <Button />
    <Button email="not-a-valid-email"/> ③
    <Button email="hi@azat.co"/>
  </div>
)
}

```

- ① Trigger no handler warning
- ② Trigger title must be a string warning
- ③ Trigger wrong email format warning

Running this code results in two warning messages displayed on your console (don't forget to open it). Mine looked like the ones shown here and in Figure 8-2. The first warning is about the function `handler` which must be specified, but I omitted it in a few buttons:

```
Warning: Failed propType: Required prop `handler` was not specified in `Button`. Check the render method of `Content`.
```

The second warning is about the wrong email format (fourth button had a wrong format):

```
Warning: Failed propType: Email validation failed! Check the render method of `Content`.
```

The third warning is about the wrong type for the title which should be a string but I provided a number in one button:

```
Warning: Failed propType: Invalid prop `title` of type `number` supplied to `Button`, expected `string`. Check the render method of `Content`.
```

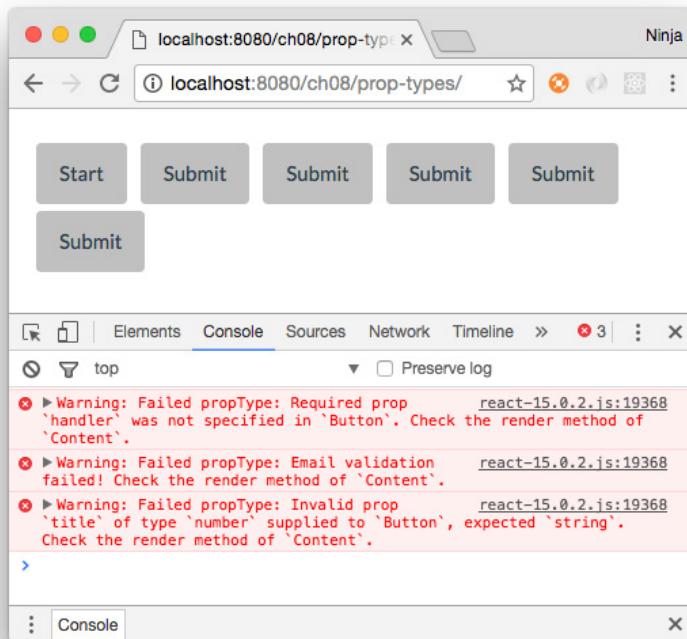


Figure 8.3 Warnings due to wrong properties types

The interesting thing is that there are more than one button with missing `handler`, we see only one warning. React will warn about each prop only once per single `render()` of `Content`.

What I love about React is that it will tell you what parent component to check. It's `Content` in our example. Imagine having 100's of components. This is very useful!

Conveniently enough, if you expand the message in DevTools, then we can spot a line number of the `Button` element which causing the trouble for that warning. See Figure 8-3 in which I first expanded the message, then located my file (`content.js`). It said that the issue is on line 9.

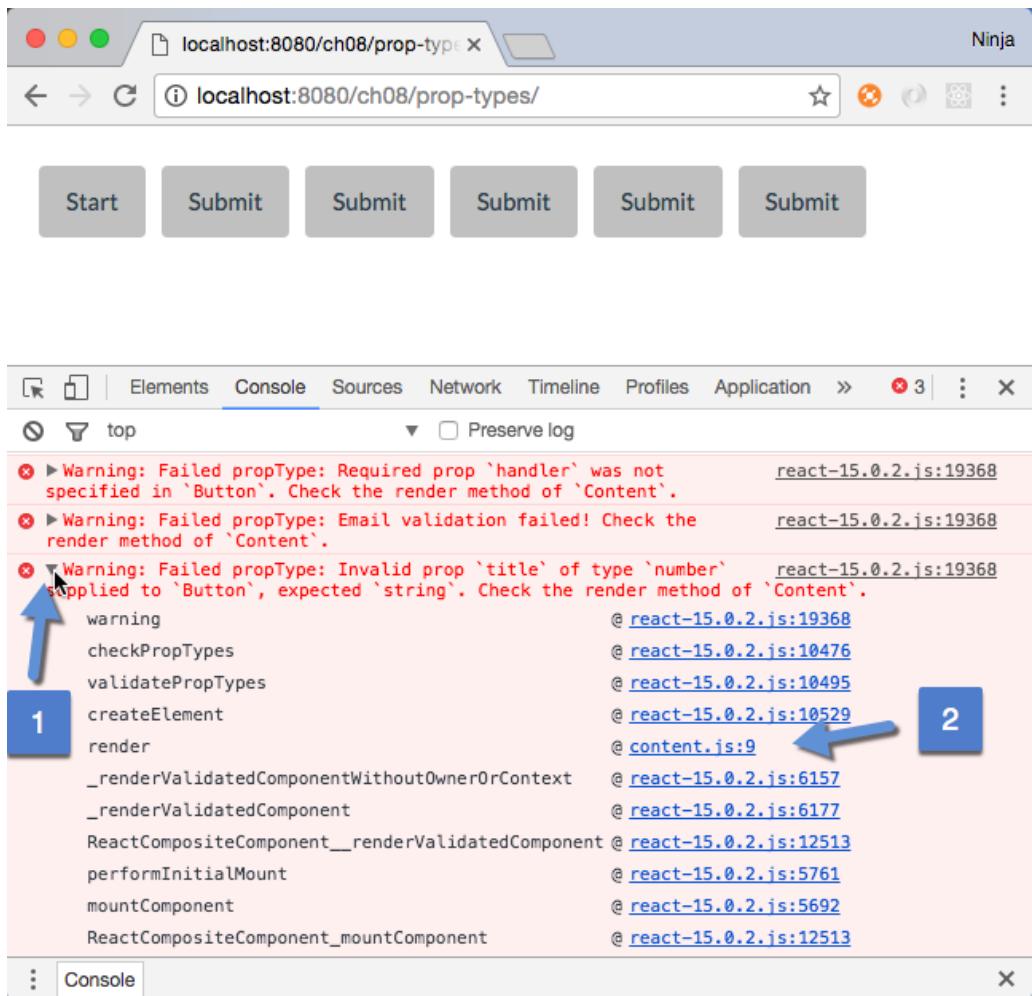


Figure 8.4 Expanding a warning revealed a the problematic line number 9

By clicking on the `content.js:9` in the console, you can open the Source tab at that line as shown in Figure 8.5.

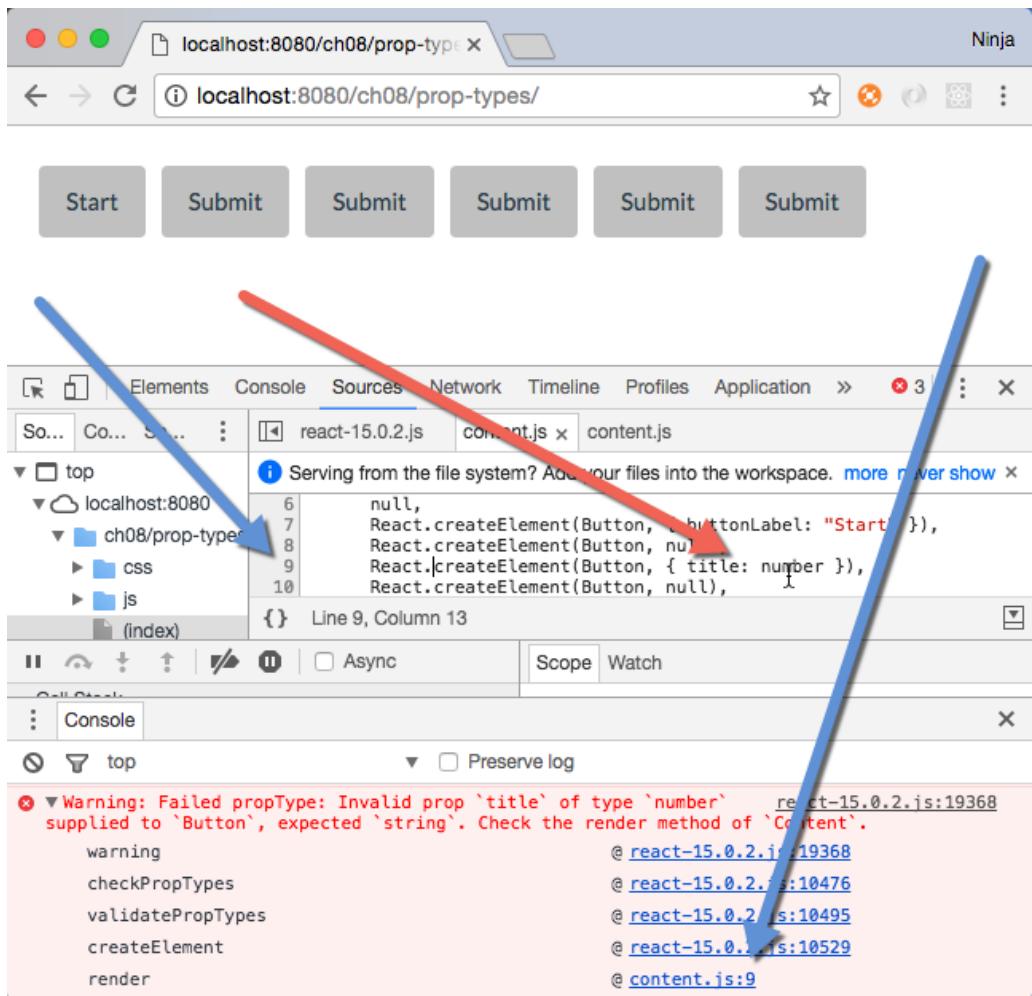


Figure 8.5 Inspecting the compiled source code is often enough to understand the problem

It clearly shows what to blame:

```
React.createElement(Button, { title: number}),
```

You don't need source maps (although we will set them up and use them in Part 2) to know that this is the third button which causing us troubles:

```
<Button title={number}>
```

I'll repeat it again: Only the unminified or uncompressed version (that is, development mode) of React shows the warnings-. ;-)

Try playing with the property types and validation yourself. It's a neat feature. Consider this code based on our example. Can you spot the problem? How many warnings do you think you'll get?

Source Maps

I got the warnings shown in Figure 8.3 because of the poorly written (on purpose) `Content`. As you can see, it's noticeable that the warning messages identify the component and where in the component problem is happening. However, the line numbers won't match your source code, because they refer to compiled JavaScript and not JSX. To get the correct line numbers, developers will need to use a source maps plugin like `source-map-support` or Webpack. We'll be covering Webpack in Part II.

It's important to know and use property types and custom validation in large projects, even if they don't have a strict enforcement or error exceptions! The benefit is that when you use someone else's component, you can verify that the supplied properties are of the right type.

There are many additional types and helper methods. Please refer to the documentation: <https://facebook.github.io/react/docs/reusable-components.html#prop-validation>.

8.3 Rendering Any Children

To continue with our fictional React project but instead of `Datepicker` (which is super robust and warns of all the missing and wrong props!), you're tasked with creation a component that is universal enough to use with any children we pass to it. It's a blog post component `Content`. It can consist of a heading and a paragraph of text:

```
<Content>
  <h1>React.js</h1>
  <p>Rocks</p>
</Content>
```

While a different blog post can consist of an image (something what you might see on Instagram or Tumblr):

```
<Content>
  
</Content>
```

They both utilize `Content`, but they pass different children to it. Wouldn't it be great to have a special way to render any children (`<p>` or ``)? There is! Please meet `children`.

The `children` property is an easy way to render all the children with `{this.props.children}`. We can also do more than rendering.

For example, we add a `div` and pass along children elements:

```
class Content extends React.Component {
  render() {
    return (
      <div className="content">
        {this.props.children}
      </div>
    )
  }
}
```

The parent of `Content` has the children `<h1>` and `<p>`:

```
ReactDOM.render(
  <div>
    <Content>
      <h1>React</h1>
      <p>Rocks</p>
    </Content>
  </div>,
  document.getElementById('content')
)
```

The end result will be that the `<h1>` and `<p>` are wrapped in the `<div>` container with a class `content` as shown in Figure 8.6. Remember, for class attributes, we use `className` in React.

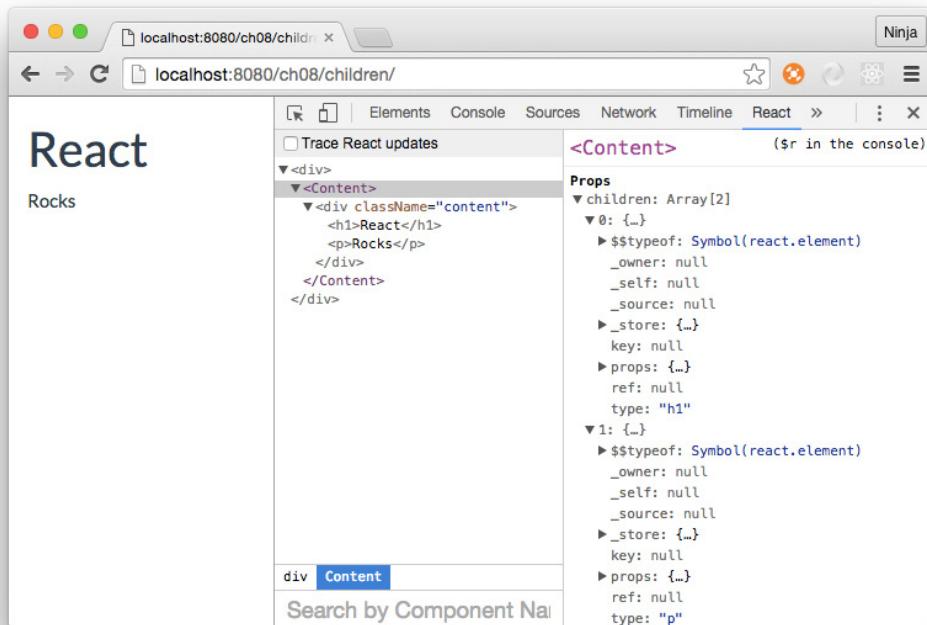


Figure 8.6 Rendering a single Content with a heading and paragraph using `this.props.children` which shows 2 items

Obviously, developers can add many more things to a component like `Content`, for example more classes for styling, layouts, and even access properties and have interactivity with events and states. I hope you got the idea that with `this.props.children` we can create pass-through components which are very flexible, powerful and universal. Let's say you need to display a link or a button in addition to text and image as shown in the previous example. The `Content` component will still be the wrapper `<div>` with the `content` class, but now there will be more different children. No changes in `Content` class itself. Simply, put the children in `Content` when you instantiate the class:

Listing 8-3: Rendering various elements using Content and its children property
`(ch08/children/jsx/script.jsx)`

```
ReactDOM.render(
  <div>
    <Content>
      <h1>React</h1>
      <p>Rocks</p>
    </Content>
    <Content>
```

```

        
    </Content>
    <Content>
        <a href="http://react.rocks">http://react.rocks</a>
    </Content>
    <Content>
        <a className="btn btn-danger" href="http://react.rocks">http://react.rocks</a>
    </Content>
</div>,
document.getElementById('content')
)

```

The resulting HTML will have two `<div>` elements with `content` classes. One will have `<h1>` and `<p>` and the other will have ``, as illustrated in the DevTools on Figure 8-6.

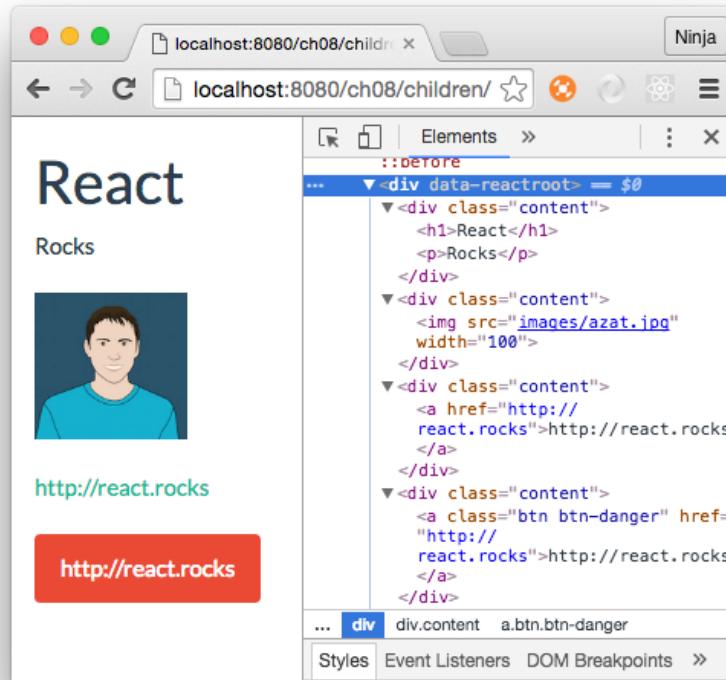


Figure 8.7 Rendering four elements with different content using a single component class

What is interesting about the `children` property is that it can be an array if there are more than one child element (as seen in Figure 8-5). You can access individual elements like this:

```
{this.props.children[0]}
{this.props.children[1]}
```

Be careful when validating children. When there's only one child element, `this.props.children` is NOT an array. If you use `this.props.children.length`, and the single children node is a string, this can lead to "bugs." That's because `length` is a valid string property. Instead, use `React.Children.count(this.props.children)` to get an accurate count of children elements. React has more helpers like `React.Children.toArray`. The most interesting (in my opinion) are:

- `React.Children.map()`
- `React.Children.forEach()`
- `React.Children.toArray()`

There's no reason here to duplicate the ever-changing list of them, so here's the link to the official documentation: <https://facebook.github.io/react/docs/top-level-api.html#react.children>

8.4 Creating React Higher-Order Components for Code Re-Use

We'll continue with the situation when you work in a large team and create components which other developers use in their projects. Let's say you are working on a piece of an interface and three of your team mates came and asked you to implement a way to load a resource (React.js website), but each of them wants to use his/her own visual representation for this "button" (there would be a button, an image and a link). Maybe, you can implement a method and call it from an event handler, but there's a more elegant solution called higher-order components. The idea is to enhance each interface with the same logic.

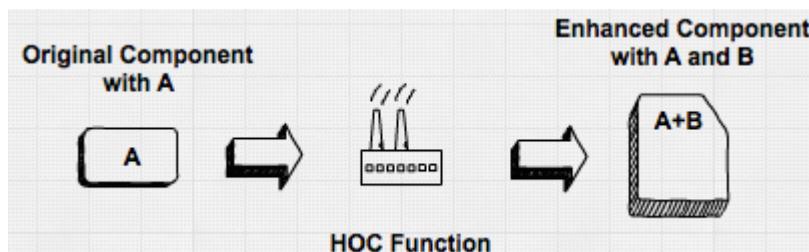


Figure 8.8 Simplified representation of the higher-order component pattern where enhanced component has props not just of A but of A and B

A higher-order component (HOC) will let you to have enhance component with additional logic. If it help you, you can think about this patterns as components inheriting some functionality when used with HOC. In other words, *HOC lets you reuse the code*. This allows you and your team to share functionality among React.js components. By doing so, developers will avoid repeating them selves (DRY—don't repeat yourself).

In essence, higher-order components are React components classes which render the original classes while addition extra functionality along the way. Defining HOC is straightforward, because it's only a function; you just it with fat arrows:

```
const LoadWebsite = (Component) => {
  ...
}
```

The name `LoadWebsite` is arbitrary, meaning you can name it anything, as long as you use the same name as when you enhance a component. Same thing with the argument to the function (`LoadWebsite`), it will be the original (not enhanced yet) component.

To demonstrate, let's set up a project for our three co-workers. The project structure goes as follows with three stateless components `Button`, `Link`, and `Logo` in `elements.jsx`, and the higher-order component function in `load-website.jsx`:

```
/hi-order
  /css
    - bootstrap.css
    - style.css
  /js
    - content.js
    - elements.js
    - load-website.js
    - react-15.0.2.js
    - react-dom-15.0.2.js
    - script.js
  /jsx
    - content.jsx
    - elements.jsx
    - load-website.jsx
    - script.jsx
  - index.html
  - logo.png
```

Going back to our co-workers, they'll need a label and a click event handler. So let's set the label, and define the `handleClick` method. Mounting events are for us to demonstrate the lifecycle:

Listing 8-4: Higher-order component implementation which adds properties to original component (ch08/hi-order/jsx/load-website.jsx)

```
const LoadWebsite = (Component) => {
  class _LoadWebsite extends React.Component {
    constructor(props) {
      super(props)
      this.state = {label: 'Run'}
      this.state.handleClick = this.handleClick.bind(this) 1
    }
    getUrl() {
      return 'https://facebook.github.io/react/docs/top-level-api.html'
    }
    handleClick(event) {
      var iframe = document.getElementById('frame').src = this.getUrl() 2
    }
  }
  return _LoadWebsite
}
```

```

        }
        componentDidMount() {
          console.log(ReactDOM.findDOMNode(this))
        }
        render() {
          console.log(this.state)
          return <Component {...this.state} {...this.props} />
        }
      }
      _LoadWebsite.displayName = 'EnhancedComponent'      ④
      return _LoadWebsite
    }
  
```

- ① Make sure that inside this method `this` is always an instance of this component
- ② Load the React website into an iframe
- ③ Pass state and props as properties using spread
- ④ Define display name for the HOC

Nothing complex, right? There are two new techniques not covered in this book before: `displayName` and the spread operator Let's quickly (as the title of this book suggests) cover them now.

8.4.1 Using `displayName`: Distinguish child components from their parent

By default, JSX will use the class name as the name of the instance (element). Thus elements created with HOC will in our example will have `_LoadWebsite` names (_ in JavaScript typically notates that this is a private attribute, variable or method and it's not intended for use as a public interface—from other modules). In cases, when developers want to change this name, there's the `displayName` static attribute. As you might know, static class attributes in ES6 must be defined outside of the class definition. (As of this writing the standard for static attributes hasn't been finalized yet.)

To sum up, `displayName` is necessary to set React element names when they need to be different from the component class name as depicted in Figure 8-8. You can see how useful it is to use the `displayName` in our `load-website.jsx` higher-order component to augment the name, because by default the component name is the function name (which might not always be a desired name).

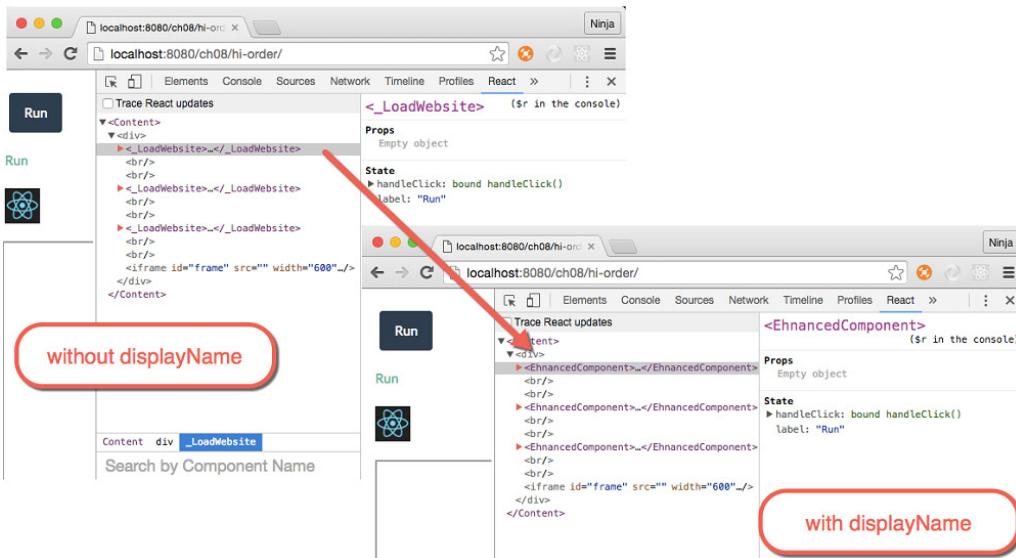


Figure 8.9 By using `displayName` static attribute, developer can change the name of the component (`LoadWebsite` to `'EnhancedComponent'`)

8.4.2 Using the Spread Operator: Pass all your attributes

Next, let's take a look at the spread operator It's part of ES6+/ES2015+ for [arrays](#) and as of the time of this writing there's a proposal to use spreads for [objects](#). It's only natural that React team added the support for spreads to JSX.

The idea is not complicated, spread operator allows to pass all of the attributes of an object (`obj`) as properties when use inside of the element:

<Component {...obj}>

We used spread already in `load-website.jsx` to pass state and prop variables to the original component when we were rendering it. We needed it because we don't know ahead of time all the props that this function will take as arguments, thus spread is a blanket statement to pass all your data (in that variable or an object).

What's good. In React and JSX, we can have more than one spread, or mix them with traditional prop declaration `key=value`. For example, we can pass all states and all props from a current class as well as `className` to a new element `Component`:

```
<Component {...this.state} {...this.props} className="main" />
```

Let's consider an example with children. In this scenario, using spread with `this.props` will pass all the properties of `DoneLink` to the anchor element `<a>`:

```

class DoneLink extends React.Component {
  render() {
    return <a {...this.props}>           ①
      <span class="glyphicon glyphicon-check"></span>
      {this.props.children}
    </a>
  }
}

ReactDOM.render(
  <DoneLink href="/checked.html">          ③
    Click here!
  </DoneLink>,
  document.getElementById('done-link')
)

```

- ① Take any props passed to `DoneLink` and copy them to `<a>`
- ② Use `Glyphicon` (`http://glyphicon.com`) to render a check icon
- ③ Pass the value for `href`

Thus in our higher-order component, we are passing all props and states to the original component when we are rendering it. By doing so, we won't have to manually add and remove the props from `render()` each time we want to pass something new or stop passing existing data from `Content` where we instantiate `LoadWebsite/EnhancedComponent` to each original element.

8.4.3 Using Higher-Order Components

So we've learned more about `displayName` and ... spread operators in JSX and React. Now we can back to the higher-order components by seeing how to use them.

Going back to the `Content` and `content.jsx`, that's where we are using `LoadWebsite`. And after defining the HOC, we need to create components using HOC in `content.jsx`:

```

const EnhancedButton = LoadWebsite(Button)
const EnhancedLink = LoadWebsite(Link)
const EnhancedLogo = LoadWebsite(Logo)

```

Now, we'll implement three components--`Button`, `Link`, and `Logo`--to reuse our code with the HOC pattern. The `Button` component will be created via `LoadWebsite` and because of that it will magically inherit its properties (that is, `this.props.handleClick` and `this.props.label`):

```

class Button extends React.Component {
  render() {
    return <button
      className="btn btn-primary"
      onClick={this.props.handleClick}>
      {this.props.label}
    </button>
  }
}

```

The Link component will be created by HOC, that's why we can use handleClick and label properties too:

```
class Link extends React.Component {
  render() {
    return <a onClick={this.props.handleClick} href="#">{this.props.label}</a>
  }
}
```

And finally, the Logo component is also using the same properties and, you guessed it, also they are magically there because we used a spread operator when we create logo in content.jsx:

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

Each of the three components have different renderings, but they all get this.props.handleClick and this.props.label from LoadWebsite. The parent component Content renders the elements as illustrated in Listing 8-5.

Listing 8.5 Applying higher-order components to share the same event handler between Button, Link and Logo (ch08/hi-order/jsx/content.jsx)

```
const EnhancedButton = LoadWebsite(Button)
const EnhancedLink = LoadWebsite(Link)
const EnhancedLogo = LoadWebsite(Logo)

class Content extends React.Component {
  render() {
    return (
      <div>
        <EnhancedButton />
        <br />
        <br />
        <EnhancedLink />
        <br />
        <br />
        <EnhancedLogo />
        <br />
        <br />
        <iframe id="frame" src="" width="600" height="500"/> ①
      </div>
    )
  }
}
```

① Declare the iframe, in which the click method loads the React site

Finally, let's not forget to render Content on the last lines of script.jsx:

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

```
document.getElementById('content')
)
```

When you open the page, it'll have the three elements (Button, Link, and Logo) 8-X. The three elements (Button, Link, and Logo) will have the same functionality. They will load the iframe when click happens as shown in figure 8.10.

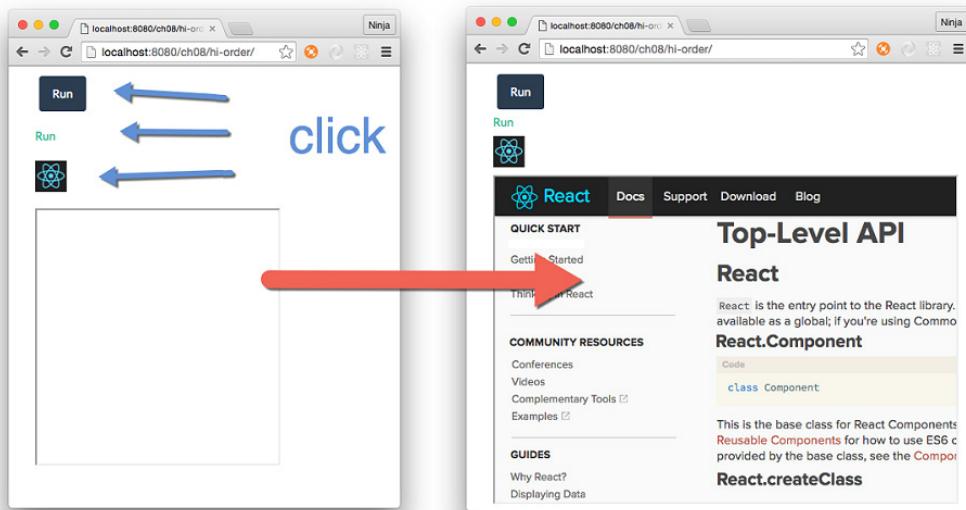


Figure 8.10 All three components will load React website thanks to the function which provided the code to load it

As you saw, HOC are great for abstracting code. You can use them to write your own mini-modules, which are re-useable React components. HOC, along with property types, are great tools for creating developer-friendly components that others will love to use.

8.5 Best Practices: Presentational vs. Container Components

There's a distinction which can allow to scale your React code in terms of code and team size: presentational vs. container components. We've touched on them in previous chapters briefly, but now since you know about passing all children and higher-order components, it'll be easier to reason about container components.

Generally speaking, by splitting your code into two type, it becomes more maintainable and simpler. Presentational components typically only add structure to DOM and styling. They take props but often don't have their own states. Most of the times, we can use functions for

stateless presentational components. For example, the `Logo` is a good illustration of a presentational component in class style:

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

Or in functional style:

```
const Logo = (props)=>{
  return 
}
```

Presentational component often use `this.props.children` when they act as a wrappers to style any children components. Examples are `Button`, `Content`, `Layout`, `Post`, etc. However, they rarely deal with data or states. That's the job of container components.

Container components are often generated by HOCs to inject data sources. They have states. Examples are `SaveButton`, `ImagePostContent`, etc. Both presentational and container components can contain other presentational or container components, but most of the times when you're starting you'd be with presentational components containing ONLY other presentational components. Container components will have either other container components or presentational ones.

The best approach is to start with components that solve your needs and once you start seeing some repeating patterns (layout?) or props which you are passing over multiple layers of nested components but not using in the interim components—introduce container component or two.

PS: You might hear terms such as dumb or skinny and smart or fat components which are just synonyms to presentational and container components.

8.6 Quiz

1. React provides robust validation, which eliminates the necessity to check input on the server side. True or false?
2. In addition to setting the properties with `defaultProps`, you can set them in constructor using `this.prop.NAME = VALUE`. True or false?
3. The `children` property can be an array or a node. True or false?
4. A higher-order component pattern is implemented via a function. True or false?
5. The main differences between minified development and unminified production versions of the React library file is that minified has warnings and unminified has optimized code. True or false? (true)

8.7 Summary

In this chapter you learned that

- You can define a default value for any component property by setting the component's `defaultProps` attribute.
- You can enforce validation checks on component property values while working with the uncompressed, development version of the React library.
- You can check the type of the property, set it to `isRequired` so it is mandatory, or define your own custom validation as required.
- If the property value fails validation, a warning appears in your browser's console.
- The minified, production version of the React library does not include these validation checks
- React allows you to encapsulate and reuse common properties, methods and events among your components by creating Higher-Order Components
- Higher-Order Components are defined as functions which take another component as an argument. This argument is the component inheriting from the HOC.
- Any HTML or React components nested within a JSX element can be accessed through the `props.children` property of the parent component.

8.8 Quiz Answers

1. False, because any front-end validation is not a substitute for the back-end validation. The front-end code is exposed to anyone and anyone can bypass it by reverse engineering how front-end app communicates with the server and send the any data directly to the server bypassing your front-end code.
2. False, because React needs the `defaultProps` as a static class field/attribute when an element is created, while `this.props` is an instance attribute.
3. True, if there's only one child then `this.props.children` is a single node.
4. True, HOC pattern is a function which takes a component and creates another component class with enhanced functionality. This new class renders the original component while passing props and states to it.

9

Project: Menu

In this chapter,

- Project Structure and Scaffolding
- Building the Menu without JSX
- Building the Menu in JSX

The next three chapters will walk you through projects gradually building on the concepts learned in chapters 1-8. These projects will also re-enforce the material by repeating some of the most important to React techniques and ideas. The first project will be minimal, but don't skip it.

Imagine you are working on a unified visual framework which will be used in all apps of your company. Having the same look and feel in various apps is important. Think how Google's material UI is used across many properties which belong to Google: AdWords, Analytics, Search, Drive, Docs, etc.

Your first task is to implement a menu (maybe like the one shown in Figure 9-1). It will be used in the layout's header across many pages of various applications. The menu items needs to changes based on the user role or what part of the application is currently being viewed. For example, admins and managers should see a menu option "Manage Users". At the same time, this layout will be used in a customer relationship which needs its own unique set of menu options. You got the idea. You'll need to have this menu generated dynamically meaning you will have some React code which generates menu options.

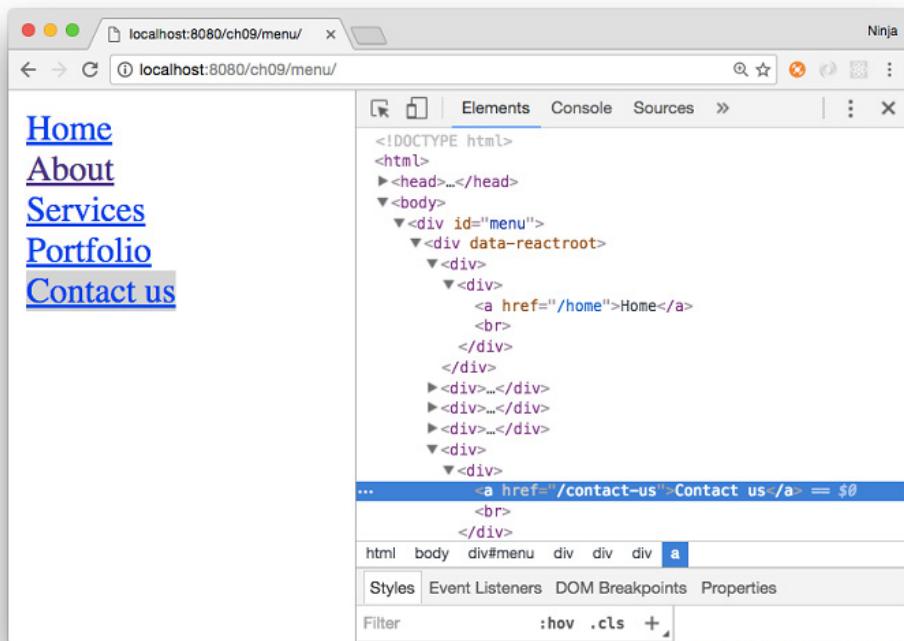


Figure 9.1: Menu

For simplicity, the menus will be just `<a>` tags. To do it, we'll create two custom React components, named `Menu` and `Link`. We create them in a way that is similar to the way we created the `HelloWorld` component in the chapter 1 (or any other component for that matter).

The project will show you how to render programmatically nested elements because manually hard-coding menu items is not a great idea (what happens when you need to change an item?—it's not dynamic!). We'll use the `map()` function for this.

The source code for the examples in this chapter is in [the ch09 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

9.1 Project Structure and Scaffolding

Firstly, we should start with an overview of the project structure which is very flat to keep it simple.

```
/menu
  index.html
  package.json
  react-dom.js
  react.js
  script.js
```

① ②

- ① The main HTML file
- ② The main script

Keep in mind, this is what you will get by the end of this walk-through. You should start with an empty folder. So let's create a new folder and start implementing our project:

```
$ mkdir menu-jsx
$ cd menu-jsx
```

Download `react.js` and `react-dom.js` of version 15 and drop 'em into the folder. Next is the HTML file.

NOTE To follow along with the project, you'll need to download the *unminified* version of React (for the helpful warnings it returns if something goes wrong). You can also download and install node.js and npm. They aren't strictly necessary for this project yet, but are useful for compiling JSX later in this chapter. As always, Appendix A covers installation of both tools.

The HTML for this project is very basic. It includes the `react.js` and `react-dom.js` files that, for simplicity, are in the same folder as the HTML file. Of course, later on you'll want to have your `*.js` files in some other folder like `js` or `src`:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="react.js"></script>
    <script src="react-dom.js"></script>
  </head>
```

The body has just two elements. One element is a `<div>` container with the ID `menu`. That's where our menu will be rendered. The second element is a `<script>` tag with our React application code:

```
<body>
  <div id="menu"></div>
  <script src="script.js"></script>
</body>
</html>
```

We are done with scaffolding. This is our foundation on which we'll build the menu... without JSX first.

9.2 Building the Menu without JSX

The `script.js` is our main application file. It will contain the main `ReactDOM.render()` as well as two components, and it looks like this:

Listing 9.1 Basic Skeleton of the Menu Script (ch09/menu/script.js)

```
class Menu extends React.Component {...}           ①
class Link extends React.Component {...}

ReactDOM.render(
  React.createElement(
    Menu,
    null           ②
  ),
  document.getElementById('menu')
)
```

- ① Define Menu
- ② Define Link which will be used by Menu
- ③ Don't pass any props to Menu

Of course, it's possible to make `Menu` dependent on an external list of menu items provided in a property such as `menuOptions` will be defined somewhere else:

```
const menuOptions = [...]
...
ReactDOM.render(
  React.createElement(
    Menu,
    {menus: menuOptions}
  ),
  document.getElementById('menu')
)
```

These two approaches are both right and developers will need to pick one or the other depending on the question: do they want `Menu` to be just about structure and styling or also about getting the information. We will continue with the latter approach in this chapter (making `Menu` self-sustained).

9.2.1 The Menu Component

Now to create the main component `Menu`. Let's step through the code. To create it, we extend `React.Component()`:

```
class Menu extends React.Component {...}
```

The `Menu` component will be rendering the individual menu items, which are link tags. Before we can render them, let's define the menu items. The menu items are hard-coded in the `menus` array like this (you can get them from a data model, store or a server in a more complex scenario):

```

    render() {
      let menus = ['Home', ❶
        'About',
        'Services',
        'Portfolio',
        'Contact us']
      ...
    }
  
```

❶ Mock data store

We'll use a `map()` function from the `Array` interface to produce four `Link` components. Remember that the `render` method must return a single element. For this reason, we use `div` to wrap around our links.

```

    return React.createElement('div',
      null,
      ...
    )
  
```

It's worth mentioning that `{}` can output not just a variable or expression, but an array too. This comes in handy when we have a list of items. So, basically, to render every element of an array, we can pass that array to `{}`. While JSX and React can output arrays, they won't output objects. So the objects must be converted to array.

Knowing that we can output an array we can proceed to generating this array. It will consist of React elements. The `map()` function is a good method to use because it returns an array. We can implement `map()` so that each element is the result of the expression `React.createElement(Link, {label: v})` wrapped in `<div>`. In this expression, `v` is a value of the `menus` array item (`Home`, `About`, `Services`, ...) while `i` is its index number (`0,1,2,3,...`).

```

    menus.map((v, i) => {
      return React.createElement('div',
        {key: i},
        React.createElement(Link, {label: v})
      )
    })
  
```

Did you notice the `key` prop set to the index? This is needed so React can access each `<div>` element in a list faster. If you don't set `key`, then there would be a warning (at least in React 15, 0.14 and 0.13):

WARNING Each child in an array or iterator should have a unique "key" prop. Check the render method of `Menu`. See <https://fb.me/react-warning-keys> for more information. in `div` (created by `Menu`) in `Menu`

Again, kudos to React for good error and warning messages. :) So each element of a list must have a unique value of a `key` attribute. They don't have to be unique across the entire app and other components, just within this list. Interestingly, since React v15, you won't see

the `key` attributes in HTML (and that's a good thing-let's not pollute HTML). However, React DevTools will show the keys to you as captured in Figure 9-2.

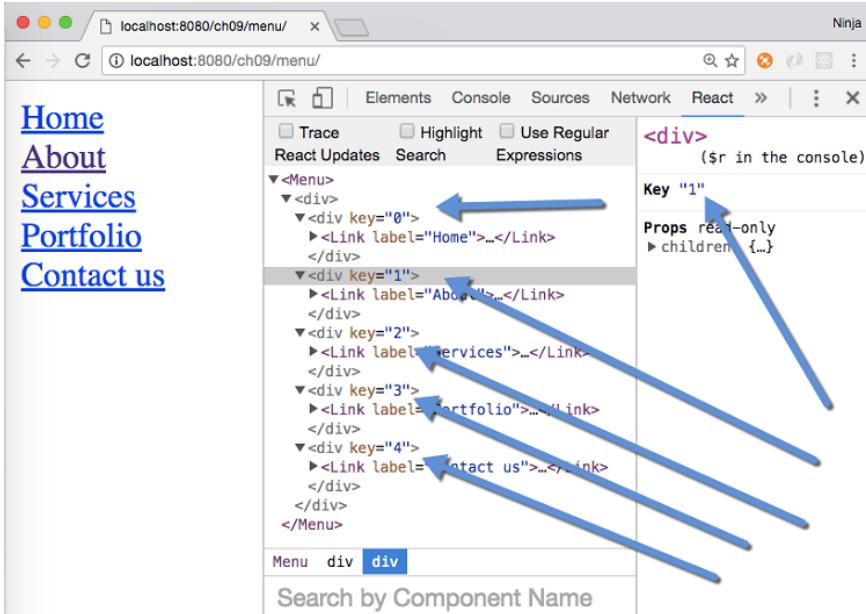


Figure 9-2: React DevTools shows you keys of the list elements

Array.map()

The mapping function from the `Array` class will be used in React components to represent list of data very often so it's worth writing a sidenote on the `map()` and its usage. This is because when developers create UIs, they do so from the data represented as an array. The UI is also an array but with slightly different elements (React elements!).

So `map()` is invoked on an array and it return a new array elements which are transformed from the original array by the function. At the very minimum when working with `map()`, developers need to pass this function:

```
[1, 2, 3].map( value => <p>value</p>) // <p>1</p><p>2</p><p>3</p>
```

There are two more arguments which can be used in this function in addition to the value of the item (`value`): `index` and `list` such so:

```
[1, 2, 3].map( (value, index, list) => {
  return <p id={index}>{list[index]}</p>
}) // <p id="0">1</p><p id="1">2</p><p id="2">3</p>
```

So the `<div>` has a `key` attribute, which is important. It allows React to optimize rendering of lists by converting them to hashes (access time for hashes is better than that of lists or arrays). Basically, we create numerous Link components in an array, and each one of them takes the property `label` with a value from the `menus` array.

Here's the full code of `Menu` for your reference – it's really simple and straightforward.

Listing 9.2 The Menu Component in full (ch09/menu/script.js)

```
class Menu extends React.Component {
  render() {
    let menus = ['Home',
      'About',
      'Services',
      'Portfolio',
      'Contact us']
    return React.createElement('div',
      null,
      menus.map((v, i) => {
        return React.createElement('div',
          {key: i},
          React.createElement(Link, {label: v})
        )
      })
    )
  }
}
```

And we are moving to the `Link` implementation.

9.2.2 The Link Component

So the call to `map()` creates a `Link` component for each item in the `menus` array. Let's have a look at the code for the `Link` component and see what happens when each `Link` component is rendered.

In the `Link` component's `render` code, we write an expression to create a URL. That URL will be used in the `href` attribute of the `<a>` tag. The `this.props.label` value is passed from the `map`'s closure in the `Menu` render function `return React.createElement(Link, {label: v})`:

```
class Link extends React.Component {
  render() {
    const url='/'
      + this.props.label
      .toLowerCase()
      .trim()
      .replace(' ', '-')
  }
}
```

The `methods` `toLowerCase()`, `trim()`, and `replace()` are standard JavaScript string functions. They perform conversion to lowercase, trimming of white space on the edges, and replacing of white spaces with dashes, respectively.

The URL expression produces the following URLs:

- /home for Home
- /about for About
- /services for Services
- /portfolio for Portfolio
- /contact-us for Contact us

In the render's return of the Link component, we pass `this.props.label` as a third argument to `createElement()`, and it becomes part of the `<a>` tag content (that is, text of the link).

To separate each link, we add a link break tag: `
`. Because the component must return only one element, we wrap `<a>` and `
` in `<div>`:

```
return React.createElement('div',
  null,
```

Each argument after the second to `createElement()` (for example, the third, fourth, and fifth) will be used as content (a.k.a. children). To create the link element, we pass it as the second argument. And to create a break element after each link, we pass the line break element `
` as the fourth argument:

```
React.createElement(
  'a',
  {href: url},
  this.props.label
),
React.createElement('br')
)
}
})
```

Listing 9.3 Link component in full (ch09/menu/script.js)

```
class Link extends React.Component {
  render() {
    const url= '/' ①
      + this.props.label
      .toLowerCase()
      .trim()
      .replace(' ', '-')
    return React.createElement('div',
      null,
      React.createElement(
        'a',
        {href: url}, ②
        this.props.label
      ),
      React.createElement('br') ③
    )
  }
}
```

① Define a function which create URL fragments out of the menu names

- ② Pass the URL fragment to set the `href` attribute
- ③ Add line break element to separate a new menu item

Let's get it running.

