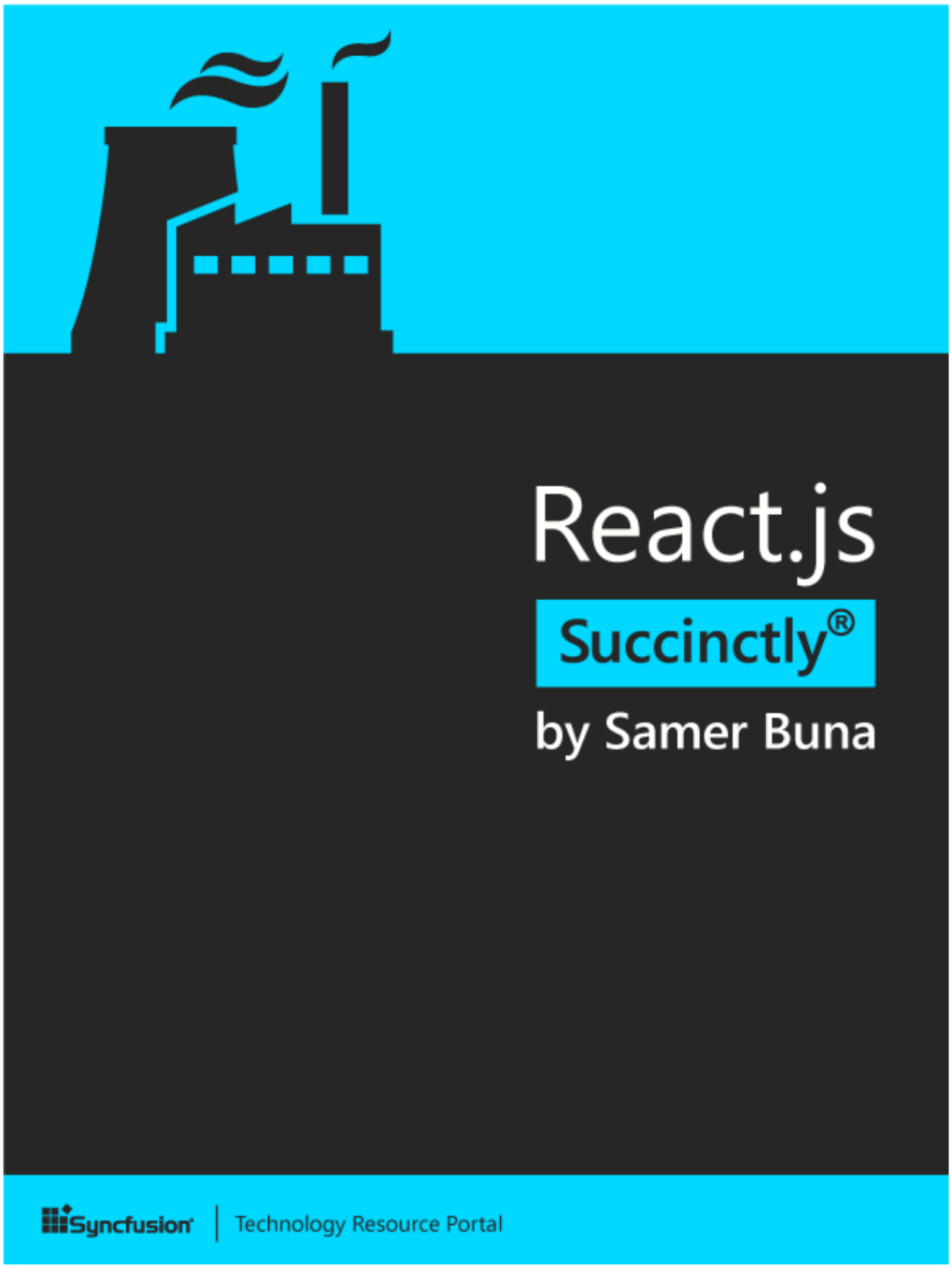
MrAlex6204 / **Books**

Branch: master ▾ **Books** / **ReactJs.pdf** Find file Copy path

MrAlex6204 ReactJs.This book will not teach you JavaScript `d297b07` on Aug 19, 2016

1 contributor

2.7 MB



React.js Succinctly

By
Samer Buna

Foreword by Daniel Jebaraj

Copyright © 2016 by Syncfusion, Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books 6

About the Author 8

Introduction 9

Chapter 1 What Is React? 11

Chapter 2 Why React? 12

Generating HTML 13

Enhancing HTML 14

React's way 15

Chapter 3 Declarative User Interfaces 19

React's language 19

To JSX or not to JSX 20

Chapter 4 React Components 22

Readability 22

Reusability 23

Composability 24

React's stateful components 25

Creating React components 26

Chapter 5 Composability 33

Chapter 6 Reusability 37

Input validation 37

Input default values 41

Shared component behavior 42

Chapter 7 Working with User Input 45

React's synthetic events 45

Working with DOM nodes in the browser 47

Controlled components 49

Chapter 8 Component Lifecycle 51

componentWillMount() 56

componentDidMount() 57

componentWillReceiveProps(nextProps) 61

shouldComponentUpdate(nextProps, nextState) 62

componentWillUpdate(nextProps, nextState) 63

componentDidUpdate(prevProps, prevState) 64

componentWillUnmount() 65

Chapter 9 Let's Build a Game with React 67

The memory grid game 67

Implementation increments 69

Challenges 101

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

It was 1993 when my geeky uncle realized my passion for computers and set me up on a path to empower that passion. I am forever grateful to him.

I was just a kid back then, with a natural love for math and physics, but when I started learning to program (using Pascal at the time), I knew I found my life-long hobby!

I am not sure I can describe coding as my *hobby*; it is certainly a career that pays the bills. However, whenever I have some free time, I often just start coding for a side project, or learn that brand-new framework which smart person X tweeted about the other day.

I don't write in Pascal anymore; my favorite server-side language has been Ruby for over a decade, and for the past few years, I have been mainly coding in JavaScript. With the modern changes that are happening to JavaScript, I am starting to favor JavaScript over Ruby.

When React was first released, I was an Ember fan. I still favor Ember over Angular today, but after learning the simple and powerful React, I left the school of MVC frameworks for good. Today, I do my front-end work exclusively with React and vanilla JavaScript.

You can stalk me on Twitter @samerbuna or email me at samer@agilelabs.com. You can also find a list of all the books and courses I have authored on my LinkedIn profile at linkedin.com/in/samerbuna.

Special thanks to my friend Julia Hunt for her excellent contributions to this book. Her feedback was spot-on as always, and this book is much better because of her.

I hope that you enjoy this book and that it will teach you the great benefits of the React library.

Samer Buna

Introduction

Since 2010 or so, Angular has been *the* MVC framework of choice for top developers. It's the most popular of all, and it offers the most impressive features out of the box. This all changed in 2015, and the change came from Facebook engineers this time. React started to claim a big portion of the market, with huge players adopting it: Netflix, Yahoo, and Airbnb, to name a few.

React is a huge revolutionary success; it's simple and effective, and it changed the way we design our views for good.

React is not an MVC framework—it's just a view library. The concepts introduced by React are what make it a big deal. Other frameworks out there, Angular included, have learned from what React does and copied it in parts or fully.

This story, however, is not about a shiny new library from Facebook that is taking over. The story here is about MVC not being the best approach for big applications, and REST APIs, which have been the standard so far, not being the best solution for data communication between clients and servers.

Facebook engineers are challenging both the MVC and the REST standards. Here are their alternative proposals, in summary:

- Instead of MVC and data-binding, a front-end system should have a one-way data flow. The views should never change the models directly; they can only read from the models. For writing, the views trigger actions that eventually get dispatched to the models (the models are referred to as "Stores" in this flow). React fits perfectly in the way it *reacts* to the data changes in the stores. React components will be listening to those changes, and React will re-render the views efficiently when those changes happen.

- Instead of REST and having the logic of what data to expose on the server-side, and instead of managing different end-points for different needs (for example, one end-point to expose top posts, one to expose a post, and one to expose a post with comments included), let the clients ask the servers for exactly what they need. The servers will then parse the clients' questions, and respond with what they asked for (no over-fetching or under-fetching). This is the concept behind the GraphQL runtime and query language that Facebook engineers released in 2015.

