3 BOOKS IN 1

THE ULTIMATE GUIDE TO GETTING STARTED WITH REACT

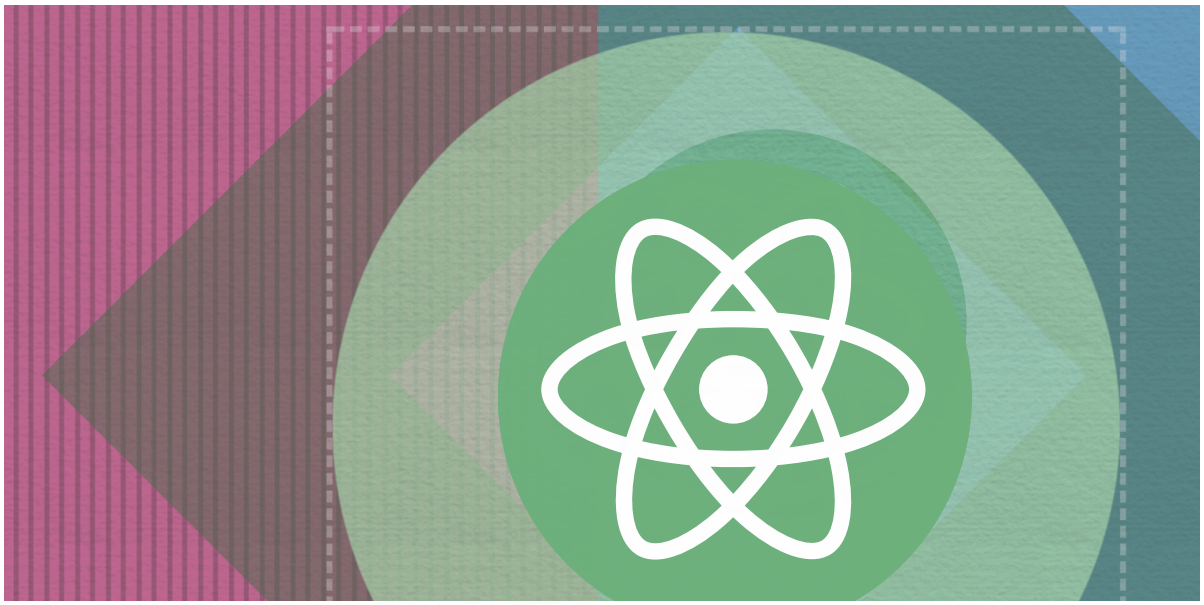
A REACT COLLECTION

SITEPOINT

Table of Contents

Learn

Your First Week With React



sitepoint

Your First Week With React

Copyright © 2017 SitePoint Pty. Ltd.

Product Manager: Simon Mackie

Cover Designer: Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

Table of Contents

Preface

React is a remarkable JavaScript library that's taken the development community by storm. In a nutshell, it's made it easier for developers to build interactive user interfaces for web, mobile and desktop platforms. One of its best features is its freedom from the problematic bugs inherent in MVC frameworks, where inconsistent views is a recurring problem for big projects. Today, thousands of companies worldwide are using React, including big names such as Netflix and AirBnB. React has become immensely popular, such that a number of apps have been ported to React --- including WhatsApp, Instagram and Dropbox.

This book is a collection of articles, selected from SitePoint's React Hub, that will guide you through your first week with the amazingly flexible library.

Who Should Read This Book

This book is for novice React developers. You'll need to be familiar with HTML and CSS and have a reasonable level of understanding of JavaScript in order to follow the discussion.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.</p>
```

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed:

```
function animate() {
    ⋮
new_variable = "Hello";
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ↪ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-
↪design-real-user-testing/?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter

1

How to Tell if React is the Best Fit for Your Next Project

by Maria Antonietta Perna

Nowadays, users expect sleek, performant web applications that behave more and more like native apps. Techniques have been devised to decrease the waiting time for a website to load on a user's first visit. However, in web applications that expose a lot of interactivity, the time elapsing between a user action taking place and the app's response is also important. Native apps feel snappy, and web apps are expected to behave the same, even on less than ideal internet connections.

A number of modern JavaScript frameworks have sprung up that can be very effective at tackling this problem. React can be safely considered among the most popular and robust JavaScript libraries you can use to create fast, interactive user interfaces for web apps.

In this article, I'm going to talk about what React is good at and what makes it work, which should provide you with some context to help you decide if this library could be a good fit for your next project.

What Is React?

React is a Facebook creation which simply labels itself as being “a JavaScript library for building user interfaces”.

It's an open-source project which, to date, has raked in over 74,000 stars on GitHub.

React is:

- **Declarative**: you only need to design *simple views for each state in your application*, and React will efficiently update and render just the right components when your data changes.
- **Component-based**: you create your React-powered apps by assembling a number of encapsulated components, each managing its own state.
- **Learn Once, Write Anywhere**: React is not a full-blown framework; it's just a library for rendering views.