9.2.3 Getting It Running

To view the page (Figure 9-3), open it as a file in Chrome, Firefox, Safari, or (maybe) Internet Explorer. The protocol in the address bar will say `file://....` This is not ideal but will do for this project. For real development, you will need a web server. With a web server the protocol will be `http://....` or `https://....` as I have in Figure 9-3.

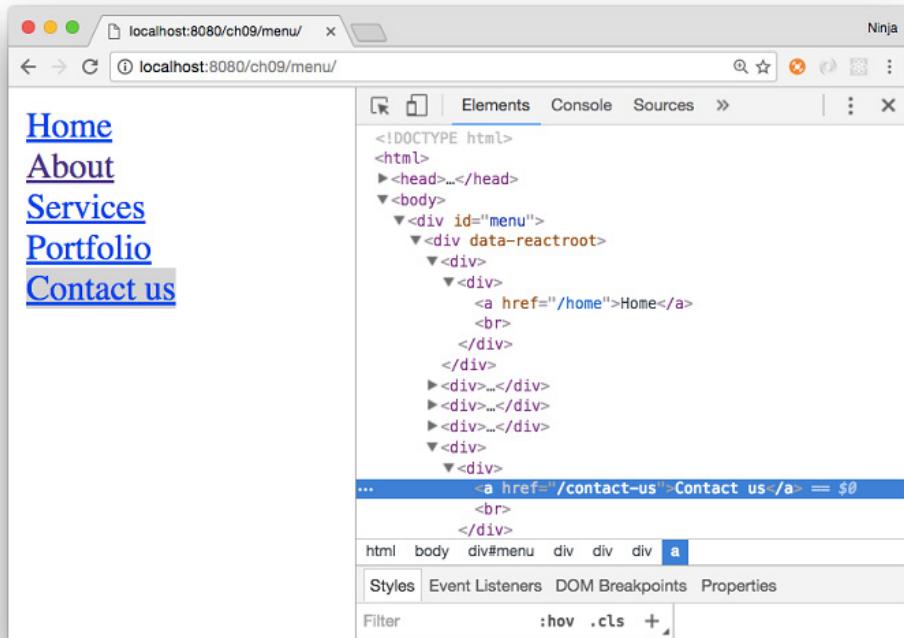


Figure 9-3: React Menu showing rendering of nested components

Yes. Even for these simple web pages like you see in the Menu project, I like to use a local web server. It makes running code more closely resemble how you'd do it in production. Plus, it allows you to use AJAX/XHR, which you can't use if you're just opening an HTML file in a browser.

The easiest way to run a local web server is to use [node-static](#) or [a similar Node.js tool](#). This is true even for Windows although I stopped using that OS many years ago. If you are hellbent on not using Node, then alternatives will include IIS, Apache httpd, nginx, MAMP, LAMP and other variations of web servers. Needless to say, Node tools are highly recommended for their minimal lightweight approach. To install `node-static`, use npm:

```
$ npm install -g node-static@0.7.6
```

Once it's installed, run this command from your project's root folder (or from a parent folder) to make the file available on <http://localhost:8080> (this is not an external link, run the command below first before clicking the link):

```
$ static
```

No compilation is needed for this project. To make my point clear, if you run `static` in `react-quickly/ch09/menu`, then the URL will be <http://localhost:8080>. Conversely, if you run `static` from `react-quickly`, then the URL needs to be <http://localhost:8080/ch09/menu>.

To stop the server on Mac OS X or Unix/Linux (Posix systems), simply press control + c. As for Windows, I don't know. :) TK

That's it. *No compilation is needed for this project.* No thrills here, but the page should display five links (or more if you add more items to the `menus` array). The result was shown in Figure 9-1. This is much better than copying and pasting five `<a>` elements and then ending up with multiple places to modify the labels and URLs.

It can be even better with JSX.

9.3 Building the Menu in JSX

The project will be more extensive containing `node_modules`, `package.json` and `JSX`:

```
/menu-jsx
  /node_modules          ①
  index.html
  package.json
  react-dom.js
  react.js
  script.js
  script.jsx            ②
```

- ① Babel dev dependency for JSX to JS transpilation
- ② The main JSX script

While it's possible to install `react` and `react-dom` as npm modules instead of having them as files (that's what we will do in Part 2), it will require additional complexity if you decide to deploy. You see, right now to deploy this app, you can just copy the files in the project folder without `node_modules`. If you install React and ReactDOM with npm, then you would have to either include that folder as well or use some bundler, or copy the JS files from `dist` into root

(where we already have them). Therefore, let's just have files in the root and not use npm for React and React DOM just yet. We will cover bundlers in Part 2 but for now let's keep things stupid simple.

As you can see, there's `node_modules` folder for developer dependencies such as Babel which is used for JSX to JS transpilation. Also, there's `node_modules` folder for developer dependencies such as Babel which is used for JSX to JS transpilation.

Create a new folder:

```
$ mkdir menu-jsx
$ cd menu-jsx
```

And create `package.json` in it with `npm init -y`. Add this to `package.json` to install and configure Babel:

Listing 9.4 `package.json` for Menu JSX (ch09/menu-jsx/package.json)

```
{
  "name": "menu-jsx",
  "version": "1.0.0",
  "description": "",
  "main": "script.jsx",
  "scripts": {
    "build": "./node_modules/.bin/babel script.jsx -o script.js -w" ①
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": ["react"] ②
  },
  "devDependencies": { ③
    "babel-cli": "^6.9.0",
    "babel-preset-react": "^6.5.0"
  }
}
```

- ① Define a build script with the watch flag
- ② Configure babel to transpile React's JSX
- ③ Include Babel CLI as well as React/JSX preset

Install the developer dependencies packages with `npm i` or `npm install`. Our setup should be ready now. Let's take a look at `script.jsx`. At a higher level it will have these parts:

```
class Menu extends React.Component {
  render() {
    ...
  }
}

class Link extends React.Component {
  render() {
    ...
  }
}
```

```

}

ReactDOM.render(<Menu />, document.getElementById('menu'))

```

Looks familiar? Right. It's the same *structure* as in Menu without JSX.

The main change in the high-level listing above is to replace `createElement()` for the `Menu` component in `ReactDOM.render()` with this line:

```
ReactDOM.render(<Menu />, document.getElementById('menu'))
```

Next, we will refactor the components.

9.3.1 Refactoring the Menu Component

The beginning of `Menu` is the same:

```

class Menu extends React.Component {
  render() {
    let menus = ['Home',
      'About',
      'Services',
      'Portfolio',
      'Contact us']
    return ...
  }
}

```

In the refactoring example of the `Menu` component, we need to output the value `v` as a `label`'s attribute value (that is, `label={v}`). In other words, we are assigning the value of `v` as a property for `label`. So, the line to create the `Link` element changes from

```
React.createElement(Link, {label: v})
```

to this JSX code:

```
<Link label={v}/>
```

The `label` property of the second argument (`{label: v}`) becomes the attribute `label={v}`. The attribute's value `v` is declared with `{}` to make it dynamic (versus a hard-coded value). *Please note, when you use curly braces to assign property values, you don't need double quotes ("").*

Also, React needs the `key={i}` attribute to access the list more efficiently. Therefore, in the end, the `Menu` component is restructured as this JSX code:

Listing 9.5 Menu with JSX (ch09/menu-jsx/script.jsx)

```

class Menu extends React.Component {
  render() {
    let menus = ['Home',
      'About',
      'Services',
      'Portfolio',
      'Contact us']
    return (
      <ul>
        {menus.map((v, i) =>
          <li key={i}>
            <Link label={v}>{v}</Link>
          </li>
        )}
      </ul>
    )
  }
}

```

```

    'Portfolio',
    'Contact us']
return <div>
  {menus.map((v, i) => {
    return <div key={i}><Link label={v}/></div>
  })}
</div>
}

```

Do you see the increase in readability? I do!

In Menu's `render()`, if you prefer to start the `<div>` on a new line, you can do so by putting `()` around it. For example, this code is identical to the snippet above, but `<div>` starts on a new line, which might be more visually appealing:

```

...
return (
  <div>
    {menus.map((v, i) => {
      return <div key={i}><Link label={v}/></div>
    })}
  </div>
)
}

```

9.3.2 Refactoring the Link Component

The `<a>` and `
` tags in the Link component also need refactoring from this:

```

...
return React.createElement('div',
  null,
  React.createElement(
    'a',
    {href: url},
    this.props.label),
  React.createElement('br')
)
...

```

to this JSX code:

```

return <div>
  <a href={url}>
    {this.props.label}
  </a>
  <br/>
</div>

```

So the entire JSX version of the Link component can look something like this:

```

class Link extends React.Component {
  render() {
    const url='/' +
      this.props.label
      .toLowerCase()
      .trim()

```

```
    .replace(' ', '-')
  return <div>
  <a href={url}>
    {this.props.label}
  </a>
  <br/>
</div>
}
```

Phew. We are done! Let's run the JSX project.

9.3.3 Getting it Running

Open your Terminal, iTerm or Command Prompt app. In the project's folder (`menu-jsx`), run the build script with `npm run build`. It will keep running to watch for any file changes and re-compile. You can refer to Chapter 3 for more details on Babel CLI.

On my computer I got this message from Babel CLI (on yours, the path will differ):

```
> menu-jsx@1.0.0 build /Users/azat/Documents/Code/react-quickly/ch09/menu-jsx
> babel script.jsx -o script.js -w
```

Now, we are good to go. With `script.js` generated, you can use `static` to serve the files over http on localhost. The application should look and work exactly like its regular JavaScript brethren (Figure 9-4).

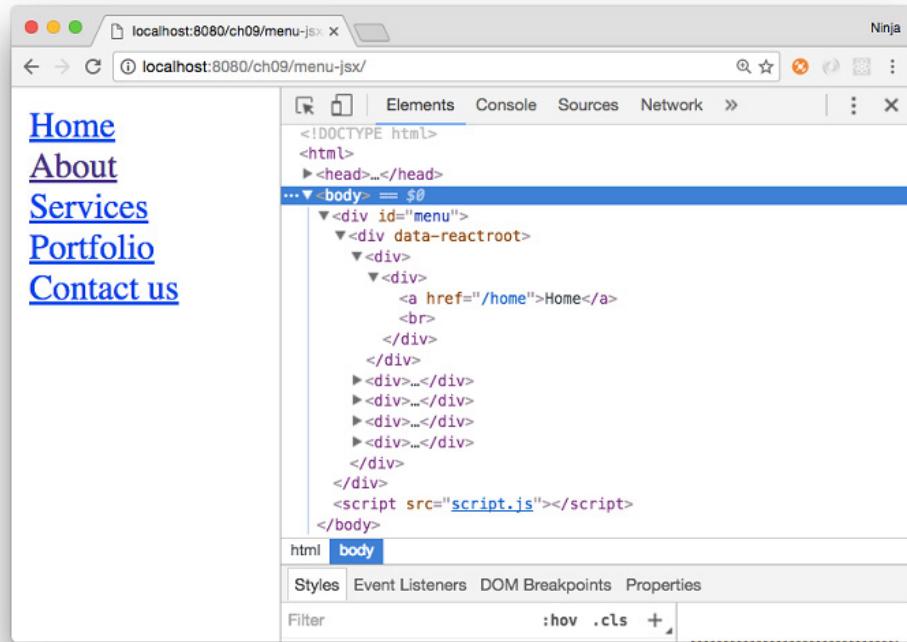


Figure 9-4: Menu with JSX

9.4 Homework

Bonus points,

- Load `menu` from `menus.json` via fetch API. See chapter 5 for inspiration on how to load data.
- Create an npm script which will grab `react.js` from `react` npm package installed in `node_modules` and copy it into the project folder to be used by `index.html`. This will replace the need of manually downloading `react.js` for future versions by using `npm i react` and then running your script.

Submit your code in a new folder under `ch09` as a pull request to this book's GitHub repository: <https://github.com/azat-co/react-quickly/>

9.5 Summary

- `key` is your friend. Set this attribute when generating lists

- `map()` is an elegant way to create a new array based on the original array. Its iterator arguments are `value`, `index` and `list`.
- For JSX to work, at a bare-minimum developers need Babel CLI, and React presets.

10

Project: Tooltip

In this chapter,

- Project Structure and Scaffolding
- The Tooltip Component
- Getting It Running

When working on websites which have a lot of text such as Wikipedia for example, it's a great idea to allow users to get some additional information without the loss of the position and context. Give them extra hint on the cursor hover!

React is all about UI and a better user experience, so let's build a component to display some helpful text (a tooltip) on a mouse-over event (Figure 10-1). While there are many good out-of-the-box solutions (like [react-tooltip](#)), the goal here is to learn React. Building a tooltip from scratch is a good exercise. Maybe you can extend my example to your liking and make it a part of your app or even a new open-source React component!

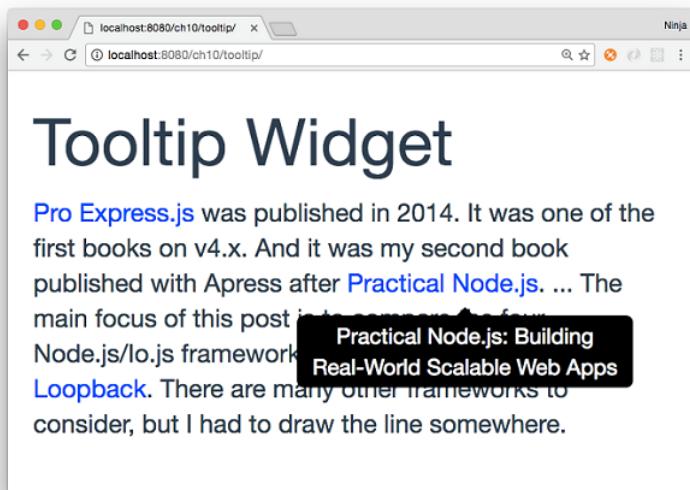


Figure 10-1: Tool tip text appear when users hover their cursors over the marked text

The key to creating the Tooltip component is to be able to take any text, hide it with CSS and then make it visible again on mouse over. We'll use if/else conditions, JSX, and other programming elements from this project. For the CSS part, we'll be using Twitter Bootstrap classes (and a special TB theme) to make the tooltip look nice in very little amount of time.

NOTE To follow along with the project, you'll need to download the unminified version of React and install node.js and npm for compiling JSX. In this example, I'm also using a Twitter Bootstrap theme called [Flatly](#) from [Bootswatch](#). This theme depends on Twitter Bootstrap. Appendix A covers installation of everything.

The source code for the examples in this chapter is in [the ch10 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

10.1 Project Structure and Scaffolding

The project structure for Tooltip is as follows:

```
/tooltip
  /node_modules
  bootstrap.css
  index.html
  package.json
  react-dom.js
  react.js
```

```
script.js
script.jsx
```

②

- ① Babel dev dependency for JSX to JS transpilation
- ② The main JSX script

As before, there's `node_modules` folder for developer dependencies such as Babel which is used for JSX to JS transpilation. The structure is flat with styles and scripts in the same folder. I did this to keep everything simple. Of course, in the real app developers put styles and scripts into separate folder.

The key parts in `package.json` are npm script to build, Babel configuration and dependencies. My `package.json` looks like this:

```
{
  "name": "tooltip",
  "version": "1.0.0",
  "description": "",
  "main": "script.jsx",
  "scripts": {
    "build": "./node_modules/.bin/babel script.jsx -o script.js -w"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": ["react"]
  },
  "devDependencies": {
    "babel-cli": "^6.9.0",
    "babel-preset-react": "^6.5.0"
  }
}
```

After you have `package.json`, make sure to run `npm i` or `npm install`.

Starting with the HTML, create `index.html`. Here it is in full:

Listing 10.1 (ch10/tooltip/index.html)

```
<!DOCTYPE html>
<html>

  <head>
    <script src="react.js"></script>
    <script src="react-dom.js"></script>
    <link href="bootstrap.css"          ①
          rel="stylesheet"
          type="text/css"/>
  </head>

  <body class="container">
    <h1>Tooltip Widget</h1>
    <div id="tooltip"></div>          ②
    <script src="script.js" type="text/javascript"></script>
  </body>
```

```
</html>
```

- ➊ Apply styles
- ➋ Define the render element for React and out Tooltip

In the `<head>`, include React, React DOM files, and Twitter Bootstrap styles. And the `body` is minimal. It contains the `div` with the ID `tooltip` and the application's `script.jsx` file.

Next, we'll create `script.jsx`. That's right, it's not a typo. The source code will be in `script.jsx`, but we include the `script.js` file in our HTML. That's because we'll be using the command-line Babel tool.

10.2 The Tooltip component

So let's see `script.jsx`. It's pretty much just the code for the component and the tooltip text we want to render. The tooltip text is a property which we set when we create `Tooltip` in `ReactDOM.render()`.

Listing 10.2 Component and Text (ch10/tooltip/script.jsx)

```
class Tooltip extends React.Component {
  constructor(props) {
    ...
  }
  toggle() { ➊
    ...
  }
  render() { ➋
    ...
  }
}

ReactDOM.render(<div>
  <Tooltip text="The book you're reading now">React Quickly</Tooltip> ➌
  was published in 2017. Its awesome!
</div>,
  document.getElementById('tooltip'))
```

- ➊ Declare a method to show and hide the help text
- ➋ Declare mandatory render method
- ➌ Provide help text as a prop while the content will be the highlighted text to hover the cursor over

Let's implement `Tooltip` and declare the `Tooltip` component with the initial state of `opacity: false`. This will command the hide or show of the help text. (We covered states in more detail in chapter 4.)

Take a look at the method `constructor()` in action.

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

```

    this.state = {opacity: false}
    this.toggle = this.toggle.bind(this)
  }
  ...
}

```

Initial state will determine hide the help text. Toggle will be changing this state and the visibility of the tooltip (that is, whether the help text is shown or not). So let's implement `toggle()`.

10.2.1 The `toggle()` Function

Next, we define the `toggle` function to switch the visibility of the tooltip by changing the state `opacity` to the opposite of what it was before (that is, true to false and false to true).

We will be utilizing the method `this.setState()` which we learned in chapter 4 to change `opacity`. `tooltipNode` will be used in bit.

```

toggle() {
  const tooltipNode = ReactDOM.findDOMNode(this)
  this.setState({
    opacity: !this.state.opacity,
    ...
  })
}

```

A tricky thing with tooltip help text is that we must place the help text close to the element the mouse is hovering over. To do so, we need to get the position of the component using `tooltipNode`.

To position the tooltip text, we use `offsetTop` and `offsetLeft` on the DOM node. These are [DOM Node](#) properties from the HTML standard (not a *React thing*):

```

    top: tooltipNode.offsetTop,
    left: tooltipNode.offsetLeft
  })
},

```

Here it is in full code for the `toggle()`:

Listing 10.3 The `toggle()` function (ch10/tooltip/script.jsx)

```

toggle() {
  const tooltipNode = ReactDOM.findDOMNode(this)
  this.setState({
    opacity: !this.state.opacity,
    top: tooltipNode.offsetTop,
    left: tooltipNode.offsetLeft
  })
}

```

Looking at the code, we can see that it changes the state and position. Do we need to re-render the view now? No, because React will update the view for us. `setState()` will invoke

re-reader automatically. It might or might not result in the actual DOM changes depending on whether the state was used in `render()`... which we will be Implementing next!

10.2.2 The `render()` function

The `render()` function will hold the CSS style object for the help text as well as Twitter Bootstrap styles. So first let's define this `style` object. We set the `opacity` and `z-index` CSS styles depending on the value of the `this.state.opacity`. We do need `z-index` to float the help text above any other elements so we will set the value reasonably high at 1000. We set the value to 1000 when it's visible and to -1000 when it's not visible:

```
zIndex: (this.state.opacity) ? 1000 : -1000,
```

For more clarity, take a look at Figure 10-2. It shows how the styles will be applied with mouse over (opacity is true).

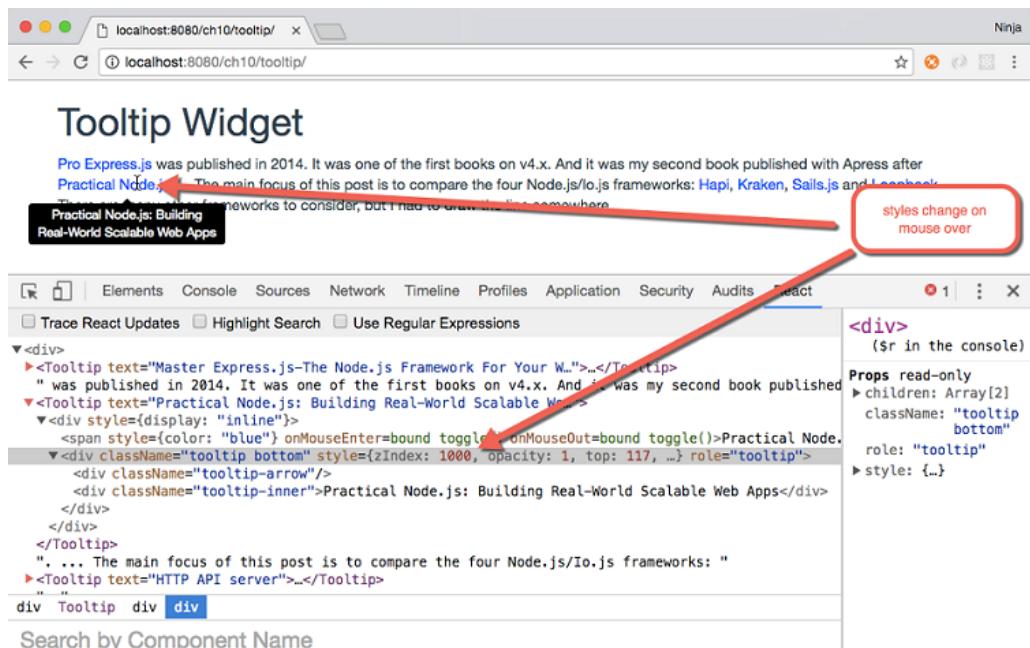


Figure 10.2 Help text is shown on mouse over by using opacity 1 and z-index 1000.

For `z-index`, we need to use `zIndex` (note the camelCase). Remember to use camelCase with React instead of dash-syntax. In other words, the CSS property `z-index` becomes React style property `zIndex`; `background-color` becomes `backgroundColor`; `font-`

family becomes `fontFamily`, etc. By using valid JavaScript names, React can update real DOM from the virtual one faster.

The state `opacity` `this.state.opacity` is a boolean true or false, but the CSS opacity is binary 0 or 1. If state `opacity` is false, the CSS opacity is 0 (zero) and if it's true, the CSS opacity is 1. We need to convert. Putting a binary operator `+` will do the conversion:

```
opacity: +this.state.opacity,
```

As far as the position of the tooltip goes, we want to place the help text near the text where the mouse is hovering over by adding 20px to the top (which is the distance from the top edge of the window to the element) and by subtracting 30px from the left (which is the distance from the left edge of the window to the element). The values were chosen visually; feel free to adjust the logic as you see fit:

```
render() {
  const style = {
    zIndex: (this.state.opacity) ? 1000 : -1000,
    opacity: +this.state.opacity,
    top: (this.state.top||0) + 20,
    left: (this.state.left||0) -30
  }
}
```

Next is the `return`. The component will render both the text on which to hover and the help text. I'm using Twitter Bootstrap classes along with my `style` object to hide the help text and to show it later.

The text on which users can hover to see a tooltip is colored blue just so we can visually tell it apart from other text. It has two mouse events for when the cursor enters the `span` and when it leaves it:

```
return (
  <div style={{display: 'inline'}}>
    <span style={{color: 'blue'}}
      onMouseEnter={this.toggle}
      onMouseOut={this.toggle}>
      {this.props.children} ①
    </span>
  </div>
  ① Output whatever inner HTML will be passing to `Tooltip` later
```

This is the actual help text code. It's very static-like, except for the `{style}`. React will change the state, and it will trigger the change in the UI.

```
<div className="tooltip bottom" style={style} role="tooltip"> ①
  <div className="tooltip-arrow"></div> ②
  <div className="tooltip-inner">
    {this.props.text} ③
  </div>
</div>
</div>
)
```

```

}

① Apply the style object to style attribute
② Use arrow class for the a pointy arrow
③ Output the text of the tooltip from the text property {this.props.text}

```

Tooltip `render()` looks like this in full.

Listing 10.4 The render function in full

```

render() {
  const style = {
    zIndex: (this.state.opacity) ? 1000 : -1000,
    opacity: +this.state.opacity,
    top: (this.state.top || 0) + 20,
    left: (this.state.left || 0) - 30
  }
  return (
    <div style={{display: 'inline'}}>
      <span style={{color: 'blue'}}
        onMouseEnter={this.toggle}           ①
        onMouseOut={this.toggle}>
        {this.props.children}               ②
      </span>
      <div className="tooltip bottom"
        style={style}                     ③
        role="tooltip">
        <div className="tooltip-arrow"></div>
        <div className="tooltip-inner">
          {this.props.text}                ④
        </div>
      </div>
    )
}

```

- ① Trigger show on mouse enter
- ② Output any text passed at content of Tooltip
- ③ Apply styles with opacity, zIndex, and proper position based on the position of the DOM node
- ④ Output the help text using Twitter Bootstrap classes

That's it. We are done with the Tooltip component!

10.3 Getting It Running

Try it out or use it in your projects by compiling JSX with npm:

```
$ npm run build
```

This Tooltip is pretty cool thanks to Twitter Bootstrap styles. Maybe it's not as versatile as some of other modules out there, but we built it *ourselves from scratch*. That's what I'm talking about! With the help of Twitter, Bootstrap classes, and React, we were able to create a

good tooltip (figure 10-3) in almost no time! It's even responsive (as it adapts to various screen sizes, thanks to dynamic positioning!) Yay!

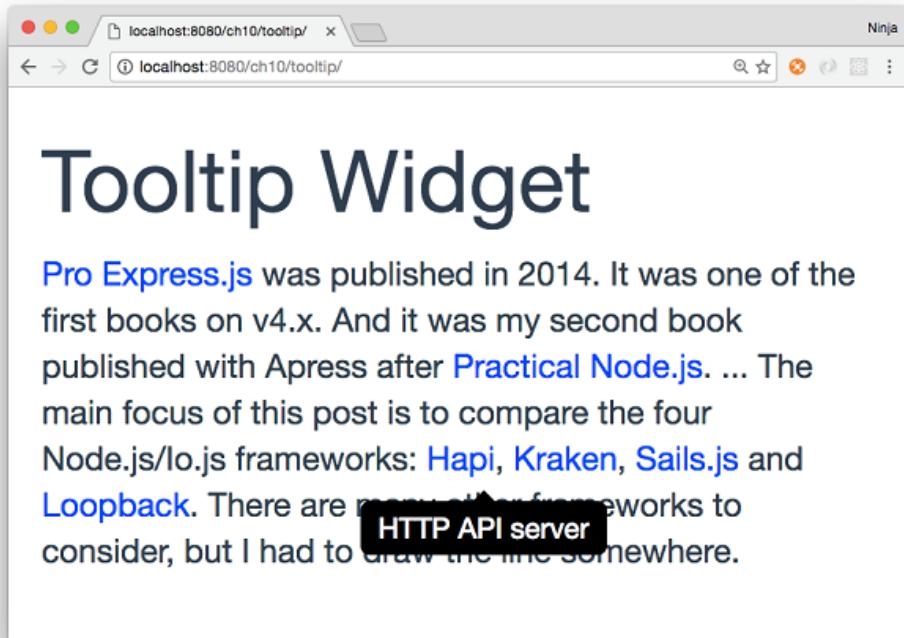


Figure 10.3 When hovering on blue text, the black container with text and a pointy arrow will appear offering additional info to users.

10.4 Homework

For bonus points,

- Create a variation which works on mouse click, i.e., shows when you click on the highlighted text and hides when you click again.
- Enhance Tooltip by making it take a property which will determine whether it's on mouse over or on click behavior.
- Enhance Tooltip by making it take a property which will position help text above the text instead of the default position below the text (hint: change TB class and change top and left).

Submit your code in a new folder under `ch10` as a pull request to this book's GitHub repository: <https://github.com/azat-co/react-quickly/>

10.5 Summary

- React style properties are camelCase unlike CSS style properties
- `this.props.children` has the content of the component
- No need to manually re-render because React will automatically re-render after `setState()`

11

Project: Timer

In this chapter,

- Project Structure and Scaffolding
- App Architecture
- The TimerWrapper Component
- The Timer Component
- The Button Component
- Get It Running

Studies have shown that meditation is great for health (calming) and productivity (focus). Who doesn't want to be healthier and more productive especially with very minimal monetary investment?!

Gurus recommend starting with as little as 5 minutes and progressing to 10, then to 15 in a span of a few weeks. 30-60 minutes of meditation per day is a target, but some people will notice improvements even with as little as 10 min per day. I can attest to that after meditating 10 minutes per day every day for 3 years it helped me to be more focus as a well as in other areas.

But how do you know when it's time to break for a day? When you got to your daily goal be it 5 or 10 minutes? You need a timer! So let's put our React and HTML5 skills to test and create a web timer (Figure 11-1). We'll make it easy on ourselves for testing purposes and only create a timer for 5, 10, and 15 seconds. :)

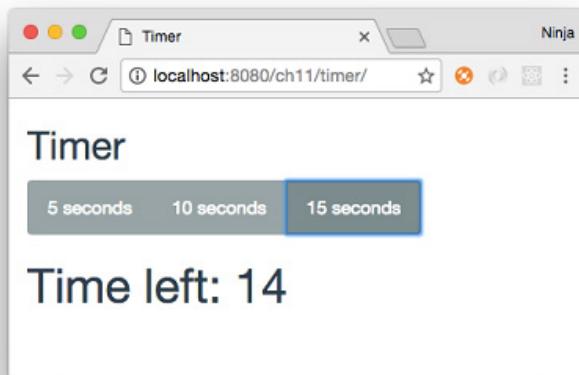


Figure 11.1 Timer in action shows 14 second left and the "pressed" 15 second button which was just pressed a second ago

The idea is to have three controls to set a countdown timer (n to 0). Think about your typical kitchen or meditation timer, but instead of minutes there'll be seconds. Click on a button, the time starts. Click on it or another button and the time will start over again.

NOTE To follow along with the project, you'll need to download the unminified version of React and install node.js and npm for compiling JSX. In this example, I'm also using a theme called Flatly from Bootswatch. This theme depends on Twitter Bootstrap. Appendix A covers installation of everything.

The source code for the examples in this chapter is in [the ch11 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

11.1 Project Structure and Scaffolding

The project structure for Timer not unlike Tooltip or Menu and is as follows:

```
/timer
  /node_modules           ①
  bootstrap.css
  flute_c_long_01.wav    ②
  index.html
  package.json
  react-dom.js
  react.js
  timer.js
  timer.jsx              ③
```

① Babel dev dependency for JSX to JS transpilation

② The sound file to signal the end of the time

③ The main JSX script

As before, there's `node_modules` folder for developer dependencies such as Babel which is used for JSX to JS transpilation. The structure is flat with styles and scripts in the same folder. I did this to keep everything simple. Of course, in the real app developers put styles and scripts into separate folder.

The key parts in `package.json` are npm script to build, Babel configuration and dependencies. My `package.json` looks like this:

```
{
  "name": "timer",
  "version": "1.0.0",
  "description": "",
  "main": "script.js",
  "scripts": {
    "build": "./node_modules/.bin/babel timer.jsx -o timer.js -w" ❶
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": ["react"]
  },
  "devDependencies": {
    "babel-cli": "^6.9.0",
    "babel-preset-react": "^6.5.0"
  }
}
```

❶

① Create an npm script to transpile JSX into JS

After you have `package.json` either by copy-pasting or by typing, make sure to run `npm i` or `npm install`.

The HTML for this project is very basic. It includes the `react.js` and `react-dom.js` files which, for simplicity, are in the same folder as the HTML file.

Listing 11.1 index.html in full (ch11/timer/index.html)

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Timer</title>
    <script src="react.js" type="text/javascript"></script>
    <script src="react-dom.js" type="text/javascript"></script>
    <link href="bootstrap.css" rel="stylesheet" type="text/css"/>
  </head>
  <body class="container-fluid">
    <div id="timer-app"/>
  </body>
  <script src="timer.js" type="text/javascript"></script>
</html>
```

This file will only include the library and point to `timer.js`, which we are going to create from `timer.jsx`. For that, you'll need Babel CLI (see chapter 3).

11.2 App Architecture

The `timer.jsx` will have three components:

- `TimerWrapper`--the main component that will do most of the things and render other components
- `Timer`--the component to display the seconds left
- `Button`--the component to render three buttons and trigger the timer (reset it)

This is how they will look on the page. on Figure 11-2 you can see `Timer` and `Button` components. `TimerWrapper` has all three buttons and `Timer` inside of it. `TimerWrapper` is a container (or smart) component while the other two are representational (or dumb).

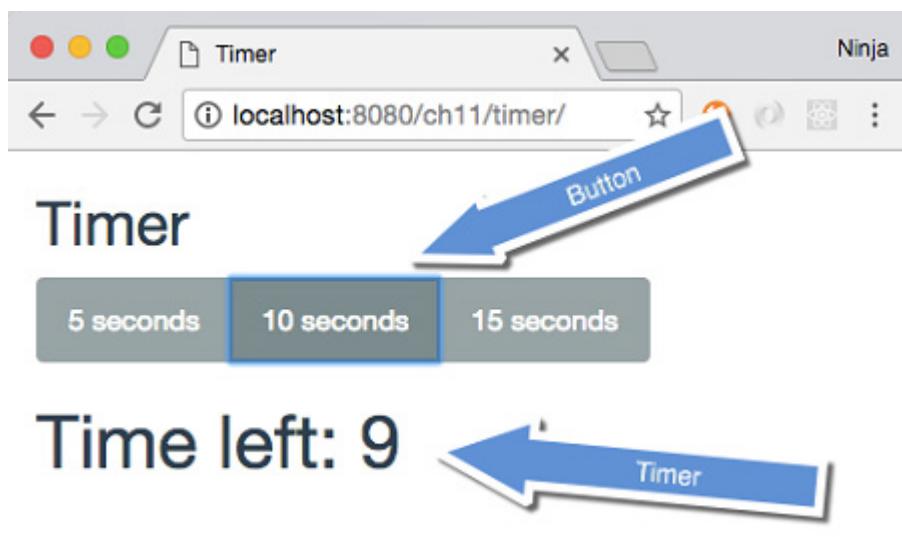


Figure 11.2 Timer and Button

The reason behind breaking down the app into three pieces is that as often happens in software engineering, things tend to change quickly with each new release. By breaking away the presentation (`Button` and `Timer`) and logic (`TimerWrapper`), we can make the app more adaptable. Moreover, we will be able to reuse elements such as buttons in other apps. The bottom line, keeping representation and business logic separate is one of the best practices when working with React.

We need TimerWrapper to communicate between Timer and Buttons. The interaction between these three components and a user is shown in figure 11.3:

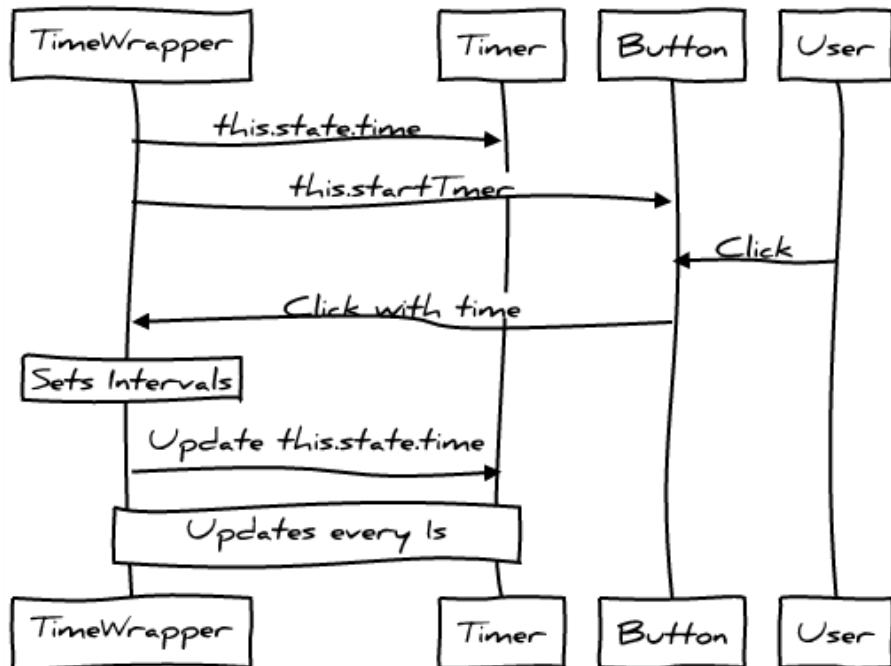


Figure 11.3 Timer app execution, top is the start

1. TimerWrapper renders Timer and Buttons by passing TimerWrapper's states as properties.
2. The user interacts with a button, which triggers an event in the button.
3. The event in the button calls the function in TimerWrapper with the time value in seconds.
4. TimerWrapper sets the interval and updates Timer.
5. The updates continue until there are 0 (zero) seconds left.

To keep things simple, we'll keep all three components in the one `timer.jsx` file, the outline of which looks like this

Listing 11.2 : timer.jsx in outline

```

class TimerWrapper extends React.Component {
  constructor(props) {           ①
    // ...
  }
  startTimer(timeLeft) {        ②
    ...
  }
}

```

```

        // ...
    }
    render() {
        // ...
    }
}

class Timer extends React.Component {
    render() {
        // ...
    }
}

class Button extends React.Component {
    startTimer(event) { ❸
        // ...
    }
    render() {
        // ...
    }
}

ReactDOM.render(
    <TimerWrapper/>, ❹
    document.getElementById('timer-app')
)

```

- ❶ Set some initial states
- ❷ Trigger the new timer (reset) for real
- ❸ Trigger the new timer (reset) from a user click—calls startTimer from TimerWrapper
- ❹ Render TimerWrapper

Let's start from the bottom of the `timer.jsx` file and render the main component (`TimerWrapper`) into the `<div` with ID `timer-app`:

```

ReactDOM.render(
    <TimerWrapper/>,
    document.getElementById('timer-app')
)

```

The `ReactDOM.render()` will be the last call in the file. It uses `TimerWrapper` so let's define this component next.

11.3 The TimerWrapper Component

`TimerWrapper` is where all the fun happens!

First, we need to be able to save time left (`timeLeft`) and reset the timer (`int` from `int — interval`, not integer). On the first app load, the timer shouldn't be running, so we need to set the time (`timeLeft`) state to `null`. This will come in handy in `Timer` because we will be able to

tell the difference between the first load (`timeLeft` is `null`), and when the time is up (`timeLeft` is `0`).

We'll also set the state `timer` property to `null`. This property will hold the reference to the `setInterval` function that will do the countdown:

```
class TimerWrapper extends React.Component {
  constructor(props) {
    super(props)
    this.state = {timeLeft: null, timer: null}
    this.startTimer = this.startTimer.bind(this)
  }
  ...
}
```

This is the `startTimer` event handler. It's called each time a user clicks a button. If a user clicks a button when the timer is already running, then we need to clear the previous interval and start anew. (We definitely don't want multiple timers running at the same time.) The first thing that the `startTimer` method does is to stop the previous countdown by clearing `setInterval`. The current timer's `setInterval` object is stored in the `this.state.timer` variable:

```
...
startTimer(timeLeft) {
  clearInterval(this.state.timer)
```

Both `clearInterval()` and `setInterval()` are browser API methods; that is, they are available from a `window` object without additional libraries. For more information on these two methods, visit the Mozilla Developer Network (MDN):

- [clearInterval\(\)](#)
- [setInterval\(\)](#)

After we clear the previous timer, we can set a new one with `setInterval()`; and let's use a fat arrow function to bind the `this` context so we can use it in the closure/callback of the `setTimetout()`.

```
let timer = setInterval(() => {
  ...
})
```

The `timeLeft` variable stands for "time left." When it's `0`, then we remove the timer with `clearInterval(this.state.timer)`. We save the current value minus `1`, and check if it reached `0`. If it did in fact reach `0`, then we clear this timer by using the `this.state.timer` variable. The reference to `timer` will be saved in the `setInterval()`'s closure, so there's no need to pull the value from the state (but we could):

```
var timeLeft = this.state.timeLeft - 1
if (timeLeft == 0) clearInterval(timer)
this.setState({timeLeft: timeLeft})
}, 1000)
return this.setState({timeLeft: timeLeft, timer: timer})
...
```

In addition, we want to save the time left in the state so another Timer component can access it. We set the state to the new value with `setState()`. The time of the interval is 1,000 milliseconds, which is 1 second. On the last line, we set the state to the new value of `timeLeft` and `timer`.

`setInterval()` is scheduled to be executed asynchronously in the JavaScript's event loop. The returned `setState()` will fire before the first `setInterval()` callback. You can easily test it by putting console logs. The following code will print 1, then 2, not 2 then 1.

```
...
startTimer(timeLeft) {
  clearInterval(this.state.timer)
  let timer = setInterval(() => {
    console.log('2: Inside of setInterval')
    var timeLeft = this.state.timeLeft - 1
    if (timeLeft == 0) clearInterval(timer)
    this.setState({timeLeft: timeLeft})
  }, 1000)
  console.log('1: After setInterval')
  return this.setState({timeLeft: timeLeft, timer: timer})
}
...
```

Lastly, there's the mandatory `render()` function for `TimerWrapper`. It returns `<h2>`, three buttons, and the `Timer` component. `row-fluid` and `btn-group` are Twitter Bootstrap classes. These classes just make buttons look better and are not essential to React.

```
render() {
  return (
    <div className="row-fluid">
      <h2>Timer</h2>
      <div className="btn-group" role="group" >
        <Button time="5" startTimer={this.startTimer}/>
        <Button time="10" startTimer={this.startTimer}/>
        <Button time="15" startTimer={this.startTimer}/>
      </div>
    </div>
  )
}
```

This code shows how we reuse Button components. We provide different values to the `time` property, which the Button component will use internally. How?

The buttons have different `time` property values. This allows the buttons to display different times on their labels and to set different timers. The `startTimer` property of Button has the same value for ALL THREE buttons (value is `TimerWrapper this.startTimer`). This method starts/resets the Timer as you know. The `time` argument will be different for 5, 10, and 15 seconds.

Next, we display the text "Time left..." which is the `Timer` component. We pass the state `time` as a property to `Timer`. The text on the page will be updated automatically by React once the state is updated. In addition, there's the audio tag to alert us when the time is up. It's just an HTML5 tag that points to a file:

```

        <Timer time={this.state.timeLeft}/>
        <audio id="end-of-time" src="flute_c_long_01.wav" preload="auto"></audio>
    </div>
)
}
}

```

Meet the meat (or tofu) of the Timer app—TimerWrapper!

Listing 11.3 The TimerWrapper Component (ch11/timer/timer.jsx)

```

class TimerWrapper extends React.Component {
  constructor(props) {
    super(props)
    this.state = {timeLeft: null, timer: null}
    this.startTimer = this.startTimer.bind(this)
  }
  startTimer(timeLeft) {
    clearInterval(this.state.timer)      ①
    let timer = setInterval(() => {
      console.log('2: Inside of setInterval')
      var timeLeft = this.state.timeLeft - 1
      if (timeLeft == 0) clearInterval(timer)
      this.setState({timeLeft: timeLeft})  ②
    }, 1000)
    console.log('1: After setInterval')
    return this.setState({timeLeft: timeLeft, timer: timer})
  }
  render() {
    return (
      <div className="row-fluid">
        <h2>Timer</h2>
        <div className="btn-group" role="group" >
          <Button time="5" startTimer={this.startTimer}/> ③
          <Button time="10" startTimer={this.startTimer}/>
          <Button time="15" startTimer={this.startTimer}/>
        </div>
        <Timer timeLeft={this.state.timeLeft}>          ④
          <audio id="end-of-time" src="flute_c_long_01.wav" preload="auto"></audio> ⑤
        </Timer>
      )
    }
}

```

- ① Clear timer to reset in case there were any other timers running
- ② Update the decremented time left every second
- ③ Render buttons which will call `startTimer` with different time
- ④ Render text "Time left:..." and play sound when it's 0
- ⑤ HTML5 audio tag which will play the alert when time is 0

Speaking of playing sounds when time is 0, we can implement Timer component next.

11.4 The Timer Component

So let's create the Timer component. The goals of this component is to show the time left and to play the sound when the time is up.

To play the sound which signifies that the time is up (file `flute_c_long_01.wav`) we can use a special HTML5 element `<audio>`. We defined it in `TimerWrapper` earlier with the `src` pointing to the WAV file and ID set to `end-of-time`. The statement with `play()` will play the audio file.

It is an HTML5 tag and has nothing to do with React. This again shows how React is cool at playing with other JavaScript things (jQuery or HTML5 or maybe even Angular?).

Okay, we play the sound when the time is up:

```
class Timer extends React.Component {
  render() {
    if (this.props.timeLeft == 0) {
      document.getElementById('end-of-time').play()
    }
    // ...
  }
}
```

Now, we need to show the time left when the timer is running, and when the timer is up, we want to display 0. To add this logic, we'll use an `if` condition to check for the value in `this.props.time` for 0.

Also, we don't want to show text with 0 seconds left initially because the timer has never run. Let's just hide the text altogether in the beginning. To differentiate, we can set the `timeLeft` value to `null` initially and 0 when the time is up.

Therefore, if the `timeLeft` property is `null` or 0, then the component will render an empty `<div>`. But if the value is not `null` or 0, then we'll see an `<h1>` element showing the rest of the time left to elapse:

```
// ...
if (this.props.timeLeft == null || this.props.timeLeft == 0)
  return <div/>
return <h1>Time left: {this.props.timeLeft}</h1>
}
```

Here's the component in full for the reference:

Listing 11.4 The timer component in full (ch11/timer/timer.jsx)

```
class Timer extends React.Component {
  render() {
    if (this.props.timeLeft == 0) {
      document.getElementById('end-of-time').play() ①
    }
    if (this.props.timeLeft == null || this.props.timeLeft == 0)
      return <div/> ②
    return <h1>Time left: {this.props.timeLeft}</h1> ③
  }
}
```

```

    }
}

① Play sound when time is up
② Display nothing initially when
③ Display text "Time left..."
```

For Timer to show seconds we need to start the timer first. This will happen when users click on buttons. Onward to those glorious buttons!

11.5 The Button Component

Let's be done with another simple component (as they should be in accordance with reactive mindset). The Button component is also pretty straightforward like the Timer.

The button will have an `onClick` event handler. This is what we need to capture users clicks on buttons. The function to start the timer is passed down to this Button component from its parent TimerWrapper in `this.props.startTimer`. But how should we pass time (5, 10, 15) to TimerWrapper's `startTimer`? Look at this code from TimerWrapper:

```

<Button time="5" startTimer={this.startTimer}/>
<Button time="10" startTimer={this.startTimer}/>
<Button time="15" startTimer={this.startTimer}/>
```

The idea is that we'll render three buttons using this component (code reuse, YAY!). To know what time was selected though, we need the value in `this.props.time`, which we pass as an argument to `this.props.startTimer`.

If we write this... it won't work!