- Instead of the standard recommendation of separating data and views, have the data requirement expressed in the view component itself. Since the view knows exactly which data elements it needs to render, these two shouldn't be separate at all. This is the concept behind the Relay.js framework that Facebook engineers released in 2015.

This book will not cover GraphQL or Relay, but an understanding of React itself is the first step toward understanding the rest of the architecture proposed by Facebook engineers.

Why did I write this book?

I love big frameworks (and I cannot lie). They serve a great purpose, especially for young teams and startups. Lots of smart design decisions are already made for us, and there is a clear path to writing good code. This allows us to focus on our application's logic.

Frameworks, however, come with some disadvantages, and for big applications and experienced developers, those disadvantages are sometimes a deal breaker.

I'll name two of the relevant disadvantages of going with a framework:

- Frameworks are not flexible, although some will claim to be. Frameworks want us to code everything a certain way; if we try to deviate from that way, the framework usually ends up fighting us about that.

- Frameworks are big and full of features. If we need to use only a small piece of them, we have to include the whole thing anyway (hopefully this will change with HTTP2).

Some frameworks are going modular, which I think is great, but I am a big fan of the Unix philosophy:

"*Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.*" —Doug McIlroy

React is a small program that does one thing, and it does it really well. There is a lot of buzz around the performance of React's virtual DOM, but I don't think that's the greatest thing about React. The virtual DOM performance is a nice plus.

What React does really well is speak a common language between developers and browsers that allows developers to declaratively describe user interfaces, and to model the state of these interfaces and not the transactions on them. Basically, React taught us and the machines a new language that we both now understand: the language of UI outcomes.

This realization was the most important one I made about React, and I think it deserves a book focused on making it clear to everyone learning React.

Who should read this book?

You need basic knowledge of JavaScript to survive this book. This book will not teach you JavaScript. If you're comfortable with JavaScript itself, but have never used a JavaScript framework or library before, this book is for you.

If you're learning React after using other JavaScript libraries, this book will also have an answer to the "Why?" question that's probably on your mind: Why bother learning something new?

Chapter 1 What Is React?

React is a JavaScript library that can be used to describe *views* (for example, HTML elements) based on some *state* (which is often in the form of data). When you're working with React in a browser, React can mount the described views in the browser's DOM (Document Object Model) and automatically update what needs to be updated whenever the original state changes.

React is a small but powerful library, with the power being more in the concepts than in the implementation. Some of the concepts under which React operates are:

Reusable, composable, and stateful components

In React, we build views using smaller components. We can reuse a single component in multiple places, with different states and properties, and components can contain other components. Every component in a React application has a private state that may change over time, and React will take care of updating the component's view when its state changes.

The nature of reactive updates

React's name is the simple explanation for this concept. When the state of a component changes, those changes need to be reflected somewhere. For example, we need to regenerate the HTML views for the browser's Document Object Model (DOM) whenever their state

the HTML views for the browser's Document Object Model (DOM) whenever their state changes. With React, we do not need to worry about how to reflect the state changes; React will simply react to the changes and automatically update the views when needed.

The virtual representation of views in memory

With React, we write HTML using JavaScript. We rely on the power of JavaScript to generate HTML that depends on some data, rather than enhancing HTML to make it work with that data. Enhancing HTML is what other JavaScript frameworks usually do. For example, Angular extends HTML with features like loops, conditionals, and others.

If we are receiving just the data from the server (with AJAX), we need something more than HTML to work with it, so it's either using an enhanced HTML, or using the power of JavaScript itself to generate the HTML. Both approaches have advantages and disadvantages, and React embraces the latter one, with the argument that the advantages are stronger than the disadvantages.

Using JavaScript to render HTML allows React to have a virtual representation of HTML in memory (which is aptly named the virtual DOM), and React uses that to render the views virtually first. Every time a state changes and we have a new HTML tree that needs to be written back to the browser's DOM, instead of writing the whole tree, React will only write the difference between the new tree and the previous tree since it has both trees in memory. This process is known as *tree reconciliation*, and I think it's the best thing that's happened in web development since AJAX!

Chapter 2 Why React?

HTML is a great language, and what makes it great, in my opinion, is how it allows us to be declarative when building our webpages. We basically tell the browsers what we want with HTML.

For example, when we tell the browser to render a paragraph of text in a page, we don't care how the browser is actually going to do that. We don't have to worry about what steps the browser needs to do to make our text look like a paragraph, such as how many pixels of white space should be around it, how many words to display per line, or what to do when the screen is resized. We just want a paragraph, and with HTML we have the power to command the browser to make it happen.

But HTML by itself is not enough. We don't have loops, if statements, or variables in HTML, which means we can't write dynamic HTML based on data. HTML works great for static content, but when we have data in the equation, we need something else.

The simple truth is data is *always* in the equation. We write HTML to represent some data, even when we use HTML for static content. When we write static content, we are manually writing HTML by hand to represent some data, and that's okay for small sites where the data is not frequently changed.

The word "change" is key here. When the data we represent with HTML is changed, we need to manually update the HTML to reflect that change. For example, when that small business for which you built a website gets a new phone number, it'll be time to drop into an HTML editor, make that change, and then upload the new HTML files to the server.

If the nature of the change is more frequent though, for example, if the business now wants to feature a different product on the home page every hour, you will not agree (I hope) to make that manual change every hour. Imagine if Facebook was all static content, and every time you update your status someone at Facebook needs to edit an HTML file and upload it to the server.

We realized a long time ago that HTML is not enough, and we have since been trying to enhance HTML to make it support dynamic content.

Let's assume that we want to represent a list of products in HTML. For every product, we have a name and a price. Here's some sample data:

Code Listing 1: Sample Products Data

```
{
  "products": [
    { "name": "React.js Essentials", "price": 2999 },
    { "name": "Pro React", "price": 2856 },
    { "name": "Learning React Native", "price": 2199 }
  ]
}
```


A small business mentality would reason that we have only these three products, and their prices rarely change. Manually writing some static HTML for these three products would be okay.

But a list of products is dynamic—some products will be added, some will be removed, and the prices will change seasonally. Manually writing HTML will not scale, so we need to come up with HTML that always represents the last *state* of the products' data.

Generating HTML

We can use database tables (or collections) to represent data in a database. For our products example, we can use a products table.

Code Listing 2: Products Table

```
create table products (
  name text,
  price integer
)
```

The rows of this table each represent a single product. We can then give the client write access on the table to allow them to add, remove, and update their list of products whenever they want.

To come up with HTML to represent these products, we can read the last state of the data from this table and process it in some programming language—like Java—to make the program concatenate together HTML strings based on the data.

Code Listing 3: HTML Strings

```
html = "<html><body><ul>";

// Loop over rows in table, and for every row:
html += "<li>" + name + " - " + price + "</li>";

// When the loop is done:
html += "</ul></body></html>";
```

We can now write the value of that **`html`** variable to an HTML file and ship it. Every time the customer updates their data table, we can regenerate and ship the new HTML.

This worked great, and it gave us a lot of power to work with dynamic data, but something was wrong: it didn't feel right.

We wanted to write HTML, not concatenate strings.

Enhancing HTML

A genius mind came up with an idea to create a new language that is somewhere between HTML and programming languages like Java. A language where we can still write normal HTML tags, but also use special tags that will act like loops and conditionals when we need them. JSP and PHP are examples of this.

Here's how we can represent our product data using an enhanced HTML language:

Code Listing 4: HTML+ (Pseudocode)

```
<html><body><ul>
  <% FOR each product in the list of products %>
    <li><%= the product's name %> - <%= the product's price %></li>
  <% END FOR %>
</ul></body></html>
```

This is clearly a bit more pleasant to work with than concatenating strings. The compiler will take care of doing the string concatenations for us, and we will get HTML that always reflects our exact data.

This has been the standard way to create dynamic websites for years. In fact, until recently, Facebook did exactly that to show you the list of updates from your friends.

But just like everything else in life, browsers evolved and became smarter. JavaScript became a standard language that is supported in all browsers, and someone invented AJAX, which allowed us to ask servers questions in the background of a webpage.

We realized that we could be more efficient in the way we represent our data. Instead of preparing the HTML server-side and shipping it ready to the client, we can send just the data to the client, and we'll have the fancy, smart browsers prepare the HTML themselves. JavaScript can do that.