How Does the Virtual DOM Work?

The **Virtual DOM** is at the core of what makes React fast at rendering user interface elements and their changes. Let's look closer into its mechanism.

The HTML Document Object Model or DOM is a

> programming interface for HTML and XML documents. ... The DOM provides a representation of the document as a structured group of nodes and objects that have properties and methods. Essentially, it connects web pages to scripts or programming languages. — MDN

Whenever you want to change any part of a web page programmatically, you need to modify the DOM. Depending on the complexity and size of the document, traversing the DOM and updating it could take longer than users might be prepared to accept, especially if you take into account the work browsers need to do when something in the DOM changes. In fact, every time the DOM gets updated, browsers need to recalculate the CSS and carry out layout and repaint operations on the web page.

React enables developers to make changes to the web page without having to deal directly with the DOM. This is done via the **Virtual DOM**.

The Virtual DOM is a lightweight, abstract model of the DOM. React uses the `render` method to create a node tree from React components and updates this tree in response to changes in the data model resulting from actions.

Each time there are changes to the underlying data in a React app, React creates a new Virtual DOM representation of the user interface.

Updating UI Changes with the Virtual DOM

When it comes to updating the browser's DOM, React roughly follows the steps below:

- Whenever something changes, React re-renders the entire UI in a Virtual DOM representation.
- React then calculates the difference between the previous Virtual DOM representation and the new one.
- Finally, React patches up the real DOM with what has actually changed. If nothing has changed, React won't be dealing with the HTML DOM at all.

One would think that such a process, which involves keeping two representations of the Virtual DOM in memory and comparing them, could be slower than dealing directly with the actual DOM. This is where efficient diff algorithms, batching DOM read/write operations, and limiting DOM changes to the bare minimum necessary, make using React and its Virtual DOM a great choice for building performant apps.

Is React Good for Every Project?

As the name itself suggests, React is great at making super reactive user interfaces — that is, UIs that are very quick at responding to events and consequent data changes. This comment about the name *React* made by Jordan Walke, engineer at Facebook, is illuminating:

> This API reacts to any state or property changes, and works with data of any form (as deeply structured as the graph itself) so I think the name is fitting. — Vjeux, "Our First 50,000 Stars"

Although some would argue that all projects need React, I think it's uncontroversial to say that React would be a great fit for web apps where you need to keep a complex, interactive UI in sync with frequent changes in the underlying data model.

React is designed to deal with stateful components in an efficient way (which doesn't mean devs don't need to optimize their code). So projects that would benefit from this capability can be considered good candidates for React.

Chris Coyier outlines the following, interrelated situations when reaching for React makes sense, which I tend to go along with:

- **Lots of state management and DOM manipulation**. That is, enabling and disabling buttons, making links active, changing input values, closing and expanding menus, etc. In this kind of project, React makes managing stateful components faster and easier. As Michael Jackson, co-author of React Router, aptly put it in a Tweet:

  > Point is, React takes care of the hard part of figuring out what changes actually need to happen to the DOM, not me. That's *invaluable*

- **Fighting spaghetti**. Keeping track of complex state by directly modifying the DOM could lead to spaghetti code, at least if extra attention isn't paid to code organization and structure.

Resources

If you're curious about how React and its Virtual DOM work, here's where you can learn more:

- React Videos from Facebook Engineers
- "The Real Benefits of the Virtual DOM in React.js", by Chris Minnick
- "The difference between Virtual DOM and DOM", by Bartosz Krajka
- "React is Slow, React is Fast: Optimizing React Apps in Practice", by François Zaninotto
- "How to Choose the Right Front-end Framework for Your Company", by Chris Lienert

Conclusion

React and other similar JavaScript libraries ease the development of snappy, event-driven user interfaces that are fast at responding to state changes. One

underlying goal can be identified in the desire to bridge the gap between web apps and native apps: users expect web apps to feel smooth and seamless like native apps.

This is the direction towards which modern web development is heading. It's not by chance that the latest update of *Create React App*, a project that makes possible the creation of React apps with zero configuration, has shipped with the functionality of creating progressive web apps (PWAs) by default. These are apps that leverage service workers and offline-first caching to minimize latency and make web apps work offline.

Chapter

2

React vs Angular: An In-depth Comparison

by Pavels Jelisejevs

Should I choose Angular or React? Today's bipolar landscape of JavaScript frameworks has left many developers struggling to pick a side in this debate. Whether you're a newcomer trying to figure out where to start, a freelancer picking a framework for your next project, or an enterprise-grade architect planning a strategic vision for your company, you're likely to benefit from having an educated view on this topic.

To save you some time, let me tell you something up front: this article won't give a clear answer on which framework is better. But neither will hundreds of other articles with similar titles. I can't tell you that, because the answer depends on a wide range of factors which make a particular technology more or less suitable for your environment and use case.