```

// Won't work. Must be a definition.
<button type="button" className='btn-default btn'
  onClick={this.props.startTimer(this.props.time)}
  {this.props.time} seconds
/>
```

Function to `onClick` must be a definition, not an invocation. So how about this?

```

// Nope.
<button type="button" className='btn-default btn'
  onClick={()=>{this.props.startTimer(this.props.time)}}
  {this.props.time} seconds
/>
```

Yes. This is right approach of having a middle step (function) to pass different time values. We can make it more elegant by creating a class method. The event handler will call this class method (`this.startTimer`) which in turn will call a function from the property `this.props.startTimer`.

```

class Button extends React.Component {
  startTimer(event) {
    ①
    return this.props.startTimer(this.props.time)
```

```

    }
    render() {           ②
      return <button type="button" className='btn-default btn'
        onClick={this.startTimer.bind(this)}>          ②
          {this.props.time} seconds
        </button>
    }
}

```

- ① Kick start or reset the time with proper time value
- ② Render Button UI
- ③ Capture onClick

The Button is stateless which you can confirm by looking at the full code of the component as shown in Listing 11-5.

Listing 11.5 The Button component (ch11/timer/timer.jsx)

```

class Button extends React.Component {
  startTimer(event) {
    return this.props.startTimer(this.props.time)
  }
  render() {
    return <button type="button" className='btn-
      default btn' onClick={this.startTimer.bind(this)}>
      {this.props.time} seconds
    </button>
  }
}

```

Obviously, you don't need to use same names in Button and TimerWrapper. You can name Button's method handleStartTimer for example. Personally, having same name helps me *mentally* to link together props, methods and states from different components.

Another naming comment is that Timer could be named TimerLabel if not for the audio method... is there a room for refactoring? Absolutely! Check the Homework section in this chapter.

We are officially over with coding. Now on to getting this thingy run.

11.6 Getting It Running

We compile the JSX into JavaScript with the following Babel 6.9.5 command, assuming we have Babel CLI and presets installed (`hint: package.json!`):

```
$ ./node_modules/.bin/babel timer.jsx -o timer.js -w
```

If you copied my build npm script from `package.json`, then you can simply run `npm run build`.

If you did everything right, enjoy your beautiful timer application (Figure 11-4)! And, turn off your music to hear that alarm when the time is up.

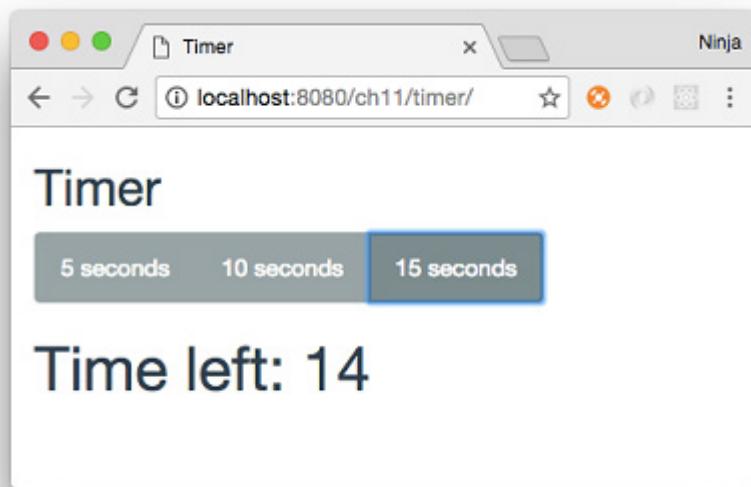


Figure 11.4 Pressing on 15 seconds launched the timer and now it says that 10 seconds left

Make sure it works *properly*: You should see a time-remaining number. It changes every second. When you click the button, a new countdown should begin; that is, the timer interrupts and starts over on each press of a button.

11.7 Homework

For bonus points,

- Convert Timer to a stateless component implemented by a fat arrow function
- Implement a stop button
- Modify the final version of this project to use 5, 10 and 15 minutes
- Decouple `<audio>` tag in `TimerWrapper` from `play()` in `Timer`
- Refactor to have 4 files: `timer.jsx`, `timer-label.jsx`, `timer-button.jsx` and `timer-sound.jsx` with as much loose coupling as possible
- Implement a slider button which changes with every time interval (chapter 6 for slider integration)

Submit your code in a *new folder under ch11* as a pull request to this book's GitHub repository: <https://github.com/azat-co/react-quickly/>

11.8 Summary

- Keep components simple and as close to representational as possible
- Pass functions as values of properties, not just data
- Two components can exchange data between each other via a parent

12

The Webpack Build Tool

In this chapter, we cover these topics:

- What does Webpack do?
- Adding Webpack to a Project
- Modularizing Your Code
- Running Webpack and Testing The Build
- Hot Module Replacement



Figure 12.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch12>

Before we can go any further with React stack (a.k.a. React and friends), let's look at a first such tool essential to most modern web development--a build tool (or bundler). We will use this tool in each chapter to bundle our many code files into the minimum number of files needed to run our applications, ready for an easy deployment. The build tool we will be using is Webpack.

If you've not come across a build tool before, or have used another one such as Grunt, Gulp or Bower, this chapter is for you. We will run cover how to set Webpack up, configure it and get

it running against one of your project. You'll also look at Hot Module Replacement, a feature of webpack that enables you to hotswap updated modules for those running on a live server. First though, let's look at what Webpack actually does for us.

The source code for the examples in this chapter is in [the ch12 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

12.1 What Does Webpack do?

Webpack's core focus is to optimize the javascript you have written so that it is contained in as few files as possible for a client to request. This reduces the strain on the server for popular sites and also reduces the page load time on the client. Of course it's not as simple as that. The JavaScript you write is often written in modules that are easy to reuse. But they often depend on other modules which may depend on other modules and so on and keeping track of what needs to be loaded when so all the dependencies resolve quickly becomes a big headache you don't need.

Let's say you have a utility module `myUtil` and you use it in many React components `accounts.jsx`, `transactions.jsx`, etc. Without a tool like Webpack, then you would have to manually keep track of the fact that each time you use one of those components you need to include `myUtil` as a dependency. Additionally, you might be loading `myUtil` unnecessarily for a second or third time because another component which depends on `myUtil` has already loaded it. Of course this is a very simplified example. In real projects there are dozens or even hundreds dependencies which are used in other dependencies of the project. Webpack helps with this.

Webpack knows how to deal with all three types of javascript module - CommonJS, AMD and ES6 - as well so no need to worry if you're working with a hotch-potch of module types. Webpack will analyze the dependencies for all the javascript in your project and:

- Ensure that all dependencies are loaded in the correct order
- Ensure that all dependencies are loaded only once
- Ensure that your javascript is bundled into as few files (called *static assets*) as possible for it to work.

Webpack also supports features called code splitting and asset hashing to which you must opt-in to further optimize your javascript and its deployment. These allow you to identify blocks of code which are only required under certain circumstances. These blocks will be split out to be loaded on demand rather than bundled in with everything else.

NOTE Code Splitting and asset hashing are outside the scope of this book. Please check out the webpack website for more information at <https://webpack.github.io/docs/code-splitting.html>

It's not just about javascript though. Webpack supports the pre-processing of other static files through the use of *loaders*. For example, we could do the following before any bundling takes place.

- Precompile our JSX, jade or CoffeeScript files into plain JavaScript
- Precompile ES6+ code into ES5 for browsers which don't yet support it
- Precompile SASS and compass files into CSS
- Optimize sprites into a single PNG, JPG file or inline data assets

There are many loaders available for all sorts of file types and also plugins for affecting webpack's behavior catalogued on the webpack homepage and if you can't find what you're looking for, there's also documentation on how to write your own.

For the rest of this book, we'll be using Webpack to do the following:

1. Manage and bundle our dependencies from npm modules so we don't have to manually download files from the Internet, and include them with `<script>` tags in HTML
2. Transpile JSX into regular JavaScript while providing source maps for easier debugging

As you'll see, you configure the order Webpack loads, pre-compiles and bundles your files using its `webpack.config.js` file. Before that however, let's look at how to install it and get it working with a project.

12.2 Adding Webpack to a Project

To illustrate how readers can get starting working with Webpack, we'll slightly modify the `email` project from chapter 7 which had an email and comment input fields, two style sheets and one component `Content`. The original app is shown again in Figure 12-1 for convenience.

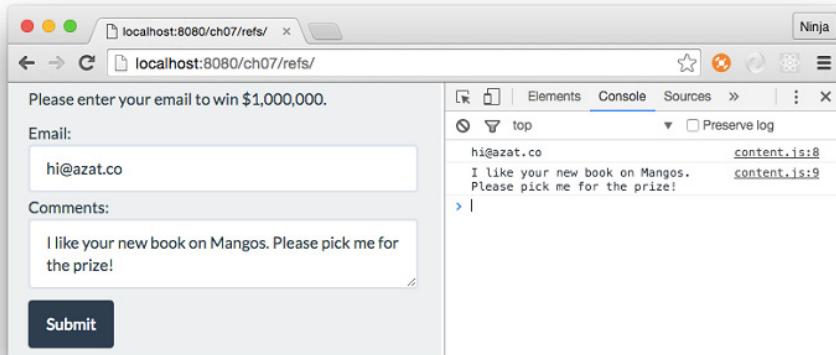


Figure 12.2 Original Email before Webpack

Here's the *new* project structure in which I point to where it differs from the project in chapter 7:

```
/email-webpack
  /css
    - bootstrap.css
    - main.css
  /js          1
    - bundle.js      2
    - bundle.map.js 3
  /jsx
    - app.jsx        4
    - content.jsx
  /node_modules 5
    - index.html
    - package.json   6
  - webpack.config.js
  - webpack.dev-cli.config.js
  - webpack.dev.config.js
```

- 1 The `js` folder does not have `react.js` or `react-dom.js` files
- 2 All the scripts
- 3 Mapping of the line numbers for DevTools
- 4 ReactDOM.render statement
- 5 Dependencies to compile (Webpack, Babel, etc.)
- 6 Babel configs and other project info
- 7 Webpack configs

Contrast that with the non-Webpack setup (from chapter 7):

```
/email
  /css
    bootstrap.css
  /js
    content.js      1
    react-15.0.2.js
    react-dom-15.0.2.js
    script.js
  /jsx
    content.jsx
    script.jsx      2
  index.html
```

- 1 Compiled script with the main component
- 2 `ReactDOM.render()` statement in JSX

Now, do you have Node and npm? This is the best time to install them. You won't be able to proceed without them. Refer to Appendix A on Node and npm installation.

Assuming you got Node and npm, now we'll do the following:

1. Install webpack
2. Install dependencies and save them to package.json
3. Configure Webpack's webpack.config.js

4. Configure Dev Server and Hot Module Replacement:

First step with Webpack setup is the dependencies.

12.2.1 Installing Webpack & Its Dependencies

To use Webpack, you'll need a few additional dependencies as noted in `package.json`:

- Webpack: The bundler tool itself (npm name is `webpack`)
- Loaders: Style, CSS, Hot Module Replacement and Babel/JSX pre-processors (npm names are `style-loader`, `css-loader`, `react-hot-loader` and `babel-loader`, `babel-core` and `babel-preset-react`)
- Webpack Dev Server: An Express development server which will allow us to use Hot Module Replacement feature (npm name is `webpack-dev-server`)

You can install each module manually, but I recommend just copying the `package.json` file (Listing 12-1) from the GitHub repository to your project root (see the project structure above). And then running `npm i` or `npm install` from project root (where you have `package.json`) to install the dependencies. This will ensure you didn't forget any of the ten modules (module is a synonym of a package in Node most of the times). It will also ensure that your versions at least somewhat close to the ones I used. Using wildly different versions is a *fantastic* way to break the app. :)

Listing 12.1 Setting up the dev environment with `package.json` for building router from scratch (ch10/naive-router/`package.json`)

```
{
  "name": "email-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "build": "./node_modules/.bin/webpack -w"          1
    "wds-cli": "./node_modules/.bin/webpack-dev-server --inline --hot --module-
      bind 'css=style!css' --module-bind 'jsx=react-hot!babel' --config webpack.dev-
      cli.config.js",
    "wds": "./node_modules/.bin/webpack-dev-server --config webpack.dev.config.js"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"           2
    ]
  },
  "devDependencies": {
    "babel-core": "^6.13.2",
    "babel-loader": "^6.2.4",                         3
    "babel-preset-react": "^6.5.0",
    "css-loader": "^0.23.1",                           4
    "react": "^15.2.1",
    "react-dom": "^15.2.1",
  }
}
```

```

    "react-hot-loader": "^1.3.0",      ⑤
    "style-loader": "^0.13.1",
    "webpack": "1.12.9",            ⑥
    "webpack-dev-server": "^1.14.1"   ⑦
  },
}

```

- ① Save Webpack build script as an npm script for convenience
- ② Tell Babel what presets to `use` (React for JSX in our case, ES6+ - optional)
- ③ Install Babel loader to `process` JSX
- ④ Install CSS loader to require `CSS` from `JavaScript` (then install `Style` loader to inject `CSS` into a webpage)
- ⑤ Install React Hot Module Replacement loader
- ⑥ Install Webpack `locally` (recommended)
- ⑦ Install Webpack Dev Server `locally` (recommended)

The `babel` property in `package.json` should be familiar to you from Part 1, therefore we won't spend too much time repeating ourselves. To remind you, we need this property to configure Babel to convert JSX to JS. If you need to support browsers which can't work with ES6, then you can add `es2015` preset to `presets`

```

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

```

And `babel-preset-es2015` to `devDependencies`:

```

"devDependencies": {
  "babel-preset-es2015": "6.18.0",
  ...
}

```

Besides new dependencies, there are new npm scripts. The commands in `scripts` in `package.json` are *optional* but highly recommended, because npm scripts for launching and building is one of the best practices when working with React and Node. Of course, you can run all the builds manually just like that: `./node_modules/.bin/webpack -w`, without using npm scripts... but why typing extra characters?

To sum up, you'll be able to run Webpack with `npm run build` or directly with `./node_modules/.bin/webpack -w`. The flag `-w` means `watch`, that is continue to monitor for any source code changes and re-build bundles if there are any. In other words, Webpack will keep running to automatically make changes. Of course, you must have all the necessary modules installed with `npm i`.

The `webpack -w` command looks for `webpack.config.js` by default. We cannot run Webpack with this configuration file. Let's create it next.

NOTE The wds and wds-cli npm scripts in package.json will be explained in the Hot Module Replacement section.

12.2.2 Configuring Webpack

Webpack needs to know what to process (the source code) and how to do it (the loaders). That's why there's webpack.config.js in the root of the project (see project structure above). In a nutshell, in this project we are using Webpack to:

- Transform our JSX files into JS files: babel-loader, babel-core and babel-preset-react
- Loading of CSS via require and resolving url and imports in the process: css-loader
- Add CSS by injecting <style> element: style-loader
- Bundle all the resulting js files into one file called bundle.js
- Provide proper source code line mapping in DevTools via sourcemaps

So Webpack needs its own configuration file webpack.config.js. By the way, you can have more than one configuration file. It comes in handy for developments, production, testing and other builds. In the project structure, you can see that I created these files:

```
webpack.dev-cli.config.js
webpack.dev.config.js
```

The naming don't really matter as long as you and your team mates can understand the meaning. The name is passed to Webpack with --config. More on these particular configuration files in the hot module section later in this chapter. For now, here's our basic config as shown in Listing 12-2.

Listing 12.2 Configuring webpack (email-webpack/webpack.config.js)

```
module.exports = {
  entry: './jsx/app.jsx',          ①
  output: {
    path: __dirname + '/js/',      ②
    filename: 'bundle.js'          ③
  },
  devtool: '#sourcemap',           ④
  module: {
    loaders: [
      { test: /\.css$/, loader: 'style!css' }, ⑤
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader' ⑥
      }
    ]
  }
}
```

- ① Define the file to start **bundle**ing (typically your main file which loads other files)

- 2 Define path for the bundled files
- 3 Define filename for the bundled file which we will be using in `index.html`
- 4 Define that you need proper mapping of compiled source code lines to the JSX source code lines: useful for debugging and shows in DevTools
- 5 Specify loader to import and then inject CSS into webpage from JavaScript
- 6 Specify loader which will perform JSX transformation (and ES6+ if you need them)

The `devtool` property is good to have for development because it'll provide source maps which will allow you to see the line numbers in source not compiled code.

Now you are ready to not only run Webpack for this project, but also to bootstrap any Webpack-based projects in the future. Of course, Webpack has a lot of feature and we covered just the basics but these basics are enough to compile JSX, provide source maps as well as inject and import CSS and bundle JavaScript.

Borrowing from just-in-time compilation concept, you might call it just-in-time learning meaning if or when you need more Webpack features, you can consult its documentation or a book like *SurviveJS*.

By now we should be ready to use some of Webpack's power in JSX.

12.3 Modularizing Your Code

If you remember, in chapter 7 the email app used global objects and `<script>`. It's fine for the demonstration purposes of this book as well as some small app. However, in large apps using globals is frowned upon because you might run into all sorts of troubles with name collision, and managing multiple `<script>` tags which might have duplicate inclusions. We can let Webpack do all the dependency management by using CommonJS syntax. Webpack will include only needed dependencies and package them into a single file `bundle.js` (according to the configs in `webpack.config.js`).

This step is not something special I recommend do for Webpack. Modularizing is one of the best practicing not only for React but for software engineer in general. You can use Browserify, SystemJS or other bundler/module loaders and still use CommonJS/Node syntax of `require` and `module.exports`. In other words, the code in this section will be transportable to other systems once we refactor it away from primitive globals.

For this reason, we refactor `app.jsx` to use `require()` and `module.exports` instead of global objects and HTML `<script>`. Due to `style-loader`, we can require CSS files as well. And because of the Babel loader, we can require JSX files.

Listing 12.3 (ch12/email-webpack/app.jsx)

```
require('../css/main.css')           ①
const React = require('react')       ②
const ReactDOM = require ('react-dom')
const Content = require('./content.jsx') ③
```

```
ReactDOM.render(
  <Content />,
  document.getElementById('content')
)
```

- ➊ Import CSS which thanks to style and css loaders will be imported and injected into the webpage
- ➋ Import React for <> syntax which is really `React.createElement()`
- ➌ Import Content

In contrast, `ch07/email/jsx/script.jsx` looks like this (so you don't have to flip the pages):

```
ReactDOM.render(
  <Content />,
  document.getElementById('content')
)
```

Yes. The old file is smaller but this is one of the rare cases when less is not more. You see, in the old file we relied on global `Content`, `ReactDOM` and `React` objects which is a bad practice for aforementioned reasons (potential name collision, hard to manage, etc.).

In `content.jsx`, we can use `require()` similarly while the code for `constructor()`, `submit()` and `render()` haven't changed:

```
const React = require('react')           ➊
const ReactDOM = require('react-dom')    ➋

class Content extends React.Component {
  constructor(props) {
    // ...
  }
  submit(event) {
    // ...
  }
  render() {
    // ...
  }
}

module.exports = Content                ➌
```

- ➊ Import React
- ➋ Import ReactDOM
- ➌ Export Content component to be used in con

The `index.html` needs to point to the bundle Webpack creates for us. That's the file `js/bundle.js`. Its name was specified this in `webpack.config.js` so now we just need to add it in. It will be created after you run `npm run build`. This is the new `index.html` code:

```
<!DOCTYPE html>
<html>

  <head>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
```

```

</head>

<body>
  <div id="content" class="container"></div>
  <script src="js/bundle.js"></script>
</body>

</html>

```

Note that we've also removed the reference to the stylesheet `main.css` from `index.html`. Webpack will inject a `<style>` element with a reference to `main.css` into `index.html` for us. This is due to `require('main.css')` in `app.jsx`. You can use `require()` for `bootstrap.css` as well.

By now, our project is refactored.

12.4 Running Webpack and Testing The Build

This is THE moment of truth. Run `$ npm run build` and compare your output with the following:

```

> email-webpack@1.0.0 build /Users/azat/Documents/Code/react-quickly/ch12/email-webpack
> webpack -w

Hash: 2ffe09fff88a4467788a
Version: webpack 1.12.9
Time: 2545ms
    Asset      Size  Chunks             Chunk Names
  bundle.js    752 kB      0  [emitted]  main
bundle.js.map  879 kB      0  [emitted]  main
+ 177 hidden modules

```

If there are no errors and you can see newly created `bundle.js` and `bundle.js.map` in `js` folder - bingo! Now spin up your favorite web server (maybe `node-static` or `http-server`) and check the web app. See that it's logging the email and the comment in the console.

As you can see, incorporating Webpack into a project is quite straightforward and yields great results.

NOTE 177 hidden modules or Webpack bundle under the hood

Yes. There are 177 modules in `ch12/email-webpack/js/bundle.js`. You can open the file and search for `__webpack_require__(1)` and `__webpack_require__(2)`, etc. until `__webpack_require__(176)` which is actually the Content component. This is the compiled code from `app.jsx` which is importing Content (lines 49-53 in `bundle.js`):

```

const React = __webpack_require__(5);
const ReactDOM = __webpack_require__(38);
const Content = __webpack_require__(176);

```

```
ReactDOM.render(React.createElement(Content, null), document.getElementById('content'))
);
```

So at the bare minimum, you are ready to use Webpack for the rest of the book. However, I strongly recommend to setup one more thing. It can speed up your development dramatically. Before we get onto more React development, let's look at one more great feature of Webpack—Hot Module Replacement.

ESLint and Flow

There are two more very nice and useful development tools which you can benefit from. Obviously, they are optional, but I just have to mention them because they are a pretty *big* deal.

Firstly, [ESLint](#) (npm name `eslint`). It can take predefined rules or sets of rules to make sure your code (JS or JSX) adheres to the same standards. For example, how many spaces is the indentation? 4 or 2? Or maybe it's a tab? Or what if you accidentally put a semi-colon in your code? (It's not a typo. Semi-colons are optional in JavaScript, and I prefer not to use them.) ESLint will give a warning even for unused variable. It can even prevent some bugs from sneaking into the code! (Not all of them of course. :-)

To get started with ESLint check out [Getting Started with ESLint](#). It's very straightforward. You will also need `eslint-plugin-react`. Make sure to add the React rules to `.eslintrc.json` (full code is in the `ch12/email-webpack-eslint-flow` folder):

```
"rules": {
  "react/jsx-uses-react": "error",
  "react/jsx-uses-vars": "error",
}
```

Example of warnings from running ESLint React on `ch12/email-webpack-lint-flow/jsx/content.jsx`:

```
/Users/azat/Documents/Code/react-quickly/ch12/email-webpack-lint-flow/jsx/content.jsx
  9:10  error  'event' is defined but never used  no-unused-vars
 12:5   error  Unexpected console statement        no-console
 12:17  error  Do not use findDOMNode            react/no-find-dom-node
 13:5   error  Unexpected console statement        no-console
 13:17  error  Do not use findDOMNode            react/no-find-dom-node
```

Next, [Flow](#) is a static type checking tool meaning you can add a special comment `// @flow` to your scripts and types. Yes! Types in JavaScript! Rejoice all the software engineers with a preference for strongly typed languages (Java, Python, C, etc.)! Once you do that, you can run Flow check to see if there are any issues. Again, this tool can prevent you from having pesky bugs.

```
// @flow

var bookName: string = 13
console.log(bookName) // number. This type is incompatible with string
```

Flow (npm name `flow-bin`) has an extensive docs and guide so we won't duplicate the instructions here considering that they are prone future changes: [Getting started with Flow](#)(<https://flowtype.org/docs/getting-started.html>) and [Flow for React](#)

You can configure Atom or any other modern code editor to work with ESLint and Flow to catch problems on-a-fly so to say (Figure 12.3).

The screenshot shows the Atom code editor interface. The main pane displays a file named 'content.jsx' with Flow annotations. Lines 13 through 21 have several 'Warning' annotations from Flow, indicating missing annotations for 'props', 'submit', 'event', and 'prompt'. The right sidebar shows a project structure for 'ch12' containing 'email-webpack', 'css', 'js', and 'jsx' subfolders, along with various configuration files like '.flowconfig', '.eslintrc.json', and 'package.json'. The bottom status bar shows the file path 'ch12/email-webpack-lint-flow/jsx/content.jsx', the line number '13:38', and other development details.

Figure 12.3 Code editor Atom supports Flow by showing you issues in the bottom pane as well as marks on the code line all during the development

I put Email code with ESLint v3.8.1 and Flow v0.33.0 into ch12/email-webpack-eslint-flow.

12.5 Hot Module Replacement

Hot Module Replacement (HMR) is one of the coolest features of Webpack and React, because it allows developers to write code and test it faster by updating the browser with changes while preserving whatever state their app had.

Say you're working on a complex single page web application and to get to the current page you're working on takes twelve clicks. If you uploaded some new code to the site, then to get it running, you would need to hit reload \ refresh in your browser and then repeat those dozen clicks again. If you're utilizing HMR on the other hand, there would be no page reloads and your changes will be reflected on the page.

Some developers even say that HMR is so ground-breaking that if React didn't have any other features, they would use it only for HMR! The major benefit of HMR is that developers can

iterate (write, test, write, test, etc.) faster., because their app will save the state when they make changes.

The process of hot-updating the code is multi-step. In its simplified form it's shown in Figure 12-3. For more nitty-gritty details of how the HMR process works see [the documentation](#) while we will cover the practical application of this technology as it pertains to your example of an email form.

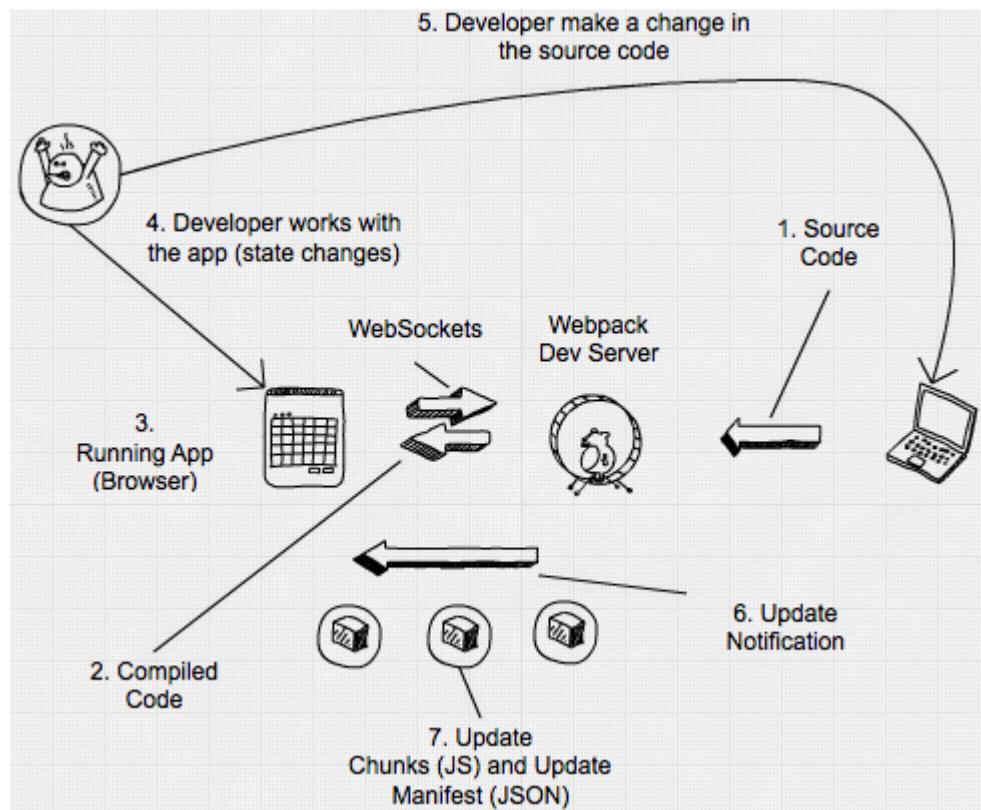


Figure 12.4 Webpack listens for code changes and sends update notifications along with updates to the running app in the browser

As shown in Figure 12.4 , Webpack HMR and the dev server utilize WebSockets which monitor update notifications from the server. If there are any, the front-end gets chunks (JavaScript code) and update manifesto (JSON) which are basically delta of the changes. The front-end apps preserves its state (for example data in input field or a screen position) but the UI and code changes. Magic. :)

To illustrate HMR on an example, we will use a new configuration file and Webpack Dev Server. It's possible to use HMR with your own server built with Express/Node. WDS is optional but it's provided by Webpack (as a separate module `webpack-dev-server`) so we will cover it.

Then, once everything is configured, we'll enter email in the form and make a few changes in the code to observe that the entered email is still there and our changes propagated to the web app.

12.5.1 Configuring Hot Module Replacement

Firstly, duplicate `webpack.config.js` by creating a copy named `webpack.dev.config.js`:

```
$ cp webpack.config.js webpack.dev.config.js
```

Next, open the newly created file `webpack.dev.config.js`. We need to add a few things such as new entry points, public path, HMR plugin and dev server flag set to true. This is the final file with annotation where you'll need to add configs:

Listing 12.4 Webpack Dev Server and HMR configuration (ch12/email-webpack/webpack.dev.config.js)

```
const webpack = require('webpack')           ①

module.exports = {
  entry: [
    'webpack-dev-server/client/?http://localhost:8080',      ②
    'webpack/hot/dev-server',                                ③
    './jsx/app.jsx'
  ],
  output: {
    publicPath: 'js/',                                     ④
    path: __dirname + '/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  module: {
    loaders: [
      { test: /\.css$/, loader: 'style!css' },
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loaders: ['react-hot', 'babel']                      ⑤
      }
    ],
    devServer: {
      hot: true                                         ⑥
    },
    plugins: [new webpack.HotModuleReplacementPlugin()]       ⑦
  }
}
```

① Import `webpack` module

- 2 Include Webpack Dev Server (called inline dev server)
- 3 Include HMR
- 4 Set path for Dev Server to make bundle.js available (it wont write to disk)
- 5 Include `react-hot` to automatically enable HMR on all JSX files
- 6 Set dev server in HMR mode
- 7 Include HMR plugin

To use this new configuration file, we need to tell Webpack Dev Server to use this particular file by providing --config option:

```
./node_modules/.bin/webpack-dev-server --config webpack.dev.config.js
```

Save it in package.json for convenience if you don't have it there already. If you remember, we had react-hot-loader in the dependencies. This module enables HMR for all JSX files (which are in turn converted to JS).

I prefer enabling HMR for all files with react-hot-loader. But if you want to have HMR just for certain modules, not all of them, then don't use react-hot-loader, and do opt in manually by adding `module.hot.accept()` statement to those particular JSX/JS modules you want to cherry-pick for HMR. This `module.hot` magic comes from Webpack. It's recommended to check that `module.hot` is available:

```
if(module.hot) {
  module.hot.accept()
}
```

Phew. That's a mouth full of configurations! Yet, there's another way to use and configure Webpack. Aren't you excited? I am! We can use command-line option to pack some configs in the commands.

So if you prefer to use a command-line, be my guest. Your config file will become smaller but the command bigger. For example, this file `webpack.dev-cli.config.js` has fewer configs:

```
module.exports = {
  entry: './jsx/app.jsx',
  output: {
    publicPath: 'js/',
    path: __dirname + '/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loaders: []
      }
    ]
  }
}
```

However, it uses more CLI options:

```
./node_modules/.bin/webpack-dev-server --inline --hot --module-bind 'css=style!css' --module-bind 'jsx=react-hot!babel' --config webpack.dev-cli.config.js
```

There are a few things happening here. First, `--inline` and `--hot` will include the entries enabling dev server and the HMR mode. Then, we pass our loaders with `--module-bind` using the following syntax:

```
fileExtension=loader1!loader2!...
```

Make sure that `react-hot` is before `babel` otherwise, there would be an error.

When it comes to using CLI or a full config file, the choice is *yours*. I find CLI approach better for simpler builds. Of course, to avoid crying because you mistyped this monstrosity of a command, save that command as an npm script in `package.json`. And no, batch/shell scripts/Make scripts are not cool anymore. Use npm scripts like all all the cool kids do. (Disclaimer: It's a joke. I'm not advocating Fashion-Driven Development.)

npm Scripts

npm scripts offer certain advantages because they are commonly used in Node and React projects. So they became a de facto standard and most developers will expect to find them when they first learn about a project. In fact, when I look at a new project or a library, npm scripts is the first place I look at after `readme.md` and sometime even instead of `readme.md` which might be out-of-date. Also npm scripts support a pre and post hooks convention and other features.

npm scripts offer a flexible way to save essential scripts for testing, building, seeding with data, running in development or other environments. In other words, any work performed via CLI and which is related to the app but which is not the app itself can be save to npm scripts.

Not many developers know that npm scripts support pre and post hooks convention which make them even more versatile. For example, I can create a task `learn-react` and two tasks with pre and post hooks: `prelearn-react` and `postlearn-react`. As you can guess, the pre will be executed before `learn-react` while post will be executed after `learn-react`. Considering these scripts:

```
"scripts": {
  "prelearn-react": "echo \"Purchasing React Quickly\",
  "learn-react": "echo \"Reading React Quickly\",
  "postlearn-react": "echo \"Creating my own React app\""
},
```

Your commands will print:

```
...
Purchasing React Quickly
...
Reading React Quickly
...
```

Creating my own React app

With `pre` and `post` hooks npm can easily replace some of the build steps performed by Webpack, Gulp or Grunt (See [the official docs](#) and [How to Use npm as a Build Tool](#) for more npm tips such as parameters and arguments). Whatever functionality is missing with npm scripts (WDS or some plugin) can be simply implemented from scratch as a Node script. The advantage is that you will have fewer dependencies on plugins for your project.

12.5.2 HMR in Action

Now, go ahead and start the dev server (with `npm run wds` or `npm run wds-cli`). Then go to <http://localhost:8080> and open the DevTools console. There will be messages from HMR and WDS (Webpack Dev Server) as follows:

```
[HMR] Waiting for update signal from WDS...
[WDS] Hot Module Replacement enabled.
```

Enter some info in the email or comment field and then change the `content.jsx`. You can modify something in `render()`, for example change text from `Email` to `Your Email`:

```
Your Email: <input ref="emailAddress" className="form-control" type="text" placeholder="hi@azat.co"/>
```

You'll see the some more logs:

```
[WDS] App updated. Recompiling...
...
[HMR] App is up to date.
```

And then finally, your changes will be on the webpage but the entered text will also be there (Figure 12-4). Great, because now you don't need to waste time entering test data or navigating deep inside of the nested UIs. Yeah! Now we can spend more time doing important things instead of typing and clicking around the front-end app, because the development will be *faster* with HMR!

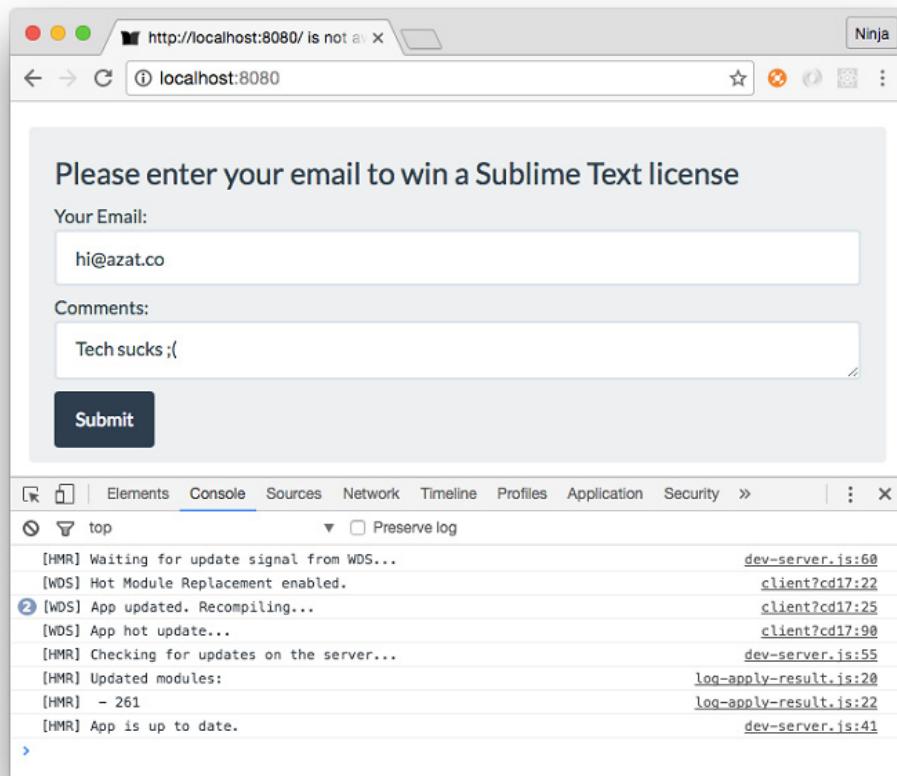


Figure 12.5 HMR just updated the view from "Email" to "Your Email" without erasing my data in the email and comment fields as shown in logs

NOTE HMR is not bullet-proof. It won't update or fail in some situations. WDS will reload the page (live reload), when that happens (this behavior is controlled by `webpack/hot/dev-server` with another option to reload manually being `webpack/hot/only-dev-server`).

All in all, Webpack is a nice tool to use with React to streamline and enhance your bundling. It's great not only for optimizing the code, images, styles, and other assets when you deploy... but also for development, thanks to WDS and HMR.

12.6 Quiz

1. What is the command to run a dev npm script ("dev": "./node_modules/.bin/webpack-dev-server --config webpack.dev.config.js")?
2. HMR is just a React term for live reloading. True or false?
3. WDS will write to disk the compiled files just like the webpack command. True or false?
4. webpack.config.js must be a valid JSON file just like package.json. True or false?
5. What loaders developers needs to use in order to import and then inject CSS into a webpack using Webpack?

12.7 Summary

In this chapter, we've covered

- To make Hot Module Replacement work have Webpack Dev Server, and `react-hot-loader` in config or `module.hot.accept()` in files
- You can utilize `require()` to load CSS with `style-loader` and `css-loader`
- The `--inline --hot` options with CLI commands will launch WDS in *hot inline mode*
- `devtool: '#sourcemap'` enables proper line numbers for compiled code
- `publicPath` is a WDS setting which will tell WDS where to "put" the bundle

12.8 Quiz Answers

1. `npm run dev`. Only `start` and `test` npm script can be run without `run`. All other scripts follow `npm run NAME` syntax.
2. False. HMR can replace live reloading and fallback to it when HMR fails, but HMR is more advanced and offers more benefits such as updating only parts of your app, and preserving the app's state.
3. False. WDS only serves file without actually writing them to disk.
4. False. `webpack.config.js` which is a default Webpack configuration file must be Node/JavaScript file with CommonJS/Node module exporting the object literal of configurations (the object can be with double quotes akin to JSON though).
5. Style and CSS loaders: Style will import and CSS will inject.

13

React Routing

This chapter covers

- Implementing Router from Scratch
- React Router
- React Router Features
- Routing with Backbone



Figure 13.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch13>

For a while, the problem with a great many single-page applications was that as you progressed through the apps, the URLs rarely, if ever, changed. There was no reason to go to the server thanks to browser rendering! Just the content in an area of the page changed. This had some unfortunate consequences.

- Refreshing your browser took you away from the page you were reading to the original form of the page you were reading.
- Hitting the back button in your browser might take you to a completely different site because its history function only recorded a single URL for the site you were reading. There were no URL changes reflecting your navigation between content.

- You couldn't share a precise page on the site with your friends.
- Search engines couldn't index your site because there were no distinct URLs to index.

Fortunately, there is URL routing. URL routing lets you configure an application to accept request URLs that do not map to physical files. Instead, you can use routing to define URLs that are semantically meaningful to users, that can help with search-engine optimization (SEO), and that can reflect your application's state. For example, a URL for a page that displays product information might be

```
https://www.manning.com/books/react-quickly
```

This is neatly mapped behind the scenes to the one page that displays products and the fact that it's the product with id `react-quickly` that needs to be displayed. It also means that as you browse various products, the URL can change and both the browser and search engines will be able to interact with the product pages as you'd expect. And if you want to avoid complete page reloads, you can use a hash (#) in your URLs, like these well known sites do.

```
https://mail.google.com/mail/u/0/#inbox
https://en.todoist.com/app?v=816#agenda%2Foverdue%2C%20today
https://calendar.google.com/calendar/render?tab=mc#main_7
```

Having URL routing is paramount, because without URLs our users won't be able to save or share links without losing the state of the application be it an SPA or traditional web app with server rendering. URLs are integral part of the web, however with some of the single-page applications, the URL routing is neglected.

In a nutshell, URL routing is a major requirement for a user-friendly well designed web app. In this chapter, we're going to build a simple React website and look at a couple of different options for implementing Routing within it. We'll look at the React router library in a while, but first, we'll try building some simple routing from scratch.

The source code for the examples in this chapter is in [the ch13 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

13.1 Implementing Router from Scratch

Although, there are existing libraries which implement routing for React, let's start by implementing our own simple router to see how easy it is to get something simple going with React and also to understand how some of the other routers might work under the hood.

The end goal of this project is to have at three pages which change along with the URL when we navigate around. We'll be using hash URLs (#) to keep things simple as non-hash URL require special server configuration. Thus, there are these pages:

- Home: / (empty URL path)
- Accounts: /#accounts

- Profile: /#profile

Figure 13.2 shows the navigation from Home page to Profile page.

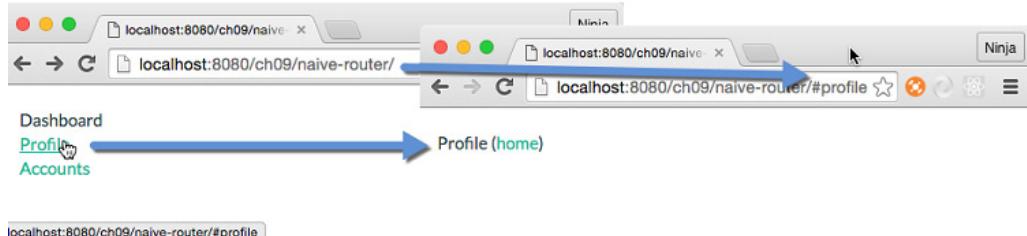


Figure 13.2 Navigating from Home page to Profile page with the URL change by clicking on a link

To implement this project, we will create a router component (`router.jsx`). This router component will take the information from the URL and update webpage accordingly. You can break down the implementation into these steps:

1. Write the mapping between the URL entered and the resource to be shown (e.i., React elements or components). Developers will need to provide different mapping for each new project.
2. Write the code to access the requested URL and check it against the mapping — router library consisting of a single Router component (`router.jsx`). This Router can be reused without modifications in various projects.
3. Add the Router component to the HTML page and supply it with the mapping we wrote in step 1

We are using JSX to use React elements for the markup. Obviously, `Router` doesn't have to be a React component - it can be a regular function or a class - but by using React component we are reinforcing concepts learned before such as event life cycles, and taking advantage of React's rendering and handing of the DOM. Also, our implementation will be closer to the React Router implementation which will help readers understand React Router better.

13.1.1 Setting up the Project

The structure of the project Router from scratch (you can call it simple or naive Router) is as follows:

```
/naive-router
/css
  - bootstrap.css
  - main.css
/js
  - bundle.js
/jsx
```

```

- app.jsx
- router.jsx
/node_modules
- index.html
- package.json
- webpack.config.js

```

Next, we install dependencies. I have them in `package.json` so you can copy the dependencies as well as babel config, and scripts, and run `npm install`.

Listing 13.2 Setting up the dev environment with `package.json` for building router from scratch (ch13/naive-router/package.json)

```

{
  "name": "naive-router",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "./node_modules/.bin/webpack -w"      ①
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"          ②
    ]
  },
  "devDependencies": {
    "babel-loader": "^6.2.4",
    "babel-preset-react": "^6.5.0",
    "webpack": "1.12.9"      ③
  },
  "dependencies": {
    "react": "^15.2.1",
    "react-dom": "^15.2.1"
  }
}

```

- ① Save Webpack build script as an npm script for convenience
- ② Tell Babel what presets to use (React for JSX in our case, ES6+ - optional)
- ③ Install Webpack locally (recommended)

This is not all. Webpack needs its own configuration file `webpack.config.js` (as explained in chapter 9). The key here is to configure the source (`entry`) and the desired destination (`output`). Also, we need to provide the loader.