Since we didn't want to concatenate strings, we needed something like JSP and PHP, but something that would work with the JavaScript engine in the browser.

Frameworks like Angular, Ember, and a handful of others were developed to make this process easier.

Just like server-side enhanced HTML languages, with Angular we do:

Code Listing 5: Angular Loop

```
<html><body><ul>
  <li *ngFor="#product of products">
    {{product.name}} - {{product.price}}
  </li>
</ul></body></html>
```

Now we're being more efficient over the wire, and we are generating dynamic HTML without having to deal with concatenating strings.

This worked great, but as we used it to build big applications, a few issues came up:

- Performance: Since we are no longer hitting the refresh button every time we click a link, our use of memory should now be managed more carefully.
- Browsers are great for rendering an HTML tree of elements dynamically, but they are still slow, especially when we need to update the tree. Every time an update is needed, there are tree traversals that need to happen, and when we reach the node and update it, the browser needs to redraw the user screen, which is an expensive process.
- In some cases, we were not modeling the state of our data directly with user interfaces, and instead we were writing code that expressed the steps and the transitions needed to update those interfaces. This is not ideal.

React's way

The engineers at Facebook, realizing the problems of all other JavaScript frameworks, wanted something better, so they went back to the drawing board. A few months later, they came up with what eventually became React.

They had a couple simple rules to guide their genius solution:

- Enhancements to HTML are noise—let's just concatenate strings in JavaScript itself, and yes, sacrifice some readability for performance.
- The DOM is a bottleneck; we should avoid it as much as possible.

HTML views are hierarchical. An HTML document is a big tree of nodes. With the power and flexibility of JavaScript, Facebook engineers realized that if HTML nodes were represented as JavaScript objects, we would have a new world of possibilities for working with them.

Having the power and flexibility of JavaScript, Facebook engineers realized that if HTML nodes were represented as JavaScript objects, we would have a new world of possibilities to work with them.

That led to their first major decision: *Represent every HTML element with a JavaScript object.*

Note: If you want to test the React code samples in this book, the easiest way to get a React environment for testing is to use one of the web development online playground applications. My favorite is JS Bin; here's a React-ready bin ***that you can fork to start testing.***

React's API today has the **`createElement`** function, which can be used to do exactly that.

Code Listing 6: Void Element

```
// Instead of:
```

```
// Instead of:
<img src="logo.png" />

// We do:
React.createElement("img", { src: "logo.png" })
```

The **`createElement`** function returns a JavaScript object. To inspect that object, go to the main React website and paste the **`createElement`** line in the developer tools JavaScript console:

```
> React.createElement("img", { src: "logo.png" })
<· ▶Object {$$typeof: Symbol(react.element), type: "
  img", key: null, ref: null, props: Object…}
>
```

*Figure 1: **`createElement`** in JS Console*

Notice the **`type`** property for that object, and take a look at the **`props`** property as well.

`<img />`, however, is a void (self-closing) element; let's take a look at another example for a normal element.

Code Listing 7: Normal Element

```
// Instead of:
<a href="/">Home</a>

// We do:
React.createElement("a", { href: "/" }, "Home");
```

To represent a parent-child relation, we can use the arguments of the **`React.createElement`** function. We can pass one or more React objects, starting from the third argument of **`React.createElement`**.

For example, to make the logo image clickable:

Code Listing 8: Parent-Child Relation

```
// Instead of:
<a href="/">
  <img src="logo.png" />
</a>

// We do:
React.createElement("a", { href: "/" },
```



```
  React.createElement("img", { src: "logo.png" })
);
```

If we inspect this last object in the console, we'll notice how the **`props`** attribute for the **`<a>`** element has a **`children`** attribute, and that in turn holds the object for the **`<img />`** element. This is how we represent a tree with React.

Of course, under the hood, React will eventually generate HTML strings from these JavaScript objects and concatenate them together.

There are big benefits to writing HTML with JavaScript:

- We have the power of JavaScript itself to work with data. Our product example becomes:

Code Listing 9: The Power of JavaScript

```
React.createElement("ul", {},
  ...products.map(product =>
    React.createElement("li", {}, `${product.name} - ${product.price}`)
  )
);
```

- It's all JavaScript, so we don't mix HTML with new tags or anything. We also don't have to learn a new language, and instead learn how to use the few JavaScript functions that React defines.

- We'll always have a representation of our HTML in memory, separate from the one in the browser. This means that we can optimize our DOM operations by using a smart diffing algorithm on the structures we have in memory. This is only relevant when things change. This concept is what we call *the virtual DOM* in React.

The virtual DOM

After rendering HTML for the product data example that we started with, we'll have three products with their names and prices displayed in the browser. With an **`Ajax`** call, we've determined that the price for product #2 has changed, and we now have a fourth product to display.

React has the original user interface state for three products stored in memory (because we wrote it in JavaScript first). Before sending anything to the browser, React will compute the new state, which now has four products.

With both states available in memory now, React can compute the difference between the new state and the original state, and it can conclude that it needs to add a node #4 and modify node #2. All other nodes will remain unchanged.

Having the original state in memory represents a big performance benefit. Other frameworks that don't have an original state in memory have two options:

- Generate the new state's UI and replace the whole tree in the browser.

- Try to change nodes #2 and #4, but rely on what is in the DOM. This results in making a bunch of READ operations in the DOM, and processing the tree to figure out where node #2 begins and where node #3 ends. These DOM operations are usually expensive.

React's virtual representation of the browser's DOM in memory (known as the virtual DOM) is clearly a better alternative. In fact, it's so good that most other JavaScript frameworks today have changed their algorithms to do exactly what React does for updating the DOM.

React's power is more than a virtual DOM with a smart diffing algorithm. Views in React have private state, so our product list view can have its original three products object represented in that private state, and when the new data comes in, we just update the private state. React triggers a DOM reload whenever the private state of the view changes. It basically hits a special refresh button in the browser.

From the point of view of a user inspecting the state, the view has been completely refreshed, and the new view represents the new state. But under the hood, React is doing this refresh operation smartly by using the diff it computed via the virtual DOM, and only applying that diff to the browser's tree. As developers, however, we don't have to worry about that; we just declare our state changes and let React take care of the steps needed to get these changes efficiently

our state changes and let React take care of the steps needed to get those changes efficiently reflected in the browser's DOM.

This is so important to understand that it's worth repeating: With React, we model the state of our user interfaces, and not the transitions on them. If there are changes, we just model the new state.

React has effectively taught both the browsers and us developers to speak a new language: the language of user interface outcomes. We can now communicate with our browsers in this higher-level language that allows us to express our user interfaces in a declarative way.

Using the React language for our products example, we're basically saying, “Browser, I have three products for you.” Then a minute later, we come back and say, “Browser, I now have four products for you.” We don't have to worry about the changes in the prices or names, since this is not part of the language; it's a transition, and we just model the new state: four products. In fact, even the number of products does not matter because of the reactive update nature of React, which will take care of re-rendering the views when that number changes. Our command to the browser ends up being, “Browser, we have a list of products for you, and changes might happen to that list.”

Chapter 3 Declarative User Interfaces

React’s language

With React's new language of outcomes, which browsers now understand, we can build declarative user interfaces (like we used to do with pure HTML), but now we can work with data as well.

For our products example, we just command the browser to display a list of products; we don’t even worry about how many products are actually on that list.

Our command to the browser is this: Display this list of products as an unordered list; for every product, display its name, then display its price.

After that, we’re done. When the data we have for the list of products changes, we don’t need to do anything to our UI to get it updated with the new data. Our command to the browser does not need to be changed.

In other words, if we have transitions on the data, we don't have to worry about the user interface—we just manage the new state of our data.

Data transitions include operations like the following:

- One product was removed from the catalog.
- Three more products were added to the catalog.
- A product name has changed.
- Five products' prices have changed.

- The order of the products in the catalog has changed.

Of course, we have to change our views if the *structure* of our data changes. For example, when we add a **`Boolean`** flag to every product to track whether the product is in stock or not, and we want to display the out-of-stock products in gray, this would be a structure transition, not just a data transition. Once we account for the new structure for the first time though, updating that new flag becomes just a data transition that we don't have to worry about in the UI.