Since we can't answer the question directly, we'll attempt something else. We'll compare Angular (2+, not the old AngularJS) and React, to demonstrate how you can approach the problem of comparing any two frameworks in a structured manner on your own and tailor it to your environment. You know, the old "teach a man to fish" approach. That way, when both are replaced by a BetterFramework.js in a year's time, you'll be able to re-create the same train of thought once more.

Where to Start?

Before you pick any tool, you need to answer two simple questions: "Is this a good tool per se?" and "Will it work well for my use case?" Neither of them mean anything on their own, so you always need to keep both of them in mind. All right, the questions might not be that simple, so we'll try to break them down into smaller ones.

Questions on the tool itself:

- How mature is it and who's behind it?
- What kind of features does it have?
- What architecture, development paradigms, and patterns does it employ?

- What is the ecosystem around it?

Questions for self-reflection:

- Will I and my colleagues be able to learn this tool with ease?
- Does is fit well with my project?
- What is the developer experience like?

Using this set of questions you can start your assessment of any tool and we'll base our comparison of React and Angular on them as well.

There's another thing we need to take into account. Strictly speaking, it's not exactly fair to compare Angular to React, since Angular is a full-blown, feature-rich framework, while React just a UI component library. To even the odds, we'll talk about React in conjunction with some of the libraries often used with it.

Maturity

An important part of being a skilled developer is being able to keep the balance between established, time-proven approaches and evaluating new bleeding-edge tech. As a general rule, you should be careful when adopting tools that haven't yet matured due to certain risks:

- The tool may be buggy and unstable.
- It might be unexpectedly abandoned by the vendor.
- There might not be a large knowledge base or community available in case you need help.

Both React and Angular come from good families, so it seems that we can be confident in this regard.

React

React is developed and maintained by Facebook and used in their own products,

including Instagram and WhatsApp. It has been around for roughly three and a half years now, so it's not exactly new. It's also one of the most popular projects on GitHub, with about 74,000 stars at the time of writing. Sounds good to me.

Angular

Angular (version 2 and above) has been around less then React, but if you count in the history of its predecessor, AngularJS, the picture evens out. It's maintained by Google and used in AdWords and Google Fiber. Since AdWords is one of the key projects in Google, it is clear they have made a big bet on it and is unlikely to disappear anytime soon.

Features

Like I mentioned earlier, Angular has more features out of the box than React. This can be both a good and a bad thing, depending on how you look at it.

Both frameworks share some key features in common: components, data binding, and platform-agnostic rendering.

Angular

Angular provides a lot of the features required for a modern web application out of the box. Some of the standard features are:

- Dependency injection
- Templates, based on an extended version of HTML
- Routing, provided by @angular/router
- Ajax requests by @angular/http
- @angular/forms for building forms
- Component CSS encapsulation
- XSS protection
- Utilities for unit-testing components.

Having all of these features available out of the box is highly convenient when you don't want to spend time picking the libraries yourself. However, it also means that you're stuck with some of them, even if you don't need them. And replacing them will usually require additional effort. For instance, we believe that for small projects having a DI system creates more overhead than benefit, considering it can be effectively replaced by imports.

React

With React, you're starting off with a more minimalistic approach. If we're looking at just React, here's what we have:

- No dependency injection
- Instead of classic templates it has JSX, an XML-like language built on top of JavaScript
- XSS protection
- Utilities for unit-testing components.

Not much. And this can be a good thing. It means that you have the freedom to choose whatever additional libraries to add based on your needs. The bad thing is that you actually have to make those choices yourself. Some of the popular libraries that are often used together with React are:

- React-router for routing
- Fetch (or axios) for HTTP requests
- A wide variety of techniques for CSS encapsulation
- Enzyme for additional unit-testing utilities.

We've found the freedom of choosing your own libraries liberating. This gives us the ability to tailor our stack to particular requirements of each project, and we didn't find the cost of learning new libraries that high.

Languages, Paradigms, and Patterns

Taking a step back from the features of each framework, let's see what kind higher-level concepts are popular with both frameworks.

React

There are several important things that come to mind when thinking about React: JSX, Flow, and Redux.

JSX

JSX is a controversial topic for many developers: some enjoy it, and others think that it's a huge step back. Instead of following a classical approach of separating markup and logic, React decided to combine them within components using an XML-like language that allows you to write markup directly in your JavaScript code.

While the merits of mixing markup with JavaScript might be debatable, it has an indisputable benefit: static analysis. If you make an error in your JSX markup, the compiler will emit an error instead of continuing in silence. This helps by instantly catching typos and other silly errors.

Flow

Flow is a type-checking tool for JavaScript also developed by Facebook. It can parse code and check for common type errors such as implicit casting or null dereferencing.

Unlike TypeScript, which has a similar purpose, it does not require you to migrate to a new language and annotate your code for type checking to work. In Flow, type annotations are optional and can be used to provide additional hints to the analyzer. This makes Flow a good option if you would like to use static code analysis, but would like to avoid having to rewrite your existing code.