```

module.exports = {
  entry: './jsx/app.jsx',           ①
  output: {
    path: __dirname + '/js/',       ②
    filename: 'bundle.js'          ③
  },
  module: {

```

```

loaders: [
  {
    test: /\.jsx?$/,
    exclude: /(node_modules)/,
    loader: 'babel-loader' ④
  }
]
}

```

① Define the file to start **bundling** (typically your main file which loads other files)
 ② Define path for the bundled files
 ③ Define filename for the bundled file which we will be using in `index.html`
 ④ Specify loader which will perform JSX **transformation** (and ES6+ if you need them)

13.1.2 Creating the Route Mapping in app.jsx

First, we create mapping with `mapping` object where keys are URL fragments and the values are the content of the individual pages. Mapping means take a value and ties/connect it to another value. The idea is that the key (URL fragment) would map to JSX. Of course, we can create separate files for each individual page, but for now let's just keep them in `app.js`:

Listing 13.2 The route mapping (app.jsx)

```

const React = require('react') ①
const ReactDOM = require('react-dom')
const Router = require('./router.jsx')

const mapping = { ②
  '#profile': <div>Profile <a href="#">home</a></div>,
  '#accounts': <div>Accounts <a href="#">home</a></div>,
  '*': <div>Dashboard<br/>
    <a href="#profile">Profile</a>
    <br/>
    <a href="#accounts">Accounts</a>
  </div>
}

ReactDOM.render(
  <Router mapping = {mapping}/>, ③
  document.getElementById('content')
)

```

- ① Use CommonJS `require()` for module importation with Webpack bundling
 ② Use route mapping object which maps routes to individual pages
 ③ Pass mapping to Router

Now we should implement `Router` in `router.jsx`.

13.1.3 Creating the Router in router.jsx

In a nutshell, Router needs to take information from the URL (e.g., #profile) and map it to the JSX using mapping prop provided to it. There's a way to access URL from `window.location.hash` of the browser API.

```
const React = require('react')
module.exports = class Router extends React.Component {
  constructor(props) {
    super(props)
    this.state = {hash: window.location.hash}
    this.updateHash = this.updateHash.bind(this)
  }
  render() {
    ...
  }
}
```

But that's not all. Next, we will need listen to any changes to it with `hashchange`. If we don't implement listening to new URLs, then our router will work only once, that's when the element Router create which is on a whole page reload. The best place to attach and remove listeners for hashchange is `componentDidMount()` and `componentWillUnmount()` lifecycle event listeners.

```
updateHash(event) {
  this.setState({hash: window.location.hash})
}
componentDidMount() {
  window.addEventListener('hashchange', this.updateHash, false)
}
componentWillUnmount() {
  window.removeEventListener('hashchange', this.updateHash, false)
}
```

componentDidMount and componentWillUnmount

The lifecycle events are described in chapter 5, but if you need a refresher: `componentDidMount()` fired when the element is mounted and has the actual DOM node. For this reason, this is the safest place to attach events which integrate with other DOM objects, and also to make AJAX/XHR calls (not used here).

On the other hand, `componentWillUnmount()` is the best place to remove event listeners. The reason for that is that our element will be unmounted and we need to remove whatever we created outside of this element (such as event listener on `window`). Leaving a lot of events listeners hanging in there, without the elements which created and used them, is a bad practice because it will lead to performance issues such as memory leaks.

In `render()`, we simply use an `if/else`. We see if there's a match with the current URL value (`this.state.hash`) and the keys/attributes/properties in the prop `mapping`. If yes, then we access `mapping` again to get the content of the individual page (JSX). If no, then we fallback to `*` for all other URLs including the empty value (home page).

Listing 13.3 Implementing URL router from scratch by listening to hashchange

(ch13/naive-router/jsx/router.jsx)

```
const React = require('react')
module.exports = class Router extends React.Component {
  constructor(props) {
    super(props)
    this.state = {hash: window.location.hash}      ①
    this.updateHash = this.updateHash.bind(this)
  }
  updateHash(event) {
    this.setState({hash: window.location.hash})
  }
  componentDidMount() {
    window.addEventListener('hashchange', this.updateHash, false) ②
  }
  componentWillUnmount() {
    window.removeEventListener('hashchange', this.updateHash, false)
  }
  render() {
    if (this.props.mapping[this.state.hash])
      return this.props.mapping[this.state.hash]      ③
    else
      return this.props.mapping['*']
  }
}
```

- ① Assign initial URL hash value
- ② Keep feeding the new URL hash values when they change
- ③ Render the content corresponding to the URL base

Finally, in `index.html`, we include CSS file and `bundle.js` which Webpack will produce when we run `npm run build` (which in turn runs `./node_modules/.bin/webpack -w`):

```
<!DOCTYPE html>
<html>

  <head>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
    <link href="css/main.css" type="text/css" rel="stylesheet"/>
  </head>

  <body>
    <div id="content" class="container"></div>
    <script src="js/bundle.js"></script>
  </body>

</html>
```

Run the bundler to get `bundle.js` and open the web page in a browser. Clicking on the links will change the URL as well as the content of the page already was shown in Figure 13-1.

As you can witness, building your own router with React is straightforward. We were able to utilize lifecycle methods to listen for changes in the hash to render various content.

Although building your own front-end router is a viable option, things might start to become more complex if you nested routes, routes parsing (extracting URL parameters), or use you need "nice" URLs without the hash #.

While we can use a router from Backbone or some other front-end MVC-like framework, there's a solution designed for React specifically (hint: it uses JSX!).

13.2 React Router

React is amazing at building UIs. If I haven't convinced you yet, go back and reread the previous chapters. :) It's also can be used to implement simple URL routing from scratch as you've observed with `router.jsx`. Simple is good, right?

However, for more sophisticated SPAs, developers will need more features. For instance, passing of a URL parameter is a common feature to signify an individual item rather than a list of items. It could look like `/posts/57b0ed12fa81dea5362e5e98` where `57b0ed12fa81dea5362e5e98` is a unique post ID. We can extract this URL parameter using a regular expression but sooner or later if your application grows in complexity, you might find your self re-inventing existing implementations for the front-end URL routing.

Semantic URLs

Semantic or nice URLs is a concept aimed to improving usability and accessibility of a website/webapp by decoupling internal implementation from UI. A non-semantic way might include usage of query strings and/or script file name. On the other hand, semantic way embraces usage of the path only in a manner and hierarchy that helps users to interpret the structure and manipulate the URL.

For example:

Non-Semantic (ok)	Semantic (better)
<code>http://webapplog.com/show?post=es6</code>	<code>http://webapplog.com/es6</code>
<code>https://www.manning.com/books/react-quickly?a_aid=a&a_bid=5064a2d3</code>	<code>https://www.manning.com/books/react-quickly/a/5064a2d3</code>
<code>http://en.wikipedia.org/w/index.php?title=Semantic_URL</code>	<code>https://en.wikipedia.org/wiki/Semantic_URL</code>

Major frameworks, for instance Ember, Backbone, and Angular, all have routing built into them. When it comes to routing and React, `react-router` is one of those ready-to-go off-the-shelf solutions. We will cover React and Backbone implementation later in this chapter as well to illustrate how nicely React plays with this MVC-like framework which many developers use for their SPAs. Right now, let's focus on React Router though.

React Router is not part of the official React core library. It came from community, but it's mature and popular enough ([a third of React projects use React Router](#)) to be considered for your projects. In fact, it's one of the default options to consider for most React engineers I talked to.

The syntax of React Router uses JSX, which is another plus because it allows developers to create more readable hierarchical definitions than with a mapping object (as seen in the previous project).

The way React Router works is very similar to our naive Router implementation in the previous section in a way that there's `Router` React component (you can say that React Router inspired my implementation of naive Router!):

1. Create mapping in which URLs will translate into React components (which turn into markup on a webpage). In React Router, this is achieved by passing `props path` and `component` as well as nesting `Route`. The mapping is done in JSX by declaring and nesting `Route` components. We must implement this part for each new project.
2. Utilize React Router components `Router` and `Route` which will do all the magic of changing the views according to the changes in URLs. Obviously, we won't implement this part, but we will need to install the library.
3. Render `Router` on a webpage by mounting it with `ReactDOM.render()` just like a regular React element. Needless to say, this part needs to be implemented for each new project.

We use JSX to create a `Route` for each page and nest them either in another `Route` or in `Router`. The `Router` object goes in the `ReactDOM.render()` function, like any other React element.

```
ReactDOM.render(
  <Router ...>
    <Route ...>
      <Route .../>
      ...
    </Route>
    <Route .../>
  </Router>
), document.getElementById('content'))
```

Each `Route` has at least two properties. One is `path`, which is the URL pattern to match to trigger this route. The other property is `component`, which will fetch and render the necessary component. You can have more properties on routes, such as event handlers or data. They'll be accessible in `props.route` in side of that route component. This is how we pass data to route components.

To illustrate, let's consider the following example of a SPA with routing to a few pages: About, Posts (like a blog), individual Post, Contact Us and Login. They have different paths and render from different components:

- **About:** /about
- **Posts:** /posts
- **Post:** /post
- **Contact:** /contact

Pages About, Posts, Post and Contact Us will utilize the same layout (Content component) and simply render inside of it. This would be the React Router code (note: this is not a complete, final version though):

```
<Router>
  <Route path="/" component={Content} >
    <Route path="/about" component={About} />
      <Route path="/about/company" .../>
      <Route path="/about/author" .../>
    <Route path="/posts" component={Posts} />
    <Route path="/posts/:id" component={Post}/>
    <Route path="/contact" component={Contact} />
  </Route>
</Router>
```

Interestingly, *developers can nest routes to reuse layouts from parents and their URLs could be independent of nesting*. For instance, it's possible to have a nested component About with the /about URL, even than the "parent" layout route Content says /app:

```
<Router>
  <Route path="/app" component={Content} >
    <Route path="/about" component={About} />
    ...
  </Route>
</Router>
```

In other words, About doesn't have to have nested URLs /app/about.

To navigate, we'll implement a menu as shown in 13-TK. The menu, as well as the header, will be rendered from Content and reused in About, Posts, Post and Contact. Take a look at end goal of implementing the About page on Figure 13-2.

There are a few things happening: About is rendered, the menu button is active, the URL reflects that we are on the About page by showing us /#/about, and the text [Node.University](#) is what we have in the About component (you'll see it later).

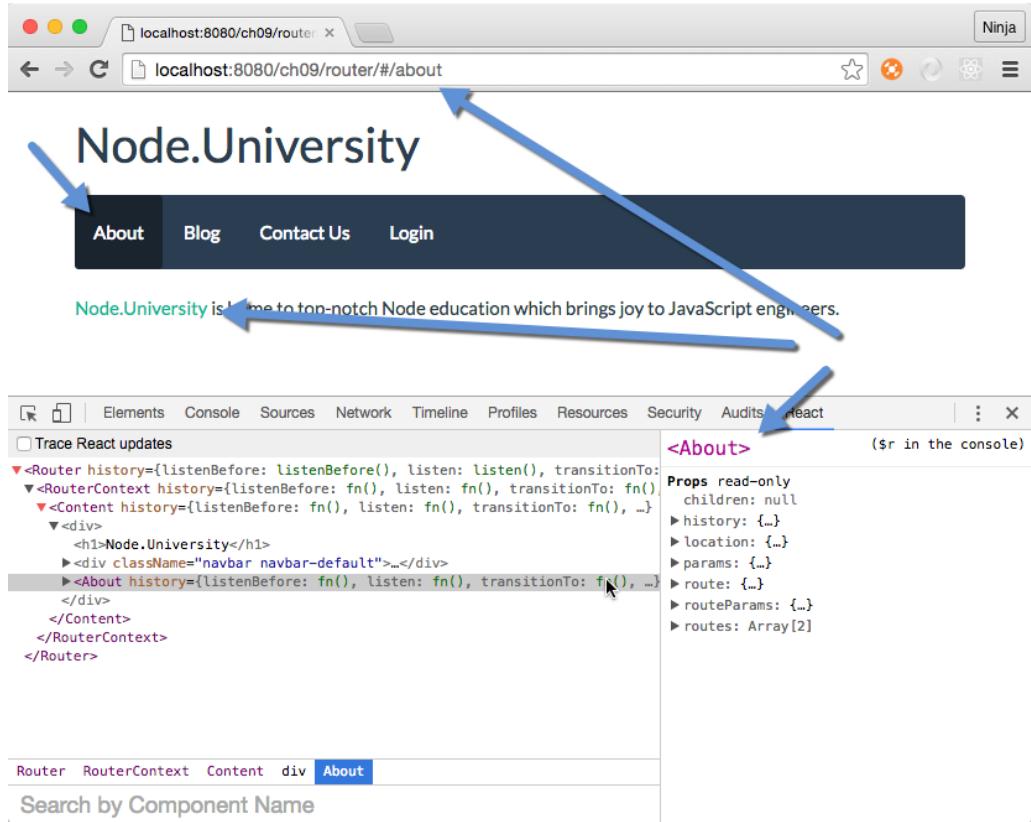


Figure 13.3 Navigating to /about will render About text inside of Content, change URL and make the button active

The About, Posts, and Contact Us pages have a common menu, so we can nest them under one route (Content).

13.2.1 React Router's JSX Style

As mentioned before, developers leverage JSX to create the `Router` element and `Route` elements nested within it (and each other). Each element (`Router` or `Route`) has at least two properties: `path` and `component` which tell router the URL path and the React component class to create and render. It's possible to have additional custom properties/attributes to pass data. We'll use it to pass `posts` array.

Let's put our knowledge together by importing the React Router objects and using them in `ReactDOM.render()` to define the routing behavior. Moreover in addition to `About` `/about`, `Posts` `/posts`, and `Contact` `/contact`, we will create a `Login` `/login`.

**Listing 13.4 Part of the app.jsx implementation where we define the Router
(ch13/router/jsx/app.jsx)**

```
const ReactRouter = require('react-router')
let { Router,
      Route,
      Link
} = ReactRouter

ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={Content} >
      <Route path="/about" component={About} />
      <Route path="/posts" component={Posts} posts={posts}/>
      <Route path="/posts/:id" component={Post} posts={posts}/>
      <Route path="/contact" component={Contact} />
    </Route>
    <Route path="/login" component={Login}/>
  </Router>
), document.getElementById('content'))
```

This last route `Login` (`/login`) lives outside of the `Content` route and won't have the menu (which is in `Content`). To summarize, anything that doesn't need the common interface provided in `Content` (`menu!`), such as the login page (Figure 13-3), can be left outside of the `Content` route. This behavior determined by the hierarchy of the nested routes.

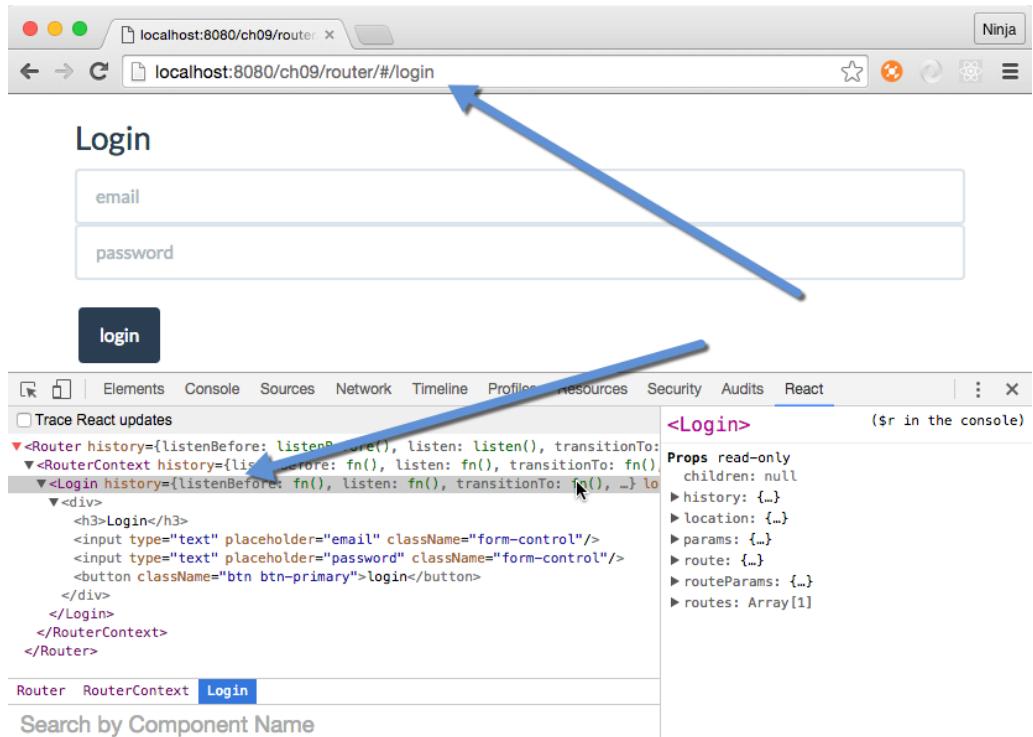


Figure 13.4 Login page `#/login` is not using common layout (Content) with the menu, hence there's no menu, only Login element

At the same time, the `Post` component will render some blog post information based on the post slug (part of the URL—think ID), which it'll get from the URL (for example, `/posts/http2`) via the `props.params.id` variable. In essence, by using a special syntax with a colon in the path, we tell router to parse that value and populate it into `props.params`.

As you can observe, the entire `Router` is passed to the `ReactDOM.render()` method. One thing to notice here is that I'm passing `history` to the `Router`. Since version 2 of React Router, we must supply a history implementation. There are two choices: you bundle with React Router history or standalone history implementations.

13.2.2 Hash History

Hash history as you can guess is relying on the hash symbol `#` which is how we navigate inside of the page without reloading it. For example, `router/#/posts/http2`. The proper name for hash is fragment identifier. Most SPA leverage hash because they need to reflex a change in

the context within their apps without causing a complete refresh (request to the server). That's what we used in implementing router from scratch.

In this example, we will also be using hash. It comes from a standalone history from `history`. We need to import the library, initialize it and pass to React Router.

During the initialization, there's a `queryKey` set to false when initializing history. This is because we want to disable the pesky query string (e.g., `?_k=v18reh`), which is there by default to support older browsers for transferring states when navigating. You can set `false`, when you define `history`:

```
const ReactRouter = require('react-router')
const History = require('history')
let hashHistory = ReactRouter.useRouterHistory(History.createHashHistory)({
  queryKey: false
})
<Router history={hashHistory}/>
```

To use a bundled hash history, simply import it from React Router like this:

```
const { hashHistory } = require('react-router')
<Router history={hashHistory} />
```

You can use a different history implementation with React Router. Old browsers love hash history, but that means you'll see the `#` hashtag. If you need URLs without it, then you can use browser history with some server modifications listed [here](#). To keep it simple here, we'll use hash history but without tokens.

13.2.3 Browser History

An alternative to hash history would be the browser HTML5 pushState history. For example, browser history URL could be `router/posts/http2` and not `router/#/posts/http2`. Another name for this browser history URLs style is real URLs.

Browser history uses regular and not fragmented URLs, so each request will trigger a server request. That's why this approach requires some server-side configuration which we won't cover it at the moment. Typically, SPAs should use fragmented/hash URLs, especially if you need to support older browsers, because browser history requires more complex implementation.

Developers can use browser history similarly to hash history. First, import the module, then plug it in, and finally, configure the server to serve the same file (and not the file from your SPAs routing).

Browser implementations come from a standalone custom package (like `history`) or from the implementation inside of React Router (`ReactRouter.browserHistory`). After we imported the browser history library, apply it to `Router`:

```
const { browserHistory } = require('react-router')
```

```
<Router history={browserHistory} />
```

Then, you need to modify the server as outlined [here](#). The idea is that once you switch to real URLs they would be hitting the server. It needs to serve the same SPA JavaScript code to every request such as `/posts/57b0ed12fa81dea5362e5e98` and `/about`.

Because hash history is the preferred way to implement URL routing in older browsers and to keep it simple (without having to implement the back-end server), we'll use hash history in this chapter.

13.2.4 React Router Dev Setup with Webpack

When working with React Router, there are libraries developers need to use and import as well as JSX compilation to run. Let's cover the development setup for the React Router using Webpack which will perform aforementioned tasks.

Here's the `devDependencies` from `package.json`. Most of them familiar to you already. From the new packages, we have `history` and `react-router`. As always the case, make sure you're using the exact versions provided otherwise we can't guarantee that the code will run.

**Listing 13.5 All dependencies we need to run React Router, React and JSX
(ch13/router/package.json)**

```
{
  ...
  "devDependencies": {
    "babel-core": "^6.11.4",
    "babel-loader": "^6.2.4",
    "babel-preset-react": "^6.5.0",
    "history": "^2.1.2",
    "react": "^15.2.1",
    "react-dom": "^15.2.1",
    "react-router": "^2.6.0",
    "webpack": "1.12.9"
  }
}
```

In addition to `devDependencies`, `package.json` must have `babel` configurations. Also, I recommend adding `npm scripts`:

```
{
  ...
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "./node_modules/.bin/webpack -w",
    "i": "rm -rf ./node_modules && npm cache clean && npm install"
  },
  "babel": {
    "presets": [
      "react"
    ]
  },
  ...
}
```

```
}
```

Please note, that because JSX will be converted to `React.createClass()` we will need to import and define `React` in those files which use JSX even when there's no usage of `React`. To illustrate my point, in `About` component (which is stateless, that is a function), it seems like there are no usages of `React`. However, when this code is transpiled, there would be the usage, hence we need to define `React`:

Listing 13.6 Defining React even in the absence of an obvious usage of it because JSX will need it (ch13/router/jsx/about.jsx)

```
const React = require('react')

module.exports = function About() {
  return <div>
    <a href="http://Node.University" target="_blank">Node.University</a>
      is home to top-notch Node education which brings joy to JavaScript engineers.
  </div>
}
```

The rest of the files and the project as whole will look like this:

```
/router
  /css
    - bootstrap.css
    - main.css
  /js
    - bundle.js  ①
    - bundle.js.map
  /jsx
    - about.jsx
    - app.jsx
    - contact.jsx
    - content.jsx
    - login.jsx
    - post.jsx
    - posts.jsx
  /node_modules
    - index.html
    - package.json
    - posts.js  ②
    - webpack.config.js
```

- ① Bundled (concatenated) file and its source map for better debugging
- ② Data for the blog posts such as URLs, titles, and text

The `index.html` will be very bare bone because it'll include only the bundled file:

```
<!DOCTYPE html>
<html>

  <head>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
    <link href="css/main.css" type="text/css" rel="stylesheet"/>
  </head>
```

```

<body>
  <div id="content" class="container"></div>
  <script src="js/bundle.js"></script>
</body>

</html>

```

The `webpack.config.js` needs to have at least an entry point `app.jsx` and `babel-loader`:

**Listing 13.7 Configuring Webpack with entry point, Babel and source maps
(ch13/router/webpack.config.js)**

```

module.exports = {
  entry: './jsx/app.jsx',
  output: {
    path: __dirname + '/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',           ①
  stats: {
    colors: true,
    reasons: true
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}

```

- ① Set `devtool` value to see proper mapping to your JSX source code, not the transpiled one

Phew, we implemented the dev setup consisting of `webpack.config.js`, and dependencies. We also have a sense of the project structure. Now, let's implement the Content layout component.

13.2.5 Creating a Layout Component

The `Content` component will serve as a layout for `About`, `Posts`, `Post` and `Contact` components because it's defined as a parent `Route`. Let's see how to implement it and how to get some useful information from the router (Figure 13.5).

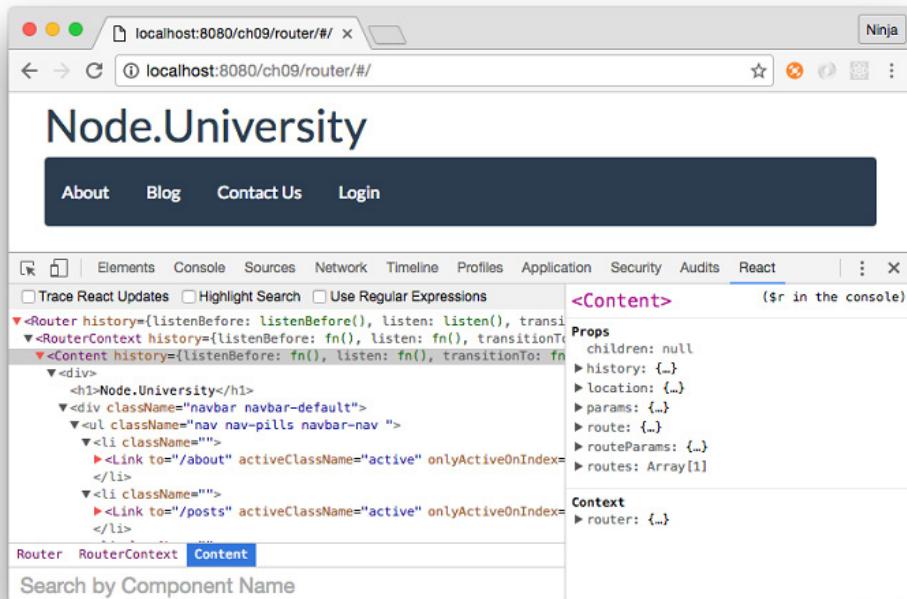


Figure 13.5 content component as the home page (no children)

First, we import React and Link from React Router. The latter is a special component to render the navigation links, that's `Link` is a special wrapper for `<a>`. `Link` has some magic attributes that the normal anchor tag doesn't, such as `activeClassName="active"`, which will add the class `active` when this route is active.

The Content's structure will look something like this with the omission of a few pieces:

```
const React = require('react')
const {Link} = require('react-router')

class Content extends React.Component {
  render() {
    return (
      <div>
        ...
      </div>
    )
  }
}

module.exports = Content
```

Inside of the `render()`, we use amazing UI library [Twitter Bootstrap](#) to declare menu item with the proper classes. The menu can be created using ready-made CSS classes, such as:

```
<div className="navbar navbar-default">
  <ul className="nav nav-pills navbar-nav ">
    <li ...>
      <Link to="/about" activeClassName="active">
        About
      </Link>
    </li>
    <li ...>
      <Link to="/posts" activeClassName="active">
        Blog
      </Link>
    </li>
    ...
  </ul>
</div>
```

We can access `isActive()` method which returns true or false. This way, an active menu link will be visually different from others links. For example,

```
<li className={(this.context.router.isActive('/about'))? 'active': ''}>
  <Link to="/about" activeClassName="active">
    About
  </Link>
</li>
```

Notice `activeClassName` attribute of `Link`. By setting this attribute to a value, link will use apply the class to an active element (selected link). But we need to set the style on ``, not just on `Link`. That's why we also access `router.isActive()`.

Then after we are done with the `Content` class definition (full implementation soon after), we define a static field/attribute `contextTypes` which will enable the usage of `this.context.router`. If you are using ES8+, you might have support for static fields, but not in ES6 or ES7 as of this writing (Aug, 2016). This static attribute will be used by React Router in a way that if it's required, React Router will populate `this.context` (from which we can access `router.isActive()`, and other methods):

```
Content.contextTypes = {
  router: React.PropTypes.object.isRequired
}
```

Having `contextType` and `router` set to required will allow us to have access to `this.context.router.isActive('/about')` which in turn will tell us when this particular route is active.

Whew. Here's the full implementation of the `Content` layout in which we use `this.props.children`. It's important to use children because that's where React Router will display nested routes and corresponding components such as `About`, `Contact` `Posts` and `Post`.

```

const React = require('react')
const {Link} = require('react-router')

class Content extends React.Component {
  render() {
    return (
      <div>
        <h1>Node.University</h1>
        <div className="navbar navbar-default">
          <ul className="nav nav-pills navbar-nav">
            <li className={((this.context.router.isActive('/about'))? 'active': '')}>
              ①
              <Link to="/about" activeClassName="active">
                About
              </Link>
            </li>
            <li className={((this.context.router.isActive('/posts'))? 'active': '')}>
              <Link to="/posts" activeClassName="active">
                Blog
              </Link>
            </li>
            <li className={((this.context.router.isActive('/contact'))? 'active': '')}>
              <Link to="/contact" activeClassName="active">
                Contact Us
              </Link>
            </li>
            <li>
              <Link to="/login" activeClassName="active">
                Login
              </Link>
            </li>
          </ul>
        </div>
        {this.props.children} ③
      </div>
    )
  }
  Content.contextTypes = {
    router: React.PropTypes.object.isRequired ④
  }
  module.exports = Content
}

```

- ① Accessing Router and its method to check the active route
- ② Using `Link` to create a navigation link
- ③ Children routes (defined in app.jsx) will be rendered here
- ④ Defining that this component needs router object in the context

The `children` statement enables us to reuse the menu on every subroutine (route nested in the / route), such as `/products`, `/product`, `/about`, and `/contact`:

```
{this.props.children}
```

Let's see another way to access router inside of an individual route besides using `contextTypes` on the example of the `Contact` implementation.

13.3 React Router Features

React Router has certain features and patterns. Let's take a look on another way how to access router from child components, how to navigate programmatically within those components and of course the chapter won't be complete without covering how to parse URL parameters and pass data around.

13.3.1 Accessing Router with withRouter HOC

Having router allows to navigate programmatically and access current route among other things. It's a good thing to have access to router in your component.

We've seen how to access router from context `this.context.router` by setting the static class attribute:

```
Content.contextTypes = {
  router: React.PropTypes.object.isRequired
}
```

We've used this approach in `Content`. However, context depends on React's context (which is an experimental approach discouraged by the React team), and static attributes which are not available in stateless components defined via functions. Thus, there's another ([some](#) might argue simpler and better) way to provide router and it's called `withRouter`.

`withRouter` is a higher-order component (more about HOCs in chapter 8) which will take a component as an argument, inject router and return another (HOC) component. To illustrate, we can inject router into `Contact` like this:

```
const {withRouter} = require('react-router')
...
<Router ...>
  ...
    <Route path="/contact" component={withRouter(Contact)} />
</Router>
```

In essence, we inject router into `Contact`. When we look into the `Contact` component implementation (a function), the `router` object will be accessible from the props (argument object to the function):

```
const React = require('react')

module.exports = function Contact(props) {
  // props.router - GOOD!
  return <div>
  ...
</div>
}
```

The advantage of `withRouter` is that it'll work with regular stateful React classes as well as with stateless functions.

NOTE Even there's no direct (visible) usage of React, we must require React because this code will be converted to the code with `React.createElement()` statements which depends on the React object. For more info, please refer to chapter 3 on how exactly JSX transforms.

13.3.2 Navigating Programmatically

One of the popular usages of `router` is to navigate programmatically meaning change the URL (i.e., `location`) from within your code based on some logic and not the user actions. To demonstrate, an app has a contact form which must be redirected to an error page, a thank you or an about page upon the form submission by the user. In this case, user types a message, submits a form and then the app navigates based on the server response.

Once we have the router, we can navigate programmatically if we need to by calling `router.push(URL)` where `URL` must be a defined route path. For instance, it's possible to navigate to About from Contact after one second:

Listing 13.8 By calling `router.push()`, you can navigate programmatically.

```
const React = require('react')

module.exports = function Contact(props) {
  setTimeout(()=>{props.router.push('about')}, 1000) ①
  return <div>
    <h3>Contact Us</h3>
    <input type="text" placeholder="your email" className="form-control"></input>
    <textarea type="text" placeholder="your message" className="form-control"></textarea>
    <button className="btn btn-primary">send</button>
  </div>
}
```

① Navigate away after one second by calling ``router.push()``

Navigating programmatically is an important feature because it allows to change the state of the application. Let's implement how we accessing URL param such as post ID.

13.3.3 URL Params and Other Route Data

As you can observe, having `contextTypes` and `router` will give us `this.context.router` object. It's an instance of `<Router/>` defined in `app.jsx` and it can be used to navigate, get active path, etc. On the other hand, there are other interesting information in `this.props` and there's no need for a static attribute to access it:

- `history`: Deprecated in v2.x, we can use `context.router`
- `location`
- `params`
- `route`
- `routeParams`
- `routes`

The `this.props.location` and `this.props.params` object have data about the current route such as path name, URL parameters (names defined with a colon :), etc.

Let's use `params.id` in `post.jsx` for the `Post` component in `Array.find()` to find the post corresponding to the URL path, e.g., `router/#/posts/http2`.

**Listing 13-9: Using URL parameter in a stateless component and rendering post data
(ch13/router/jsx/post.jsx)**

```
const React = require('react')

module.exports = function Product(props) {
  let post = props.route.posts.find(element=>element.slug === props.params.id) ①
  return (
    <div>
      <h3>{post.title}</h3>
      <p>{post.text}</p>
      <p><a href={post.link} target="_blank">Continue reading...</a></p>
    </div>
  )
}
```

① Finding a post by its slug property

When you navigate to the Posts page (Figure 13-5), there's a list of posts. To remind you, the route definition is this:

```
<Route path="/posts" component={Posts} posts={posts}/>
```

Clicking on each of them will navigate to `#/posts/ID`, and that page will reuse the layout of the `Content` component as well.

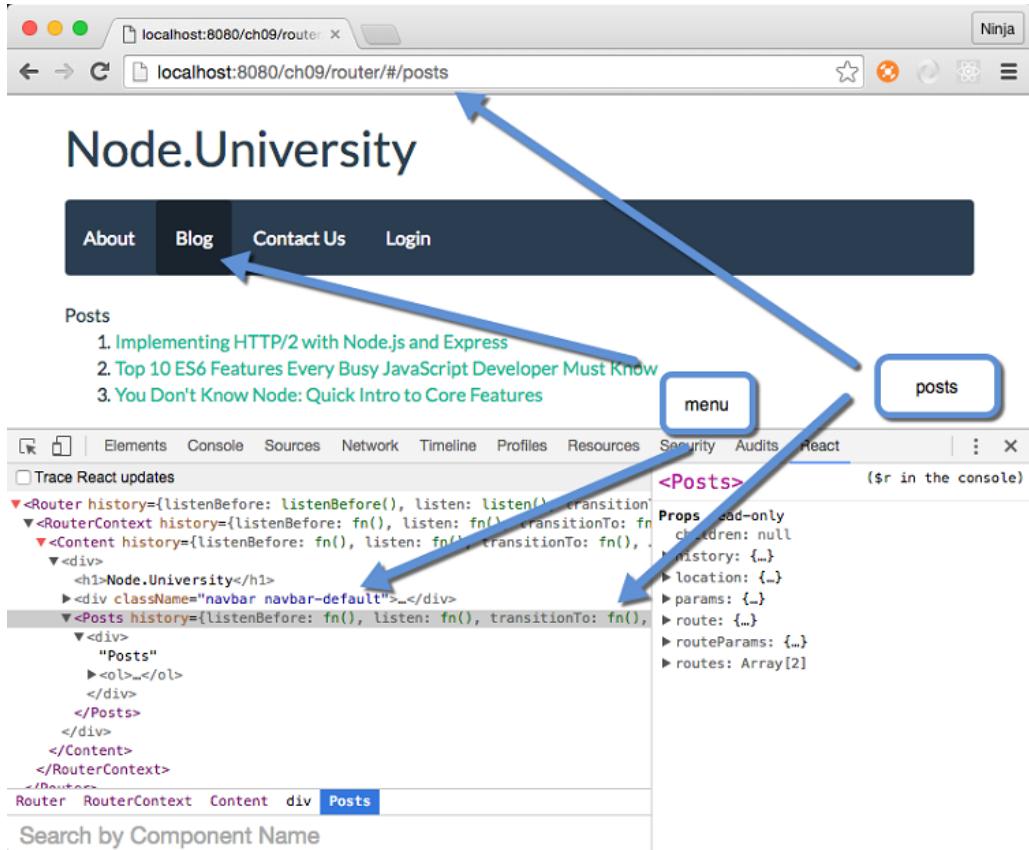


Figure 13.6 The Posts page renders the Posts component inside of the Content (menu) component as it is defined as a child route to Content in app.js

Now we are moving on to working with data.

13.3.4 Passing Props in React Router

Developers often need to pass data to nested routes. It's easy to do. In fact in our example, Posts needs to get the data about posts.

Here's the listing of Posts in which we are accessing a property passed to us in `<Route/>` in `app.jsx`, that is `posts` from the `posts.js` file. Thus, it's possible to pass any data to a route as an attribute, e.g., `<Route path="/posts" component={Posts} posts={posts}/>` and then access this data in `props.route`, e.g., `props.route.posts`.

Needless to say, this value of this data can be a function. This way you can pass to stateless components event handlers and implement them only in the main component such as `app.jsx`.

```
const {Link} = require('react-router')
const React = require('react')

module.exports = function Posts(props) {
  return <div>Posts
    <ol>
      {props.route.posts.map((post, index)=>          ①
        <li key={post.slug}><Link to={`/posts/${post.slug}`}>{post.title}</Link></li>
      )}
    </ol>
  </div>
}
```

① Access attribute which we defined in the route declaration

Woo. We are done with all the major parts. Now, we are ready to launch this project now! You can do so by running an npm script `npm run build` or `./node_modules/.bin/webpack -w` directly. Wait for the build to finish, and you'll see something like this:

```
> router@1.0.0 build /Users/azat/Documents/Code/react-quickly/ch13/router
> webpack -w

Hash: 07dc6eca0c3210dec8aa
Version: webpack 1.12.9
Time: 2596ms
    Asset      Size  Chunks      Chunk Names
  bundle.js    976 kB       0  [emitted]  main
bundle.js.map  1.14 MB       0  [emitted]  main
+ 264 hidden modules
```

In a new window, open your favorite static server (I use `node-static` but you can create your own using Express) and navigate to the location in your browser. Try going to `/` and `/#/about` (the exact URL will depend if you are running your static server from the same folder or a parent folder).

NOTE The full source code for this example won't be listed in the book. If you would like to play with it or use it as boilerplate, or if you found the preceding snippets somewhat confusing when taken out of context, then take a look at `ch13/router`, [GitHub](#).

13.4 Routing with Backbone

When you need routing for your React single-page application, it's rather straightforward to use React with other routing or MVC-like libraries. For example, Backbone is one of the most popular front-end frameworks which has front-end URL routing built-in.

Developers can easily leverage Backbone router to render React component by:

1. Defining a router class with `routes` object as a mapping from URL fragments to functions
2. Rendering React elements in the methods/functions of the Backbone `Router` class
3. Instantiating and starting the Backbone `Router` object

This is the project structure:

```
/backbone-router
  /css
    - bootstrap.css
    - main.css
  /js
    - bundle.js
    - bundle.map.js
  /jsx
    - about.jsx
    - app.jsx
    - contact.jsx
    - content.jsx
    - login.jsx
    - post.jsx
    - posts.jsx
  /node_modules
    ...
  - index.html
  - package.json
  - posts.js
  - webpack.config.js
```

The main logic's source is in the `app.jsx` where we perform all three aforementioned tasks:

```
// Include Libraries
const Router = Backbone.Router.extend({
  routes: {
    '' : 'index',
    'about' : 'about',
    'posts' : 'posts',
    'posts/:id' : 'post',
    'contact' : 'contact',
    'login': 'login'
  },
  ...
})
```

Once the `routes` object is defined, we can define the methods. The values in `routes` will must be used as method names.

```
// Include Libraries
const Router = Backbone.Router.extend({
  routes: {
    '' : 'index',
    'about' : 'about',
    'posts' : 'posts',
    'posts/:id' : 'post',
    'contact' : 'contact',
    'login': 'login'
```

```
  },
  index: function() {
    ...
  },
  about: function() {
    ...
  }
...
})
```

Each URL fragment will map to a function for example, `#/about` will trigger `about`. Thus, we can define these functions and render our React components in them. The data will be passed as a prop (`router` or `posts`).