This mental model about modeling the final state is much easier to understand and work with, especially when our views have lots of data transitions.

For example, consider the view that tells you how many of your friends are online. The view state will be just one single number of how many of friends are currently online. This view does not care that a moment ago, three friends came online, then one of them disconnected, then two more joined. It just knows that at this current moment, four friends are online.

To JSX or not to JSX

Writing HTML in JavaScript gives us a lot of power and advantages over using templates and display logic. But working with HTML directly has great advantages that we would sacrifice in this case. HTML is familiar and concise, and if you work with designers, they would certainly hate to work with our JavaScript-created elements.

JSX is the optional compromise. It is a simple HTML-like JavaScript syntax extension used to create React elements. You can think of JSX as an enhancement of JavaScript to allow for syntax that looks like HTML. We can use it in the return statement of a component's render function.

JSX is labeled "HTML-like" because it can't be exactly HTML. Some element attributes have to be used the way the DOM API defines them; **`class`** and **`for`** are both examples of this. To use them in React, we use **`className`** and **`htmlFor`** instead.

The following example is an email input field interface.

Code Listing 10: JSX

```
// The desired HTML:
<form>
  <label for="email">Email:</label>
  <input type="email" id="email" class="form-control" />
</form>

// How to represent it with React:
React.createElement(
  "form",
  null,
  React.createElement(
    "label",
    { htmlFor: "email" },
    "Email:"
  ),
  React.createElement(
    "input",
    { type: "email", id: "email", className: "form-control" }
  )
);

// How to represent it with JSX:
<form>
```

```
  <label htmlFor="email">Email:</label>
  <input type="email" id="email" className="form-control" />
</form>
```

It’s important to understand that JSX is completely optional, and not required to use React. We can write React in plain JavaScript; we don’t need the JSX extension.

JSX is actually not shipped with React at all; we will need third-party tools to make our project work with JSX. The Facebook-recommended tool is Babel.

Writing components render functions in JSX instead of pure JavaScript has the advantage of being concise and familiar, and that’s especially true for designers. Balanced opening and closing tags are much easier to read and parse with the human eye than JavaScript functions and object literals.

Chapter 4 React Components

User interfaces are defined as components in React. The term *component* is used by many other frameworks. We can also write web components natively using HTML5 features like custom elements and HTML imports.

Components have many advantages, and whether we are working with them natively using HTML5, or using a library like React, we get the following great benefits.

Readability

Consider this UI:

Code Listing 11: HTML-Based UI

```
<a href=”http://facebook.com”>
  <img src=”facebook.png” />
</a>
```

What does this UI represent? If you speak HTML, you can parse it quickly here and say, “it’s a clickable image.” If we’re to convert this UI into a component, maybe **`ClickableImage`** is a good name for it.

Code Listing 12: Component-Based UI

```
<ClickableImage />
```

When things get more complex, this parsing of HTML becomes harder, so components allow us to know quickly what the UI represents using the language that we’re comfortable with (English in this case).

Here’s a bigger example:

Code Listing 13: Component-Based UI

```
<TweetBox>
   <TextAreaWithLimit limit={140} />
   <RemainingCharacters />
   <TweetButton />
</TweetBox>
```

Without looking at the actual HTML code, we know exactly what this UI represents. Furthermore, if we need to modify the output of the remaining characters section, we know exactly where to go.

Reusability

Think of components as functions. This analogy is actually very close to the truth, especially in React. Functions take input, they do something (possibly on that input), and then give us back an output.

In functional programming, we also have *pure* functions, which are basically protected against any outside state; if we give them the same input, we'll always get the same output.

In React, a component is modeled after a function. Every component has private properties, which act like the input to that function, and we have a virtual DOM output. If the component does not depend on anything outside of its definition (for example, if it does not use a global variable), then we label that component pure as well.

All React components can be reused in the same application and across multiple applications. Pure components, however, have a better chance at being reused without any problems.

For an example, let's implement our **`ClickableImage`** component.

*Code Listing 14: **`ClickableImage`** Render Function*

```
var ClickableImage = function(props) {
  return (
    <a href={props.href}>
      <img src={props.src} />
    </a>
  );
};

ReactDOM.render(
  <ClickableImage href="http://google.com" src="http://goo.gl/QlB7wl" />,
  document.getElementById("react")
);
```

Having variables for both the **`href`** and the **`src`** properties is what makes this component reusable.

Note how we defined the component as an actual function. We can create React components in multiple ways. The simplest way is to use a normal JavaScript function that receives the component's **`props`** as an argument.

The function’s output is the virtual HTML view, which this component represents.

Don't worry about the syntax now—just focus on the concepts. To reuse this component, we could do something like render **`ClickableImage`** with a Google logo:

```
props = { href: "http://google.com", src: "google.png" }
```

We can simply reuse the same component with different props:


```
props = { href: "http://bing.com", src: "bing.png" }
```

The **`src`** properties should be replaced with actual images, as we did in the render function of the previous example.

Composability

We create components to represent views. For ReactDOM, the React components we define will represent HTML DOM nodes.

The **`ClickableImage`** component in the last example was composed of two HTML elements. We can think of HTML elements as *built-in* components in the browser. We can also use our own custom components to compose bigger ones. For example, let's write a component that displays a list of search engines.

*Code Listing 15: **`SearchEngines`** Mockup*

```
var SearchEngines = function(props) {
  return (
    <div className="search-engines">
      <ClickableImage href="http://google.com" src="google.png" />
      <ClickableImage href="http://bing.com" src="bing.png" />
    </div>
  );
}
```

If, for example, we have the data in this format:

*Code Listing 16: **`SearchEngines`** Data*

```
var data = [
  { href: "http://google.com", src: "google.png" },
  { href: "http://bing.com", src: "bing.png" },
  { href: "http://yahoo.com", src: "yahoo.png" }
];
```

Then, to make **`<SearchEngines data={data} />`** work, we just map the data array from a list of objects to a list of **`ClickableImage`** components:

*Code Listing 17: **`SearchEngines`** Render Function*

```
var SearchEngines = function(props) {
  return (
    <List>
      {props.data.map(engine => <ClickableImage {...engine} />)}
    </List>
```



```
  );
};

...

ReactDOM.render(
  <SearchEngines data={data} />,
  document.getElementById("react")
);
```

The three dots in **`...engine`** mean spread the attribute of an engine as flat properties for **`ClickableImage`**, which is equivalent to doing:

```
href={engine.href} src={engine.src}
```

This **`SearchEngines`** component is now reusable. It can work with any list of search engines we give to it. We also used the **`ClickableImage`** component to *compose* the **`SearchEngines`** component.

React's stateful components

Everything changes eventually. In React, a component manages its changes using a *state* object. In addition to the data we pass to components as props, React components can also have a private state, which can change over time.

For example, a timer component can store its current timer value in its state.

Code Listing 18: Timer Component

```
<Timer initialSeconds={42} />
```

This timer will start at 42 seconds and count down to 0, decrementing its state every second. At second 0, its private state will be 42; at second 1, the private state will be 41; and at second 42, the private state will be 0.

With React, we just make the timer component display its private state.

Code Listing 19: Timer Render Function

```
function() {
  return (
    <div>
      {this.state.counter}
    </div>
  )
```



```
}
```

Every second, we decrement the counter state.

Code Listing 20: Changing the State (Pseudocode)

```
Every second:
  state.counter = state.counter - 1
  if state.counter reaches 0
    stop the timer
```

Here's the great news: React components recognize the private state changes. When changes happen, React automatically hits the Refresh button for us and re-renders the UI of a component. This is where React gets its name—it will react to the state changes, and reflect them in the UI.

Creating React components

It's time to officially learn how to create React components.

Let's define our **`ClickableImage`** component, which understands two input properties, **`href`** and **`src`**. Once we have this **`ClickableImage`** component ready, we can mount it in the browser using the **`ReactDOM`** render function.

Code Listing 21: Rendering a React Component to the DOM

```
ReactDOM.render(
  <ClickableImage href="google.com", src="google.com" />,
  document.getElementById("react")
```

```
);
```

Note: ReactDOM is a library that's maintained separately and can be used with React to work with a browser's DOM. To be able to use it, you need to include its CDN entry or import it in your project. This JSBin template has a working example of a component mounted with ReactDOM.