- Further reading: Writing Better JavaScript with Flow

Redux

Redux is a library that helps manage state changes in a clear manner. It was inspired by Flux, but with some simplifications. The key idea of Redux is that the whole state of the application is represented by a single object, which is mutated by functions called reducers. Reducers themselves are pure functions and are implemented separately from the components. This enables better separation of concerns and testability.

If you're working on a simple project, then introducing Redux might be an over complication, but for medium- and large-scale projects, it's a solid choice. The library has become so popular that there are projects implementing it in Angular as well.

All three features can greatly improve your developer experience: JSX and Flow allow you to quickly spot places with potential errors, and Redux will help achieve a clear structure for your project.

Angular

Angular has a few interesting things up its sleeve as well, namely TypeScript and RxJS.

TypeScript

TypeScript is a new language built on top of JavaScript and developed by Microsoft. It's a superset of JavaScript ES2015 and includes features from newer versions of the language. You can use it instead of Babel to write state of the art JavaScript. It also features an extremely powerful typing system that can statically analyze your code by using a combination of annotations and type inference.

There's also a more subtle benefit. TypeScript has been heavily influenced by Java and .NET, so if your developers have a background in one of these languages, they are likely to find TypeScript easier to learn than plain JavaScript (notice how we switched from the tool to your personal environment). Although Angular has been the first major framework to actively adopt TypeScript, it's also possible to use it together with React.

- Further reading: An Introduction to TypeScript: Static Typing for the Web

RxJS

RxJS is a reactive programming library that allows for more flexible handling of asynchronous operations and events. It's a combination of the Observer and Iterator patterns blended together with functional programming. RxJS allows you to treat anything as a continuous stream of values and perform various operations on it such as mapping, filtering, splitting or merging.

The library has been adopted by Angular in their HTTP module as well for some internal use. When you perform an HTTP request, it returns an Observable instead of the usual Promise. Although this library is extremely powerful, it's also quite complex. To master it, you'll need to know your way around different types of Observables, Subjects, as well as around a hundred methods and operators. Yikes, that seems to be a bit excessive just to make HTTP requests!

RxJS is useful in cases when you work a lot with continuous data streams such as web sockets, however, it seems overly complex for anything else. Anyway, when working with Angular you'll need to learn it at least on a basic level.

- Further reading: Introduction to Functional Reactive Programming with RxJS

We've found TypeScript to be a great tool for improving the maintainability of our projects, especially those with a large code base or complex domain/business logic. Code written in TypeScript is more descriptive and easier to follow. Since TypeScript has been adopted by Angular, we hope to see even more projects

using it. RxJS, on the other hand, seems only to be beneficial in certain cases and should be adopted with care. Otherwise, it can bring unwanted complexity to your project.

Ecosystem

The great thing about open source frameworks is the number of tools created around them. Sometimes, these tools are even more helpful than the framework itself. Let's have a look at some of the most popular tools and libraries associated with each framework.

Angular

Angular CLI

A popular trend with modern frameworks is having a CLI tool that helps you bootstrap your project without having to configure the build yourself. Angular has Angular CLI for that. It allows you to generate and run a project with just a couple of commands. All of the scripts responsible for building the application, starting a development server and running tests are hidden away from you in `node_modules`. You can also use it to generate new code during development. This makes setting up new projects a breeze.

- Further reading: The Ultimate Angular CLI Reference

Ionic 2

Ionic 2 is a new version of the popular framework for developing hybrid mobile applications. It provides a Cordova container that is nicely integrated with Angular 2, and a pretty material component library. Using it, you can easily set up and build a mobile application. If you prefer a hybrid app over a native one, this is a good choice.

Material design components

If you're a fan of material design, you'll be happy to hear that there's a Material component library available for Angular. Currently, it's still at an early stage and slightly raw but it has received lots of contributions recently, so we might hope for things to improve soon.

Angular universal

Angular universal is a seed project that can be used for creating projects with support for server-side rendering.

@ngrx/store

@ngrx/store is a state management library for Angular inspired by Redux, being based on state mutated by pure reducers. Its integration with RxJS allows you to utilize the push change detection strategy for better performance.

- Further reading: Managing State in Angular 2 Apps with ngrx/store

Other Tools

There are plenty of other libraries and tools available in the Awesome Angular list.

React

Create React App

Create React App is a CLI utility for React to quickly set up new projects. Similar to Angular CLI it allows you to generate a new project, start a development server and create a bundle. It uses Jest, a relatively new test runner from Facebook, for unit testing, which has some nice features of its own. It also

supports flexible application profiling using environment variables, backend proxies for local development, Flow, and other features. Check out this brief introduction to Create React App for more information.

React Native

React Native is a platform developed by Facebook for creating native mobile applications using React. Unlike Ionic, which produces a hybrid application, React Native produces a truly native UI. It provides a set of standard React components which are bound to their native counterparts. It also allows you to create your own components and bind them to native code written in Objective-C, Java or Swift.