```
const Router = Backbone.Router.extend({
  routes: {
    ...
  },
  index: function() {
    ReactDOM.render(<Content router={router}/>, content)
  },
  about: function() {
    ReactDOM.render(<Content router={router}>
      <About/>
    </Content>, content)
  },
  posts: function() {
    ReactDOM.render(<Content>
      <Posts posts={posts}/>
    </Content>, content)
  },
  post: function(id) {
    ReactDOM.render(<Content>
      <Post id={id} posts={posts}/>
    </Content>, content) ②
  },
  contact: function() {
    ReactDOM.render(<Content>
      <Contact />
    </Content>, content)
  },
  login: function() {
    ReactDOM.render(<Login />, content) ③
  }
})
let router = new Router() ④
Backbone.history.start()
```

① Create Content and About inside of it. It is possible to pass the router as a prop

② Pass necessary data to Post such as URL parameter `(id)` and `posts` data

③ Render Login without Content

④ Instantiate Router and start browser history

A `content` variable is simply a DOM node (which we declare before the router):

```
let content = document.getElementById('content')
```

Comparing to the React Router example, the nested components such as `Post` will get their data not in `props.params` or `props.route.posts` but in `props.id` and `props.posts`. In my opinion, that's less magic which is always good. On the other side, we don't get to have declarative JSX syntax and must use more imperative style.

The goal of this example is to give you a head start if you have a Backbone system or planning on using it, but also even if you're not planning to use Backbone to show you yet again how React is amazing at working with other libraries

The rest of the code for this project is in `ch13/backbone-router` and on [GitHub at azat-co/react-quickly](https://github.com/azat-co/react-quickly).

13.5 Quiz

1. Developers must provide a history implementation to React Router v2.x (the one covered in this chapter) because by default it won't use one. True or false?
2. What history implementation is better supported by older browsers: hash history or browser HTML5 pushState history?
3. What developers need to implement to have access to the `router` object in a route component when using React Router v2.x ?
4. How would you access URL parameters in a route component (stateless or statefull) when using React Router v2.x?
5. React Router requires the usage of Babel and Webpack. True or false?

13.6 Summary

In this chapter, we've covered

- Implementing routing with React can be done in naive way by listening to `hashchange`
- React router provides JSX syntax for defining routing hierarchy: `<Router><Route/></Router>`.
- Nested routes don't have to have a nested URLs relative to their parent routes, that's path and nestedness are independent.
- You can use hash history without tokens by setting `queryKey` to `false`.
- Developers must include `React (require('react'))` when they are using JSX even there's no visible usages of `React`, because JSX converts to `React.createElement()` which needs `React`.

All in all, React Router is a nice library to use with React to implement routing. It shares the same syntax and has a lot of features.

13.7 Quiz Answers

1. True, in version 1.x of React Router it used to load a history implementation by default but in version 2.x, developers must provide an library either from a standalone package or one bundled with the router library.
2. Hash history is supported better by older browsers.
3. Static class attribute `contextTypes` with `router` as required object.
4. From `props.params` or `props.routeParams`
5. False. You can use it plain, and/or with other build tools such as Gulp+Browserify.

14

Working with Data Using Redux and GraphQL

This chapter covers

- Unidirectional Data Flow in React
- Flux Data Architecture
- Redux Data Library
- Server Data with Redux, GraphQL and Node



Figure 14.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch14>

So far we've been utilizing React to create user interfaces. This is the most common use case for React. However, most UIs need to work with data. This data comes from either a server (back-end), or another browser component.

When it comes to working with data, React allows for many options such as:

- Integrating with MVC-like frameworks: This option is ideal if you're already using or are planning to use an MVC-like framework for your single-page application, e.g., using

Backbone and Backbone models

- Writing your own data method or a library: This option is well suited for small UI components, e.g., fetching a list of accounts for a List of Accounts grid.
- Using React stack (a.k.a. React and friends): This option offers the most compatibility (your code will integrate with less friction) and the most adherence to React philosophy

This chapter will cover some of the most popular options for the third approach: Redux and GraphQL. Let's start by outlining how data flows in React components.

Important clarification: there's Flux architecture and then there's the flux library (npm name `flux`) by Facebook. Code wise, I will be teaching and using Redux over Flux library, because Redux is being more actively used in projects. Flux serves more of a proof of concept for the Flux architecture which Redux adheres and implements. Think of Redux and `flux` (the library) as the two implementations of the Flux architecture. (I will cover Flux architecture but not the library.)

The source code for the examples in this chapter is in [the ch14 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

14.1 React supports Unidirectional Data Flow

React is a view layer that is designed to work with a **unidirectional data flow**. A unidirectional data pattern (a.k.a. one-way binding) exists when there are no mutable (or two-way) references between concerns. Concerns are parts with different functionality. For example a view and a model cannot have two-way references. We had a diagram in chapter 7 on this but let me show it again in Figure 14.2. We will talk about bi-directional flow again in a few moments.

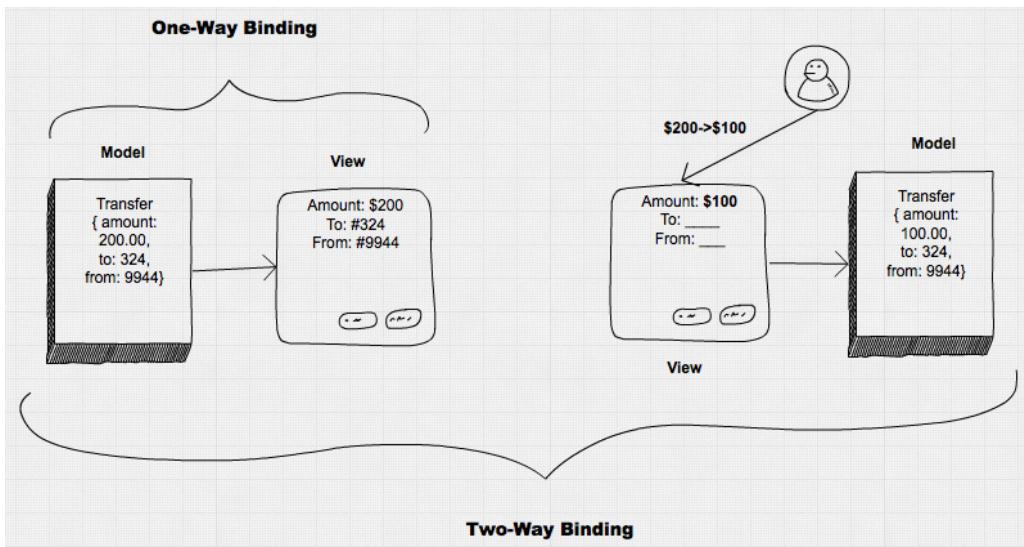


Figure 14.2 Different between uni and bi (one or two-way)

To illustrate, if there are account model and account view, then data can flow only from account model to account view and not vice versa. In other words, changes in model will cause changes in view. (Figure 14.3.) The key to understand is that view cannot modify models directly.

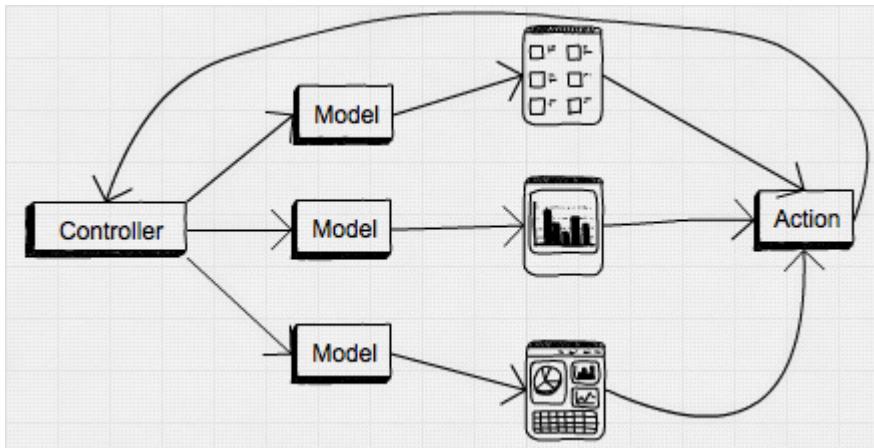


Figure 14.3 A simplified view of unidirectional data flow in which view cannot modify models directly

The unidirectional data flow ensures that for any given input into our components, we'll get the same predictable result of a `render()` expression. This pattern of React is in stark contrast to the bidirectional, two-way binding pattern of Angular.js and Backbone.js.

For example, in bidirectional data flow, changes in models will cause changes in views, **and** changes in views (user input) will cause changes in models. For this reason, with the bidirectional data flow, the state of views is less predictable making it harder to understand, debug and maintain (Figure 14-3). The key to remember is that views CAN modify models directly. This is in stark contrast with the unidirectional flow.

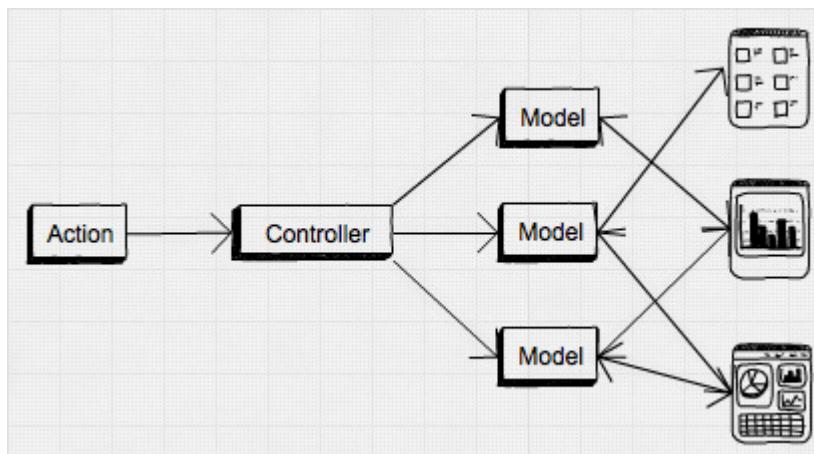


Figure 14.4 A simplified view of bidirectional data flow typical for MVC-like architecture

Interestingly enough, bidirectional data flow or two-way binding is considered by some Angular developers as a benefit. Without getting into a debate, it's true that with bidirectional flow, developers have to write less code.

For example, let's say you have an input field (maybe like the one shown in Figure 14.2). All you need to do is define a variable in the template, and the value will be updated in the model when the user types something. At the same time, the value on the web page will be updated if there's a change in the model (as a result of XHR GET request, for example). Therefore, changes are possible in two directions: from view to model and from model to view. This is great for prototyping, but not so great for complex UIs with performance, debugging, development scaling, and so on. This might sound controversial. Please bear with me.

I've built a lot of complex UI applications with MVC or MVW frameworks that have bidirectional flows. They'll do the job! In a nutshell, the problems arise because various views can manipulate various models and vice versa. That's OK when you have one or two models and views in isolation, but the bigger the application, the more models and views you have

updating each other. It's becomes harder and harder to reason why one model or view is in a given state because you cannot easily determine which models \ views updated it and in which order. That's why the bidirectional data flow in MVC frameworks (such as Angular) is not favored by many developers who find this anti-pattern hard to debug and scale.

On the other hand, with unidirectional flow, the model updates the view and that's that. As an added bonus, unidirectional data flow allows for server-side rendering because views are an immutable function of state (that is, isomorphic/universal JavaScript).

For now, just keep in mind that unidirectional data flow is one of the major selling points when it comes to React:

- Code readability and reasoning due to one source of truth (`state/model->view`).
- Debuggable code with [time travel](#); for example, it's trivial to send a dump with history to the server on exceptions and bugs.
- Server-side rendering without a headless browser; [isomorphic](#) or [universal](#) JavaScript as some call it.

Indeed, Angular 2 has borrowed React's unidirectional data flow and is breaking away from always having two-way binding as in Angular 1.

14.2 Flux Data Architecture

[Flux](#) is an architecture pattern for data flow developed by Facebook to be used in React apps. The gist of Flux is unidirectional data flow and elimination of the complexity of MVC-like patterns.

Let's consider a typical MVC-like pattern (Figure 14.5). You've got actions that trigger events in the controller, which handles models. Then, according to the models, the app renders the views, and the madness starts. Each view updates the models—not just its own model, but the other models too—and the models update the views (bidirectional data flow). It's very easy to get lost in this architecture. It's difficult to understand and debug.

MVC

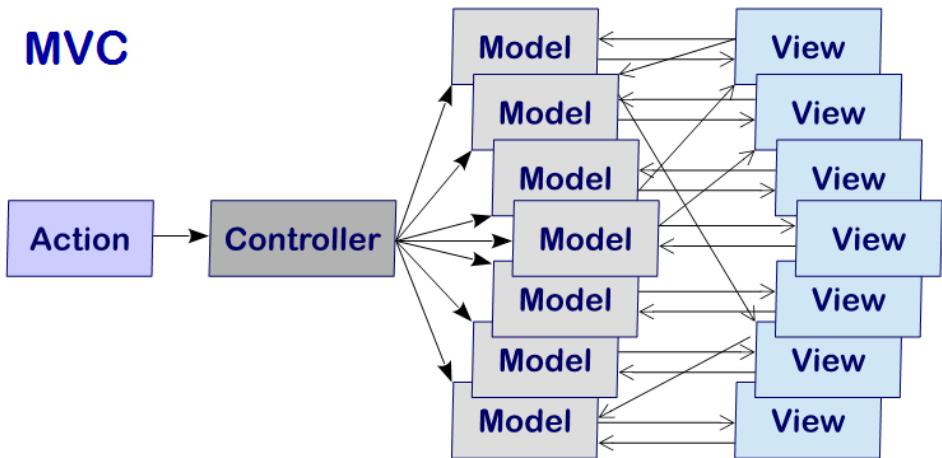


Figure 14.5 MVC-like architecture introduces a lot of complexity by allowing Views to trigger changes on ANY model and vice versa

Conversely, Flux suggests using a unidirectional data flow, as shown in figure X. In this case, you have actions from views going through a dispatcher, which in turn calls the data store. (The Flux is a replacement for MVC. Not just a new terminology.) The store is responsible for the data and the representation in the views. Views don't modify the data, but have actions that go through the dispatcher again.

Flux

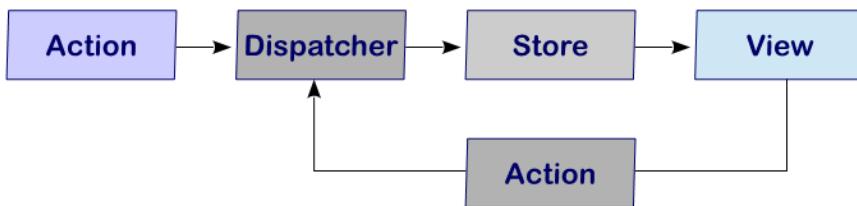


Figure 14.6 Flux architecture simplifies the data flow by using one direction (from Store to View)

The unidirectional data flow enables better testing and debugging. The official Flux library by Facebook is available at <https://github.com/facebook/flux>. A more detailed diagram of Flux architecture is shown in Figure 14.7.

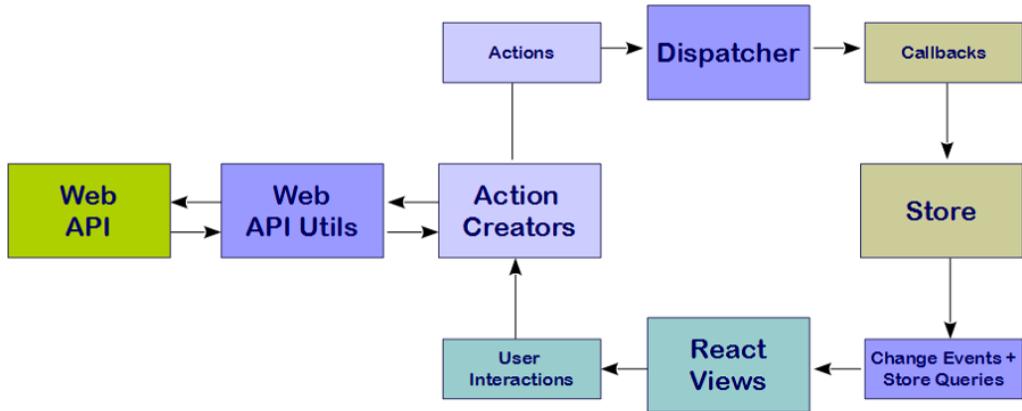


Figure 14.7 Flux architecture in a nutshell in which Actions trigger Dispatcher which triggers Store which renders Views

Historically, Flux was an architecture and only later Facebook team released `flux` module which can be used with React to implement Flux. The `flux` module is more or less a proof of concept for the Flux architecture. React developers rarely use it. There's a better module which adheres to Flux and provide more benefits. Its name is Redux.

There's no reason to duplicate great mind who already spoke on Flux better than me. For this reason, feel free to watch [this video from the official Flux website](#).

Personally, I find Flux too confusing...and I'm not alone. There are many implementations of Flux with Redux and Reflux and other Libraries. Early Manning Access Program readers of React Quickly, my anecdotal [evidence](#) and hard data of npm downloads, they all tell that **Redux is more popular than Flux** or Reflux which was in the first version of this book (I took it out). Given these points, we'll be using Redux, which some might argue is a better alternative to Flux.

14.3 Redux Data Library

Redux is one of the most popular implementations of the Flux architecture. Redux offer the following advantages over Facebook Flux:

- Ecosystem
- Simplicity
- Developer Experience (DX)
- Reducer Composition
- Server-side Rendering

Redux (redux) is a stand-alone library that implements a state container. It's like a huge variable that contains all the data your application works with, stores and changes in the run time. You can use Redux alone or on the server. As already mentioned before, Redux is very popular in combination with React. This combination is implemented in another library that is called `react-redux`.

There are few moving parts when you use Redux in your React apps:

- Store that stores all the data and provides methods to manipulate this data and is created with `createStore` function
- Provider component that makes it possible for any of your component to take data from the Store
- `connect()` function that wraps any of your components and allows to *map* certain parts of your application state from the Store to component's props.

If you remember the Flux architecture diagram Figure 14-7, then you can see while there's store. The only way to mutate the internal state is to dispatch an action. Actions are in the store.

Flux

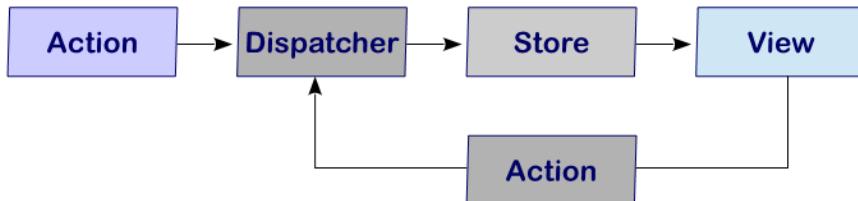


Figure 14.8 Flux architecture simplifies the data flow by using one direction (from Store to View)

Every change in the Store is performed via *actions*. Each action tells your application what exactly happened and what part of the Store should be changed. Actions can also provide data, and you'll find this very useful since, well, every app works with data that changes.

The way the data in the Store changes is prescribed by *reducers* which are pure functions. They have `(state, action) => state` signature. In other words by applying action to the current state, you'll get new state. This allows for predictability and the ability to rewind actions (undo, debugging) to previous states.

Here's the reducer file for a Todo list app in which `SET_VISIBILITY_FILTER` and `ADD_TODO` are actions:

```

function todoApp(state = initialState, action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER': ①
  }
}
  
```

```

        return Object.assign({}, state, {
          visibilityFilter: action.filter
        })
      case 'ADD_TODO':
        return Object.assign({}, state, {
          todos: [
            ...state.todos,
            {
              text: action.text,
              completed: false
            }
          ]
        })
      default:
        return state
    }
}

```

- ① Define an action
- ② Apply reducer to create new state by [copying](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) current state and the `visibilityFilter` values
- ③ Define the action `ADD_TODO`
- ④ Apply reducer to create new state by copying current state and the new TODO values text and completed as the last item of the todos array
- ⑤ Define a default fallback which in this case returns current state

You may have one or many reducers (or none at all) in your Redux application, and every time you call an action, every reducer is called. Reducers are responsible for changing the data in Store, this is why you need to be careful about what any of your reducers do per certain type of action.

Typically, a reducer is a function of state and action that has a huge `switch/case` statement inside, but you'll see there's a handy library that makes your reducers look more functional and, surprise!, more easily readable. For example, an action can be fetch a movie which we get by using a reducer. It describes how an action transforms the state into the next state.

These are the suggested before-going-to-bed reading on Redux from its creator Dan Abramov ([gaearon](#) on GitHub): [Why use Redux over Facebook Flux?](#) and [What could be the downsides of using Redux instead of Flux.](#)

14.3.1 Redux Netflix Clone

We all like good old Hollywood movies, right? Let's make a very classic app that shows you a list of movies, that's a Netflix clone (only the home page part—no streaming or anything like that). The end result of the Netflix clone is to have a grid of movies when you go to the home page with each movie showing in a detailed view once you click on the movie image.

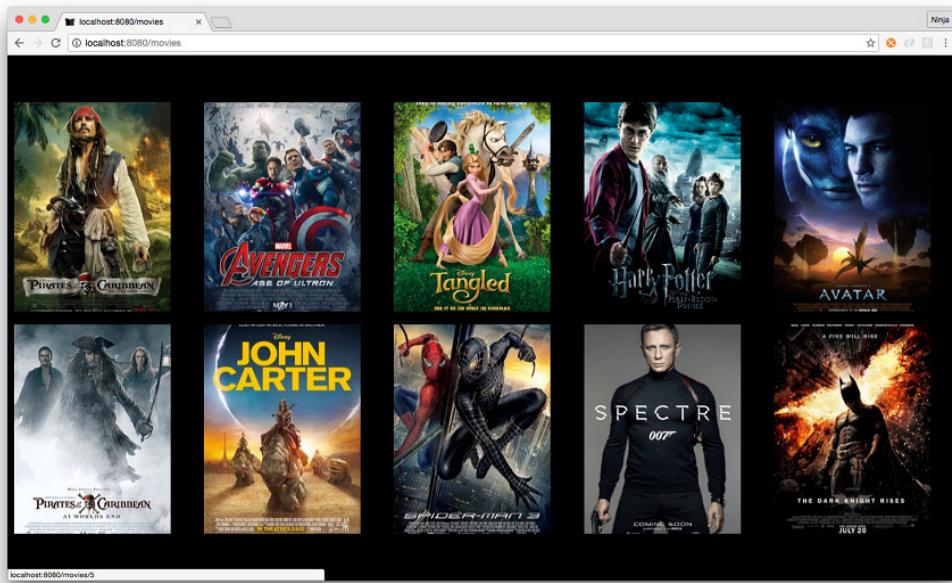


Figure 14.9 Our Netflix clone shows a grid of movies on the home page

The goal of this tutorial is to learn how to use Redux in real-life scenario to feed the data to React components. This data will be coming from a JSON file to keep things simple stupid. Each individual movie detail view will be facilitated with React Router which we've learned how to use in the previous chapter.

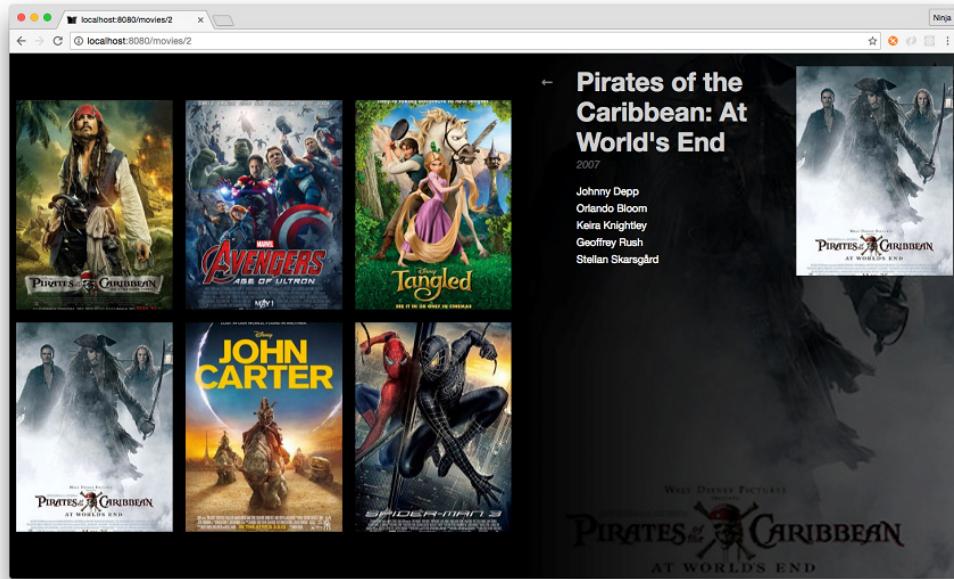


Figure 14.10 Details of a movie are shown when you click on its poster

The project will have three components: `App`, `Movies` and `Movie`. Each component will have a CSS file and live in its own folder for better code organization (that's the best practice to encapsulate React components together with styles). The project structure will go as follows:

```
/redux-netflix
  /build
    - index.js
    - styles.css
  /node_modules
  ...
  /src
    /components
      /App
        - App.css
        - App.js
      /Movie
        - Movie.css
        - Movie.js
    /Movies
      - Movies.css
      - Movies.js
  /modules
    - index.js
    - movies.js
  - index.js
```

```

- movies.json      8
- routes.js       9
- index.html
- package.json
- webpack.config.json

```

- 1 Build folder where Webpack will write bundles
- 2 App folder for the layout component
- 3 Movie folder for individual movie view component
- 4 Movies folder for the grid of movies
- 5 File which will combine reducers and expose them
- 6 Redux reducers to fetch movies and a single movie data
- 7 The entry point of the project which defines Redux Provider with reducers
- 8 Movies data
- 9 React Router routes

Okay so the folder structure of the project is ready. Now, let's take a look at dependencies and build configuration first.

14.3.2 Dependencies and Configs

There are a number of dependencies to setup for this project. We will be using [Webpack](#) to bundle all our files for live use and an additional plugins for it called `extract-text-webpack-plugin` to bundle styles from multiple `<style>` includes (inline) into one `style.css`. Also, there are Webpack loaders:

- json-loader
- style-loader
- css-loader
- babel-loader

Other modules include:

- [Babel](#) and its presets to transpile ECMAScript 6 into browser-friendly old school Javascript aka ECMAScript 5: `babel-polyfill` emulates a full ES2015 environment, `babel-preset-es2015` (for ES6/ES2015), `babel-preset-stage-0` (cutting-edge new ES7+ features) and `babel-preset-react` (for JSX).
- [react-router](#) to show hierarchy of components based on current location. It will also help arrange components into a hierarchy based on URL location.
- [redux-actions](#) to organize the reducers better.
- [ESLint](#) and its plugins to maintain proper JavaScript/JSX style
- [Concurrently](#) a Node tool to have processes (such as Webpack builds) run concurrently, i.e., at the same time

The `package.json` will list all dependencies, Babel configs and npm scripts should at least have the data shown in Listing 14-1. As always, you can install modules manually with `npm i NAME`, type the `package.json` and run `npm i` or copy `package.json` and run `npm i`.

Listing 14.1 Netflix clone needs all of these dependencies (ch14/redux-netflix/package.json)

```
{
  "name": "redux-netflix",
  "version": "0.0.1",
  "description": "A sample project in React and Redux that copies Netflix's features and work flow",
  "main": "./build/index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "concurrently \"webpack --watch --config webpack.config.js\" \"webpack-dev-server\""
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/azat-co/react-quickly.git"
  },
  "author": "Azat Mardan (http://azat.co)",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/azat-co/react-quickly/issues"
  },
  "homepage": "https://github.com/azat-co/react-quickly#readme",
  "devDependencies": {
    "babel-core": "^6.11.4",
    "babel-eslint": "^6.1.2",
    "babel-loader": "^6.2.4",
    "babel-polyfill": "^6.9.1",
    "babel-preset-es2015": "^6.9.0",
    "babel-preset-react": "^6.11.1",
    "babel-preset-stage-0": "^6.5.0",
    "concurrently": "^2.2.0",
    "css-loader": "^0.23.1",
    "eslint": "^3.1.1",
    "eslint-plugin-babel": "^3.3.0",
    "eslint-plugin-react": "^5.2.2",
    "extract-text-webpack-plugin": "^1.0.1",
    "json-loader": "^0.5.4",
    "style-loader": "^0.13.1",
    "webpack": "^1.13.1",
    "webpack-dev-server": "^1.14.1"
    "react": "^15.2.1",
    "react-dom": "^15.2.1",
    "react-redux": "^4.4.5",
    "react-router": "^2.6.0",
    "redux": "^3.5.2",
    "redux-actions": "^0.10.1"
  }
}
```

- ➊ Define the script to build and run Webpack Dev Server using `concurrently` tool
- ➋ Install various Babel plugins, loaders and `modules` (continues on several lines)
- ➌ Install `concurrently` to run npm scripts faster
- ➍ Install `extract-text-webpack-plugin` to combine inline styles into one bundle
- ➎ Install `react-redux` to work with data
- ➏ Install `redux-actions` to organize Redux reducers better

Because we use Webpack to bundle the dependencies, all of the necessary packages will be inside of `bundle.js`. For this reason, I put all dependencies in `devDependencies`. I'm very picky what's get deployed. I don't want any unused modules to be in my deployment environment just sitting their idly and causing security vulnerabilities. `npm` ignores `devDependencies` when `--production` flag is use as in `npm i --production`.

Okay. Now let's define our build process by creating `webpack.config.js`:

Listing 14.2 Webpack configuration file for Netflix clone (ch14/redux-netflix/webpack.config.js)

```
const path = require('path')
const ExtractTextPlugin = require('extract-text-webpack-plugin')

module.exports = {
  entry: {
    index: [
      'babel-polyfill',          1
      './src/index.js'          2
    ]
  },
  output: {
    path: path.join(__dirname, 'build'),      3
    filename: '[name].js'
  },
  target: 'web',
  module: {
    loaders: [{}                      4
      loader: 'babel-loader',
      include: [path.resolve(__dirname, 'src')],
      exclude: /node_modules/,
      test: /\.js$/,
      query: {
        presets: ['react', 'es2015', 'stage-0'] 5
      }
    ],
    {                                6
      loader: 'json-loader',
      test: /\.json$/
    },
    {                                7
      loader: ExtractTextPlugin.extract('style',
        'css?modules&localIdentName=[local]_[hash:base64:5]'),
      test: /\.css$/,
      exclude: /node_modules/
    }
  },
  resolve: {
    modulesDirectories: [
      './node_modules',
      './src'
    ]
  },
  plugins: [
    new ExtractTextPlugin('styles.css') 8
  ]
}
```

- 1 Apply polyfill to *fully* emulate ES2015 environment
- 2 Specify an entry `point` (does not always has to be `*.jsx`)
- 3 Specify the output folder using `path.join()` to make it more robust for cross-platform `usage` (such as on Windows)
- 4 Apply loaders as an array
- 5 Specify Babel `presets` (i.e., what to do with the code)
- 6 Apply JSON loader to mock our database of movies from JSON files
- 7 Apply loader from a plugin to extract styles and combine them into one `file` (instead of having a lot of files)
- 8 Provide plugin for text extraction

Enough with configurations. Next, we'll start working with Redux.

14.3.3 Enabling Redux

To make Redux work in your React application, a hierarchy of components needs to start with the `Provider` component at the top level. This component comes from the `react-redux` package and does a trick of injecting data from Store into components. That's right. By putting `Provider` as the top-level component, all children will have the store. Neat. :)

To make `Provider` work, we need to provide store to its `store` prop. The Store is an object that represents the application state. Redux (`react-redux`) comes with a function called `createStore()` which takes reducer(s) from `ch14/redux-netflix/src/modules/index.js` and returns the Store object.

To render `Provider` component and its whole subtree of components, use [react-dom's](#) `render()` function. It takes your first argument (`<Provider>`) and mounts it into the element you pass as the second argument (`document.getElementById('app')`).

Combining all together, the entry point of your application should now look like the code shown in Listing 14-3 in which we define a provider by passing store instance (with reducers) in a JSX format.

Listing 14.3 Main app entry point in which we define Provider and reducers (ch14/redux-netflix/index.js)

```
const React = require('react')
const { render } = require('react-dom')
const { Provider } = require('react-redux')
const { createStore } = require('react-redux')
const reducers = require('./modules')
const routes = require('./routes')

module.exports = render((
  <Provider store={createStore(reducers)}>
    {routes}
  </Provider>
), document.getElementById('app'))
```

For your whole application to able to use Redux features, there is some code we still need to implement in child components such as connecting the store. The `connect()` function from the

same package `react-redux` accepts a few arguments. It returns a function which then wraps your component so that it can receive certain parts of the Store into its props. You'll see it in a bit.

As far as `index.js` goes we are done with it. Provider component takes care of delivering data from the store to all the connected components, so there's no need to pass props directly. But a few parts are missing here such as routes, reducers and actions. Let's discover them one by one.

14.3.4 Routes

With [react-router](#), you can declare certain hierarchy of components per browser location. I covered React Router in the previous chapter so it should be familiar to you. We used it for client-side routing. React Routing is NOT strictly connected with server-side routes, however sometimes you may want to do so (especially in conjunction with techniques from chapter 16 on Universal JavaScript).

The gist of React Router is that every route can be declared by a couple of nested `Route` components, each taking two props:

- `path` means location, either relative to parent `Route` component's path or absolute (when starts with "/"), and
- `component` takes the reference to component which will be rendered when a user goes to this particular location, with all the parent components up to `Provider`.

What we need is to show a collection of movie covers on both root and `/movies` location. In addition, we need to show details on a certain movie on `/movies/:id` location.

Route configuration for this case would use `IndexRoute` and look like:

Listing 14.4 Defining URL routing with React Router (`ch14/redux-netflix/src/routes.js`)

```
const React = require('react')
const {
  Router,
  Route,
  IndexRoute,
  browserHistory
} = require('react-router')
const App = require('components/App/App')
const Movies = require('components/Movies/Movies')
const Movie = require('components/Movie/Movie')

module.exports = (
  <Router history={browserHistory}> ①
    <Route path="/" component={App}>
      <IndexRoute component={Movies} /> ②
      <Route path="movies" component={Movies}>
        <Route path=":id" component={Movie} /> ③
      </Route>
    </Route>
  </Router>
)
```

```

        </Route>
      <Router>
    )
  
```

- ① Provide either browser or hash history to the Router
- ② Define the index route, that's the route for the empty URL ` '/'`
- ③ Define the movie ID URL parameter with a colon, i.e., `':id'`

You can see that both `IndexRoute` and `Route` are nested into the topmost route. This makes the `Movies` component render for both root and `/movies/locations`. The individual movie view needs a movie ID to fetch info about that particular movie from Redux store so we define the path with a URL parameter. To do so, we use the colon syntax, i.e., `path=":id"`. In Figure 14-10 I show how the individual view and its URL will look on a small screens, thanks to responsive CSS. Notice that the URL is `movies/8` with 8 being a movie ID.

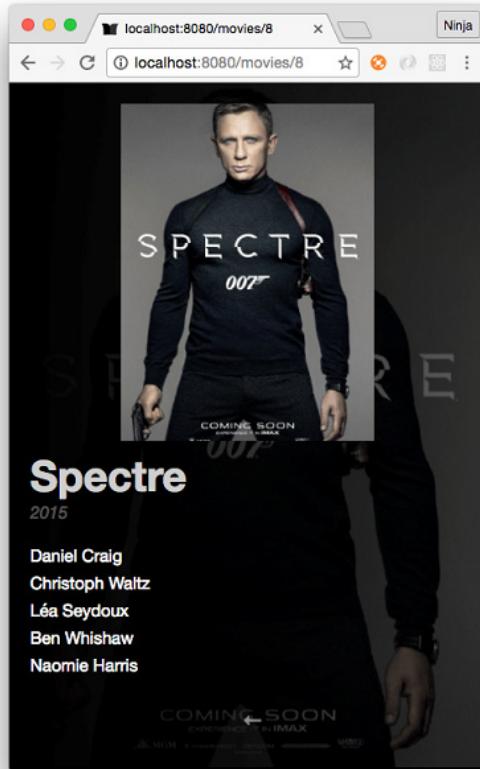


Figure 14.11 Individual movie view on a small screen with the URL which has the movie ID

In a moment, we'll cover fetching the data with Redux reducers.

14.3.5 Combining Reducers

Now, let's discover what are the modules that the `createStore()` function in `src/index.js` is applied to:

```
...
const reducers = require('./modules')           ①
...

module.exports = render(
  <Provider store={createStore(reducers)}>      ②
    {routes}
  </Provider>
), document.getElementById('app'))
```

① Import (combined) reducers from `./modules` which is actually `./modules/index.js`
 ② Apply reducers

Combine? What does it do? You see, you'll need to store movie data in the Store. Perhaps in future other parts of the Store will appear, like user account or other entities. So let's use Redux's own feature that allows you to create as many distinct parts of the Store as you need, although you only need one at the moment. In a way, we are creating a better architecture with this middle step of combining reducers so later our app can be extended effortlessly by the addition of more reducers into `./modules/index.js` (or `./modules` using the plugin Node [pattern](#)). This approach is also called [Splitting Reducers](#).

Each reducer can change data in the Store, but to make this operation safe, you may need to divide application state into separate parts, and then combine them into a single Store. This divide&conquer approach is recommended for larger app in which you'll start having more and more reducers and actions. We can easily combine multiple reducers with the `combineReducers()` function from `redux`:

Listing 14.5 ch14/redux-netflix/src/modules/index.js

```
const { combineReducers } = require('redux') ①
const {
  reducer: movies                      ②
} = require('./movies')

module.exports = combineReducers({          ③
  movies
  // more reducers go here
})
```

① Import `combineReducers` from the property `combineReducers` in `redux`
 ② Apply ES6/ES2015 destructuring assignment to create a reducer object `movies` from the `reducer` property of `./movies.js`
 ③ Export the combined reducer `movies`

You can pass as many reducers as you like and create independent branches in the Store. You can name them as you like. In this particular case, the `movies` reducer is imported and then passed into `combineReducers()` function as a property of a plain-object with key "`movies`".

This way, you declare a separate part of the Store and call it "movies". Every action that reducer from `./movies` is responsible for will now only touch this part and nothing else.

14.3.6 Reducer for Movies

Next, let's implement the "movies" reducer.

A reducer, in terms of Redux, is a function that runs *every time* any action is dispatched. It is executed with a few arguments:

- The first argument `state` is reference to the part of the state this reducer manages, and
- The second argument `action` is an object that represents the action that has just been dispatched.

Reducers in JavaScript

The term **reducers** actually comes from functional programming and most of us know that JavaScript has somewhat functional nature.

In a nutshell, a `reduce` method is an operation which summarizes a list of items so that on the input we have multiple values and on the output a single value. The list could be an array as is the case with JavaScript or another data structure outside of JavaScript.

An example would be to return a number of occurrences of a name in a list of names. List of names is input and number of occurrences is output.

The way a reducer works is that we call a method and pass a reducing function which accepts an accumulation value (what is passed to the next iteration and what will eventually become the output), and a current value (items of the list). With each iteration over items of the list (or array in JS), reducer function will be getting the accumulation value. In JavaScript, the method is `Array.reduce()`. For example, to get a name frequency we can run this code:

```
const users = ['azat', 'peter', 'wiliam','azat','azat']
console.log(users)
    .reduce(
      (acc, curr)=>(
        (curr=='azat')?++acc:acc
      ), 0
    )
```

In Redux Reducers, the accumulated value is the state object. The accumulated items are the actions. So on the input we have actions (previous actions) and a current actions, then on the

output we get state. If your reducers are pure functions without side-effects (which they should be) then you get all the goodie of Redux+React such as hot reloading and time travel.

Also, avoid putting API calls into reducers. Why? Because I said so. Just kidding. Remember that reducers are suppose to be pure functions? No side-effects. Just a state machines. They shouldn't do any asynchronous operations such as HTTP calls to an API. The best place to put these type of async call is in [middleware](#) or `dispatch()` [action creator](#) (action creators is a function that creates actions). You will see `dispatch()` later in this chapter in a component.

Back to reducers. A typical reducer is a function with a huge `switch/case` statement inside:

```
const FETCH_MOVIES = 'movies/FETCH_MOVIES'
const FETCH_MOVIE = 'movies/FETCH_MOVIE'

const initialState = {
  movies: [],
  movie: {}
}

function reducer(state = initialState, action) {
  switch(action.type) {
    case FETCH_MOVIES:
      return {
        ...state,
        all: action.movies ①
      }
    case FETCH_MOVIE:
      return {
        ...state,
        current: action.movie ②
      }
  }
}

module.exports = {
  reducer ③
}
```

- ① Save or change the list of all movies in the Store
- ② Save or change a certain movie in the Store
- ③ Export an object with the `reducer` property with of the `reducer` function value

But `switch/case` is considered [a bad practice](#) by then luminary Douglas Crockford in his classic JavaScript The Good Parts (O'Reilly Media, 2008)/

There is a handy library [redux-actions](#) which can bring this `reducer` function into a cleaner and more functional form. Instead of a huge `switch/case` statement as you observed prior, we can have a more robust object.

Let's use `handleActions` from [redux-actions](#). This `handleActions` takes a Map-like plain-object where keys are action types and values are functions. This way only a single function is called per certain action type and this function is cherry-picked by action type.

So the same function re-written with `redux-actions` and `handleActions` will look like this:

Listing 14.6 (ch14/redux-netflix/src/modules/movies.js)

```
const { handleActions } = require('redux-actions')

const FETCH_MOVIES = 'movies/FETCH_MOVIES'
const FETCH_MOVIE = 'movies/FETCH_MOVIE'

const initialState = {
  movies: [],
  movie: {}
}

module.exports = {
  reducer: handleActions({
    [FETCH_MOVIES]: (state, action) => ({
      ...state,
      all: action.movies
    }),
    [FETCH_MOVIE]: (state, action) => ({
      ...state,
      current: action.movie
    })
  }, initialState)
}
```

This new code looks very similar to `switch/case`, but it's more about mapping functions to actions rather than selecting them inside a potentially huge conditional statement.

14.3.7 Actions

To change data in Store, you use actions. Just to clarify: actions could be anything, not just a user input in a browser. It could be a result of an async operation. So basically any code can become an action. Actions are the only way to sources of information for the store. This info is sent from an app to the store. Actions are sent via `store.dispatch()` briefly mentioned in a prior section or via a helper `connect()`. But before calling an action, let's cover its type.

Every action is represented by a plain-object that has at least one property `type`, and it can have as many other properties as you want, usually to pass data into the Store. So, every action has type. Just like this:

```
{
  type: 'movies/I_AM_A_VALID_ACTION'
}
```

See that action type is a string. It's a commonly used pattern when you name your actions in uppercase and also precede them by module name in lowercase. Note however that the module name can be omitted if you are perfectly sure duplicates will never occur.

In modern Redux development, action types are declared as constant strings, for example:

```
const FETCH_MOVIES = 'movies/FETCH_MOVIES'
```

```
const FETCH_MOVIE = 'movies/FETCH_MOVIE'
```

You can see that there are two action types declared, both are strings that consist of two parts: the name of Redux module and the name of action type. This practice may be useful when you have different reducers with similarly named actions.

Every time you want to change the application state, you need to dispatch a corresponding action. An appropriate reducer function is executed, and you end up with the updated application state. Think about data you receive from API or any form a user fills in: it all can be placed, and then updated, in the Store.

```
Dispatch an action -> Reducer is executed -> New state
```

14.3.8 Action Creators

To change anything in the Store, you need to run an action through all the reducers. A reducer then changes the application state based on action type. So you need to always know the action type. There's, though, a shortcut that allows you to conceal action types under *action creators*:

1. Action creator -> Action
2. Dispatch an action -> Reducer is executed -> New state

An action creator is a function that returns an action, as straightforward as this:

```
function fetchedMovies(movies) {
  return {
    type: FETCH_MOVIES,
    movies
  }
}
```

With action creators, you can hide complex logic into a single function call. In this case, though, there's no logic at all and the only operation this function performs is returning an action, a plain-object with property `type` that defined this action and also property `movies` that has the value of array of movies. If you were to extend this Netflix clone so it can add a movie, then you'll need an action creator `addMovie()` like this:

```
function addMovie(movie) {
  return {
    type: ADD_MOVIE,
    movie
  }
}
```

Or how about `watchMovie()`?