The first argument to the **`ReactDOM.render`** method is the React element that needs to be rendered, and the second argument is where to render that element in the browser. In this case, we're rendering it to the HTML node with **`id="react"`**.

There are three main ways to define a React component:

- Stateless function components
- React.createClass

- React.Component

Stateless function components

Since components are modeled after functions, we can use a vanilla JavaScript function to write pure components:

Code Listing 22: Stateless Function Component

```
var ClickableImage = function(props) {
  return (
    <a href={props.href}>
      <img src={props.src} />
    </a>
  );
};
```

When we use a function component, we're not creating an instance from a component *class*; rather, the function itself represents what would be the render method in a regular component definition. If we design our application in a functional and declarative way, most of our components could just be simple stateless function components.

Stateless function components can't have any internal state, they don't expose any lifecycle methods, and we can't attach any refs to them. If we need any of these features (which I will explain in later chapters), we will need a class-based component definition.

With the new ES2015 arrow function syntax, the **`ClickableImage`** component can be defined more concisely with:

Code Listing 23: Stateless Function Component with Arrow Function

```
var ClickableImage = props => (
  <a href={props.href}>
    <img src={props.src} />
  </a>
);
```

React.createClass

React has an official API to define a stateful component. Our simple **`ClickableImage`**

React has an official API to define a stateful component. Our simple **`ClickableImage`** component would be:

*Code Listing 24: **`React.createClass`** Syntax*

```
var ClickableImage = React.createClass({
  render: function() {
    return (
```



```
      <a href={this.props.href}>
        <img src={this.props.src} />
      </a>
    );
  }
});
```

The **`createClass`** function takes a single argument, a JavaScript configuration object. That object requires one property, the **`render`** property, which is where we define the component's function that describes its UI.

Note how with **`createClass`**, we don't pass the **`props`** object to the **`render`** call. Instead, elements created from this component class can access their properties using **`this.props`** within the **`render`** function.

The **`this`** keyword references the *instance* of the component that we mounted in the DOM (using **`ReactDOM.render`**). Every time we mount a **`<ClickableImage />`** element, we're creating an instance of the **`ClickableImage`** component class.

In object-oriented programming terms, **`ClickableImage`** is the class, and **`<ClickableImage />`** is the object instantiated from that class.

With **`createClass`**, we can use the private state of the component object, and we can invoke custom behavior in its lifecycle methods. To demonstrate both concepts, let's implement a **`Timer`** component.

First, here's how we're going to use this **`Timer`** component:

Code Listing 25: Rendering the Timer Component

```
ReactDOM.render(
  <Timer initialSeconds={42} />,
  document.getElementById("react")
);
```

The simple definition of this component, before considering the private state or the tick operation, is:

*Code Listing 26: Timer Component **`render()`** Function*

```
var Timer = React.createClass({
  render: function() {
    return (
      <div>{this.state.counter}</div>
    );
  }
});
```

The **`counter`** variable is part of the private state within a component instance. The private state object can be accessed using **`this.state`**.

Our **`Timer`** component has the property **`initialSeconds`**, which is where the counter should start. Properties of a component instance can be accessed using **`this.props`**, so if we need to read the value that we're passing to the **`initialSeconds`** property, we use **`this.props.initialSeconds`**.

The state of a React component can be initialized using a **`getInitialState`** function in the **`createClass`** definition object. Anything we return from the **`getInitialState`** function will be used as the initial private state for a component instance.

We want the initial state for our counter variable to start as **`this.props.initialSeconds`**, so we do the following:

Code Listing 27: Timer Component Initial State

```
var Timer = React.createClass({
  getInitialState: function() {
    return {
      counter: this.props.initialSeconds
    };
  },
  render: function() {
    return (
      <div>{this.state.counter}</div>
    );
  }
});
```

Let's now define the “tick” operation. We can use a vanilla **`setInterval`** function to tick every second (1000 milliseconds). Inside the interval function, we need to change our component state and decrement the counter.

The ticking operation should start after the component gets rendered to the DOM so that we’re sure there is a **`div`** in the DOM whose content we can now control. For that, we need a lifecycle method.

Lifecycle methods act like hooks for us to define custom behavior at certain points in the lifecycle of a React component instance. The one we need here is **`componentDidMount()`**, and it allows us to define a custom behavior right after the component is mounted in the DOM.

*Code Listing 28: Timer Interval in **`componentDidMount()`***

```
var Timer = React.createClass({
  getInitialState: function() {
    return { counter: this.props.initialSeconds };
  },
  componentDidMount: function() {
```

```
    var component = this;
    setInterval(function() {
      component.setState({
        counter: component.state.counter - 1
      });
    }, 1000);
  },
  render: function() {
    return <div>{this.state.counter}</div>;
  }
});
```

There are a couple of things to notice here:

- We had to use a closure around the **`setInterval`** so that we have access to the **`this`** keyword in there. It's an old-school JavaScript trick (which we don't need anymore with the new ES2015 arrow function syntax).
- To change the state of the component, we used the **`setState`** function. In React, we should never mutate the state directly as a variable. All changes to the state should be done using **`setState`**.

This **`Timer`** component is ready, except that the timer will not stop, and it will keep going to the negative side. We can use the **`clearTimeout`** function to stop the timer. Go ahead and try to do that for our component, and come back to see the following full solution.

Code Listing 29: Timer Component Full Definition

```
var Timer = React.createClass({
  getInitialState: function() {
    return { counter: this.props.initialSeconds };
  },
  componentDidMount: function() {
    var component = this, currentCounter;
    component.timerId = setInterval(function() {
      currentCounter = component.state.counter;
      if (currentCounter === 1) {
        clearInterval(component.timerId);
      }
      component.setState({ counter: currentCounter - 1 });
    }, 1000);
  },
  render: function() {
    return <div>{this.state.counter}</div>;
  }
});

ReactDOM.render(
```



```
  <Timer initialSeconds={42} />,
  document.getElementById("react")
);
```

React.Component

ES2015 was finalized in 2015, and with it, we can now use the class syntax. A class is syntax sugar for JavaScript's constructor functions, and classes can inherit from each other using the extends keyword.

extends keyword.

Code Listing 30: ES2015 Class Syntax

```
class Student extends Person { }
```

With this line, we define a new **Student** class that inherits from a **Person** class.

The React API has a class that we can extend to define a React component. Our **ClickableImage** definition becomes:

*Code Listing 31: **React.Component** Syntax*

```
class ClickableImage extends React.Component {
  render() {
    return (
      <a href={this.props.href}>
        <img src={this.props.src} />
      </a>
    );
  }
}
```

Within the class definition, the **render** function is basically the same, except that we used a new ES2015 syntax to define it. The word **function** can be completely avoided in ES2015.

Let's look at our **Timer** example using the ES2015 syntax. Try to identify the differences.

Code Listing 32: Timer Component Using Class Syntax

```
class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = { counter: this.props.initialSeconds };
  }
  componentDidMount() {
    let currentCounter;
    this.timerId = setInterval(() => {
```



```
      currentCounter = this.state.counter;
      if (currentCounter === 1) {
        clearInterval(this.timerId);
      }
      this.setState({ counter: currentCounter - 1 });
    }, 1000);
  }
  render() {
    return (
      <div>{this.state.counter}</div>
    );
  }
}
```

Here are the differences explained:

- Instead of **getInitialState**, we now use the ES2015 class constructor function, and just assign a value to **this.state** in the constructor. We need to invoke the **super()** call there as well to fire the **React.Component** constructor call.

- In the **`setInterval`**, we used an arrow function, i.e. **`() => { }`**. With an arrow function, we don't need to do the closure trick we did before because arrow functions have a lexically bound "**`this`**" that captures the enclosing context by default.

- All the functions are defined using the new function property syntax. For example, **`componentDidMount() { ... }`**.

Component classes, elements, and instances

Sometimes you'll find these terms mixed up in guides and tutorials. It's important to understand that we have three different things here:

- What's usually referred to as "Component" is the class. The blueprint. The global definition. In the **`Timer`** example, the variable **`Timer`** itself is the component class.

- **`<Timer />`** on the other hand, is a React element that we constructed from the **`Timer`** component class. This is a stateless, immutable virtual DOM object.