Material UI

There's a material design component library available for React as well. Compared to Angular's version, this one is more mature and has a wider range of components available.

Next.js

Next.js is a framework for the server-side rendering of React applications. It provides a flexible way to completely or partially render your application on the server, return the result to the client and continue in the browser. It tries to make the complex task of creating universal applications as simple as possible so the set up is designed to be as simple as possible with a minimal amount of new primitives and requirements for the structure of your project.

MobX

MobX is an alternative library for managing the state of an application. Instead of keeping the state in a single immutable store, like Redux does, it encourages you to store only the minimal required state and derive the rest from it. It provides a set of decorators to define observables and observers and introduce reactive

logic to your state.

- Further reading: How to Manage Your JavaScript Application State with MobX

Storybook

Storybook is a component development environment for React. It allows you to quickly set up a separate application to showcase your components. On top of that, it provides numerous add-ons to document, develop, test and design your components. We've found it to be extremely useful to be able to develop components independently from the rest of the application. You can learn more about Storybook from a previous article.

Other Tools

There are plenty of other libraries and tools available in the Awesome React list.

Adoption, Learning Curve and Development Experience

An important criterion for choosing a new technology is how easy it is to learn. Of course, the answer depends on a wide range of factors such as your previous experience and a general familiarity with the related concepts and patterns. However, we can still try to assess the number of new things you'll need to learn to get started with a given framework. Now, if we assume that you already know ES6+, build tools and all of that, let's see what else you'll need to understand.

React

With React, the first thing you'll encounter is JSX. It does seem awkward to write for some developers. However, it doesn't add that much complexity --- just

expressions, which are actually JavaScript, and a special HTML-like syntax. You'll also need to learn how to write components, use props for configuration and manage internal state. You don't need to learn any new logical structures or loops since all of this is plain JavaScript.

The official tutorial is an excellent place to start learning React. Once you're done with that, get familiar with the router. The react router v4 might be slightly complex and unconventional, but nothing to worry about. Using Redux will require a paradigm shift to learn how to accomplish already familiar tasks in a manner suggested by the library. The free Getting Started with Redux video course can quickly introduce you to the core concepts. Depending on the size and the complexity of your project you'll need to find and learn some additional libraries and this might be the tricky part, but after that everything should be smooth sailing.

We were genuinely surprised at how easy it was to get started using React. Even people with a backend development background and limited experience in frontend development were able to catch up quickly. The error messages you might encounter along the way are usually clear and provide explanations on how to resolve the underlying problem. The hardest part may be finding the right libraries for all of the required capabilities, but structuring and developing an application is remarkably simple.

Angular

Learning Angular will introduce you to more new concepts than React. First of all, you'll need to get comfortable with TypeScript. For developers with experience in statically typed languages such as Java or .NET this might be easier to understand than JavaScript, but for pure JavaScript developers, this might require some effort.

The framework itself is rich in topics to learn, starting from basic ones such as modules, dependency injection, decorators, components, services, pipes, templates, and directives, to more advanced topics such as change detection,

zones, AoT compilation, and Rx.js. These are all covered in the documentation. Rx.js is a heavy topic on its own and is described in much detail on the official website. While relatively easy to use on a basic level it gets more complicated when moving on to advanced topics.

All in all, we noticed that the entry barrier for Angular is higher than for React. The sheer number of new concepts is confusing to newcomers. And even after you've started, the experience might be a bit rough since you need to keep in mind things like Rx.js subscription management, change detection performance and bananas in a box (yes, this is an actual advice from the documentation). We often encountered error messages that are too cryptic to understand, so we had to google them and pray for an exact match.

It might seem that we favor React here, and we definitely do. We've had experience onboarding new developers to both Angular and React projects of comparable size and complexity and somehow with React it always went smoother. But, like I said earlier, this depends on a broad range of factors and might work differently for you.

Putting it Into Context

You might have already noted that each framework has its own set of capabilities, both with their good and bad sides. But this analysis has been done outside of any particular context and thus doesn't provide an answer on which framework should you choose. To decide on that, you'll need to review it from a perspective of your project. This is something you'll need to do on your own.

To get started, try answering these questions about your project and when you do, match the answers against what you've learned about the two frameworks. This list might not be complete, but should be enough to get you started:

1. How big is the project?
2. How long is it going to be maintained for?

3. Is all of the functionality clearly defined in advance or are you expected to be flexible?
4. If all of the features are already defined, what capabilities do you need?
5. Are the domain model and business logic complex?
6. What platforms are you targeting? Web, mobile, desktop?
7. Do you need server-side rendering? Is SEO important?
8. Will you be handling a lot of real-time event streams?
9. How big is your team?
10. How experienced are your developers and what is their background?
11. Are there any ready-made component libraries that you would like to use?