```
function watchMovie(movie, watchMovieIndex, rating) {
  return {
    type: WATCH_MOVIE,
    movie,
    index: watchMovieIndex,
```

```

        rating: rating,
        receivedAt: Date.now()
    }
}

```

Overall,

1. Data->Action Creator->Action
2. Dispatch an action (from 1) -> Reducer is executed -> New state

Remember, an action *must* have the `type` property!

To be able to dispatch actions, we must connect components to the store. This is getting more interesting now because we are close to the state updates.

14.3.9 Connecting Component to the Store

After you learned how to put data into Store, let's take it and bring to your React components. The `Provider` component takes care of bringing tools to do that right into your components. But to use it, you need to *connect* your component to the Store. In other words, we must connect to the store to be able to dispatch:

1. Data->Action Creator->Action
2. Connect components to store to use 3
3. Dispatch an action (from 1) -> Reducer is executed -> New state

A connected component can take any data from the Store and right into its props. To do this, you need to *map state to props*. In some tutorials, you might find a function called the same way, `mapStateToProps()`, although it doesn't have to be an explicitly declared function. You'll see that using simply an anonymous arrow function is as clean and straightforward.

You need to use `connect()` function from `react-redux` to connect a component to the Store. By default, a component is not connected, and simply having it somewhere within the hierarchy of topmost `Provider` component is not enough. Why? Well, think of adding `connect()` as an explicit opt-in for only certain components. There are two types of components according to the best React practices: presentational (dumb) and container (smart) as discussed in Part 1, chapter 8. Presentational components should NOT need the store. They should just consume props. At the same time, container component needs the store and the dispatcher. Even the [definition](#) of container components according to Redux official documentation is that it subscribes to store.

On the contrary, all `Provider` is doing is providing a store to all components automatically so some of them can subscribe/connect to it. So for container components you need both, `Provider` and the store.

The `connect` function comes with `react-redux` package and accepts up to four arguments, however you'll use just one at the moment. Think about your root component, `App`. It will be

using `Movies` which minimally should have this code to display the list of movies (the actual `Movies` has a bit more code):

```
class Movies extends React.Component {
  render() {
    const {
      movies = []
    } = this.props

    return (
      <div className={styles.movies}>
        {movies.map((movie, index) => (
          <div key={index}>
            {movie.title}
          </div>
        )))
      </div>
    )
  }
}
```

Currently, it's not connected. Let's do it by adding

```
const { connect } = require('react-redux')
class Movies extends React.Component {
  ...
}
module.exports = connect()(Movies)
```

Here, `connect()` function returns a function which is then applied to `Movies` component. As a result, the component becomes connected to the Store.

Now, the `Movies` component can receive any data from the Store and dispatch actions. But to receive this data, you need to map state to component props. Remember, it's the first argument of `connect()` function. Here it goes:

```
module.exports = connect(state => state)(Movies)
```

With this setup, you take *the whole* application state from the Store and put it into props of `Movies` component. You'll find that, usually, you only need a limited subset of the state. You will also see that if the Store is empty, component won't receive any extra props because, well, there's none.

The magic of Redux that happens next is that every time a part of the Store is updated, all the component that depend on that part will receive new props, and therefore be re-rendered. You know that it happens when you dispatch an action, which means your components are now loosely interdependent and only update when Store is updated, and any of them can cause this update by dispatching a proper action. There's no need to use classic callback functions passed as props and stream them from the topmost component down to the most deeply nested. Just connect components to the store, and it will work.

14.3.10 Dispatching an Action

To apply changes to data in the Store, you need to dispatch an action. Once you have connected the component to the Store, not only you can receive props mapped to certain properties of the application state, but also you receive the `dispatch` prop.

The `dispatch` prop is a function that takes any action and dispatched is into the Store. So, you can do it simply passing an action to it:

```
componentWillMount() {
  this.props.dispatch({
    type: 'FETCH_MOVIE',
    movie: {}
  })
}
```

The `type`, as you know, is a string, and so Redux lib applies all the reducers with this object. Only a part of one or more reducers will be executed that is responsible for that action.

After the action has been dispatched, which means you changed the Store in most cases, all the components that are connected to the Store and have props mapped from updated part of the application state will be re-rendered. There's no need to check if the component should update or do anything. Simply rely on new props in component's `render()` function and that's it.

You know about action creators, so you may replace a bare action (object with `type`) with one of them (function `fetchMovie()`):

```
componentWillMount() {
  this.props.dispatch(fetchMovie())
}
```

Since `fetchMovie()` returns a plain-object that is identical to previous one, it's okay to simply call this function and pass the result to `dispatch()`.

1. Data->Action Creator->Action
2. Connect components to store to use 3
3. Dispatch an action (from 1) -> Reducer is executed -> New state

14.3.11 Passing Action Creators into Component Props

There's another way to use action creators: you can put them right into component props. To do that, you can use the second argument of `connect()` function, and simply pass a plain-object where keys are names of the props you want to inject and values are functions that return actions:

```
const {
  fetchMovies
} = require('modules/movies')

class Movies extends Component {
  render() {
```

```

const {
  movies = []           ①
} = this.props

return (
  <div className={styles.movies}>
    {movies.map((movie, index) => ( ②
      <div key={index}>
        {movie.title}
      </div>
    )))
  </div>
)
}

module.exports = connect(state => ({
  movies: state.movies.all   ②
}), {
  fetchMovies
})(Movies)

① Grab this.props.movies or assign to an empty array (ES6 destructuring)
② Map the data to populate movies prop

```

Now, you can refer to `fetchMovie()` via props without `dispatch()` just like this:

```

componentWillMount() {
  this.props.fetchMovie()
}

```

So the code above is instead of using `dispatch()` to pass an action. This new action creator will be automatically wrapped into a valid `dispatch()` call. You don't even need to worry about doing it yourself. Awesome!

You can also see that the first argument to `connect()` in `ch14/redux-netflix/src/components/Movies/Movies.js`, which is a function that maps state to component props, is taking the whole state (`state`) as the only argument and returns a plain-object with a single property `movies`:

```

...
module.exports = connect(state => ({
  movies: state.movies.all
}), {
  fetchMovies
})(Movies)

```

You can make the code more eloquent with destructuring of `state.movies`:

```

module.exports = connect(({movies}) => ({
  movies: movies.all
}), {
  fetchMovies
})(Movies)

```

In `render()` function of the component `Movies`, the value of `movies` is picked from props and is rendered into a collection of sibling DOM elements, each is a `div` element with inner text set to `movie.title`. This is a typical approach to rendering an array into a fragment of sibling DOM elements. Wonder what the final component `Movies` will look like? Here's the code.

Listing 14.7 Passing action creators into Movies component props (ch14/redux-netflix/src/components/Movies/Movies.js)

```
const React = require('react')
const { connect } = require('react-redux')
const { Link } = require('react-router')
const movies = require('../movies.json')      ①
const {
  fetchMovies
} = require('modules/movies')
const styles = require('./Movies.css')

class Movies extends React.Component {
  componentWillMount() {
    this.props.fetchMovies(movies)      ②
  }

  render() {
    const {
      children,
      movies = [],
      params = {}
    } = this.props

    return (
      <div className={styles.movies}>
        <div className={params.id ? styles.listHidden : styles.list}>
          {movies.map((movie, index) => (
            <Link
              key={index}
              to={`/movies/${index + 1}`}>
              <div
                className={styles.movie}
                style={{backgroundImage: `url(${movie.cover})`}} />
              </Link>
            )))
          </div>
          {children}      ③
        </div>
      )
    }
  }

  module.exports = connect(({movies}) => ({      ④
    movies: movies.all
}), {
  fetchMovies
})(Movies)
```

- ① Load mock database from a JSON file (thanks to ``json-loader``) into ``movies``

- 2 Fetch the list of movies using the action creator for `FETCH_MOVIES` and the JSON object `movies` - this could be substituted for an AJAX/XHR call to an API server
- 3 Pass children as defined in React Router hierarchy
- 4 Connect component to a store, provide state to props mapping and `fetchMovies` action creator

As depicted in the annotation for the `Movies` component, swapping for an `async` data is straightforward: make a `fetch` and then call `dispatch` in `componentWillMount()`. Or even better, let's use `componentDidMount()` which recommended by React team for AJAX/XHR calls:

```
componentWillMount() {
  // this.props.fetchMovies(movies)
}
componentDidMount() {
  fetch('/src/movies.json', {method: 'GET'})
    .then((response)=>{return response.json()})
    .then((movies)=>{
      this.props.fetchMovies(movies)          ①
    })
}
```

- 1 Fetch the JSON file which will be served by Webpack Dev Server
- 2 Dispatch the action with the data which came asynchronously from the server via a GET request

And as far as `POST`, `PUT` or other HTTP calls, you can do the same thing we did with `GET`.

As a bonus, take a look at the implementation of `Movie` which uses a different state to props mapping by taking a movie ID from a React Router's URL parameter and using it as an index:

```
class Movie extends React.Component {
  ...
  componentWillMount() {
    this.props.fetchMovie(this.props.params.id)
  }
  ...
}

module.exports = connect(({movies}) => ({
  movie: movies.current
}), {
  fetchMovie
})(Movie)
```

14.3.12 Running the Netflix Clone

It has come time to run the project. Of course, you could have done it in the beginning because the start script is in `package.json`:

```
"start": "concurrently \"webpack --watch --config webpack.config.js\" \"webpack-dev-server\\"",
```

Simply run from the project folder (`ch14/redux-netflix`): `npm start`. Click around to see that the routing is working and the images are loading whether you used mock data or loaded it via the GET request.

It's time to wrap-up this lesson.

14.3.13 Wrap-up

As you can see, the idea of Redux is to provide a single place to store the whole application data, while the only way to change it is through the actions. This makes Redux universal and possible to be used anywhere, not only in React apps. However with `react-redux` library, you can use `connect()` function to connect any component to the Store and make it react to any change in there.

This is the basic idea of reactive programming: an entity A that observes changes in entity B will react to those changes as they occur, but not in the opposite direction. Here, A is any of your components, while B is the Store.

As you connect your component and map properties of Store into its props, you can refer to the latter in `render()` function. Usually, you need to first update the Store with data to refer to that data. This is why you call an action to do so in component's `componentWillMount()` function. By the moment component is mounted for the first time and `render()` is called, the particular part of the Store component refers to may be empty. However, once updated, it is preserved. This is why you can see that the list of movies is still present even after you travel across the app's locations.

The data doesn't disappear from the Store after a particular component is unmounted, as opposed to the situation when you use component's own state (`remember this.state() and this.setState()?`), which makes your Redux store able to serve different parts of your application that require same data without having this data reloaded again and again.

It is also **safe to update component props even in `render()` function via Store** just because this operation is deferred. In the opposite, `setState()` is prohibited from using at any point component may be being updated, i.e. `render()`, `componentWillMount()` or `componentWillUpdate()`. This feature of Redux adds up to its flexibility.

14.4 Server Data with GraphQL and Node

Thus far we've been importing a JSON file as our back-end data store, or making RESTful calls to fetch the same file emulating a GET endpoint. Ah, mocking APIs. It's fine for prototyping, and when you need you can to a persistent storage via a back-end server. Typically a REST API or if you have to SOAP.

Imagine this API has to be developed by another team. You agree on the JSON (or XML) data format over the course of a few meetings. They deliver. The hand-shake is working. It seems fine. Your front-end app is getting all the info. Then, as usually the case product owners talk to clients and decide they need a new field to show stars and ratings of the movies. What happens when you need to get an extra field? There is a new endpoint to be implemented `movies/:id/ratings` or the back-end team will bump up the version of the old endpoint and add an extra field. x

Maybe the app is still in the prototyping phase. In this case, it's likely the field could be added to the existing `movies/:id...`. It's easy to see that with time you'll have more such requests with changes to formats and structure (what if the ratings must be in `movies` as well? New nested fields from other collections such as recommended by friends?). In the age of rapid agile development and lean startup methodology, flexibility is an advantage. The faster these fields and data can be adapted to the end product which is the browser app in this case, the better. There's an elegant solution called GraphQL which solves many of these hurdles.

Let's continue developing our simple Netflix clone by adding a server to it. This server will provide GraphQL API, a modern way of exposing data to React apps. When you work with Redux, the server could be built using anything (Ruby, Python, Java, Go, Perl), not necessarily Node.js, but that's what I recommend and that's what we will be using in this section because it allows to use JavaScript across the entire development tech stack!

Also, the server could use REST or older SOAP standards, but with newer GraphQL pattern developers and their systems reverse the control by allowing their clients (front-end or mobile apps) to dictate what data they need instead of coding this logic into server endpoints/routes. The way GraphQL works is by interpreting queries, hence its name QL: query language. The query is written in JSON-like format, e.g.,

```
{
  user(id: 734903) {
    id,
    name,
    isViewerFriend,
    profilePicture(size: 50) {
      uri,
      width,
      height
    }
  }
}
```

While the response is good old JSON:

```
{
  "user": {
    "id": 734903,
    "name": "Michael Jackson",
    "isViewerFriend": true,
    "profilePicture": {
```

```
        "uri": "https://twitter.com/mjackson",
        "width": 50,
        "height": 50
    }
}
```

GraphQL is often used with Relay, but as you'll see in this example you can use GraphQL with Redux or any other browser data library. In a nutshell, GraphQL uses query strings (as shown above) which are interpreted by a server (typically Node) which in turn returns data in a specified by those queries formats. In other words, the control is inverted from having rules and power in the RESTful API servers (or SOAP) to having them on the client (browser). Some of the advantages of this inverted approach and GraphQL:

- Client-specific queries: Clients get *exactly* what they need
 - Structure, Arbitrary code: Uniform API with server-side flexibility
 - Strong typing: More robust validation and certainty in responses with the ease of data consumption by strongly typed languages such as Swift, Java or Objective-C
 - Hierarchical queries: Queries follow the data they return which is important because data is used by hierarchical views
 - Faster prototyping: There's not need for extensive back-end development or large separate back-end and API teams since there's just a single endpoint for the query
 - Fewer API calls: Front-end app ends up making fewer server requests because the data structure is dictated by the front-end app and can contain what was previously obtainable only via several REST endpoints.

TIP: RELAY Another the way to consume a GraphQL API in your React application is called [Relay](#)(graphql-relay-js and react-relay on npm). Some developers prefer to use Relay instead of Redux when working with GraphQL back-end. If you take a look at examples provided in the documentation you may even see very close similarity to how Redux makes your components connected, while instead of Store, you have a remote GraphOL API.

If you or your team plan on using GraphQL seriously, then I highly recommend looking into Relay as well.

For more info on advantages of GraphQL such as strong typing, take a look at this article [GraphQL Introduction](#). We will move on with our implementation.

To deliver data to your React app, you can use a simple server made with [Express](#) and [GraphQL](#). Simply put, Express is great at organizing and exposing API endpoints while GraphQL takes care of making your data accessible in a browser-friendly way, as JSON.

The project structure will look like following (we are re-using a lot of components code from the `redux-netflix`):

```
/redux-graphql-netflix
  /build
    /public
      - index.js
      - style.css
    - server.js
  /client
    /components
      /App
        - App.css
        - App.js
      /Movie
        - Movie.css
        - Movie.js
      /Movies
        - Movies.css
        - Movies.js
    /modules
      - index.js
      - movies.js
    - index.js
    - routes.js
  /node_modules
  /server
    - index.js
    - movies.json
    - schema.js
  - index.html
  - package.json
  - webpack.config.js
  - webpack.server.config.js
```

- 1 Compiled files
- 2 Compiled front-end files
- 3 Compiled back-end file
- 4 React source code files for front-end
- 5 Express source code file for back-end
- 6 GraphQL schema

The data will still be taken from JSON file but this time it's a server file so readers can easily replace the JSON file `movies.json` with database calls in `server/schema.js`. But before we go into schemas, let's install all the dependencies including Express.

Here's the `package.json`. Do you know what to do? Copy it as best as you can and run `npm i` of course!

Listing 14.8 ch14/redux--graphql-netflix/package.json)

```
{
  "name": "redux--graphql-netflix",
  "version": "1.0.0",
  "description": "A sample project in React, GraphQL, Express and Redux that copies Netflix's features and workflow",
```

```

"main": "index.js",
"scripts": { ①
  "start": "concurrently \"webpack --watch --config webpack.config.js\" \"webpack --watch -
    -config webpack.server.config.js\" \"webpack-dev-
    server\" \"nodemon ./build/server.js\""
},
"repository": {
  "type": "git",
  "url": "git+https://github.com/azat-co/react-quickly.git"
},
"author": "Azat Mardan (http://azat.co)",
"license": "MIT",
"bugs": {
  "url": "https://github.com/azat-co/react-quickly/issues"
},
"homepage": "https://github.com/azat-co/react-quickly#readme",
"devDependencies": {
  "babel-core": "^6.11.4",
  "babel-eslint": "^6.1.2",
  "babel-loader": "^6.2.4",
  "babel-polyfill": "^6.9.1",
  "babel-preset-es2015": "^6.9.0",
  "babel-preset-react": "^6.11.1",
  "babel-preset-stage-0": "^6.5.0",
  "concurrently": "^2.2.0",
  "css-loader": "^0.23.1",
  "eslint": "^3.1.1",
  "eslint-plugin-babel": "^3.3.0",
  "eslint-plugin-react": "^5.2.2",
  "extract-text-webpack-plugin": "^1.0.1",
  "json-loader": "^0.5.4",
  "nodemon": "^1.10.0", ①
  "style-loader": "^0.13.1",
  "webpack": "^1.13.1",
  "webpack-dev-server": "^1.14.1",
  "axios": "^0.13.1", ②
  "clean-tagged-string": "0.0.1-b6", ③
  "react": "^15.2.1",
  "react-dom": "^15.2.1",
  "react-redux": "^4.4.5",
  "react-router": "^2.6.0",
  "redux": "^3.5.2",
  "redux-actions": "^0.10.1"
},
"dependencies": {
  "express": "^4.14.0", ④
  "express-graphql": "^0.5.3", ⑤
  "graphql": "^0.6.2" ⑥
}
}

```

- ① Add the start script which will compile browser and server code and launch the server
- ② Add `nodemon` dev tool to start and re-start Express
- ③ Add Axios to make HTTP calls with Promises (similar to fetch) to use on the front-end
- ④ Add utility to remove spaces from ES6 string templates and do other cleaning
- ⑤ Add Express Node web server framework to use on a back-end
- ⑥ Add GraphQL plugin for Express to use both on the back-end and front-end

7 Add GraphQL to use both on the back-end and front-end

Next, we will implement the main server file `server/index.js`.

14.4.1 Installing GraphQL on a Server

The powerhouse of the web server implemented with Express and Node is its starting point (sometime referred to as an entry point): `index.js` in the `server` folder. It's in the `server` folder because this file will be used only on the back-end and must not be exposed to clients due to security concerns (it can have API keys and passwords).

The high-level structure of our `index.js` is this:

```
const path = require('path')
const express = require('express')
const graphqlHTTP = require('express-graphql')      ①
// ...
const app = express()

app.use('/q',           ②
  // ...
)

app.use('/dist',        ③
  // ...
)

app.use('*',            ④
  // ...
}

app.listen(PORT, () => console.log(`Running server on port ${PORT}`))      ⑤
```

- ① Import dependencies including GraphQL for Express
- ② Define a single GraphQL route which will serve all kinds of data
- ③ Define a route to serve the front-end app with its static assets from `/dist` URL
- ④ Serve the main HTML page for any requests which are not for `/dist/*` URLs
- ⑤ Boot up the server

Let's fill in those missing pieces marked by ellipses in comments (`// ...`).

First, keep in mind that you need to deliver the same file, `index.html`, for *every* route, except the API endpoint and bundle files. This is necessary because when you use HTML5 History API and go to a certain location (using hash-less URL like `/movies/8`), refreshing the page will make the browser query exactly that location.

You've probably already noticed that in the previous Netflix clone version when you refresh/reload the page on an individual movie (`/movies/8`) it won't show you anything. The reason is that we need to implement something additional for browser history to work. This

code needs to be on the server and it is responsible for sending out main HTML file `index.html` on all requests even when they are `/movies/8/`.

In Express, when you need to declare a single operation per every route, you can use `*` (asterisk):

```
app.use('*', (req, res) => {
  res.sendFile('index.html', {
    root: PWD
  })
})
```

Simply sending the HTML file per any location doesn't do the trick. You'll end up with 404 errors because, in this HTML, you have references to compiled CSS and JS files (`/dist/styles.css` and `/dist/index.js`). So you need to catch those locations:

```
app.use('/dist/:file', (req, res) => {
  res.sendFile(req.params.file, {
    root: path.resolve(PWD, 'build', 'public')
  })
})
```

Or I recommend using Express middleware called `express.static()` like this (for info on more middleware and tips on Express refer to Appendix C and my top-sellers *Pro Express.js* and *Express API Reference*):

```
app.use('/dist',
  express.static(path.resolve(PWD, 'build', 'public'))
)
```

WARNING : Static, Public and Dist

The importance of having this `public` folder *inside of build* CANNOT be overstated because if you do not restrict the act of serving resources (such as files) over a subfolder, all your code will be exposed to anyone who is visiting the server. Even the back-end code such as `server.js` can be exposed if forgo the usage of a subfolder as in:

```
// Anti-pattern. Don't do this or you'll be fired
app.use('/dist',
  express.static(path.resolve(PWD, 'build'))
)
```

Will expose for attackers `server.js` which might contain secrets, API keys, passwords and the details of implementation over the `/dist/server.js` URL.

By using a subfolder such as `public`, exposing it to the rest of the world, and putting ONLY the front-end files into this exposed subfolder, you can restrict unauthorized access to other files.

For the GraphQL API to work, you need to set up one more route `/q` in which we use `graphqlHTTP` library along with a schema (`server/schema.js`) and session (`req.session`) to respond with data:

```
app.use('/q', graphqlHTTP(req => ({
  schema,
  context: req.session
})))
```

And finally, to make the server work, you need to make it listen to incoming requests at a certain port:

```
app.listen(PORT, () => console.log(`Running server on port ${PORT}`))
```

Here, `PORT` is considered an *environment variable*. It's such a variable that you can pass into the process from the command line interface, just like

```
PORT=3000 node ./build/server.js
```

If you remember in my `package.json`, I use `nodemon` as in

```
nodemon ./build/server.js
```

`nodemon` is the same as running `node`, only `nodemon` will restart the code if you made changes to it.

In the previous example we used port 8080 because that's the default value for Webpack Development Server. There's nothing wrong with using 8080 for this example's Express server, but by some weird historical reason the convention emerged that Express apps run on ports 3000.

You could see another variable declared with uppercase, `PWD`. It is an environment variable, too; however it's set by Node itself to the project directory, i.e. to the path to a folder where `package.json` file is located, which is precisely the root folder of your project.

And finally, there were `graphqlHTTP` and `schema`. The first one is a variable that you receive from the `express-graphql` package, while `schema` is your data schema built using GraphQL definitions.

Altogether, the server setup will look like this:

Listing 14.9 Express server to provide data and static assets (ch14/redux-graphql-netflix/server/index.js)

```
const path = require('path')
const express = require('express')
const graphqlHTTP = require('express-graphql')
const schema = require('./schema')
const {
  PORT = 3000,
  PWD = __dirname
}
```

```

} = process.env
const app = express()

app.use('/q', graphqlHTTP(req => ({
  schema,
  context: req.session
})))

app.use('/dist', express.static(path.resolve(PWD, 'build', 'public')))

app.use('*', (req, res) => {
  res.sendFile('index.html', {
    root: PWD
  })
})

app.listen(PORT, () => console.log(`Running server on port ${PORT}`)) ②

```

- ① Save working directory of this file (PWD=print working directory)
- ② Boot up the server using 3000 as the port value (not 8080)

GraphQL is strongly-typed meaning it uses schemas as you saw in /q. The schema is define in `server/schema.js` if you remember from the project structure listing. Let's see what is our data looks like first because the structure of the data will determine the schema which we will be using.

14.4.2 Data Structure

The app is a UI that represents data about movies. Therefore, you need to have this data somewhere. The easiest option is to save it into JSON file (`server/movies.json`). The file contains all the movies, while each movie can be represented by a plain-object with a bunch of properties, so the whole file is an array of objects:

```
[
  {
    "title": "Pirates of the Caribbean: On Stranger Tides"
    ...
  },
  {
    "title": "Pirates of the Caribbean: At World's End"
    ...
  },
  {
    "title": "Avengers: Age of Ultron"
    ...
  },
  {
    "title": "John Carter"
    ...
  },
  {
    "title": "Tangled"
    ...
  },
  {
    "title": "Spider-Man 3"
    ...
  },
  {
    "title": "Harry Potter and the Half-Blood Prince"
    ...
  }
]
```

```

}, {
  "title": "Spectre"
  ...
}, {
  "title": "Avatar"
  ...
}, {
  "title": "The Dark Knight Rises"
  ...
}]

```

With each object containing movie information such as title, cover URL, year released, cost to produce in million dollars USD, and actors starring. For example, Pirates of the Caribbean has this data:

```
{
  "title": "Pirates of the Caribbean: On Stranger Tides",
  "cover": "https://upload.wikimedia.org/wikipedia/en/c/c6/On_Stranger_Tides_Poster.jpg",
  "year": "2011",
  "cost": 378.5,
  "starring": [
    {
      "name": "Johnny Depp"
    },
    {
      "name": "Penélope Cruz"
    },
    {
      "name": "Ian McShane"
    },
    {
      "name": "Kevin R. McNally"
    },
    {
      "name": "Geoffrey Rush"
    }
  ]
}
```

NOTE the data is top 10 of the most expensive movies, source: Wikipedia.

You can see that each movie now is an object that only has titles. Later on, you can add as many properties as you want, but right now let's focus on data schema.

14.4.3 GraphQL Schema

You can use any data source with GraphQL, be it an SQL database, an object storage, a bunch of files or a remote API. Two things that matter are:

- Purity of data: identical requests return identical responses, and
- It should be possible to represent data with JSON.

Now, as you have the list of movies stored in a JSON file, you can simply import it:

```
const movies = require('./movies.json')
```

A typical GraphQL schema defines query with fields and arguments. For the example data schema, when there's only a list of objects and each object only has a single property `title`, schema definition would look like:

```

const movies = require('./movies.json')      ①
new graphql.GraphQLSchema({
  query: new graphql.GraphQLObjectType({
    name: 'Query',
    fields: {
      movies: {
        type: new graphql.GraphQLList(new graphql.GraphQLObjectType({
          name: 'Movie',
          fields: {
            title: {                                     ②
              type: graphql.GraphQLString
            }
          }
        })),
        resolve: () => movies      ③
      }
    }
  })
})

```

- ① Import movies from a file (mock database)
- ② Define the field `title` in the schema as a string
- ③ Define the "getter" for this query which will send data from the JSON file (could be a database call)

Looks pretty chubby but as the variety of your data grows, you'll see it makes sense. The core idea is that when `query` is performed, the function that is assigned to `resolve` key is executed. After that, only those properties of objects that are requested are picked from the result of this function call. These properties will be in resulting objects, while the fields that are not listed won't appear.

It means, you need to tell what properties you want to receive every time you perform a query. It makes your API flexible and efficient: you can arrange parts of the data as you want, in the run-time.

But we have 2 types of queries and more fields, so let's see how we can implement them:

Listing 14.10 ch14/redux-graphql-netflix/server/schema.js

```

const {           ①
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLList,
  GraphQLString,
  GraphQLInt,
  GraphQLFloat
} = require('graphql')
const movies = require('./movies.json')

const movie = new GraphQLObjectType({           ②
  name: 'Movie',
  fields: {
    title: {                                     ①
      type: GraphQLString
    },
  }
})

```

```

cover: {
  type: GraphQLString
},
year: {
  type: GraphQLString
},
cost: {
  type: GraphQLFloat      ③
},
starring: {
  type: new GraphQLList(new GraphQLObjectType({
    name: 'starring',
    fields: {
      name: {
        type: GraphQLString
      }
    }
  }))
}
})

module.exports = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      movies: {
        type: new GraphQLList(movie),
        resolve: () => movies      ④
      },
      movie: {
        type: movie,
        args: {
          index: {
            type: GraphQLInt
          }
        },
        resolve: (r, {index}) => movies[index - 1]      ⑤
      }
    }
  })
})

```

- 1 Set the name of the object as movie so we can use it in two queries
- 2 Define all the fields with proper types
- 3 Use float for the cost
- 4 Send back the entire array of movies
- 5 Send back only a single movie using the index (comes from the URL parameter)

Phew. Now we can move to the front-end and see how to query this neat little thing.

14.4.4 Querying the API and Saving Response into Store

To get the list of movies, you need to query the server and after the response has been received, pass it to the Store. This operation is asynchronous and involves an HTTP request, so it's time to unveil axios.

NOTE : PROMISES AND CALLBACKS The `axios` lib implements Promise-based HTTP requests. It means it returns a Promise immediately after calling a function. Since an HTTP request doesn't guarantee to be performed immediately, you'll need to wait until this Promise is resolved.

To get data from a Promise once it's resolved, use its `then` property. It accepts a function as a callback which will be called with a single argument, and this argument will be the result of the initial operation, in this case, HTTP call.

```
getPromise(options)
  .then((data)=>{
    console.log(data)
  })
```

Promise and callback (in `then`) is an alternative to using callback in a sense that the code above can be re-written without promises:

```
getResource(options, (data)=>{
  console.log(data)
})
```

There's still quite a controversy associated with Promises. While some people prefer Promises and callbacks over plain callbacks due to the `catch` all syntax, for other Promises is not worth the hassle especially considering that they can carry an error and fall silently. Now Promises is part of ES6/ES2015 but new patterns such as generators and `async/await` emerge as the next evolution of writing async code.

The bottom line is that while you can do it all with just plain callbacks, most of the modern (especially front-end) code is or will be using Promises or `async/await`. For this reason, we use Promises with `fetch()` and `axios` in this book.

For more information on Promise API refer to [MDN](#), and [Top 10 ES6 Features Every Busy JavaScript Developer Must Know](#).

If you haven't worked with `axios`, it uses Promise-based HTTP requests not unlike `fetch` which we used before. To perform a GET HTTP call, use `getProperty` of `axios`:

```
axios.get('/q')
```

Since it returns a Promise, you can immediately access its `then` property:

```
axios.get('/q').then(response => response)
```

The function you pass as the argument to `then` returns into context of the Promise and not context of your component's method. You need to call an action creator to deliver new data into Store:

```
axios.get('/q').then(response => this.props.fetchedMovie(response))
```

Now let's build a proper query against your GraphQL API. To do that, you can use multi-line template string (notice backticks instead of single quotes):

```
axios.get(`/q?query={  
  movie(index:1) {  
    title,  
    cover  
  }  
}`).then(response => this.props.fetchedMovie(response))
```

When you use multi-line template literal, it preserves line breaks, and it isn't making any good to a query string in the API endpoint URL. To remove unnecessary spaces and line breaks, use [clean-tagged-string](#) lib which does only that: brings your huge multi-line template string into a less huge single-line one. The result of `

```
clean`/q?query={  
  movie(index:1) {  
    title,  
    cover  
  }  
}`
```

Would look like this:

```
'/q?query={ movie(index:1) { title, cover } }'
```

Notice the syntax: there are no round brackets after `clean` and it's attached to the template string itself instead. It's a valid syntax and it is called [tagged strings](#). Perhaps you'll meet them in the future, but right now you can just learn that this syntax is right, too.

Okay, so we can get the first movie with the index of 1:

```
const clean = require('clean-tagged-string').default  
  
axios.get(clean`/q?query={  
  movie(index:1) {  
    title,  
    cover  
  }  
}`).then(response => this.props.fetchedMovie(response))
```

Next, we need to implement code to get any movie by its ID as well as request more fields not just title and cover to display view shown in Figure 14.12.

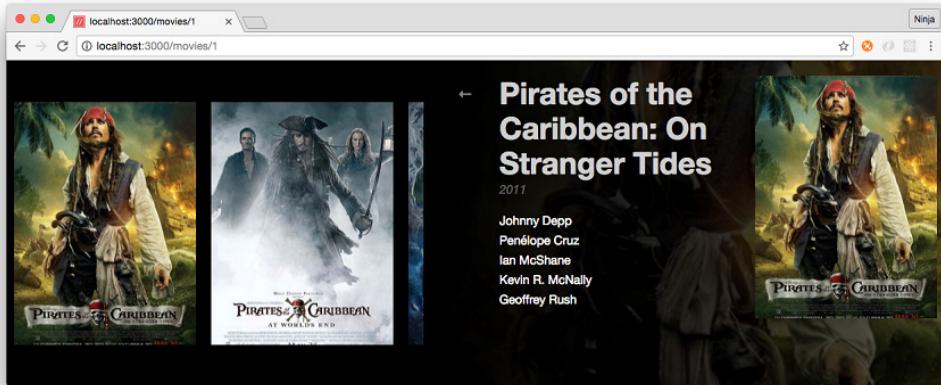


Figure 14.12 Single movie view server from Express server (port 3000) with browser history (no hash signs!)

It's good to know that now the single movie "page" won't be lost on reload because we added the special server code to `sendFile()` for `*` catch all route which send `index.html`.

Okay so first, fetching the data for a single movie from the API in the lifecycle component using our favorite Promise-based HTTP agent `axios`:

```
componentWillMount() {
  const query = clean`{
    movie(index:${id}) {
      title,
      cover,
      year,
      starring {
        name
      }
    }
  `

  axios.get(`/q?query=${query}`).then(response => this.props.fetchedMovie(response))
}
```

You can see that the amount of requested properties of movie entity is a little longer: not just `title` and `cover` but also `year` and `starring`. Since `starring` is itself an array of objects, you also need to declare what properties of those objects you want to request. In this case, it's only `name`.

Response from the API goes to `fetchedMovie` action creator, and after that, the Store is updated with an actual movie a user wants to be looking at.

Connect it:

```
module.exports = connect(({movies}) => ({
  movie: movies.current
}), {
  fetchedMovie
})(Movie)
```

And render:

```
render() {
  const {
    movie = {
      starring: []
    }
  } = this.props;
  return (
    <div>
      <img src={`url(${movie.cover})`} alt={movie.title} />
      <div>
        <div>{movie.title}</div>
        <div>{movie.year}</div>
        {movie.starring.map((actor = {}, index) => (
          <div key={index}>
            {actor.name}
          </div>
        )))
      </div>
      <Link to="/movies">
        ←
      </Link>
    </div>
  )
}
```

To organize the code better, let's add a method `fetchMovie()` in already familiar to your Movie component:

Listing 14.11 ch14/redux-graphql-netflix/client/components/Movie/Movie.js

```
// ...
fetchMovie(id = this.props.params.id) { ①
  const query = clean`{
    movie(index:${id}) { ②
      title,
      cover,
      year,
      starring {
        name
      }
    }
  `

  axios.get(`/q?query=${query}`).then(response => { ③
    this.props.fetchedMovie(response)
  })
}
// ...
```

- 1 Use React Router param from the URL to set id
- 2 Form the query using ID, template string and clean
- 3 Make the request to /q
- 4 Dispatch action with the data from the server

Now we can move to getting the list of movies.

14.4.5 Showing the List of Movies

Showing a list of movies is only a little bit different: the query to API is different, and it's rendered in a different way than when fetching a single movie.

So, you fetched the data from GraphQL server using a valid GraphQL query, via an asynchronous GET request performed with Axios lib, and put this data into Store via an action. The next thing is to show this data to the user. Time to *render* it.

You already know that, to take data from the Store, a component needs to be connected, e.g. wrapped with `connect()` function call that maps state to props and actions to props as well. In `render()` function of the component, you simply rely to its props, while making sure these props will have valid values at some point in time, usually after the component has mounted for the first time.

Let's declare a component that would pick data from the Store, take it from props and render it. First connect it:

```
const React = require('react')
const { connect } = require('react-redux')
const {
  fetchedMovies
} = require('modules/movies')

class Movies extends Component {
  // ...
}

module.exports = connect(({movies}) => ({
  movies: movies.all
}), {
  fetchedMovies
})(Movies)
```

You can see that `connect()` function takes two arguments, the first is mapping Store to component props, and the second is mapping action creators to component props. After that, the component will have two new props: `movies` and `fetchedMovies()`.

Next, let's fetch them and as the data has been received, place it into the Store via the action creator. Usually, data may be requested from a remote API at the very first moment the component starts its lifecycle, i.e. `componentWillMount()`:

```
class Movies extends React.Component {
  componentWillMount() {
```

```

const query = clean`{
  movies {
    title,
    cover
  }
}`

axios.get(`/q?query=${query}`).then(response => {
  this.props.fetchedMovies(response)
})
// ...
}
// ...

```

And finally, let's render this data:

```

// ...
render() {
  const {
    movies = []
  } = this.props

  return (
    <div>
      {movies.map((movie, index) => (
        <Link
          key={index}
          to={`/movies/${index + 1}`}
          <img src={`url(${movie.cover})`} alt={movie.title} />
        </Link>
      ))}
    </div>
  )
// ...

```

You can see that every movie has `cover` and `title` properties and link to a movie is basically a reference to its position inside the array of movies. It's not stable when you have thousands of elements in a collection because, well, the order is never guaranteed, but for now it's okay. A better way would be to use a unique ID which is typically auto-generated by a database (like MongoDB).

The component now renders the list of movies, although it lacks styles. Don't forget to check out sources for this chapter to see how it all works with styles and three-level hierarchy of components.

14.4.6 Wrap-up

Adding GraphQL support on a very basic level is straightforward and transparent. The way GraphQL works is different from how a typical RESTful API does, for you can query any properties, at any nesting level, of any subset of entities that API provides. This makes

GraphQL efficient on datasets of complex objects while REST design usually requires multiple requests to get the same data.

All in all, Redux is a nice library to use with React to implement data. It might be an overkill for a small app, so start gradually and add Redux when you feel like your actions and data flow are getting out of control.

At the same time, GraphQL is a promising pattern of implementing server-client hand-shake. It allows for more control from the client and structure of the data than REST API.

14.5 Quiz

1. Name two main arguments of a reducing function in the `reduce()` function in JavaScript?
2. What is the command to create a GraphQL schema? new `graphql.GraphQLSchema()`, `graphql.GraphQLSchema()` or `graphql.getGraphQLSchema()`?
3. Redux offers simplicity, larger ecosystem and better DX (Developer Experience) over Facebook Flux (flux). True or false?
4. How would you create a store and Provider? new `Provider(createStore(reducers))`, <`Provider`
`store={createStore(reducers)}`> or `provider(createStore(reducers))`?
5. Is it okay to put API calls into reducers? Yes or no?

14.6 Summary

In this chapter, we've covered

- Unidirectional data flow provides predictability and the ease of maintenance to React apps.
- Flux is the recommended architecture when working with React and unidirectional data flow
- Redux is one of the most popular implementations of Flux architecture
- With Redux, you can dispatch an action or put in into the props object
- Redux `connect()` enables container (smart) components and dispatcher of actions
- Redux Provider populates access to store to children so developers don't have to pass store in props manually
- Reducer is a file with a reducing function which uses (typically) a `switch/case` statement or `handleActions` to *apply* actions to a new state, i.e., current state and actions as input and new state as output
- Redux `combineReducers` conveniently merges multiple reducers to allow developers to split the code for those reducers into various modules/files
- To enable browser history and hash-less URL with React Router, use `.sendFile()` in * Express route to serve `index.html`

- GraphQL's URL structure is /q?query=... where query has the value of your data query

14.7 Quiz Answers

1. Accumulated value and current value are the main two arguments because without them we won't be able to summarize a list
2. new graphql.GraphQLSchema()
3. True
4. <Provider store={createStore(reducers)}>
5. No. Avoid putting API calls into reducers. It's better to put them into components (container/smart components to be specific)

15

Unit Testing React with Jest

In this chapter we'll cover testing react apps. The topics covered are

- Why use Jest (vs. Mocha or Others)
- Unit Testing with Jest
- UI Testing React with Jest and the TestUtils Add-on



Figure 15.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch15>

In modern software engineering, testing is very important. It's at least as important as using Agile methods, writing well-documented code, and having enough coffee supplies, sometimes even more so. Proper testing will save you many hours spent later on debugging. The code is not an asset anymore, it's a liability, so our goal is to make it as easy to maintain as possible. Using test-driven/behavior-driven development (TDD/BDD) can make the maintenance easier. It can make your company more competitive by allowing it to iterate faster and yourself more productive by giving you the confidence that your code works.

Code is a Liability?

The phrase "code is not an asset, it's a liability" gives 191M results which makes it harder to pinpoint the origins of this phrase. While I can't find an author, I can tell you the gist of the idea: when a real estate developer builds a house, they produce an asset because the house will be generating rent.

In contrast, software developers build apps, but the code is a tool to make apps assets. The code is not an asset by itself. It's more of a necessary evil to get to the end goal of having a working application. Thus, code is a liability because developers have to maintain it.

Some of the best ways to minimize the cost of maintaining the code is to make it simple, robust, and flexible for future changes and enhancements.

Using test-driven/behavior-driven development (TDD/BDD) can make the maintenance easier. It can make your company more competitive by allowing it to iterate faster and yourself more productive by giving you the confidence that your code works.

There are multiple types of testing. Most commonly, the type of testing you're performing can be separated into three categories: unit, service, and UI, as shown in Figure 15-1. Here's a high-level overview of each category, listed here from the lowest to highest level:

1. Unit testing. The system tests standalone methods and classes. There are no or very few dependencies or interconnected parts. The code for the tested subject should be enough to verify that the method works as it should work. For example, a module that generates random passwords is tested by invoking a method from a module and comparing the output against a regular expression pattern. The unit testing category also includes tests that might involve a few parts or modules working together to produce one piece of functionality. For example, a few components have to work together in order to provide the functionality of a password input with a strength check. They are tested by supplying the value to one component (input) and monitoring changes in the strength check (sufficient or not). This category should contain roughly 70% of your tests and should definitely outnumber any other types of tests.
2. Service (or integration testing): Tests typically involve other dependencies and requires a separate environment. Integration tests should be roughly 20% of all your tests (see Figure 15-1). Once you have a solid foundation of unit tests and the assurance of functional tests, you don't want to have too many integration tests because maintaining them will slow down development. Each time there is a UI change, your integration tests need to be updated. This often leads to flaky UI tests and having no integration testing at all, which is even worse.
3. UI (or acceptance testing): Tests often mimic Agile user stories and/or involves testing of the whole system, which obviously has all the dependencies and complexities imaginable.

The industry's best practice is described in the Testing Pyramid depicted in Figure 15-1. It shows that UI tests are more fragile and harder (more expensive) to maintain, thus they

should have be only ~10% of your overall tests. On the other hand, unit tests should be 70% because they are more durable.

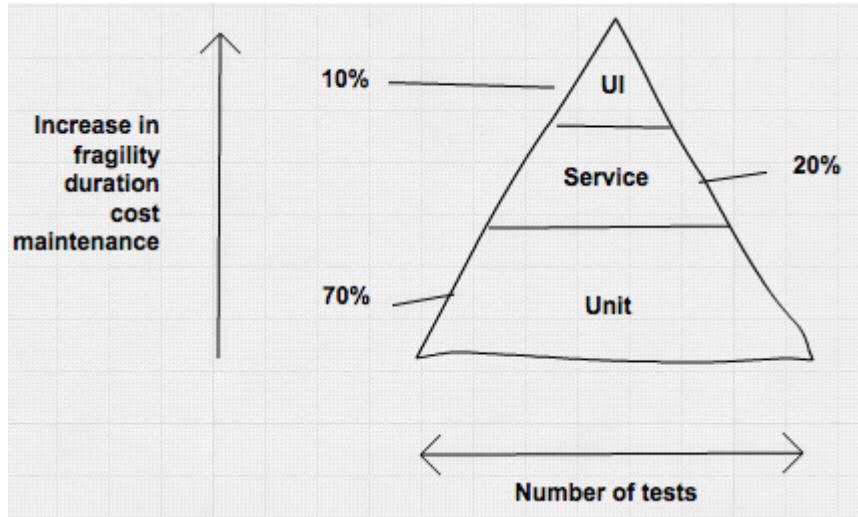


Figure 15.2 Testing pyramid according to the software engineering's best practice

In this chapter, we'll cover unit testing of React apps with a bit of UI testing of React components, thanks to the mock DOM rendering of React and Jest. We'll also be using our standard toolchain of Node, npm, Babel and Webpack.

To start unit testing, let's learn to use Jest.

The source code for the examples in this chapter is in [the ch15 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

15.1 Why Jest (vs Mocha or Others)

Jest is developed by Facebook and often used together with React by the community. Of course, there are other testing libraries like Mocha and Jasmine for example. Jest has these features:

- [Automocking](#) of JavaScript/Node modules by default makes it easier to isolate a particular code to unit test it
- Less setup to get started than with other test runners such as Mocha where you need to import Chai or standalone Expect, also Jest will find tests in the `__tests__` folder
- Sandboxed and parallel execution of tests to run them [faster](#)
- Static analysis with the support of Facebook's [Flow](#)—a static type checker for JS
- Modularity, configurability, and adaptability (via the support of Jasmine assertions)

Automocking, Static Analysis and Jasmine

If you are not familiar with a term mocking, it means faking a certain part of a dependency to be able to test current code. Automocking simply means that mocking will be done automatically for you. In Jest, every imported dependency is automocked which can be useful if you rely on mocking. The automocking can be turned off if needed.

Static analysis means that the code can be analyzed before running it which typically involves type checking. Flow is a library that adds type checking to otherwise type-less (more or less) JavaScript.

Jasmine is a feature-rich testing framework which comes with an assertion language. Jest extends and builds upon Jasmine under the hood so developers don't need to import or configure anything. Thus, developers have the best of both worlds: they can tap into the common interface of Jasmine without having extra dependencies or setup.

Now, there are many styles and opinions around what test framework is better for what job. Most projects use Mocha which has a lot of features. Jasmine came more from a front-end development, but is very interchangeable with Mocha and Jest because all of them use the same constructs to define test suites and tests: `describe`, `before`, `after`, `it`.

Without getting into a heated debate (on the pages of this book) what framework is the best, I encourage you to keep an open mind and explore Jest because of the aforementioned features and because it's coming from the same community that develops React. By doing so, you would be able to make a better judgement what framework to use for your next React project.

Most of the modern frameworks like Mocha, Jasmine and Jest will be similar for the most of the tasks with differences depending on your preferred style (maybe you prefer automocking... maybe not!) and on the edge cases of your particular project (do you need all the features Mocha provides or you need something more light weight like [TAP's node-tap](#)). Hence, Jest is a good start (if you're new), because by learning how to use Jest with React utilities and methods, you can leverage this knowledge to utilize other test runners and testing frameworks such as Mocha, Jasmine or node-tap.

15.2 Unit Testing with Jest

Jest by Facebook is a command-line tool based on Jasmine. It has a Jasmine-like interface. If you've worked with Mocha (another testing framework), you'll find Jest looks similar to it, and is easy to learn too! To illustrate my point, all three frameworks Mocha, Jasmine and Jest use the same constructs:

- `describe`: Test suite
- `it`: Test case
- `before`
- `beforeEach`
- `after`
- `afterEach`

If you never worked with any of the testing frameworks mentioned, then don't worry. Jest is straightforward to learn. The main statements are `describe` and `it`. The former is a test suite which act as a wrapper for tests. The latter is individual test which is called test case. Test cases are nested within the test suite.

Other constructs such as `before`, `after` and their each brethren will execute either before or after test suite or test case. Adding `Each` will execute the piece of code many time comparing to just one time without each.

Writing of tests consists of creating test suites, cases and assertions. Assertions are like true or false questions only in a nice readable format (called BDD).

Take a look at the example below sans assertions for now:

```
describe('Noun: method or a class/module name', ()=>{
  before((done)=>{
    // This code will be called just once before all it statements
    done()
  })
  beforeEach((done)=>{
    // This code will be called many times before all it statements
    done()
  })
  it('Verb describing the behavior', (done)=>{
    // Assertions
    done()
  })
  it('Verb describing the behavior', (done)=>{
    // Assertions
    done()
  })
  ...
  after((done)=>{
    // This code will be called just once after all it statements
    done()
  })
  afterEach((done)=>{
    // This code will be called many times after all it statements
    done()
  })
})
```

We have to have at least one `describe` and `it` but their number is not limited while everything else such as `before` or `after` are optional. Jest resides at <https://facebook.github.io/jest>, and the API documentation is at <https://facebook.github.io/jest/docs/api.html#content>.

We won't be testing any React components just yet. Because before we can work with React components, we need to learn a little bit more about Jest by just working on a Jest example which has not UI.

As an example of unit testing and Jest usage, we will unit test a module (and create it of course) which generates random passwords. Imagine you're working on a sign up page for your cool new chat app. You got to have the ability to generate passwords, right? This module will automatically generates random passwords. To keep things simple, the format will be 8 chars of alpha numeric characters.

The project (module) structure goes like this:

```
/generate-password
  /__test__
    - generate-password.test.js
  /node_modules
    - generate-password.js
    - package.json
```

We'll be using CommonJS/Node module syntax which is widely supported in Node (duh) and also browser development via Browserify and Webpack. Kindly examine the module in the `generate-password.js` file:

Listing 15-1: Module for generating passwords (`ch15/generate-password.js`)

```
module.exports = function() {
  return Math.random().toString(36).slice(-8)
}
```

Just as a refresher, in this file we export the function via the `module.exports` global. This is Node.js and CommonJS notation. You can use it on the browser with some extra tools like Webpack or [Browserify](#).

The function itself is using `Math.random()` to generate a number and converting it to a string. The string length is eight characters, which is specified by `slice(-8)`.

To test the module, we can run this eval Node command from the terminal. It will import our module, invoke its function and print the results:

```
node -e \"console.log(require('./generate-password.js')())\""
```

You can additionally improve this module by making it work with different number of characters, not just 8.

15.2.1 Writing Unit Test in Jest

To start with Jest, you need to create a new project folder and `npm init` it to create `package.json`. If you don't have npm, this is the best time to install it. You can follow my instructions in appendix B.

Once you've gotten npm and created the `package.json` file in a brand new folder, install Jest with:

```
$ npm install jest-cli@13.2.3 --save-dev
```

I'm using version 13.2.3; make sure your version is the same or compatible. The `--save-dev` will add the entry to the `package.json` file. Open it and manually change the `test` entry to `jest`:

Listing 15.2 Saving test CLI command in the project metadata file

(ch15/jest/package.json)

```
{
  "name": "jest",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "jest", ①
    "start": "node -e \"console.log(require('./generate-password.js'))()\""②
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "devDependencies": {
    "jest-cli": "13.2.3"③
  }
}
```

- ① Replace the default test script with `jest`
- ② Save the Node eval command to get a random password
- ③ Remove ^ to ensure the exact version

Now, create a folder named `__tests__`. The name is important because Jest will pick up the tests from that folder. Then, create the first Jest test in `__tests__/generate-password.js`.

The test file will use a method from Jest API called `jest.dontMock()`, because we don't want to automock the library which we are testing. Typically we only automock the dependencies to isolate the library we are currently unit testing. Jest automatically mocks every required file, so we need to use `jest.dontMock` or `jest.autoMockOff()` to avoid this for `generate-password.js`. This is one way to do it:

```
jest.dontMock('../generate-password.js')
```

The test file has just a single suite (only one `describe`), which expects the value to match the `/^[a-zA-Z0-9]{8}$/` regular expression pattern (only alphanumerics and exactly eight characters) to satisfy our condition of a strong password (we don't want our chat users to be hacked by brute force).

Listing 15.3 Test file (suite and case) for generate-password mini-module

(ch15/generate-password/__tests__/generate-password.test.js).