- When a React element is mounted in the browser's DOM, it becomes a component instance, which is stateful. The result of a **`ReactDOM.render`** call is a component instance.

Chapter 5 Composability

Several React components can be combined to produce another React component. This is one of the best features of React. It's a simple concept with great advantages.

Composability enables abstraction and allows us to understand code without having to care about all the details all the time. If a **`Profile`** component is composed of a **`ProfilePicture`** component and a **`ContactInformation`** component, we'll have a pretty good idea about what's going on there without looking at the details of each component.

Composability also enables a more uniform behavior. If we have a **`ContactInformation`** component, whenever we need to display a person's contact information, whether that person is a guest, client, vendor, or employee of your business, we can uniformly use the **`ContactInformation`** component. This also means less repetition of code in any UI that represents a contact information section.

Every time we reuse a component to write another, we are cashing out our original time investment that we put into creating the original one.

The most important benefit of composability, however, is that it allows us to separate the different concerns of our applications with great flexibility.

Let me explain composability with an example. Let's try to describe Twitter's account page with components.

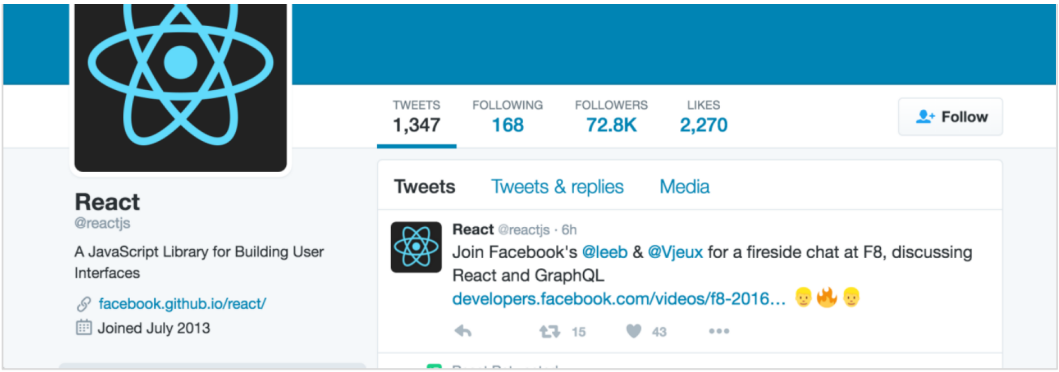


Figure 2: Twitter's User Profile Page

A simplified component list for this page could be something like the following (omitting the details of every component for brevity):

Code Listing 33: Twitter ***`UserHomePage`***

```
class Avatar extends React.Component {
  render() {
    return <img ... />;
  }
}

class UserInfo extends React.Component {
  render() {
    // name, bio, ...
  }
}

class Counts extends React.Component {
  render() {
    // tweets, following, ...
  }
}

class Tweets extends React.Component {
  render() {
    // The list of tweets.
  }
}

class UserHomePage extends React.Component {
  render() {
    return (
      <div>
        <Avatar src={} />
        <UserInfo user={} />
        <Counts tweets={} following={} ... />
        <Tweets list={} />
      </div>
    );
  }
}
```

Understanding the hierarchical relationships between components here is important. Since **`Avatar`**, **`UserInfo`**, **`Counts`**, and **`Tweets`** all appear in the **`render`** method for **`UserHomePage`**

Avatar, **UserInfo**, **Counts**, and **Tweets** all appear in the **render** method for **UserHomePage**, this makes **UserHomePage** the owner of them.

An owner component can set the **props** of the components it uses in the **render** method. For example, the **src** property for the **Avatar** component was passed to it from its owner, **UserHomePage**.

Component children

Just like we can include HTML elements within the opening and closing tags of another HTML element, we can also include React elements within the opening and closing tags of another React element. We can mix HTML elements and React elements in both cases.

Code Listing 34: Children Example

```
<Counts>
  <TweetsCount />
  <FollowingCount />
  <FollowersCount />
  <LikesCount />
</Counts>
```

The inner elements (**TweetsCount**, **FollowingCount**, etc.) are called the “children" of the outer element (**Counts**). Within the definition of the **Counts** component, we can access the list of children used within any instance of the component using the special **this.props.children** property.

*Code Listing 35: **this.props.children***

```
class Counts extends React.Component {
  render() {
    <div id="counts-headers">
      {this.props.children}
    </div>
  }
}
```

React’s tree reconciliation process uses the order of the children in the diffing algorithm. This unfortunately means that if a child is removed from the tree, all siblings after it will need to be updated in the process. Be aware of this problem.

This problem becomes a serious one when the children are driven by a dynamic array that can be shifted and unshifted. It’s also a problem when the elements of that array are shuffled.

To force React to respect the identity and state of each child in a list, we need to uniquely identify each child instance by assigning it a **key** prop.

*Code Listing 36: **key** Prop*

```
class ProductList extends React.Component {
  render() {
    return (
      <div>
        {this.products.map(product =>
           <Product key={product.id} product={product} />)}
      </div>
```

```
    );
  }
}
```

We used **`product.id`** here as the special **`key`** prop for the **`Product`** component.

React will actually warn us if we do not supply a unique key to a mapped array. Take that warning—and all React's warnings, really—seriously.

Note that this **`key`** prop cannot be used within the **`Product`** component definition—it will not be available as **`this.props.key`**.

Do not use the index of the array elements as the **`key`** value here. If you do that, when you remove an element from the array, you would practically be changing the identities of the children, and you'll get very unpredictable behavior.

One-way data binding flow

When owners set the props of owned components, they are doing one-way data binding there. The owner will most likely bind the props of its owned components based on some computations on its own props and state. This process happens recursively, and any changes in the computations for the top-level owner component will be automatically reflected everywhere down the chain where they're passed to other components.

Chapter 6 Reusability

Function components are the best components when it comes to reusability because they are pure function with no state. They are predictable—the same input will always give us the same output.

Stateful components can be reusable too, as long as the state being managed is completely private to every instance of the component. When we start depending on an external state, reusability becomes more challenging. There are a few things we can add to a React component to enhance its reusability score.

Input validation

If you are the only one using your component, you probably know exactly what that component needs in terms of input. What are the different props needed? What data type does it need for every prop?

When that component is reused in multiple places, the manual validation of input becomes a problem. Sometimes you end up with a component that does not work, but no errors are being reported.

Even if you are the only one using your component, and you use it only once, you may not have complete control over the input values. For example, you could be feeding the component something that is read from an API, which makes the component dependent on what the API returns, and that could change any time.

Let's say you have a component that takes an array of unique numbers and displays each one of them in a **`div`**. The component also takes a number value, and it highlights that value in the array if it exists.

Here's a possible implementation with a pure stateless component:

Code Listing 37: ***`NumbersList`***

```
const NumbersList = props => (
  <div>
    {props.list.map(number =>
      <div key={number} className={`highlight-${number === props.x}`}>
        {number}
      </div>
    )}
  </div>
);
```

`props.list` is the array of numbers, and **`props.x`** is the number to highlight if it exists in **`props.list`**.

To style highlighted elements differently, let's just give them a different color.

```
.highlight-true {
  color: red;
}
```

Here's how we can use this **`NumbersList`** component in the DOM:

Code Listing 38: Using ***`NumbersList`***

Code Listing 38: Using **`NumbersList`**

```
ReactDOM.render(
  <NumbersList list={[5, 42, 12]} x={42} />,
  document.getElementById("react")
);
```

Now, try to feed this component the string **`"42"`** instead of a number.

Code Listing 39: Wrong Prop Type

```
ReactDOM.render(
  <NumbersList list={[5, 42, 12]} x="42" />,
  document.getElementById("react")
);
```

Not only will things not work as expected, but we will also not get any errors or warnings about the problem.

It would be great if we could perform input validation on this **`x`** prop (and all other props), effectively telling the component to expect only an integer value for **`x`**.

React has a way for us to do exactly that, through **`React.PropTypes`**.

We can set a **`propType`** property on every component. In that property, we can provide an object where the keys are the input props that need to be validated, and the values map to what data type React should use to validate. For example, our **`NumbersList`** component should have **`x`** validated as a **`number`**.