If you're starting a big project and you would like to minimize the risk of making a bad choice, consider creating a proof-of-concept product first. Pick some of the key features of the projects and try to implement them in a simplistic manner using one of the frameworks. PoCs usually don't take a lot if time to build, but they'll give you some valuable personal experience on working with the framework and allow you to validate the key technical requirements. If you're satisfied with the results, you can continue with full-blown development. If not, failing fast will save you lot of headaches in the long run.

One Framework to Rule Them All?

Once you've picked a framework for one project, you'll get tempted to use the exact same tech stack for your upcoming projects. Don't. Even though it's a good idea to keep your tech stack consistent, don't blindly use the same approach every time. Before starting each project, take a moment to answer the same questions once more. Maybe for the next project, the answers will be different or the landscape will change. Also, if you have the luxury of doing a small project with a non-familiar tech stack, go for it. Such experiments will provide you with

invaluable experience. Keep your mind open and learn from your mistakes. At some point, a certain technology will just feel natural and *right*.

Chapter

Getting Started with React: A Beginner's Guide

3

by Michael Wanyoike

In this guide, I'll show you the fundamental concepts of React by taking you through a practical, step-by-step tutorial on how to create a simple Message App using React. I'll assume you have no previous knowledge of React. However, you'll need at least to be familiar with modern JavaScript and NodeJS.

React is a remarkable JavaScript library that's taken the development community by storm. In a nutshell, it's made it easier for developers to build interactive user interfaces for web, mobile and desktop platforms. One of its best features is its freedom from the problematic bugs inherent in MVC frameworks, where inconsistent views is a recurring problem for big projects. Today, thousands of companies worldwide are using React, including big names such as Netflix and AirBnB. React has become immensely popular, such that a number of apps have been ported to React --- including WhatsApp, Instagram and Dropbox.

Prerequisites

As mentioned, you need some experience in the following areas:

- functional JavaScript
- object-oriented JavaScript
- ES6 JavaScript Syntax

On your machine, you'll need:

- a NodeJS environment
- a Yarn setup (optional)

If you'd like to take a look first at the completed project that's been used in this guide, you can access it via GitHub.

What is React?

React is a JavaScript library for building UI components. Unlike more complete

frameworks such as Angular or Vue, React deals only with the view layer. Hence, you'll need additional libraries to handle things such as data flow, routing, authentication etc. In this guide, we'll focus on what React can do.

Building a React project involves creating one or more React **components** that can interact with each other. A React component is simply a JavaScript class that requires the `render` function to be declared. The `render` function simply outputs HTML code, which is implemented using either JSX or JavaScript code. A React component may also require additional functions for handling data, actions and lifecyle events.

React components can further be categorized into **containers/stateful components** and **stateless components**. A stateless component's work is simply to display data that it receives from its parent React component. It can also receive events and inputs, which it passes up to its parent to handle. A React container or stateful component does the work of rendering one or more child components. It fetches data from external sources and feeds it to its child components. It also receives inputs and events from them in order to initiate actions.

Understanding the React DOM

Before we get to coding, you need to be aware that React uses a **Virtual DOM** to handle page rendering. If you're familiar with jQuery, you know that it can directly manipulate a web page via the **HTML DOM**. In a lot of use cases, this direct interaction poses little to no problems. However, for certain cases, such as the running of a highly interactive, real-time web application, performance often takes a huge hit.

To counter this, the concept of the Virtual DOM was invented, and is currently being applied by many modern UI frameworks including React. Unlike the HTML DOM, the Virtual DOM is much easier to manipulate, and is capable of handling numerous operations in milliseconds without affecting page performance. React periodically compares the Virtual DOM and the HTML DOM. It then computes a

diff, which it applies to the HTML DOM to make it match the Virtual DOM. This way, React does its best to ensure your application is rendered at a consistent 60 frames per second, meaning that users experience little or no lag.

Enough chitchat! Let's get our hands dirty ...

Start a Blank React Project

As per the prerequisites, I assume you already have a NodeJS environment setup. Let's first install or update npm to the latest version.

```
$ npm i -g npm
```

Next, we're going to install a tool, Create React App, that will allow us to create our first React project:

```
$ npm i -g create-react-app
```

Navigate to your project's root directory and create a new React project using the tool we just installed:

```
$ create-react-app message-app

…
Success! Created message-app at /home/mike/Projects/github/message-app
Inside that directory, you can run several commands:
```

```
    yarn start
    Starts the development server.

    yarn build
    Bundles the app into static files for production.

    yarn test
    Starts the test runner.

    yarn eject
    Removes this tool and copies build dependencies,
    configuration files and scripts into the app directory.
    If you do this, you can't go back!

We suggest that you begin by typing:

    cd message-app
    yarn start

Happy hacking!
```

Depending on the speed of your internet connection, this might take a while to complete if this is your first time running the `create-react-app` command. A bunch of packages gets installed along the way, which are needed to set up a convenient development environment --- including a web server, compiler and testing tools.