```
jest.dontMock('../generate-password.js')

describe('method generatePassword', ()=>{
  let password
  generatePassword = require('../generate-password')①
  it('returns a generated password of lower-
```

```

    case characters and numbers with the length of 8', (done)=>{
      password = generatePassword()
      expect(password).toMatch(/^[a-zA-Z0-9]{8}$/)
      done() ②
    })
})

```

- ① Use `require`, a special Node.js global that imports the module into your script.js file
- ② Invoke done if you defined an argument— needed for asynchronous tests and optional for synchronous (our is sync)

You can run the test with `$ npm test`. You will see something like this as the terminal output:

```

Using Jest CLI v13.2.3, jasmine2
PASS  __tests__/generate-password.test.js (0.031s)
1 test passed (1 total in 1 test suite, run time 1.339s) ①

```

- ① How many tests passed and how many in total you have

15.2.2 Jest Assertions

Jest is using [the behavior-driven development \(BDD\) syntax](#) powered by [the Expect syntax](#) (by default). Unlike other frameworks like Mocha where developers need to install additional module for the syntax support in Jest it's automatic.

[Expect](#) is a popular language which is a replacement for TDD assertions. Expect has many flavors. Jest is using somewhat simplified version (in my opinion): <https://facebook.github.io/jest/docs/api.html>.

A few words about TDD. TDD could mean test-driven development or TDD syntax with assertions. Test-driven development in brief: write the test; then run it (failing); then make it work (passing); then make it right (refactor). Developers most certainly can perform test-driven development with BDD. The main benefit of BDD style is that it is intended for communicating with **every member of a cross-functional team**, not just software engineers. To put it differently, TDD is more techie language. BDD format makes it easier to read tests, ideally the spec title should tell you what you are testing:

```

describe('method generatePassword', ()=>{ ①
  ...
  it('returns a generated password of lower-
    case characters and numbers with the length of 8', ()=>{ ②
    ...
    expect(password).toMatch(/^[a-zA-Z0-9]{8}$/) ③
  })
})

```

- ① Use a noun to describe test suite
- ② Use verbs to describe behavior for a test case
- ③ Use Expect statement to implement test case

Here's a list of [the main Expect methods](#) that Jest supports (there are [many more](#)). The usage is to pass the actual values—returned by the program—to `expect()`, and compare it using the following methods with expected values which are hard-coded in the tests:

- `.not`: Inverses the next comparison in the chain
- `expect(OBJECT).toBe(value)`: Expects to be equal with `==` (`value&type`)
- `expect(OBJECT).toEqual(value)`: Expects to be deep-equal
- `expect(OBJECT).toBeFalsy()`: Expects to be falsy (refer to the sidebar below)
- `expect(OBJECT).toBeTruthy()`: Expects to be truthy
- `expect(OBJECT).toBeNull()`: Expects to be null
- `expect(OBJECT).toBeUndefined()`: Expects to be undefined
- `expect(OBJECT).toBeDefined()`: Expects to be defined
- `expect(OBJECT).toMatch(regexp)`: Expects to match the regular expression

In JavaScript/Node, a `truthy` value is a value which translates to true when evaluated as a Boolean in an `if/else` statement. A `falsy`, on the other hand, will evaluate to `false` in a `if/else`. Official definition is the value is `truthy` if it's not `falsy`, and there are only six `falsy` values:

- `false`
- `0`
- `""`: Empty string
- `null`
- `undefined`
- `NaN`: Not a number

Everything else not listed above is `truthy`.

To summarize, Jest can be used for unit testing which should be the most numerous of your tests. They are more low-level, and for this reason it's more solid and less brittle which makes them less costly to maintain.

Thus far, you've created a module and tested its method with Jest. This is a typical unit test. There were no dependencies involved, only the tested module itself. This skill should be enough for us to continue with testing React components. Now, let's take a look at more complicated UI testing. The next section deals with the React testing utility which will enable us to perform UI testing.

15.3 UI Testing React with Jest and TestUtils

Generally speaking, in UI testing, which is recommended to be only 10% of your all tests, we'll test entire components, their behavior and even the entire DOM trees. We can test them manually... which is a terrible idea! Humans make mistakes and take long time to test. Manual UI testing should be at the minimum or even zero.

How about automated UI testing? Yay! Developers can test automatically using [headless browsers](#) which are like real browsers only without GUI. That's how most Angular 1 apps are being tested. It's totally possible to use this process with React but this process is not easy and often slow and requires a lot of processing power.

Another automated UI testing approach, which is the case with React, is to leverage React's virtual DOM accessible via a browser-like testing JavaScript environment implemented by jsdom.

To leverage React's virtual DOM, we'll need a utility which is closely related to the React core library but not part of it meaning we'll need to use a separate module `react-addons-test-utils`.

NOTE It's worth noting, there are other [React add-ons](#) as well. Most of them are still in experimental stage which in practice means React team might change their interface or stop supporting them. All of them follow the naming convention of `react-addons-NAME`.

TestUtils is an add-on, and like other React add-ons, it is installed via npm. You cannot use TestUtils without npm which is a Node package manager and which comes with Node.js). You can get npm by following the instructions in Appendix A. (Note that most of the add-ons are in the experimental stage, meaning that their exact APIs are more likely to change in the future than the core React library.)

The npm name of TestUtils is `react-addons-test-utils`. I'm using version 15.2.1, and you can install it with npm:

```
$ npm install react-addons-test-utils@15.2.1 --save-dev
```

[The TestUtils add-on](#) is a React utility to test its components. In a nutshell, TestUtils will allow us to create components and render them into the fake DOM. Then we can poke around, looking at the elements by tags or classes. Isn't it wonderful? It's all done from the command line, without the need for browsers (headless or not).

TestUtils has a few main methods. They are aimed at rendering a component, simulating events such as click, mouseOver, etc. and finding elements in a rendered component. We will start with rendering of a component and introduce other methods as we go (and as we need them in our project).

To illustrate the render method of TestUtils, this code will render an element into a `div` variable without using a headless (or real for that matter) browser:

Listing 15.4 Rendering React element in Jest and checking for props

(ch15/testutils/_tests_/render-props.js)

```
...
const HelloWorld = React.createClass({
  render() {
```

```

        return <div>{this.props.children}</div>
    }
})
let hello = TestUtils.renderIntoDocument(<HelloWorld>Hello Node!</HelloWorld>)
expect(hello.props).toBeDefined()
console.log('my hello props:', hello.props) // my div: Hello Node!
...

```

Note: `renderIntoDocument` will only work on custom components not on DOM components like `<p>`, `<div>`, `<section>`, etc. So if you see an error:

```
Error: Invariant Violation: findAllInRenderedTree(...): instance must be a composite component
```

Make sure, that you're rendering a custom (your own) component class and not a standard class. See [this commit](#) and [a thread](#) on GitHub for more details.

Once you got that `hello` (officially called React component tree), you can look inside of it with one of the `find` element methods. For example, you can get the `<div>` from inside the `<HelloWorld/>` element:

Listing 15.5 Rendering React element in Jest and finding its child element

```
(ch15/testutils/__tests__/scry-div.js)
...
const HelloWorld = React.createClass({
  render() {
    return <div>{this.props.children}</div>
  }
})
let hello = TestUtils.renderIntoDocument(<HelloWorld>Hello Node!</HelloWorld>)
expect( TestUtils.scryRenderedDOMComponentsWithTag(hello, 'div').length).toBe(1)
console.log('found this many divs: ', TestUtils.scryRenderedDOMComponentsWithTag(hello, 'div').length)
...
```

So `scryRenderedDOMComponentsWithTag()` allows us to get an array of elements by their tag names (e.g., `div`). Are there any other way to get elements?

15.3.1 Finding Elements with TestUtils

In addition to the `scryRenderedDOMComponentsWithTag()`, we have a few other ways to get either a list of elements (prefixed with `scry` and has plural `Components`) or a single element (prefixed with `find` and has singular `Component`). Both of them will use element class (not to confuse with component class which is a different thing). For example, `btn`, `main`, etc.

In addition to tag names, we can get elements by type (component class) or by their CSS classes. For example, `HelloWorld` is a type, while `div` is a tag name (we used it to pull the list of criteria).

We can mix and match `scry` and `find` with class, type or tag name to get six methods depending on the needs. This is the list of what each method returns:

- `scryRenderedDOMComponentsWithTag()`: Many elements and you know their tag name
- `findRenderedDOMComponentWithTag()`: Just a single element and you know its unique tag name, that is there are no other elements with a similar tag name in the component
- `scryRenderedDOMComponentsWithClass()`: Many elements and you know their class name
- `findRenderedDOMComponentWithClass()`: Just a single element and you know its unique class name
- `scryRenderedComponentsWithType()`: Many elements and you know their type
- `findRenderedComponentWithType()`: Just a single element and you know its type

As you can see, there's no shortage of methods when it comes to pulling the necessary element(s) from your components. If you need some guidance, I would say use classes or types (component classes), because they will allow you to target elements more robustly. For instance, if think you can use tag names now because there's just one `<div>`, if you decide to add more elements of the same tag names to your code (more `<div>`, then you'll need to rewrite your test. If you use HTML class to test an `<div>` then your test will work fine after adding more `<div>` to the tested component.

The only case when using tag names might be appropriate is when you need to test all of the elements of a specific tag name (`scryRenderedDOMComponentsWithTag()`), or your components is so small than there would be no additional elements of the same tag name(`findRenderedDOMComponentWithTag()`). For example, a stateless component which wraps an anchor tag `<a>` to add a few HTML classes to it—there would be 0 additional anchor tags for sure.

15.3.2 UI Testing Password Widget

Consider a UI widget which you can be used on a sign up page to provide an ability to automatically generate passwords of a certain strength. It can look as one shown in Figure 15-2 where we have an input field, Generate button and the list of criteria.



Figure 15.3 Password widget which allows to auto generate a password according to the given strength criteria

The full walk-through of the project will be in the following section. For now we are only focusing on using TestUtils and its interface.

Once TestUtils and other dependencies (such as Jest) are installed, you can create the Jest test file to UI test our widget; let's call it `password/_tests_/password.test.js`, since we are testing a password component. The structure of this test will go like this:

```
jest.autoMockOff()

describe('Password', function() {
  it('changes after clicking the Generate button', (done)=>{
    // Importations
    // Perform rendering
    // Perform assertions on content and behavior
    done()
  })
})
```

The reason why there's a `autoMockOff()` is that this is more of a UI test rather than a unit test. For this reason we need a few modules without mocking. Let's define these dependencies inside of the `describe`. Also I've created a shortcut for `fD` for `ReactDOM.findDOMNode()` just because we'll use it a lot.

```
const TestUtils = require('react-addons-test-utils')
const React = require('react')
const ReactDOM = require('react-dom')
const Password = require('../jsx/password.jsx')
const fD = ReactDOM.findDOMNode
```

To render a component, we need to use `renderIntoDocument()`. For example, this is how you can render a `Password` component and save a reference to the object in the `password` variable. The props that I'm passing will be the keys of the rules for the password strength. For example, `upperCase` is for requiring at least one uppercase character.

```
let password = TestUtils.renderIntoDocument(<Password
  upperCase={true}
  lowerCase={true}
  special={true}
  number={true}
  over6={true}
/>
)
```

The example above is in JSX because Jest will automatically use `babel-jest` when you have install this module (`npm i babel-jest --save-dev`) and set Babel configuration to use `"presets": ["react"]`. It's also possible to not use JSX in Jest if you don't want to include `babel-jest`. In this case, simply call `createElement()`:

```
let password = TestUtils.renderIntoDocument(
  React.createElement(Password, {
    upperCase: true,
    lowerCase: true,
    special: true,
    number: true,
    over6: true
  })
)
```

Once we've got the component rendered with `renderIntoDocument()`, it's straightforward to extract the needed elements—children of `Password`—and execute expect assertions to see how our widget is working. Think of the extraction calls as your `jQuery`; you can use tags or classes. At the bare minimum, our test should be checking for these things:

1. `Password` element has list items (``) which would be the strength criteria
2. The first of the strength list items has a certain text
3. The second criteria list item is not fulfilled (strikethrough)
4. There's a Generate button (class `generate-btn`)—click it!
5. After clicking on Generate, the second criteria list item become fulfilled (strikethrough)

Clicking on Generate will fulfil all criterias and make the password visible (so users can memorize it) but we won't test it in this book. That's your homework for next week. ;-)

We can start with the number one. There's `TestUtils.scryRenderedDOMComponentsWithTag()` which gets all elements from a

particular class. In our case, this class is `li` for the `` elements because that's what the criteria list will be using: ``. `toBe()` works like a triple equal `====`.

```
let rules = TestUtils.scryRenderedDOMComponentsWithTag(password, 'li')
expect(rules.length).toBe(5)
```

For the number two in our list of things to check which is "The first of the strength list items has a certain text", we leverage `toEqual()`. We expect the first item to say that an uppercase character is required. This will be one of the rules for the password strength:

```
expect(fD(rules[0]).textContent).toEqual('Must have at least one uppercase character')
```

NOTE: `toBe()` vs. `toEqual()`

`toBe()` and `toEqual()` are NOT the same in Jest. They behave differently. The easiest way to remember is `toBe()` is `====` strict equal while `toEqual()` check that two objects have the same value. Thus both assertions will be correct:

```
const copy1 = {
  name: 'React Quickly',
  chapters: 19,
}
const copy2 = {
  name: 'React Quickly',
  chapters: 19,
}

describe('Two copies of my books', () => {
  it('have all the same properties', () => {
    expect(copy1).toEqual(copy2) // correct
  })
  it('are not the same object', () => {
    expect(copy1).not.toBe(copy2) // correct
  })
})
```

However, when comparing literals such as number 5 and string 'Must have at least one uppercase character'. The `toBe()` and `toEqual()` will produce the same results:

```
expect(rules.length).toBe(5) // correct
expect(rules.length).toEqual(5) // correct
expect(fD(rules[0]).textContent).toEqual('Must have at least one uppercase character') // correct
expect(fD(rules[0]).textContent).toBe('Must have at least one uppercase character') // correct
```

For numbers three to five, we find a button, click it and compare the values of the second criteria (it must change from being a text to strike).

There's a `TestUtils.findRenderedDOMComponentWithClass()` method which is similar to `TestUtils.scryRenderedDOMComponentsWithTag()`, but returns only one element. It'll throw an error if you have more than one element.

For simulating user actions, there's `TestUtils.Simulate` object which has methods with the names of events in camelCase. For example, `Simulate.click` or `Simulate.keyDown` and even `Simulate.change`.

Let's use `findRenderedDOMComponentWithClass()` to get button and then `Simulate.click` to press it. All done in the code without actual browsers!

```
let generateButton = TestUtils.findRenderedDOMComponentWithClass(password, 'generate-btn')
expect(fD(rules[1]).firstChild.nodeName.toLowerCase()).toBe('#text')
TestUtils.Simulate.click(fD(generateButton))
expect(fD(rules[1]).firstChild.nodeName.toLowerCase()).toBe('strike')
```

In this test, I'm checking that the `` component will have a `<strike>` element (to make text strikethrough) when the button is clicked. The button generates a random password that satisfies the second (`rules[1]`) criterion (as well as others), which is to have at least one lowercase character. Done here, moving to the next tests.

You saw `TestUtils.Simulate` in action but it can trigger not just clicks, but other interactions as well, like change of value (input fields) or even an Enter keystroke (`keyCode` is 13):

```
ReactTestUtils.Simulate.change(node)
ReactTestUtils.Simulate.keyDown(node, {
  key: "Enter",
  keyCode: 13,
  which: 13})
```

Please note, that developers have to manually pass data which will be used in the component such as `key` or `keyCode` because `TestUtils` won't autogenerate it. There are methods in `TestUtils` for every user action supported by React.

Yet, there's another way to render React elements.

15.3.3 Shallow Rendering

In some cases, developers might want to just test a single level of rendering, that is the result of `render()` in a component *without rendering its children* (if any). This will simplify the testing because it doesn't require having a DOM—the system simply create an element and we can assert the facts about it.

To illustrate the point, take a look at the same `Password` element being tested with the shallow rendering approach. In this case, we create a renderer, then pass a component to it to get its shallow rendering:

```
const passwordRenderer = TestUtils.createRenderer()
passwordRenderer.render(<Password/>)
```

```
let p = passwordRenderer.getRenderOutput()      ①
expect(p.type).toBe('div')                    ②
expect(p.props.children.length).toBe(6)
```

- ① Perform shallow rendering
- ② Perform assert on the results of shallow rendering

Now, if we log the `p` as in `console.log(p)`, the result contain the children but the object `p` won't be a React instance:

```
{ '$$typeof': Symbol(react.element),
  type: 'div',
  key: null,
  ref: null,
  props:
    { className: 'well form-group col-md-6',
      children: [ [Object], [Object], [Object], [Object], [Object], [Object] ] },
    _owner: null,
    _store: {} }
```

Contrast it with the logs of the results of `renderIntoDocument(<Password/>)` which will produce an instance `Password` of React element:

```
 Password {
  props: {},
  context: {},
  refs: {},
  updater:
  {...},
  state: { strength: {}, password: '', visible: false, ok: false },      ①
  generate: [Function: bound generate],
  checkStrength: [Function: bound checkStrength],
  toggleVisibility: [Function: bound toggleVisibility],
  _reactInternalInstance:
  { _currentElement:
    { '$$typeof': Symbol(react.element),
      type: [Function: Password],
      key: null,
      ref: null,
      props: {},
      _owner: null,
      _store: {} },
    ...
  }
}
```

- ① You get state which you don't get with shallow rendering
- ② You get element which looks like the result of shallow rendering

Needless to say, developers can't really test user behavior and nested elements with the shallow rendering. However shallow rendering can be use to test the first level of children in a component as well as its type. Developers can use this feature for custom (composable) component classes.

In the real world, you'd use shallow rendering for very targeted (almost unit-like) testing of a single component and its rendering. Shallow will allow you to avoid getting into children. You can use it when there's no need to test children, user behavior or changing states of a component. In other words, you would only need to test the `render()` function of a single element. A general rule of thumb, start with shallow rendering, then if that's not enough—continue with regular rendering.

For standard HTML classes, they can simply inspect and assert `el.props` hence there's no need for shallow renderer for standard HTML elements. For example, this is how we can create an anchor element and test that it has the expected class name and the tag name:

```
let el = <a className='btn' />
expect(el.props.className).toBe('btn')
expect(el.type).toBe('a')
```

We've covered a big chunk of TestUtils and Jest. This is enough to start using it in your projects. In fact, that's exactly what we'll be doing in the Practicum: Part III of this book—use Jest and TestUtils to BDD React components. The aforementioned Password widget is in that chapter as well if you want to take a look at the Webpack setup and all dependencies in real world so to say.

For more information on TestUtils, refer to [the official documentation](#). Jest is an extensive topic, and its full coverage is outside the scope of this book. Feel free to consult [the official API documentation](#) to learn more.

Lastly, there's [an Enzyme library](#) which provides a few more features and methods than TestUtils as well as more compact names for the methods. It's [developed by Airbnb](#) and require TestUtils as well as jsdom (which comes with Jest so you'll need jsdom only if you're not using Jest).

In this chapter we've covered the beast that is testing. It's so frightful that some developers skip it altogether, but not you. You stuck 'til the end. Congrats. Your code will be of better quality, and you'll develop faster and live a happier life. You won't have to wake up in the middle of the night to fix a broken server...or at least, not as frequently as someone without tests.

15.4 Quiz

1. The Jest tests must be in the folder named: `tests`, `__test__`, or `__tests__`?
2. TestUtils is installed with npm from `react-addons-test-utils`. True or false?
3. What TestUtils method allows you to find a single component by its HTML class?
4. What is the `expect` expression to compare objects (deep comparison)?
5. How do you test the user behavior when he/she hovers with a mouse: `TestUtils.Simulate.mouseOver(node)`, `TestUtils.Simulate.onMouseOver(node)`, `TestUtils.Simulate.mouseDown(node)`?

15.5 Summary

In this chapter we've covered:

- To install Jest, use `npm i jest-cli --save-dev`
- To test a module, turn automocking off for it with `jest.dontMock()`
- Use `expect.toBe()` and [other Expect functions](#)
- To install TestUtils, use `npm i react-addons-test-utils --save-dev`
- Use `TestUtils.Simulate.eventName(node)` where `eventName` is React event (without the `on` prefix) to test trigger DOM events
- Use `scry...` methods to fetch multiple elements
- Use `find...` methods to fetch a single element (it will give you an error if you have more than one elements: Did not find exactly one match (found: 2+))

15.6 Quiz Answers

1. `__tests__` because this is the convention Jest follows
2. True, the TestUtils is a separate npm module
3. `findRenderedDOMComponentWithClass()`
4. `expect(OBJECT).toEqual(value)` will compare objects on sameness without comparing that they are the same objects (which is done by `==` or `toBe()`)
5. `TestUtils.Simulate.mouseOver(node)` because `mouseOver` event is triggered by hovering the cursor

16

React on Node and Universal JavaScript

This chapter covers

- Why React on the Server and What is Universal JavaScript
- React on Node
- React and Express
- Universal JavaScript with Express and React



Figure 16.1 Watch this chapter's introduction video by scanning this QR code with your phone or going to <http://reactquickly.co/videos/ch16>

React is mostly a front-end library to build full-blown single-page application or simple UIs on *the browser*. So why would you be concerning yourself with using it on the server? Isn't it the old way to render HTML on the server? Well, yes and not. It turns out that when developers build web apps which *alwaysrender* on the browser—they miss out on a few key goodies. In fact, they and their apps miss out big time to the point of not being able to rank high in Google search results or losing millions of dollars in revenue. Arghhh.

How is this possible? Read on to find out about why. You can skip this chapter in only one case: if you are completely oblivious to the performance of your apps (that is, if you're a newbie developer). All others, please proceed. You'll gain precious knowledge that you can use

to build amazing apps and be able to look smart during a developers' happy hour (by using the term "Universal JavaScript").

Also, you'll grasp how to use React with Node and build Node servers, and understand how to build universal JavaScript apps with React.js and Express.js (the most popular Node.js framework).

In this chapter, we use Express to write concise Node code. If you've not come across Express before, take a look at *Express in Action* (Manning, 2015) or at my book *Pro Express.js* (Apress, 2014). If you are familiar with Express but need a refresher, download my Express.js cheatsheet for free or view it online on Github. Express installation meanwhile is covered in Appendix A.

The source code for the examples in this chapter is in [the ch16 folder](#) of the GitHub repository [azat-co/react-quickly](#). And some demos can be found at <http://reactquickly.co/demos>.

16.1 Why React on the Server and What is Universal JavaScript

You might have heard about universal JavaScript in relationship to web development. It's become such a buzzword in recent months that it seems like every web tech conference in 2016 had not one but several presentations about it! There are even a few synonyms to "universal JavaScript" such as "isomorphic JavaScript", or "fullstack JavaScript". For simplicity, I'll stick with: universal for this chapter. This section will help you finally understand what isomorphic/universal JavaScript is about.

But before we define universal JavaScript, let's discuss some of the issues developers face when building SPAs. The three main problems developers face are:

- No search engine optimization (SEO): SPAs generate HTML 100% on the browser and search crawlers don't like that
- Poor performance: Huge bundled files and AJAX calls slows down performance (especially on the first page load when it's critical)
- Poor maintainability: often SPAs lead to the duplication of code on browser and server

Let's take a closer look at each of those problems.

16.1.1 Ability for Search Engines to Index Pages Properly

Single-page applications (SPAs) built with frameworks like Backbone.js, Angular.js, Ember.js, or others are widely used for protected apps—that is, apps you need to enter a username and password to access (for example Figure 16-1). Most SPAs serve protected resources and don't need indexing, but the vast majority of websites are not protected behind logins.

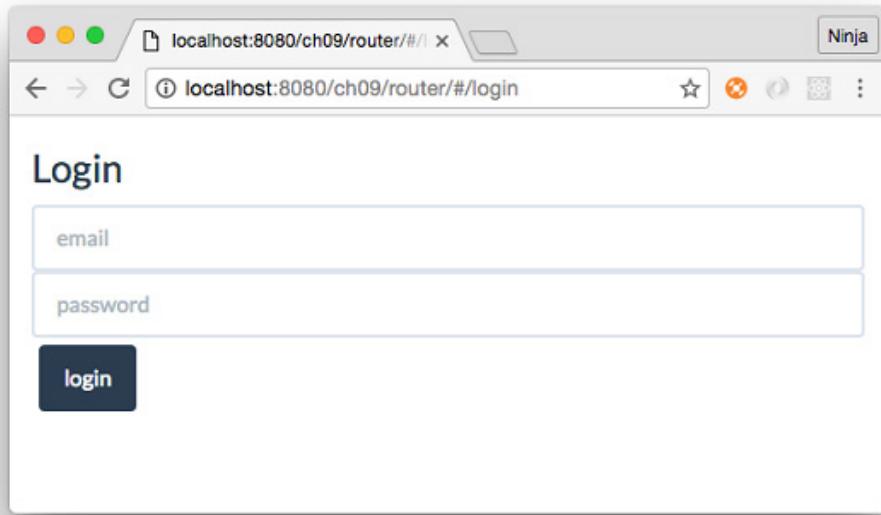


Figure 16.2 SPA which won't need SEO support because it's behind the login screen

For such public apps, SEO is important and actually mandatory, because their business depends heavily on search indexing and organic traffic. This category of website is the majority.

Unfortunately, when developers try to use SPA architecture for public facing websites, which should have good search engine indexing, it's not straightforward at all. SPA rely on browser rendering thus developers need to either re-implement the templates on the server or pre-generate static HTML pages using headless browsers just for the search engine crawlers.

Google's Support for Browser Rendering is Lacking

Recently, Google added JavaScript rendering capability to its crawlers. You might think that now your browser rendered HTML will be indexed correctly. You might think that by using Angular with a REST API server you don't need any server-side rendering. Unfortunately this might not be the case since, as it's stated in "["Understanding web pages better."](#): "Sometimes things don't go perfectly during rendering, which may negatively impact search results for your site." The gist is that Google itself is not recommending to rely on their indexing of SPAs. They can't guarantee that what's in their cache, index, and search results is exactly what your SPA rendered. So, to be on the safe side, we need to render without JavaScript as closely as possible to the JavaScript-enabled rendering.

With universal JavaScript and React in particular, developers can generate HTML on the server for the crawlers from the same components that browsers will be using to generate HTML for users. No need for bulky headless browsers to generate HTML on the server. Win-win!

16.1.2 Better Performance with Faster Loading Time

While some applications must have proper search engine indexing, others thrive on faster performance. Websites like [mobile.walmart.com \(article\)](#) and [Twitter.com \(article\)](#) have done research that showed that they needed to render the first page (first load) on the server to make the performance better. That literally cost companies millions of dollars because users will just leave if the first page is not fast enough.

Being web developers and working and living with good internet connection speed, we might forget that our website will be accessible on a slow connection. What loads in a split second will take half a minute in other cases. Suddenly, the bundle which is over 1Mb is too large. And loading the bundled file is just half of the story because now the SPA needs to make AJAX requests to the server to load the data... all while your users patiently stare at the "Loading..." spinner. Yeah, right. Some of them already left while others are frustrated.

We want to show users a functional webpage as fast as we possibly can, not just a skeleton HTML with "Loading...". Other code can be loaded later while the user browses the webpage.

With universal JavaScript, it's easy to generate HTML for to show the first page *on the server*. As a result, when users load the first page they won't see this obstructing "Loading..." message. The data will be in the HTML for users to enjoy. They will see a functional page, thus having a better user experience.

The performance boost is coming from the fact that users won't need to wait for the AJAX calls to resolve. There are other opportunities to optimize performance as well such as pre-loading the data and caching it on the server before AJAX calls came to the server (in fact, that's exactly what [we did at DocuSign by implementing data router](#)).

16.1.3 Better Code Maintainability

Code is a liability. The more there is, the more you and your team will need to support it. For these reasons, you want to avoid having different templates and logic for the same pages. Luckily Node.js, which is an essential part of Universal JavaScript, makes it effortless to use front-end/browser modules on the server. Many template engines, such as Handlebars, Mustache, Dust, and others, can be used on the server.

Given these problems and knowing that Universal JavaScript can solve them, what is a practical application?

16.1.4 Universal JavaScript with React and Node

In short, *universal*, in regard to web development, means using the same code (typically written in JavaScript) on both the server side and the client side. A narrow use case of universal JavaScript is rendering on the server and client from the same source. Universal JavaScript often implies the use of JavaScript and Node.js, because this language and platform combination allows for the reuse of the libraries.

Simply put, browser JavaScript code can be run in the Node.js environment with very few modifications. As a consequence of this interchangeability, the Node.js and JavaScript ecosystem has a wide variety of isomorphic frameworks, such as [React.js](#), [LazoJS](#), [Rendr](#), [Meteor](#) and others.

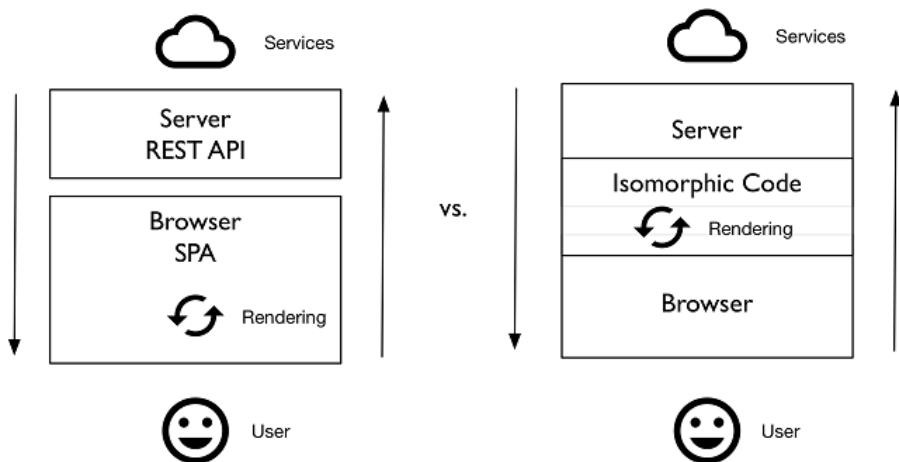


Figure 16.3 Universal HTML generation and code sharing between browser and server vs. no code sharing in traditional SPA

In a practical application Universal JavaScript architecture will consist of

- Client-side React code for the browser : an SPA or just some simple UIs making AJAX requests)
- A Node.js server generating HTML for the first page on the server and serving browser React code with the same data. This can be implemented using Express and either a template engine or React components as a template engine.
- Webpack to compile JSX for both the server and the browser.

Take a look at Figure 16-3 for the visual mental model.

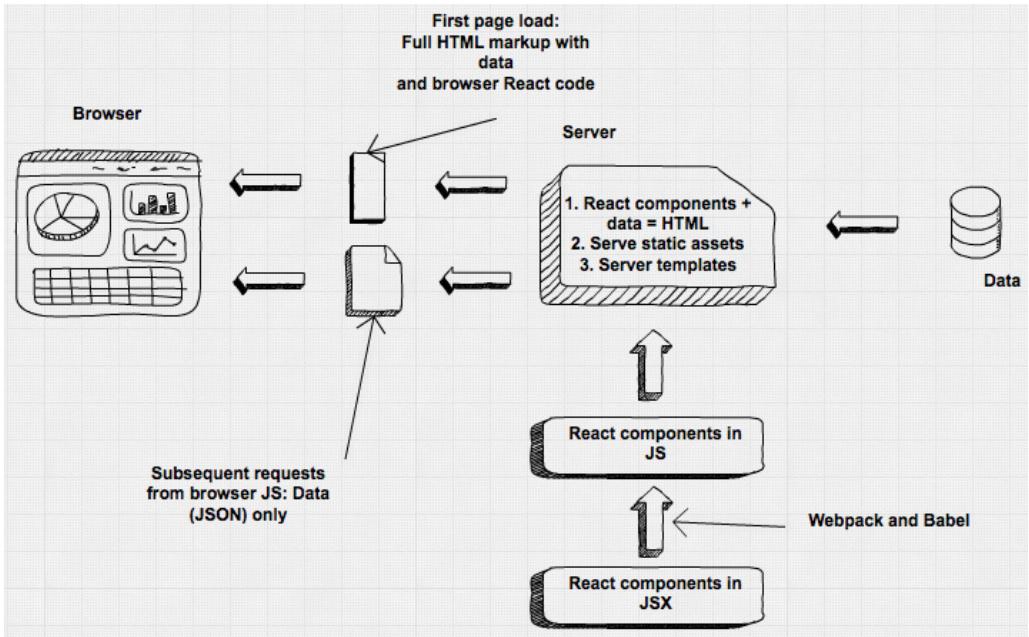


Figure 16.4 Practical application of universal JavaScript with React, Node and Express

You might be thinking: "Show me how to use this wonder, this universal JavaScript already!" All right, we are moving to a hands-on example of rendering React components on the server. We'll do so in a gradual way because there are several components (as in parts, not React components!) to leveraging Universal JavaScript pattern. We'll need to learn these things:

1. Generate HTML from React components: Just React components as input and plain HTML as output, no HTTP(S) servers yet
2. Render HTML code generated from React components in Express servers: Similar to number 1, but now we use React inside of a template engine for 100% server-side rendering (no browser React yet)
3. Implement and serve React browser files via Express: At some point you'll need an HTTP(S) server and Express is on of the options, that is there's no server side HTML generation, just serve built static assets

In the end, we'll be using React to generate server-side HTML while loading browser React at the same time—the holy grail of universal JavaScript. But before we can fly, let's learn to walk!

16.2 React on Node

We'll start with a basic use case of just generating HTML from a Node script. No servers or anything complex yet. Just importing of components and generating HTML.

There are only a handful of methods to learn to generate HTML from React components on the server. First, you'll need the npm modules `react` and `react-dom`. You can install React and npm following the instructions in appendix A. We are using React and React DOM version 15.

For those new to writing server-side Node code, you might wonder where does this server-side code go? It goes in a plain text file. Let's name it `index.js` and the React component will be in `email.js` (we'll cover non-JSX plain JavaScript for now). Those two files must be in the same folder (`ch16/node`)

The project structure goes like this:

```
/node
  /node_modules      ①
  email.js          ②
  email.jsx         ③
  index.js
  package.json
```

① Dependencies
② Email component
③ Node code

First, include the modules in your server-side code in `node/index.js`:

Listing 16.1 Server-side setup code (index.js)

```
const ReactDOMServer = require('react-dom/server')           ①
const React = require('react')
const Email = React.createFactory(require('./email.js'))      ②
...
...
```

- ① Import React DOM Server class
② Create a function which returns elements of Email class

What's up with that `createFactory()`? you might ask. Well, if we just import `email.js` that would be a component class, but we need a React element. Thus we can use JSX, `createElement()` or `createFactory()`. The latter will give a function which when invoked will give us an element.

Once you've imported your components, simply run `renderToString()` from `ReactDOMServer`:

```
const emailString = ReactDOMServer.renderToString(Email())
```

The code fragment from `index.js`:

```

const ReactDOMServer = require('react-dom/server')
const React = require('react')
const Email = React.createFactory(require('./email.js'))

const emailString = ReactDOMServer.renderToString(Email())
console.log(emailString)
// ...

```

Now, is `email.js` regular JavaScript? In this case, it has to be. You can "build" JSX into regular JS with Webpack.

TIP : Importing JSX

To use JSX, Another approach is to convert it on the fly. There's a library which will enhance `require` to do just that so we can configure our `require` once and then import JSX just as any other JS files.

To import JSX, developers can use `babel-register` as shown below in their `index.js` in addition to installing `babel-register` and `babel-preset-react`(use `npm` to install them).

```

require('babel-register')({
  presets: [ 'react' ]
})

```

Listing 16.2 Server-side Email (node/email.jsx)

```

const React = require('react')

const Email = (props)=> {
  return (
    <div>
      <h1>Thank you {(props.name)?props.name:''}for signing up!</h1>
      <p>If you have any questions, please contact support</p>
    </div>
  )
}

module.exports = Email

```

You'll get strings rendered by React components. You can use these strings in your favorite template engine to show on a web page or somewhere else (like in email HTML). In my case, the `email.js` (`ch16/node/email.js`) with a heading and a paragraph will render into this HTML string:

Listing 16-3: Email (node/email.jsx) rendered into strings with Universal React attributes

```

<div data-reactroot="" data-reactid="1" data-react-checksum="1319067066">
  <h1 data-reactid="2">
    <!-- react-text: 3 -->Thank you <!-- /react-text -->
    <!-- react-text: 4 -->
    <!-- /react-text -->
    <!-- react-text: 5 -->for signing up!<!-- /react-text -->
  </h1>
  <p data-reactid="6">If you have any questions, please contact support</p>

```

```
</div>
```

Now, what is happening with those attributes `data-reactroot`, `data-reactid` and `data-react-checksum`? We didn't put them in there, but React did. Why? That's for browser React and universal JavaScript (next section).

If you won't need the React markup that browser React needs (for example, if you're creating an email HTML), use the `ReactDOMServer.renderToStaticMarkup()` method. It works similarly to `renderToString()`, but strips all the `data-reactroot`, `data-reactid` and `data-react-checksum` attributes. In this case, React will be just like any other static template engine.

For example, you can load the component from `email.js` and generate HTML with `renderToStaticMarkup()` instead of `renderToString()`:

```
const emailStaticMarkup = ReactDOMServer.renderToStaticMarkup(Email())
```

The resulting `emailStaticMarkup` will be sans React attributes as follows:

```
<div><h1>Thank you for signing up!</h1><p>If you have any questions, please contact support</p></div>
```

So while for email you won't need the browser React, the original `renderToString()` is what we'll be using for universal JavaScript architecture with React. The way it works: server-side React adds some secret sauce to the HTML in the form of checksums (`data-react-checksum`). Those checksums will be compared by the browser React, and if they match, browser components won't regenerate/repaint/rerender unnecessarily. There would be no flash of content (which often happens due to rerendering). The checksums will match if the data supplied to the server-side components is *exactly* the same as that on the browser. But how to supply the data to the components create on the server? As props!

If you need to pass some props, pass them as an object parameter. For example, you can provide the name (Johny Pineappleseed) to the `Email` component:

```
const emailStringWith Name = ReactDOMServer.renderToString(Email({
  name: 'Johny Pineappleseed'
}))
```

The full `index.js` with three way to render HTML: static, string and string with a property is shown below

```
const ReactDOMServer = require('react-dom/server')
const React = require('react')
const Email = React.createFactory(require('./email.js'))

const emailString = ReactDOMServer.renderToString(Email())
const emailStaticMarkup = ReactDOMServer.renderToStaticMarkup(Email())
console.log(emailString)
console.log(emailStaticMarkup)

const emailStringWith Name = ReactDOMServer.renderToString(Email({name: 'Johny Pineappleseed'}))
```

```

        ))
console.log(emailStringWithNames)

```

That was rendering React components into HTML just in a plain Node, no servers and no thrills. Next, we'll cover using React in an Express server.

16.3 React and Express: Rendering on the Server-side From Components

Express.js is one of the most popular, or even *the* most popular, Node.js frameworks. It's simple yet highly configurable. There are hundreds of plugins called middleware that you can use with Express.js.

In a bird's-eye view of the tech stack, Express and Node takes the place of an HTTP(S) server, effectively replacing technologies like [Microsoft IIS](#), [Apache httpd](#) or [nginx](#) or [Apache Tomcat](#). What is unique about Express and Node is that they allow you to build highly scalable and performant systems, thanks to the [non-blocking I/O nature](#) of Node. Express's advantage is that it has a vast ecosystem of plugins (called middleware), and a very mature and stable codebase.

Unfortunately, a detailed overview of the framework is out of the scope of this book, but what we can do is to learn how to create a small Express app and render React in it. In no way this is a deep dive in Express.js, but it'll get you started with the most widely-used Node.js web framework. Call it an express Course in Express if you wish. :-)

NOTE As mentioned earlier, Appendix A covers how to install both node.js and Express if you want to follow along with this example

16.3.1 Rendering Simple Text Server-side

Let us build an HTTP and HTTPS servers using Express and then generate HTML server-side using React as schematically shown in Figure 16-4.

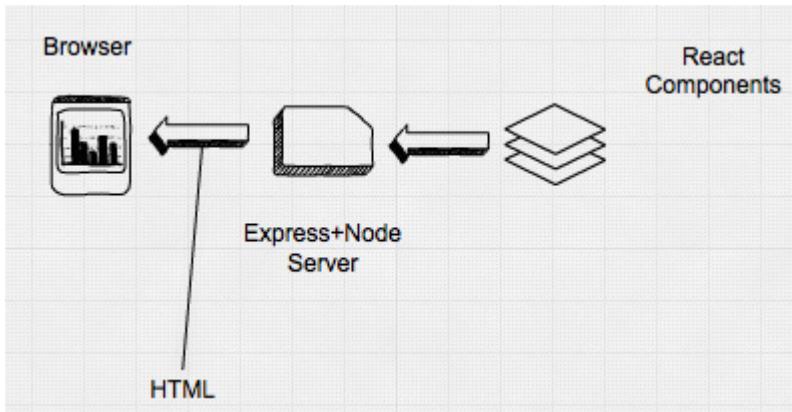


Figure 16.5 Express/Node server will be generating HTML and sending it to the browser

The most basic example of using React in Express, would be to generate an HTML string without markup and send it as a response to the request:

Listing 16.4 A simple use of React on Express which show HTML on a page

```
const express = require('express')
const app = express()           ①
const http = require('http')

const ReactDOMServer = require('react-dom/server')
const React = require('react')
const About = React.createFactory(require('./components/about.js'))      ②

app.get('/about', (req, res, next) => {
  const aboutHTML = ReactDOMServer.renderToString(About())
  response.send(aboutHTML)          ③
})

http.createServer(app)           ④
.listen(3000)
```

- ① Import express library
- ② Import About component and create a React object
- ③ Send HTML string back to client in response to /about request
- ④ Instantiate the HTTP server and boot it up

This will work but the /about won't be a complete page with <head> and <body>. Also you might be wondering what is `app.get()` and `app.listen()`. Let's take another example and all will be revealed.

16.3.2 Rendering An HTML Page

Let's cover a more interesting example in which we will also use some external plugins and a template engine.

The idea for the app is the same: serve HTML generated from React using Express. The page will have some text which will be generated from `about.jsx` as you can observe on Figure 16-5. No thrills but this way it's better because it's simple and starting with simple is good.

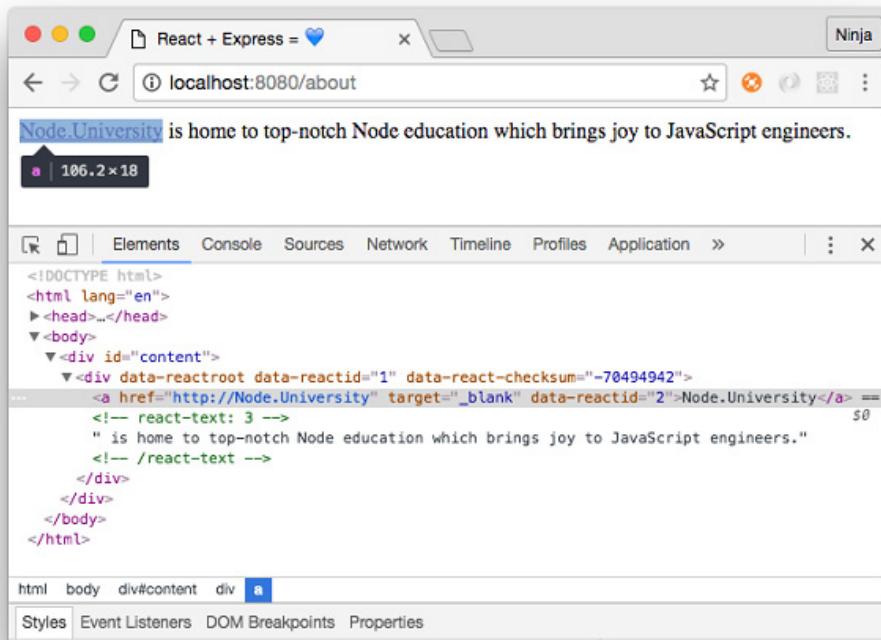


Figure 16.6 Rendering from the React component server-side

Create a folder called `react-express`. (This example is in `ch16/react-express`). The end project structure will look as follows:

```
/react-express
  /components
    - about.jsx
  /views
    - about.hbs
  - index.js
```

```
- package.json
```

Firstly, create `package.json` with `npm init -y`, then install Express with `npm` like this:

```
$ npm install express@4.14.0 --save
```

As with any Node applications, let open an editor and create a file. Typically, you'd create the server file name `index.js`, `app.js` or `server.js` which you'll later start with the `node` command (e.g., `node index.js`). You can name it `index.js`.

The file has these parts:

1. Imports: Require the dependencies such as `express` itself and its plugins
2. Configurations: Set certain configuration values such as what template engine to use
3. Middleware: Define common actions performed for all incoming requests, e.g., validation, auth, compression, etc.
4. Routes: Define the URLs handled by this server such as `/accounts`, `/users`, etc., as well as their actions
5. Error handlers: Show meaningful messages or webpages when errors happen
6. Bootup: Start HTTP and/or HTTPS server(s)

Listing 16.5 High-level overview of the Express&Node server file

```
const fs = require('fs')                                     ①
const express = require('express')
const app = express()
const errorHandler = require('errorhandler')
const http = require('http')
const https = require('https')

// ...

app.set('view engine', 'hbs')                                ②

app.get('/',                                                 ③
  // ...
)
app.get('/about',
  // ...
)

// ...

app.use(errorHandler)                                       ④

http.createServer(app)                                      ⑤
  .listen(3000)

// ...
if (typeof options != 'undefined')
  https.createServer(app, options)                         ⑥
    .listen(443)
```

① Import modules

- 2 Set configurations
- 3 Define routes (no pure middleware in this project)
- 4 Define error handlers (type of a middleware)
- 5 Boot up http server
- 6 Boot up https server

Now we will go deeper. The imports sections is straightforward. In it you would require dependencies and instantiate objects. For example to import the Express.js framework itself and to create an instance write these lines:

```
var express = require('express')
var app = express()
```

CONFIGURATION

In the configurations sections, set configurations with `app.set()` where the first argument is a string and the second is a value. For example, to set the template engine to hbs, use this configuration `view engine`:

```
app.set('view engine', 'hbs')
```

`hbs` is an Express template (or view) engine for [the Handlebars template language](#). Most of you probably worked with Handlebars or some close relatives of it such as Mustache, Blaze, etc. Ember also uses [Handlebars](#). Thus, it's a common and easy-to-get-started template, that's why we'll use it here.

One caveat, you must install `hbs` package in order for Express to properly use the view engine. Do so by executing: `npm i hbs --save`.

MIDDLEWARE

The next section is for setting up middleware which is like plugins. For example, to enable the app to serve static assets, use static middleware:

```
app.use(express.static(path.join(__dirname, 'public')))
```

Static middleware is great because it turns Express into a static HTTP(S) server which proxies requests to files in a specified folder (`public` in our example) just like `nginx` or `apache httpd` would.

Next is routes. They called endpoints, resources, pages and many more names. The idea is that we define a URL pattern which will be matched by Express against real URLs of incoming requests. In case there's a match, Express will execute the logic associate with this URL pattern which is called handling of a request. It can be anything from displaying a static HTML for a 404: Not Found page or making a request to another service, and caching the response before sending it back to the client.

ROUTES

So routes are the most important part. We define routes with `app.NAME()` pattern where NAME is a name of an HTTP method in lower case. For example, this is a syntax of GET / (home page or empty URL) endpoint which will just send back the string "Hello":

```
app.get('/', (request, response, next)=>{
  response.send('Hello!')
})
```

For the `/about` page/route, we can change the first argument—URL pattern. Also, we can render the HTML string.