Here's how we can do that:

Code Listing 40: React **`PropTypes`**

```
NumbersList.propTypes = {
  x: React.PropTypes.number
};
```

If we now try to pass **`x`** as a string instead of a number, React will give us an explicit warning about it in the console.


```
▶ Warning:        console_runner-19b53204114bb6697f7c32c3c848fd19.js:1
Failed propType: Invalid prop `x` of type `string` supplied to
`NumbersList`, expected `number`.
```

Figure 3: Invalid Prop Type Warning

If we pass a correct numeric value for **`x`**, we won't get any warnings.

While these validation errors only show up in development (primarily for performance reasons), they are extremely helpful in making sure all developers use the components correctly. They also make debugging problems in components an easier task.

`React.PropTypes` has a range of validators we can use on any input. Here are some examples:

- JavaScript primitive validators:
  - **`React.PropTypes.array`**
  - **`React.PropTypes.bool`**

  - **`React.PropTypes.number`**
  - **`React.PropTypes.object`**
  - **`React.PropTypes.string`**
- **`React.PropTypes.node`**: Anything that can be rendered.
- **`React.PropTypes.element`**: A React element.
- **`React.PropTypes.instanceOf(SomeClass)`**: This uses the **`instanceof`** operator.
- **`React.PropTypes.oneOf(['Approved', 'Rejected'])`**: For ENUMs.
- **`React.PropTypes.oneOfType([..., ...])`**: Either this or that.
- **`React.PropTypes.arrayOf(React.PropTypes.number)`**: An array of a certain type.
- **`React.PropTypes.objectOf(React.PropTypes.number)`**: An object with property values of a certain type.
- **`React.PropTypes.func`**: A function reference.

By default, all props we pass to components are optional. We can change that using **`React.PropTypes.isRequired`**, which can be chained to other validators.

For example, to make **`x`** required in our **`NumbersList`** example, we can do the following:

*Code Listing 41: **`isRequired`***

```
NumbersList.propTypes = {
  x: React.PropTypes.number.isRequired
};
```

And if we try to omit **`x`** from the props at this point, we'll get the following:

```
▶Warning:        console_runner-19b53204114bb6697f7c32c3c848fd19.js:1
Failed propType: Required prop `x` was not specified in
`NumbersList`.
>
```

Figure 4: Required Prop Type Warning

In addition to the list of validators we can use with **`React.PropTypes`**, we can also use custom validators. These are just functions that we can use to do any custom checks on inputs. For example, to validate that the input value **`tweetText`** is not longer than 140 characters, we can do something like:

*Code Listing 43: Custom **`PropTypes`***

```
Tweet.propTypes = {
  tweetText: (props, propName) => {
    if (props[propName] && props[propName].length > 140) {
      return new Error('Too long');
    }
  }
}
```

For components created with **`React.createClass`**, **`propTypes`** is just a property on the input object. For the regular class syntax, we can use a static property (which is a proposed feature in JavaScript).

*Code Listing 44: **`propTypes`** Syntax*

```
// With React.createClass syntax:
let NumbersList = React.createClass({
  propTypes: {
    x: React.PropTypes.number
  },
});

// For class syntax:
class NumbersList extends React.Component {
  static propTypes = {
    x: React.PropTypes.number
  };
```



```
}

// For functions syntax (and also works for class syntax):
NumberList.propTypes = {
  x: React.PropTypes.number
}
```

Input default values

One other useful thing we can do on any input prop for a React component is assign it a default value in case we use the component without passing a value to that prop.

For example, here's an alert box component that displays an error message in an alert **`div`**. If we don't specify any message when we use it, it will default to "Something went wrong."

*Code Listing 45: React **`defaultProps`***

```
const AlertBox = props => (
  <div className="alert alert-danger">
    {props.message}
  </div>
);

AlertBox.defaultProps = {
  message: "Something went wrong"
};

ReactDOM.render(
  <AlertBox />,
  document.getElementById("react")
);
```

The syntax to use **`defaultProps`** with **`React.Component`** is similar to **`propTypes`**. For **`React.createClass`**, we use the method **`getDefaultProps`** instead:

*Code Listing 46: **`getDefaultProps()`***

```
const AlertBox = React.createClass({
  getDefaultProps() {
    return { message: "Something went wrong" };
  }
});
```


Shared component behavior

Sometimes, reusing the whole component is not an option; we'll have cases where some components share most of their functionalities but are different in a few areas.

If we need different components to share common behavior, we have two options:

- If we're using the **`React.createClass`** syntax, we can use *mixins*.
- If we're using the **`React.Component`** class syntax, we can create a new class and have all components extend it. We can also manually inject the external methods we'd like our classes to have in their constructor functions.

Mixins are objects that we can "mix" into any component defined with **`React.createClass`**.

For example, in one application, we have links and buttons with unique **`id`**s, and we would like to track all clicks that happen on them. Every time a user clicks a button or a link, we want to hit an API endpoint to log that event to our database.

We have two components in this app, **`Link`** and **`Button`**, and both have a click handler **`handleClick`**.

To log every click before handling it, we need something like this:

*Code Listing 47: **`LogClick`** Function*

```
logClick() {
  console.log(`Element ${this.props.id} clicked`);
  $.post(`/clicks/${this.props.id}`);
}
```

We can add this method to both components, but instead of repeating it twice, we can put **`logClick()`** in a mixin, and include the mixin in every component where we need to log the clicks.

Here's how we put the **`logClick`** feature in a mixin and use it in both **`Link`** and **`Button`**:

*Code Listing 48: **`LogClickMixin`***

```
const logClicksMixin = {
  logClick() {
    console.log(`Element ${this.props.id} clicked`);
    $.post(`/clicks/${this.props.id}`);
  },
};

const Link = React.createClass({
  mixins: [logClicksMixin],
  handleClick(e) {
```

```
    this.logClick();
    e.preventDefault();
    console.log("Handling a link click...");
  },
  render() {
    return (
      <a href="#" onClick={this.handleClick}>Link</a>
    );
  }
});

const Button = React.createClass({
  mixins: [logClicksMixin],
  handleClick(e) {
    this.logClick();
    e.preventDefault();
    console.log("Handling a button click...");
  },
  render() {
    return (
      <button onClick={this.handleClick}>Button</button>
    );
  }
});

ReactDOM.render(
  <div>
    <Link id="link1" />
    <br /><br />
    <Button id="button1" />
  </div>,
  document.getElementById("react")
);
```

This is a better approach, since now the **`logClick`** implementation is abstracted in one place. If in the future we need to change that implementation, we only need to do it in that one place.

If we want to do the exact same thing with **`React.Component`** class syntax (which is vanilla JavaScript where we don't have mixins), we have many options. Here's one possible way: Create a new class **`Loggable`**, implement the **`logClick`** in there, and then make both **`Button`** and **`Link`** components extend **`Loggable`**.

*Code Listing 49: The **`Loggable`** Class*

```
class Loggable extends React.Component {
  logClick() {
    console.log(`Element ${this.props.id} clicked`);
    $.post(`/clicks/${this.props.id}`);
```

```
  }
}

class Link extends Loggable {
  handleClick(e) {
    this.logClick();
    e.preventDefault();
    console.log("Handling a link click...");
  }
  render() {
    return (
      <a href="#" onClick={this.handleClick.bind(this)}>Link</a>
    );
  }
}

class Button extends Loggable {
  handleClick(e) {
    this.logClick();
    e.preventDefault();
    console.log("Handling a button click...");
  }
  render() {
    return (
      <button onClick={this.handleClick.bind(this)}>Button</button>
    );
  }
}

ReactDOM.render(
  <div>
    <Link id="link1" />
    <br /><br />
    <Button id="button1" />
  </div>,
  document.getElementById("react")
);
```


Chapter 7 Working with User Input

We modeled our data state and declaratively defined our UI as a function of our state. It's now time to start accounting for UI events and capturing input from the user.

The nature of user input can be described as a reverse data flow when compared to how

React's components represent data. In fact, with active user input on a page, the data starts depending on the UI.

Suppose, for example, we have a UI that displays a list of comments. We start with an array of comments in the data layer, make the UI represent that, and then add a form on the page to add a new comment. Our array of data now depends on the user interactions with this form.

If we handle the input events, capture the data that users enter, and then manipulate our original data source to account for the new input, React will refresh the views that are using that data.