Navigate to the newly created `message-app` folder and open the `package.json` file.

```
{
    "name": "message-app",
    "version": "0.1.0",
    "private": true,
    "dependencies": {
    "react": "^15.6.1",
    "react-dom": "^15.6.1",
    "react-scripts": "1.0.12"
    },
    "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
    }
}
```

Surprise! You expected to see a list of all those packages listed as dependencies, didn't you? Create React App is an amazing tool that works behind the scenes. It creates a clear separation between your actual code and the development environment. You don't need to manually install Webpack to configure your project. Create React App has already done it for you, using the most common options.

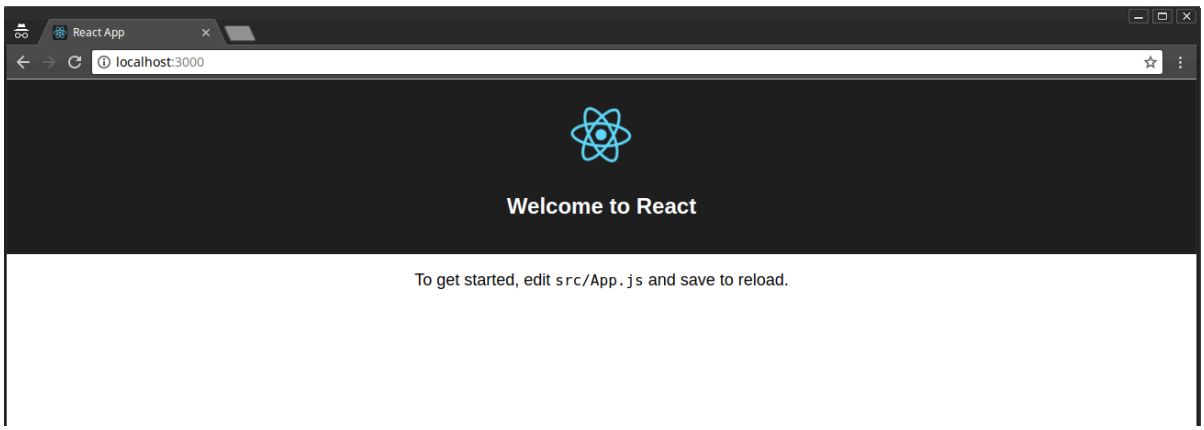
Let's do a quick test run to ensure our new project has no errors:

```
$ yarn start

Starting development server…
```

```
Compiled successfully!

You can now view message-app in the browser.

    Local:            http://localhost:3000/
    On Your Network:  http://10.0.2.15:3000/

Note that the development build is not optimized.
To create a production build, use yarn build.
```

If you don't have Yarn, just substitute with npm like this: `npm start`. For the rest of the article, use `npm` in place of `yarn` if you haven't installed it.

Your default browser should launch automatically, and you should get a screen like this:



One thing to note is that Create React App supports **hot reloading**. This means any changes we make on the code will cause the browser to automatically refresh. For now, let's first stop the development server by pressing `Ctrl + C`. This step isn't necessary, I'm just showing you how to kill the development

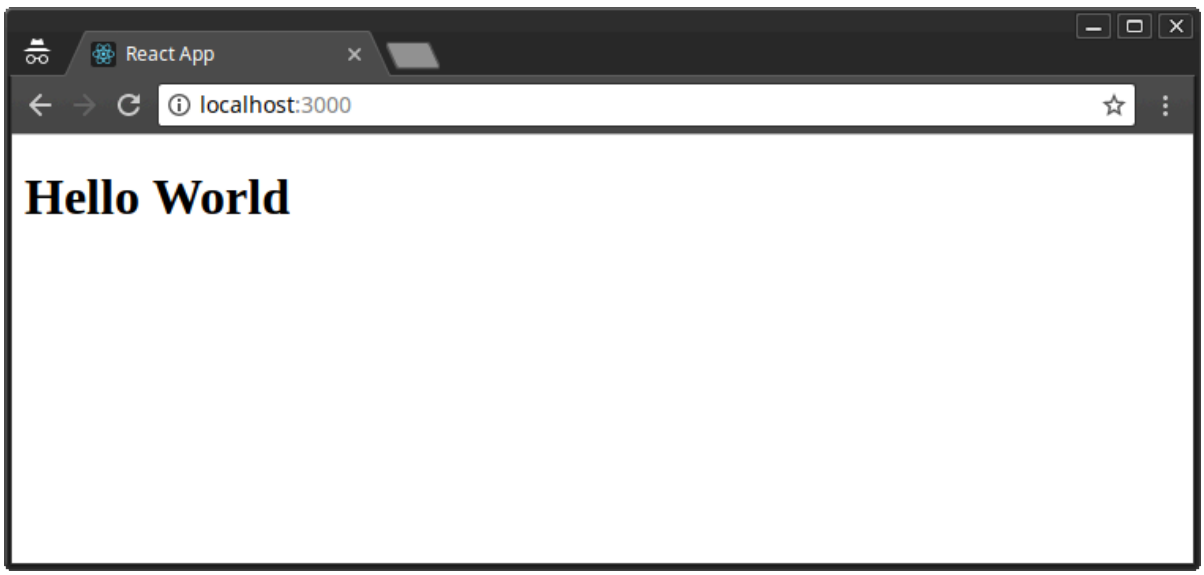
server. Once the server has stopped, delete everything in the `src` folder. We'll create all the code from scratch so that you can understand everything inside the `src` folder.