```
app.get('/about', function(req, res, next) {
  response.send(`<div>
    <a href="http://Node.University" target="_blank">Node.University</a>
    is home to top-notch Node education which brings joy to JavaScript engineers.
  </div>`)
})
```

LAYOUT WITH HANDLEBARS

Now, we want to render React HTML from the Handlebars template because Handlebars will provide us with an overall layout such as `<html>` and `<body>`. In other words, we keep React for UI elements and Handlebars for the layout.

To do so, create a new folder `views` and a template `about.hbs` in this new and empty (but not for long time) folder `views`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>React + Express = ☀</title>
    <meta name="author" content="Azat" />
  </head>

  <body>
    <div id="content">{{{about}}}</div> ❶
  </body>
</html>
```

- ❶ Use triple curly braces to output unescaped HTML from the `about` variable (supplied in `index.js`)

RENDERING THE PAGE

Then, in the route change `response.send()` to `response.render()` like this:

```
// ...
const React = require('react')
require('babel-register')({ ❶
  presets: [ 'react' ]
})
const About = React.createFactory(require('../components/about.jsx')) ❷
```

```
// ...
app.get('/about', (request, response, next) => {
  const aboutHTML = ReactDOMServer.renderToString(About())
  response.render('about', {about: aboutHTML})      ③
})
// ...
```

- ① Enhance require to convert **JSX** on the fly which will enable us importing/requiring **JSX** files
- ② Prepare About component
- ③ Generate React **HTML** string with React markup
- ④ Pass React **HTML** string to Handlebars template about.hbs

To sum it up, Express routes can render from templates Handlebars (with data such as `about` string variable) or just send response in a string format.

Do I have to use a different template engine for the server rendering and layouts?

Please note, it's possible to use React for layouts instead of Handlebars. There's a `express-react-views` library for that. It's only for static markup and not for browser React. We won't be using or covering it, because it requires extensive use of `dangerouslySetInnerHTML`, is not supporting all HTML it's most often confuses beginner Express-React developers. In my opinion, there's very little benefit of using React for layouts.

HANDLING ERRORS

Error handlers are similar to middleware, e.g., they can be imported from a package such as `errorhandler`:

```
const errorHandler = require('errorhandler')
...
app.use(errorHandler)
```

Or create right there in `index.js`:

```
app.use((error, request, response, next) => {
  console.error(request.url, error)
  response.send('Wonderful, something went wrong... ')
})
```

The way error handlers work is by invoking `next(error)` with an error object from without a request handler.

BOOTING UP THE SERVER

Lastly, to start your app run implement `listen()` by passing a port number and a callback (optional):

```
http.createServer(app).listen(portNumber, callback)
```

In our example, it'll look like this:

```
http.createServer(app)
```

```
.listen(3000)
```

Here's the full server code of index.js to make sure nothing has slipped through the cracks

```
const fs = require('fs')
const express = require('express')
const app = express()
const errorHandler = require('errorhandler')
const http = require('http')
const https = require('https')

const React = require('react')
require('babel-register')({
  presets: [ 'react' ]
})
const ReactDOMServer = require('react-dom/server')
const About = React.createFactory(require('./components/about.jsx'))

app.set('view engine', 'hbs')
app.get('/', (request, response, next)=>{
  response.send('Hello!')
})

app.get('/about', (request, response, next) => {
  const aboutHTML = ReactDOMServer.renderToString(About())
  response.render('about', {about: aboutHTML})
})

app.all('*', (request, response, next)=> {
  response.status(404).send('Not found... did you mean to go to /about instead?')1
})
app.use((error, request, response, next) => {
  console.error(request.url, error)
  response.send('Wonderful, something went wrong...')2
})

app.use(errorHandler)

http.createServer(app)
  .listen(3000)

try {
  const options = {
    key: fs.readFileSync('./server.key'),
    cert: fs.readFileSync('./server.crt')2
  }
} catch (e) {
  console.warn('Create server.key and server.crt for HTTPS')
}
if (typeof options != 'undefined')
  https.createServer(app, options)
    .listen(443)
```

- 1** Implement catch all fallback. You would't believe how many people in my classes would implement a server and then go to a non-existing URL and think there's a error when in fact they should be viewing /about
- 2** Load key and certificate for SSL/HTTPS. You can look up how to generate them in [Easy HTTP/2 Server with Node.js and Express.js](#)

Now, everything should be ready to run the server with `node index.js` or its shortcut `node .` to see the server response once you navigate to <http://localhost:3000/about>. If something missing or you have errors when starting the server and navigation to the address, please refer to the project source code in `ch16/react-express`.

The end result should render a proper HTML page with header and body and inside of the body it should have React markup such as checksum and reactroot as shown in Figure 16.7.

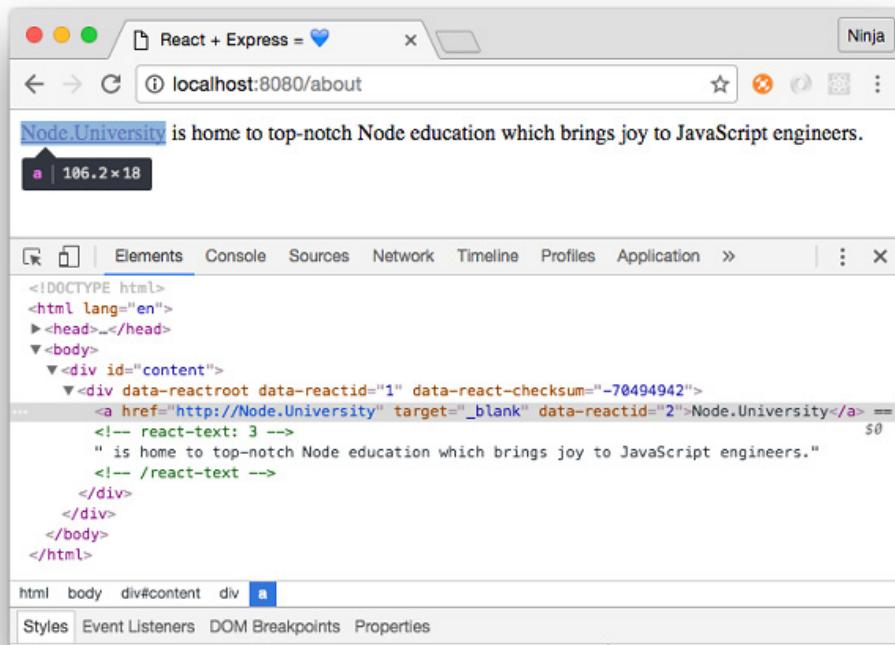


Figure 16.7 Rendering React markup from a Handlebars layout using Express will give an HTML page

Why did we use markup rendering and not just static HTML string or `express-react-views`? We'll need this markup with checksums later for the browser React, that's universal JavaScript architecture.

In the next section, we are putting all these React on browser, Express and React on Node skills together to implement universal JavaScript architecture.

16.4 Universal JavaScript with Express and React

In this section on Universal JavaScript with Express and React all the skills of this chapter (and most of the book?!) will come together. We will need to render component(s) on the server, plug it in the template, and enable browser React.

To learn this Universal JavaScript, we will build a Message Board. There will be three components: Header, Footer and MessageBoard as can be seen on Figure 16-7. The Header and Footer component will have static HTML just to display some text while MessageBoard will have the form to post messages on the board and a list of messages. The Message board will be using AJAX calls to get list of messages and post a new message to the back-end server which in turn will utilize MongoDB NoSQL database.

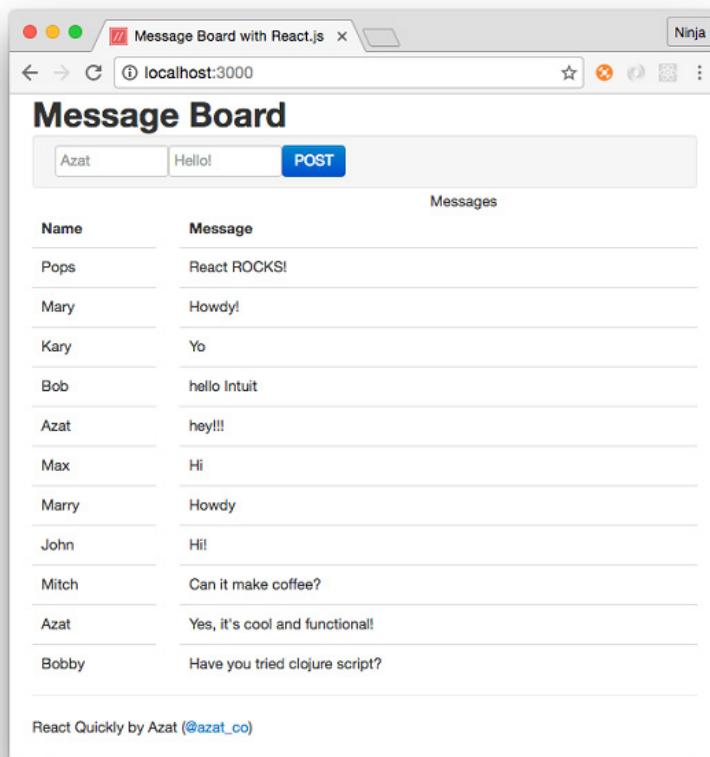


Figure 16.8 Message Board with a form to post a message and a list of existing messages

In a nutshell for universal React, you'll need to:

1. Set up server so that it hydrate data to the template and render HTML (components and props), e.g., `index.js`
2. Create a template that outputs data (a.k.a. locals) unescaped, e.g., `views/index.hbs`
3. Include the browser React file (`ReactDOM.Render`) in the template for interactivity , e.g., `client/app.jsx`
4. Create components, `Header`, `Footer` and `MessageBoard`
5. Set up build processes with Webpack, e.g., `webpack.config.js`

See the diagram of the practical application of universal JavaScript with React and Express again on Figure 16-8.

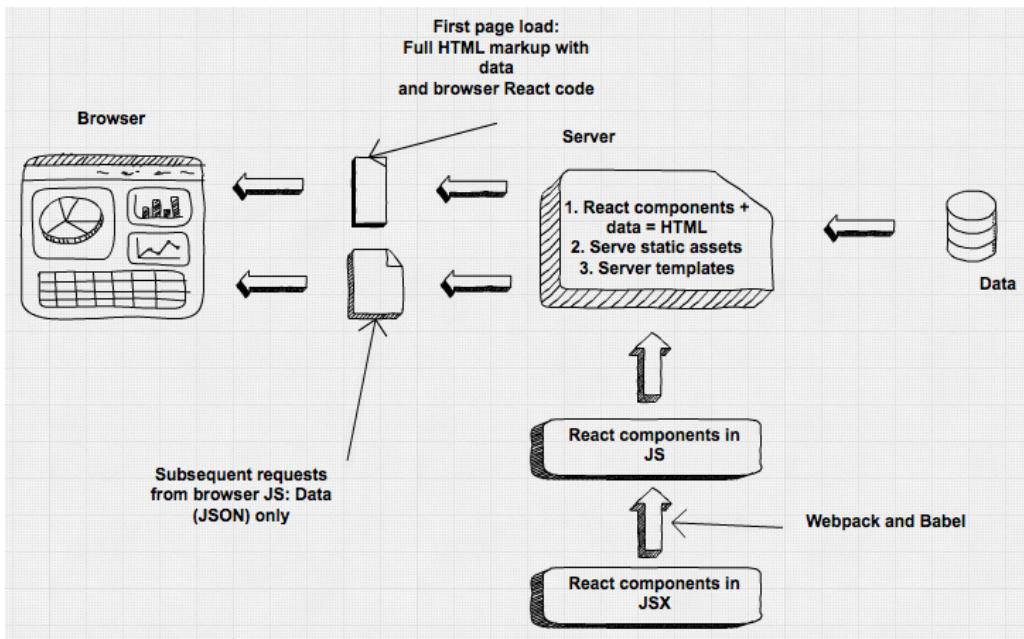


Figure 16.9 Gist of Universal JavaScript with React and Express

NOTE You'd also need to install and launch MongoDB for this example to work. You can read about installation on its website or in the appendix of this book. Once you install MongoDB, simply run `mongod` and leave it running. This will allow our Express server to connect to it using the magic URL: `url = 'mongodb://localhost:27017/board'`.

You can look up the full source code of `index.js` in the GitHub repository.

16.4.1 Project Structure and Configuration

The project goes as follows:

```
/client
  - app.jsx          ①
/components
  - board.jsx        ②
  - footer.jsx
  - header.jsx
/node_modules
/public
  /css
  /js               ③
    - bundle.js
    - bundle.js.map
/views
  - index.hbs
  - index.js         ④
  - package.json
  - webpack.config.js
```

- ① Client/browser code
- ② Shared code between client/browser and server
- ③ Compiled and bundled by Webpack scripts
- ④ Server code

The server dependencies will include these packages (quote from `package.json`):

```
...
"dependencies": {
  "babel-register": "^6.11.6",           ①
  "body-parser": "^1.13.2",
  "compression": "^1.5.1",
  "errorhandler": "^1.4.1",
  "express": "^4.13.1",                  ②
  "hbs": "^4.0.0",
  "express-validator": "^2.13.0",
  "mongodb": "^2.2.6",                  ③
  "morgan": "^1.6.1",
  "react": "^15.3.0",
  "react-dom": "^15.3.0"                ④
},
...
```

- ① Load JSX from Node with `require`
- ② Use Express framework
- ③ Use MongoDB for storing messages (this is the driver; you need both, the driver and the database)
- ④ Use React for rendering on the server

A few words about the middleware used in this project, for those completely new to Express, Express is not a large framework which does almost everything for you. Contrary, it's a base foundation layer on top of which Node engineers build their very custom systems, virtually their own frameworks. They turn out to be fit precisely to the task at hand which is not always

the case with all-in-one frameworks. You get only what you need with Express and its ecosystem of plugins. Those plugins are called middleware because they use middleware pattern with Express implementing the middleware manager.

Every Express engineer has their favorite middleware packages which he/she uses from project to project. I tend to start with these and then add along more packages if/when I need them:

- `compression`: Automatically compresses responses using gzip algorithm. This makes responses smaller and faster to download — useful thing.
- `errorhandler`: Rudimentary error hanler such as 404 and 500 errors
- `express-validator`: Validation of payload of incoming requests. Always a good idea to have.
- `morgan`: Logging of reqests on the server. Supports multiple formats.
- `body-parser`: Enable automatic parsing of JSON and urlencoded data format into Node/JS object accessible in `request.body`

For information on middleware `compression`, `body-parser` and `errorhandler`, as well as for the list of additional Express middleware, refer to the Appendix C or [Pro Express](#).

Now, we can set up the server in `message-board/index.js`.

16.4.2 Setting up the Server

Just as we did in the previous examples, we'll implement the server side of things in `index.js` and then work through the five sections so you can see how it breaks down. First of all, here it is in full.

Listing 16.6 The server-side of the message board app (ch16/message-board/index.js)

```
require('babel-register')({  
  presets: [ 'react' ]  
}  
  
const express = require('express'),  
  mongodb = require('mongodb'),  
  app = express(),  
  bodyParser = require('body-parser'),  
  validator = require('express-validator'),  
  logger = require('morgan'),  
  errorHandler = require('errorhandler'),  
  compression = require('compression'),  
  url = 'mongodb://localhost:27017/board',  
  ReactDOMServer = require('react-dom/server'),  
  React = require('react')  
  
const Header = React.createFactory(require('./components/header.jsx')),  
  Footer = React.createFactory(require('./components/footer.jsx')),  
  MessageBoard = React.createFactory(require('./components/board.jsx'))
```

```

mongodb.MongoClient.connect(url, function(err, db) {           ③
  if (err) {
    console.error(err)
    process.exit(1)
  }

  app.set('view engine', 'hbs')

  app.use(compression())
  app.use(logger('dev'))
  app.use(errorHandler())
  app.use(bodyParser.urlencoded({extended: true}))
  app.use(bodyParser.json())
  app.use(validate())
  app.use(express.static('public'))

  app.set('view engine', 'hbs')

  app.use(function(req, res, next){
    req.messages = db.collection('messages')           ④
    return next()
  })

  app.get('/messages', function(req, res, next) {
    // ...
  })
  app.post('/messages', function(req, res, next){
    // ...
  })

  app.get('/', function(req, res, next){
    // ...
  })

  app.listen(3000)
})

```

- 1 Import JSX and compile it on-the-fly to JS
- 2 Define the address of the local MongoDB instance as well as the db name (board)
- 3 Connect to MongoDB instance using the URI
- 4 Set collection as a property of a request object for an easier access in other routes and their modularization

CONFIGURATION

Again, we'll need to use `babel-register` to import JSX, (after installing `babel-register` and `babel-preset-react` with npm).

```

require('babel-register')({
  presets: [ 'react' ]
})

```

In `index.js`, we are implementing our Express server. Let's import the components using a relative paths `./components/`:

```

const Header = React.createFactory(require('./components/header.jsx')),
Footer = React.createFactory(require('./components/footer.jsx')),

```

```
MessageBoard = React.createFactory(require('./components/board.jsx'))
```

For the purpose of rendering React apps, you need to know that Express.js can utilize pretty much any template engine. Let's consider Handlebars, which is very close to regular HTML. You can enable Handlebars with this statement, assuming `app` is the Express.js instance:

```
app.set('view engine', 'hbs')
```

The `hbs` module must be installed (I have it in `package.json`).

MIDDLEWARE

These middleware provide a lot of functionality to our server which otherwise we would have had to implement ourselves. I point to the most essential ones for this project:

```
// ...
app.use(compression())
app.use(logger('dev'))           ①
app.use(errorHandler())
app.use(bodyParser.urlencoded({extended: true}))
app.use(bodyParser.json())        ②
app.use(validate())
app.use(express.static('public')) ③
// ...
```

- ① Enable server logs for requests to help debugging and development
- ② Enable parsing of the incoming JSON data payloads
- ③ Enable access to all the files under `public` such as `bundle.js`

SERVER-SIDE ROUTES

Then, in your route—let's say, `/`—you would call `render` on `views/index.handlebars` (`res.render('index')`), because the default template folder is `views`:

```
app.get('/', function(req, res, next){
  req.messages.find({}, {sort: {_id: -1}}).toArray(function(err, docs){
    if (err) return next(err)
    res.render('index', {data})
  })
})
```

The `req.message.find()` call is just a MongoDB method to fetch documents. Although, you have to have MongoDB installed and running for this example to work verbatim (without any *changes*), I don't like to enforce my preference for a database on you. It's rather easy to replace calls to MongoDB with whatever you want. Most modern RDBMS and NoSQL databases have Node drivers. Most of them even have ORM/ODM libraries written in Node. Therefore, you can safely ignore my DB call, if you're not planning to use MongoDB.

If you do, Appendix D has [a cheatsheet](#) for you. The idea is that in the request handler, you can make a call to an external service (e.g., using `axios` to get Facebook user information) or use PostgreSQL. How you get the data in Node is not the focus of this chapter.

The most important thing here for us in regards to universal React is `res.render()` with two arguments. The first argument is the name of the template that is `index.hbs` which is in the `views` directory. The second argument to `res.render()` is the `locals`, which basically means data that will be used in the templates.

Listing 16.7 Rendering HTML generated from React components (ch16/message-board/index.js)

```
...
app.set('view engine', 'hbs')           ①
...

app.get('/', function(req, res, next){
  req.messages.find({}, {sort: {_id: -1}}).toArray(function(err, docs){
    if (err) return next(err)
    res.render('index', {
      header: ReactDOMServer.renderToString(Header()),
      footer: ReactDOMServer.renderToString(Footer()),
      messageBoard: ReactDOMServer.renderToString(MessageBoard({
        messages: docs          ④
      })),
      ...
    })
  })
})
```

- ① Apply template engine Handlebars
- ② Request array of messages from MongoDB
- ③ Send an HTML string from Header to the index template
- ④ Send an HTML string from MessageBoard generated with list of message to the index template

At this point, you'll have an Express server which renders handlebar template with three HTML strings from React components. This is not very exciting by itself. We could have done it without React. We could have used Handlebars or Pug or Mustache or any other template engine to render everything, not just the layout. Why do we need React? Well, we'll be using React on the browser and the browser React will take our server HTML and add all the events and states. All the magic. That's why!

We are not done with the server yet. There are two more things to implement for our example; they are the APIs:

- GET `/messages`: Get a list of messages from a database
- POST `/messages`: Create a new message in a database

These routes will be used by browser React when it will make AJAX/XHR requests to GET and POST data. The code for the routes goes in Express which is `index.js` as follows:

```
app.get('/messages', function(req, res, next) {
  req.messages.find({}, {sort: {_id: -1}}).toArray(function(err, docs){ ①
    if (err) return next(err)
    return res.json(docs)          ②
  })
})
```

```
    })
})
```

The route to handle creation of messages (POST /messages) will use express-validator to make sure that the incoming data is present (notEmpty()). express-validator is a convenient middleware because you can set up all kinds of validation rules. Input validation is paramount to securing your apps. Always sanitize your data. Developers get one-sided because they work with the code and the system. They know what data it supports. Developers get biased. *Developers should think that every user either an malicious attacker or just negligent person who never reads your instructions and always sends some weird data.*

The route will also use the reference to the database from req.messages to perform the insertion of a new message:

```
app.post('/messages', function(req, res, next){
  req.checkBody('message', 'Invalid message in body').notEmpty()           ①
  req.checkBody('name', 'Invalid name in body').notEmpty()
  var errors = req.validationErrors()
  if (errors) return next(errors)
  req.messages.insert(req.body, function (err, result) {                      ②
    if (err) return next(err)
    return res.json(result.ops[0])                                              ③
  })
})
① Check that message is present in the request body
② Insert the request body into the database
③ Output the ID of the new document which is autogenerated by the database
```

node-dev

As mentioned before, I recommend a tool called nodemon or something similar. node-dev is one of those alternatives. node-dev monitors for file changes and restarts the server when changes are detected. It can save you HOURS! To install node-dev simply run:

```
npm i node-dev@3.1.3 --save-dev
```

In the package.json, you can add the command node-dev . to the start npm script.

```
...
"scripts": {
  ...
  "start": "./node_modules/.bin/webpack && node-dev ."
},
...
```

The boot up is rather primitive compare the the previous section in which we used https. Obviously, the port number can be changed or taken from environment variables.

```
app.listen(3000)
```

If you remember, we have / route which handles all the GET requests to / or `http://localhost:3000/` in our case (it's implemented above and uses a template called `index` in `res.render()`) Now, let's implement the template.

16.4.3 Server-Side Layout Templates with Handlebars

It's possible to use any template engine on the server to render React HTML. For example, Handlebars is a good option because it's very similar to HTML which mean little modification when transitioning from HTML to this template engine.

```
index.hbs
<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- meta tags and CSS -->
  </head>

  <body>
    <div class="container-fluid">
      <!-- header -->
      <!-- props -->
      <!-- messageBoard -->
      <!-- footer -->
    </div>
    <script type="text/javascript" src="/js/bundle.js"></script>
  </body>
</html>
```

In the Handlebars file `index.hbs`, we use triple curly braces `{{{...}}}` to output components and props (unesaped output) such as HTML. For example, `{{{props}}}` will output a script tag `<script/>` so we can define a variable `messages` in it. The `index.hbs` code to render unescaped HTML string for props is:

```
<div>{{{props}}}</div>
```

The rest locals (data) will be outputted similarly:

```
<div id="header">{{{header}}}</div>
...
<div>{{{props}}}</div>
...
<div class="row-fluid" id="message-board" />{{{messageBoard}}}</div>
...
<div id="footer">{{{footer}}}</div>
```

This is how we can output HTML string from Header component in Handlebars:

Listing 16.8 Outputting HTML generated by React in a Handlebars (ch16/message-board/views/index.hbs)

```
...
<div class="container-fluid">
  <div class="row-fluid">
    <div class="span12">
      <div id="header">{{header}}</div>
    </div>
  </div>
...

```

What about the data? In order to get the benefit of server-side React working together with browser React, we must use the same data on browser and server when we create React elements. Let's pass the data from the server to the browser React without having to have AJAX calls. We can embed the data as a JS variable right in the HTML!

So when we are passing header, footer, messageBoard, we can add `props` in / Express route. In `index.hbs`, print the values with triple curly braces as well as include the script `js/bundle.js` which will be generated by Webpack later:

Listing 16.9 Server-side layout which renders HTML from React components (ch16/message-board/views/index.hbs)

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Message Board with React.js</title>
    <meta name="description" content="Message Board" />
    <meta name="author" content="Azat Mardan" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link type="text/css" rel="stylesheet" href="/css/bootstrap.min.css" />
    <link type="text/css" rel="stylesheet" href="/css/bootstrap-responsive.min.css" />
  </head>
  <body>
    <div class="container-fluid">
      <div class="row-fluid">
        <div class="span12">
          <div id="header">{{header}}</div> ①
        </div>
      </div>
      <div>{{props}}</div> ②
      <div class="row-fluid">
        <div class="span12">
          <div id="content">
            <div class="row-fluid" id="message-board" />{{messageBoard}}
          </div>
        </div>
      </div>
      <div class="row-fluid">
        <div class="span12">
          <div id="footer">{{footer}}</div>
        </div>
      </div>
    </div>
  </body>

```

```
<script type="text/javascript" src="/js/bundle.js"></script>
</body>
</html>
```

3

- 1 Output HTML generated from Header component
- 2 Output HTML containing `<script>` with list of messages as an array
- 3 Include browser React

We also used some Twitter Bootstrap styling in the template, but it's not essential for the project or the universal JavaScript example.

We used a few variables (a.k.a. `locals: header, messageBoard, props, footer`) in our templates which we need to provide in `render()` of an Express request handler. Just to refresh, this is `index.js` code which we already implemented and which is using the template above by calling it by name `index` which is a convention for `index.hbs`:

```
res.render('index', {
  header: ReactDOMServer.renderToString(Header()),
  footer: ReactDOMServer.renderToString(Footer()),
  messageBoard: ReactDOMServer.renderToString(MessageBoard({messages: docs})),
  props: '<script type="text/javascript">var messages='+JSON.stringify(docs)+'</script>'
})
```

The values will be generated from React components. This way we will be using the same components on the server and on the browser. Ability to easily render on the server (with Node)—that's the beauty of React.

Next, we move on to the variable like `props, header, footer, etc.`

16.4.4 Composing React Components on the Server

Okay, so we are finally doing what we did for the rest of 15 chapters which is creating React components. Isn't it good to get back to familiar once in a while? Yes. But where do they come from? These components live in `components` folder. As mentioned, the components will be used on the browser and the server that's why we are putting them in a separate folder `components` and not just creating them in `client`. (Other options for components folder names are `shared`, and `common`.)

To expose these components, each of them must have `module.exports` assigned a value of the component class or stateless function. For example, you would require React, implement the class or a function, and then export the `Header` as follows:

```
const React = require('react')      1
const Header = ()=>{             2
  return (
    <h1>Message Board</h1>
  )
}

module.exports = Header          3
```

- ➊ Although there's no mentioning of React in the code, it's used by JSX
- ➋ Declare the stateless component
- ➌ Export the stateless component

Message board will be using AJAX/XHR calls to get list of messages and post a new message. The calls are in `board.jsx`. The file will have `MessageBoard`. It's our container (smart) component so the call are in that component.

The interesting place to look at is where we are making AJAX calls in `MessageBoard`... it's in `componentDidMount()`, because this lifecycle event will **NEVER** be called on the server!

Listing 16.10 Fetching list of messages and sending a new message with Axios
 (ch16/message-board/components/board.jsx)

```
const request = require('axios')
const url = 'http://localhost:3000/messages'          ➊
const fD = ReactDOM.findDOMNode
...
class MessageBoard extends React.Component {
  constructor(ops) {
    super(ops)
    this.addMessage = this.addMessage.bind(this)
    if (this.props.messages)
      this.state = {messages: this.props.messages}
  }
  componentDidMount() {
    request.get(url, (result) => {                  ➋
      if(!result || !result.length){
        return;
      }
      this.setState({messages: result})
    })
  }
  addMessage(message) {
    let messages = this.state.messages
    request.post(url, message)                      ➌
      .then(result => result.data)
      .then((data) =>{
        if(!data){
          return console.error('Failed to save')
        }
        console.log('Saved!')
        messages.unshift(data)
        this.setState({messages: messages})
      })
  }
  render() {
    return (
      <div>
        <NewMessage messages={this.state.messages} addMessageCb={this.addMessage} /> ➍
        <MessageList messages={this.state.messages} />
      </div>
    )
  }
}
```

- 1 Create a variable for server address. It can be changed later
- 2 Make a GET request with axios and update the state on success with the list of messages
- 3 Make a POST request with axios and on success add the message to the list of messages by updating the state
- 4 Pass the method to add messages to NewMessage representational/dumb component which will create a form and event listeners

You can look up the implementation of NewMessage and MessageList in the same file ch16/message-board/components/board.jsx. I won't bore you here. They are just *representational* components with very little logic or no logic—just the description of UI in the form of JSX.

We are done with rendering our React (and layout) HTML on server. Now, let's sync up the markup with the browser React, otherwise there would be no adding of messages—no interactive browser JavaScript events!

16.4.5 Client-side React Code

If we stop our implementation at this point, there would be just static markup from the rendering of React components on the server. There would be no saving of the new messages because the `onClick` event for the POST button won't work. We need to plug in the browser React to take over where server static markup rendering has left us.

We create the `app.jsx` as a browser *only* file. It won't be executed on the server (unlike our components). This is *the place* to put `ReactDOM.render()` calls to enable browser React.

```
ReactDOM.render(<MessageBoard messages={messages}/>,
  document.getElementById('message-board')
)
```

We also need to use global `messages` as prop to `MessageBoard`. The `messages` prop value will be populated by server-side template and `{{{props}}}` data (see section on the server-side layout). In other words, the `messages` array of messages will be populated from the `index.hbs`, when in the `Express.js` route / the template get data (called local) from `props` variable.

Failure to provide the same prop `messages` to `MessageBoard` on the server and on the browser will result in browser React re-painting the entire component, because it will consider the views to be different. Under the hood, React will use `checksum` attribute to compare the data which is already in the DOM (from our server-side rendering) with what browser React comes up with. React uses `checksum` because it's quicker than doing the actual tree comparison (which could take a while).

Thus, in the `app.js` file, we need to require some front-end libraries. Then render out components in DOM.

Listing 16.11 Rendering client React components on browser (ch16/message-board/client/app.jsx)

```
const React = require('react')
const ReactDOM = require('react-dom')

const Header = require('../components/header.jsx')
const Footer = require('../components/footer.jsx')
const MessageBoard = require('../components/board.jsx')

ReactDOM.render(<Header />, document.getElementById('header'))
ReactDOM.render(<Footer />, document.getElementById('footer'))
ReactDOM.render(<MessageBoard messages={messages}/>, document.getElementById('message-board'))
```

Meh, browser code is tiny.

16.4.6 Setting up Webpack

Final step is setting up Webpack to bundle browser code into one file, manage dependencies and convert JSX code.

Firstly, we need to configure Webpack like this with the entry point as `client/app.jsx`, also have the output set to `public/js` in the project folder, and use Babel loaders. The `devtool` setting is for getting proper source code lines in Chrome DevTools (not the lines from the compiled JS code).

```
module.exports = {
  entry: './client/app.jsx',
  output: {
    path: __dirname + '/public/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  stats: {
    colors: true,
    reasons: true
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}
```

To convert JSX to JS, you can use `babel-preset-react` and specify Babel configs in `package.json`:

```
...
  "babel": {
    "presets": [
```

```

    "react"
  ],
},
...

```

The browser React dependencies in `package.json` will be dev dependencies because Webpack will bundle everything what's needed into `bundle.js`, thus we won't need them at run-time (make sure you use exact versions provided here, otherwise all the new stuff that will come out when I'm done writing this paragraph, will break the project—only half joking):

```
{
  ...
  "devDependencies": {
    "axios": "^0.13.1",
    "babel-core": "^6.10.4",
    "babel-jest": "^13.2.2",
    "babel-loader": "^6.2.4",
    "babel-preset-react": "^6.5.0",
    "node-dev": "^3.1.3",
    "webpack": "^1.13.1"
  }
}
```

Also, while you are in `package.json`, add an npm build script to it (it's optional but more convenient):

```

...
  "scripts": {
    ...
    "build": "./node_modules/.bin/webpack -w"
  },
...

```

I personally love to use `watch` for Webpack (`-w`). In the `package.json`, you can add the option `-w .` to the npm build script.

```

...
  "scripts": {
    "build": "./node_modules/.bin/webpack -w",
    ...
  },
...

```

Consequently, every time you run `npm run build`, Webpack will use Babel to convert JSX into JS and stitch all the files with their dependencies into a giant ball, which will be put in `/public/js/app.js` in this case.

Thanks to our `include` in the template `views/index.handlebars` right before ending `</body>` tag, we got the browser code working (line below is what we have in the template):

```
<script type="text/javascript" src="/js/bundle.js"></script>
```

When I run this default task with `npm run build`, I see these logs:

```
Hash: 1d4cfcb6db55f1438550
Version: webpack 1.13.1
Time: 733ms
    Asset      Size  Chunks             Chunk Names
  bundle.js    782 kB     0  [emitted]  main
bundle.js.map  918 kB     0  [emitted]  main
+ 200 hidden modules
```

That's a good sign. If you have another message or even errors, please compare your project with the code on [GitHub](#).

16.4.7 Running the App

That's it as far as rendering React.js components in Express.js apps goes. Typically, all you need (assuming you have build process and components):

1. A template that outputs locals/data unescaped
2. To hydrate data to the template and render it (components and props)
3. To include the browser React file (`ReactDOM.Render`) in the template for interactivity

Was this confusing? If so, then get the tested and working code for the project from the repository and poke around. You can remove code in `app.js` to disable browser React (no interactivity such as mouse clicks), or remove code in `index.js` to disable server React (slight delay when loading a page).

The source code is in ch16/message-board and on [GitHub](#).

To run the project, simply have MongoDB running (`$ mongod`; for more instructions see appendix F) and in the project folder run these commands:

```
$ npm install
$ npm start
```

Oh and don't forget to either have Webpack running builds in the watch mode (`npm run build`) or re-start the app every time you make a change to browser code.

Open <http://localhost:3000> in your browser and you'll see the Message Board. Nothing has changed much, except that the first load is faster because the HTML is rendered on the server.

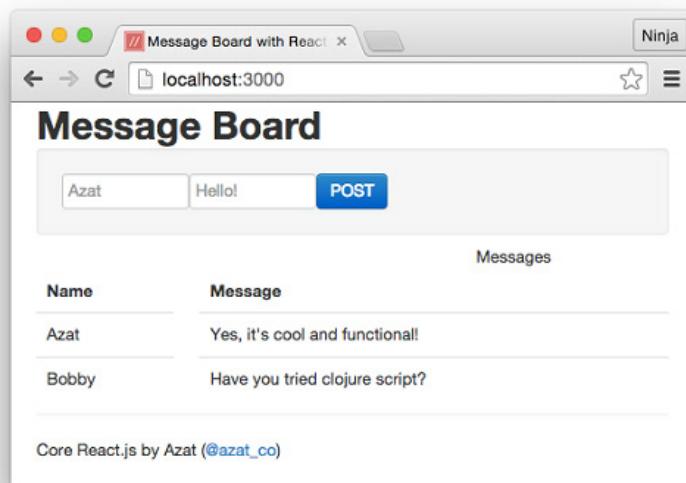


Figure 16.10 Universal app in action with server and browser rendering

You can compare timing by looking at the Network tab and localhost resource (this is server-side rendering) vs. all time after messages call. My results were ~30ms vs. ~300ms as captured in Figure 16.11.

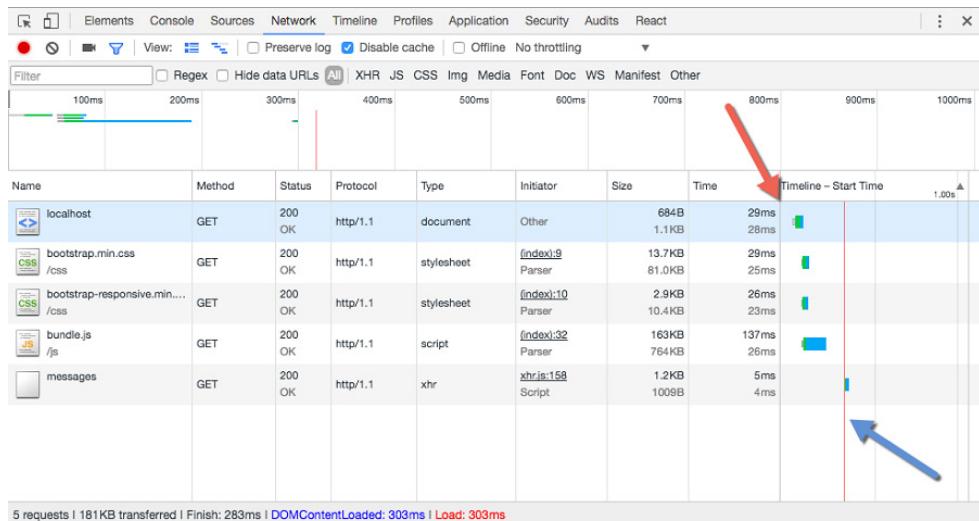


Figure 16.11 Loading of the server-side HTML is 10x faster than complete loading which is made slower by bundle and GET /messages

Of course, the bulk of total loading time is taken up by the `bundle.js`. After all it has over 200 modules! :) The GET `/messages` is not taking too long. Just a few milliseconds. With the universal approach, localhost has all the data and that load in mere 20-30ms. With browser only React, localhost has only bare-bone skeleton HTML. Users will have to wait ~10x. Everything above 150ms is usually noticeable for humans. See the results of localhost for yourself in Figure 16.12.

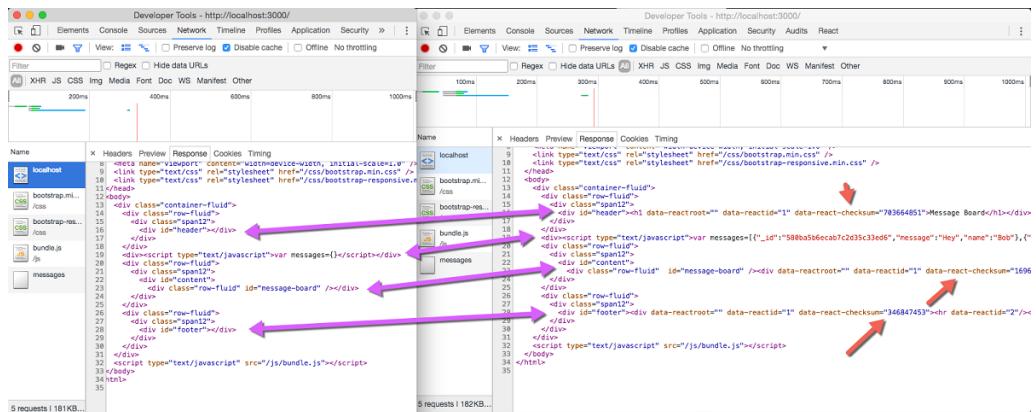


Figure 16.12 Localhost (first response) for browser only rendering on the left side vs. server-side rendering on the right side

You can play around by commenting the rendering statements in `index.js` (Express.js) or `app.jsx` (browser React). For example, if you comment out `server-side Header` but leave the browser render for `Header` intact, then there might be a flash where you don't see `Header` for a few moments and then it comes.

Also, if you comment out passing of the `props` variable on server or modify its value, then browser React will update DOM after getting the list of messages for `axios`. React will give you a warning that checksums don't match.

Universal Routing and Data

Sooner or later your application might grow and you'll need to use libraries for routing and data such as React Router and Redux which I covered in the previous chapters. Interestingly enough, these libraries already support Node and React Router even supports Express. For example, you can pass React Router routes to Express for the server-side support via `match` and `RouterContext` which will render components server-side:

```
const { renderToString } = require('react-dom/server')
const { match, RouterContext } = require ('react-router')
const routes = require('./routes')
// ...
app.get('/', (req, res) => {
  match({ routes, location: req.url }, (error, redirectLocation, renderProps) => {
    // ...
    res.status(200).send(renderToString(<RouterContext {...renderProps} />))
  })
})
```

And for Redux, there's the same `createStore()` which we can use inside of an Express middleware. For example, there's an `App` component:

```
const { createStore } = require('redux')
const { Provider } = require('react-redux')
const reducers = require('./modules')
const routes = require('./routes')

// ...
app.use((req, res) => {
  const store = createStore(reducers)          ①
  const html = renderToString(                 ②
    <Provider store={store}>
      <App/>
    </Provider>
  )
  const preloadedState = store.getState()       ③
  res.render('index', {html, preloadedState})   ④
})
```

① Create a new Redux store instance

- 2 Render the component to a string
- 3 Access the initial state from the Redux store
- 4 Render page back to the client using HTML and data

The `index` template will looks like:

```
<div id="root">${html}</div>
<script>
  window.__PRELOADED_STATE__ = ${JSON.stringify(preloadedState)}
</script>
<script src="/static/bundle.js"></script>
```

So you can see that Redux is using the same approach to the one which we used in Message Board: rendering HTML and data in a `<script>` tag.

The full usage examples with explanations are [here](#) and [here](#).

This concludes the chapter on isomorphic (or universal, as some prefer to call it) JavaScript. The uniformity and code reuse it provides are tremendous benefits. They help us, developers, to be more productive and live happier work lives!

16.5 Quiz

1. What is the method used to render a React component on the server?
2. Rendering the first page on the server improves performance. True or false?
3. CommonJS or Node.js module syntax, that's `require()` (along with Webpack) allows to "require" or import npm modules in browser code. True or false?
4. What is a symbol to output unescaped string in Handlebars? `<%...%>`, `{}{...}`, `{}{{...}}` or `dangerouslySetInnerHTML=...?`
5. Where is the best place to put AJAX/XHR calls in browser React so they won't be triggered on server?

16.6 Summary

In this chapter you learned

- To use and render React on the server you need `react-dom/server` and `renderToString()`
- The data must be the same to sync server React HTML with browser one. React uses checksums for comparison
- `renderToString()` vs. `renderToStaticMarkup()` is checksums vs. not checksums
- For universal JS to work: render React on the server, supply browser React with the same data and render browser React components
- Use triple curly braces `{}{{html}}{}}` to output unescaped HTML content in Handlebars

16.7 Quiz Answers

1. `ReactDOMServer.renderToString()`, because `renderToStaticMarkup()` won't render checksums
2. True, you get all the data on the first page load without having to wait for `bundle.js` and AJAX requests
3. True, you can use `require()` and `module.exports` syntax right out of the box with Webpack. Just by setting an entry point in the `webpack.config.js`, Webpack will traverse all the dependencies from there and include only needed ones.
4. `{{{...}}}` is the right syntax. For escaped variables use `{{data}}` to ensure safer usage.
5. `componentDidMount()` because it will never be called on the server rendering.