I'll first explain the simple way to work with user input: *refs attributes*. Then I'll explain how to work with user input using the recommended way: *controlled components*.

Before we go into that, however, we need to understand the synthetic events system in React.

React's synthetic events

We can define native browser events on DOM elements. For example, to **`console.log`** the content of a **`div`** when it's clicked, we can do the following:

Code Listing 50: HTML Event

```
<div onclick="console.log(this.textContent);">
  You might not need React...
</div>
```

To do that exact same thing in React, we use an **`onClick`** synthetic event.

Code Listing 51: React Event

```
<div onClick={se => console.log(se.target.textContent)} >
  You might not need React...
</div>
```

The differences between these events are:

- Casing is important. React uses **`camelCase`** for the event attributes (**`onClick`**, **`onKeyPress`**).

- We don't use strings with React events; we use JavaScript directly.
- We use a function for the value. The function gets defined when we define the component, and it will be invoked when we click on the element.

An **`onClick`** used within a React component works with a synthetic event. That's why I named the variable passed to the function "**`se`**". React's synthetic events are wrappers around the browser's native events. We can still access the native event object itself using the **`nativeEvent`** attribute on any synthetic event. We can see the native event if we **`console.log(se.nativeEvent)`** in the previous example.

Browsers historically had significant differences in the way they handled and invoked their native events. This is getting a lot better with the W3C standards now, but browser quirks are not to be trusted. With React synthetic events, we don't have to worry about that. React will make sure events work identically across all browsers.

One notable example of an event that React made standard between multiple inputs and in all browsers is the **`onChange`** event. The native **`onChange`** event works differently among different input types, and it usually does not mean "change," but rather that "the user is done changing."

With React events, the **`onChange`** event means that on any change, anywhere, anytime, and for any input:

- If we're typing in an **`input`** or **`textarea`** element, React will fire the **`onChange`** event on every key stroke (much like keyboard events, but also with support for paste and automated entries).
- If we select or clear a check box, React will fire the **`onChange`** event.
- When we select an option from a drop-down select element, React will fire the **`onChange`** event.

Here are examples of some of the popular events that we can use with React:

- Keyboard events: **`onKeyDown`**, **`onKeyPress`**, **`onKeyUp`**
- Focus events: **`onFocus`**, **`onBlur`**
- Form events: **`onChange`**, **`onInput`**, **`onSubmit`**
- Touch events: **`onTouchStart`**, **`onTouchEnd`**, **`onTouchMove`**, **`onTouchCancel`**
- Mouse events: **`onClick`**, **`onDrag`**, **`onMouseOver`**, **`onMouseOut`**, etc.

There are also clipboard events, UI events, wheel events, composition events, media events, and image events.

We can even work with events that are not supported in React by using **`addEventListener`** (usually in **`componentDidMount`**) and **`removeEventListener`** (in **`componentWillUnmount`**).

Working with DOM nodes in the browser

React's powerful performance feature, the virtual DOM, frees us from ever needing to touch the DOM. When adding React's synthetic events to the mix, React (in a way) has created a "fake browser" that respects all the standards, is way faster, and is a lot easier to work with.

This, however, does not mean that we can't use the original browser's DOM and events with React if we need to. Sometimes we need to do special things with the DOM, such as integrate our components with third-party libraries.

If we need to access a native DOM node in any React component, we can use React's special attribute, **`ref`**.

We can pass that attribute either a string or a function. You'll find a lot of examples that use a string value (which acts like a unique identifier), but don't do that; instead, use functions.

To work through an example, let's say we have an email input field where the user is supposed to type an email, and we have a Save button. Before we save, we want to make sure that the input is a valid email. If we use the **`<input type="email" />`** element, we can use the native API method **`element.checkValidity()`** to figure out whether or not the user entered a valid email. To invoke that API method, we need access to the native DOM element.

Here's the full example implemented with React's **`ref`** attribute:

Code Listing 52: Ref Attributes

```
const EmailForm = React.createClass({
  handleClick() {
    if (this.inputRef.checkValidity()) {
```

```
      console.log(`Email Ok. Saving Email as ${this.inputRef.value}`);
    }
  },
  render() {
    return (
      <div>
        <input type="email" ref={inputRef => this.inputRef = inputRef} />
        <button onClick={this.handleClick}>Save</button>
      </div>
    );
  }
})

ReactDOM.render(<EmailForm />, document.getElementById("react"));
```


When we define a **`ref`** attribute on the input element and use a function for its value, React will execute that function when the input element gets mounted with the **`EmailForm`** component. React will also pass a reference to the DOM input element (**`inputRef`** in the example) as an argument to that **`ref`** function. Inside the **`ref`** function, we can access the **`EmailForm`** component instance via the **`this`** keyword, so we can store the input reference as an instance variable.

Once we have the reference to the native DOM element (**`this.inputRef`**), we can access all of its API normally. In **`handleClick`**, we are calling the native **`checkValidity()`** function on the DOM reference.

We can use the same trick to read the value of any input at any point. In **`handleClick`**, the **`console.log`** line actually reports the email input text value (using the native API property "**`value`**") when the input is valid.

Changing a React component's state with native API calls

We can use React's **`ref`** attributes to read input from the user by assigning component variables to input fields like we did in the previous example. However, every time the user types something into an input field, they're practically changing the "state" of that field. Since we are representing elements with stateful components in React, not reflecting the input state change back to the enclosing component's state means that our component is no longer an exact representation of its current state that is defined in React.

Here's an example to make that point easy to understand:

Code Listing 53: Input State

```
const EmailForm = React.createClass({
  getInitialState() {
    return {
      currentEmail: this.props.currentEmail
    };
  },
  render() {
    return (
      <div>
        <input type="email" value={this.state.currentEmail} />
        <button>Save</button>
      </div>
```

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

ReactDOM.render(
  <EmailForm currentEmail="mark@fb.com" />,
  document.getElementById("react")
);
```


We rendered the same **`EmailForm`** to save an email field, but this time we're displaying an initial value of the email field, and users can change it if they want (an "edit" feature).

Since the email is something that can be changed in this component, we put it on the state of the component. React rendered the input email and used the component state to display the default value. If we change the **`currentEmail`** state in memory, React will update the value displayed in the input box.

However, if the user is allowed to change the value of that input field directly using the browser API (by typing in the text box), then the displayed DOM in the browser will be different than the current copy of the virtual DOM that we have in memory because that one is reflecting the **`currentEmail`** state, which has not changed. If something else changes in the state of the **`EmailForm`** component, React will re-render the **`EmailForm`** component, and the value the user typed in the email will be lost. This is why React would not allow the user to type in this example's email input.

React components should always represent their state in the DOM, and the input field is part of that state. If we can make the input field always represent the React component's state, even when the user types in it, then we can just read React's state whenever we need to read the new input from the user.

A component where we control the input to always reflect the state is called a *controlled component*.

Controlled components

By using controlled components, we don't have to reach to an input field's native DOM element to read its value since whatever the user types in there will be reflected on the component state itself.

To achieve this level of control, we use an **`onChange`** event handler. Every time a change event is fired, we update the component state associated with the input.

Here's the previous example updated to be a controlled component:

Code Listing 54: Controlled Component

```
const EmailForm = React.createClass({
  getInitialState() {
    return { currentEmail: this.props.currentEmail };
  },
  setCurrentEmailState(se) {
    this.setState({ currentEmail: se.target.value });
  },
  handleClick() {
    console.log(`Saving New Email value: ${this.state.currentEmail}`);
  },
  render() {
```

```
    return (
      <div>
        <input type="email" value={this.state.currentEmail}
               onChange={this.setCurrentEmailState} />
        <button onClick={this.handleClick}>Save</button>
      </div>
    );
  }
})

ReactDOM.render(
  <EmailForm currentEmail="mark@fb.com" />,
  document.getElementById("react")
);
```

By using the **`setCurrentEmailState`** **`onChange`** handler, we're updating the state of the **`EmailForm`** component every time the user types in the email field. This way we make sure that the new DOM in the browser and React's in-memory virtual DOM are both in sync, reflecting the current state of the **`EmailForm`** component.

Note how in the **`console.log`** line, we can read any new input value the user entered using **`this.state.currentEmail`**.

More Pages