Introducing JSX Syntax

Inside the `src` folder, create an `index.js` file and place the following code in it:

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

        ReactDOM.render(<h1>Hello World</h1>,
         document.getElementById('root'));
```

Start the development server again using `yarn start` or `npm start`. Your browser should display the following content:



This is the most basic "Hello World" React example. The `index.js` file is the root of your project where React components will be rendered. Let me explain how the code works:

- Line 1: React package is imported to handle JSX processing
- Line 2: ReactDOM package is imported to render React components.
- Line 4: Call to render function
  - `<h1>Hello World</h1>`: a JSX element
  - `document.getElementById('root')`: HTML container

The HTML container is located in `public/index.html` file. On line 28, you should see `<div id="root"></div>`. This is known as the **root DOM** because everything inside it will be managed by the **React DOM**.

JSX (JavaScript XML) is a syntax expression that allows JavaScript to use tags such as `<div>`, `<h1>`, `<p>`, `<form>`, and `<a>`. It does look a lot like HTML, but there are some key differences. For example, you can't use a `class` attribute, since it's a JavaScript keyword. Instead, `className` is used in its place. Also, events such as `onclick` are spelled `onClick` in JSX. Let's now modify our Hello World code:

```
const element = <div>Hello World</div>;

ReactDOM.render(element, document.getElementById('root'));
```

I've moved out the JSX code into a variable named `element`. I've also replaced the `h1` tags with `div`. For JSX to work, you need to wrap your elements inside a single parent tag. This is necessary for JSX to work. Take a look at the following example:

```
const element = <span>Hello,</span> <span>Jane</span>;
```

The above code won't work. You'll get a syntax error telling you must enclose adjacent JSX elements in an enclosing tag. Basically, this is how you should enclose your elements:

```
const element = <div>
  <span>Hello, </span>
  <span>Jane</span>
</div>;
```

How about evaluating JavaScript expressions in JSX? Simple, just use curly braces like this:

```
const name = "Jane";
        const element = <p>Hello, {name}</p>
```

... or like this:

```
const user = {
    firstName: "Jane",
    lastName: "Doe"
}
```

```
const element = <p>Hello, {user.firstName} {user.lastName}</p>
```

Update your code and confirm that the browser is displaying "Hello, Jane Doe". Try out other examples such as `{ 5 + 2 }`. Now that you've got the basics of working with JSX, let's go ahead and create a React component.

Declaring React Components

The above example was a simplistic way of showing you how `ReactDOM.render()` works. Generally, we encapsulate all project logic within React components, which are then passed via the `ReactDOM.render` function.

Inside the `src` folder, create a file named `App.js` and type the following code:

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

class App extends Component {

    render(){
    return (
        <div>
        Hello World Again!
        </div>
    )
    }

}
```

```
export default App;
```

Here we've created a React Component by defining a JavaScript class that is a subclass of `React.Component`. We've also defined a render function that returns a JSX element. You can place additional JSX code within the `<div>` tags. Next, update `src/index.js` with the following code in order to see the changes reflected in the browser:

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

import App from './App';

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

First we import the `App` component. Then we render `App` using JSX format, like this: `<App/>`. This is required so that JSX can compile it to an element that can be pushed to the `React DOM`. After you've saved the changes, take a look at your browser to ensure it's rendering the correct thing.

Next, we'll look at how to apply styling.

Styling JSX Elements

There are are two ways of styling JSX elements:

1. JSX inline styling

2 External Stylesheets

Below is an example of how we can implement JSX inline styling:

```
// src/App.js

…
    render() {
    const headerStyle = {
        color: '#ff0000',
        textDecoration: 'underline'
    }
    return (
        <div>
        <h2 style={headerStyle}>Hello World Again!</h2>
        </div>
    )
    }
…
```

React styling looks a lot like regular CSS but there are key differences. For example, `headerStyle` is an object literal. We can't use semicolons like we normally do. Also, a number of CSS declarations have been changed in order to make them compatible with JavaScript syntax. For example, instead of `text-decoration`, we use `textDecoration`. Basically, use camel case for all CSS keys except for vendor prefixes such as `WebkitTransition`, which must start with a capital letter. We can also implement styling this way:

```
// src/App.js

…
    <h2 style={{color:'#ff0000'}}>Hello World Again!</h2>
…
```

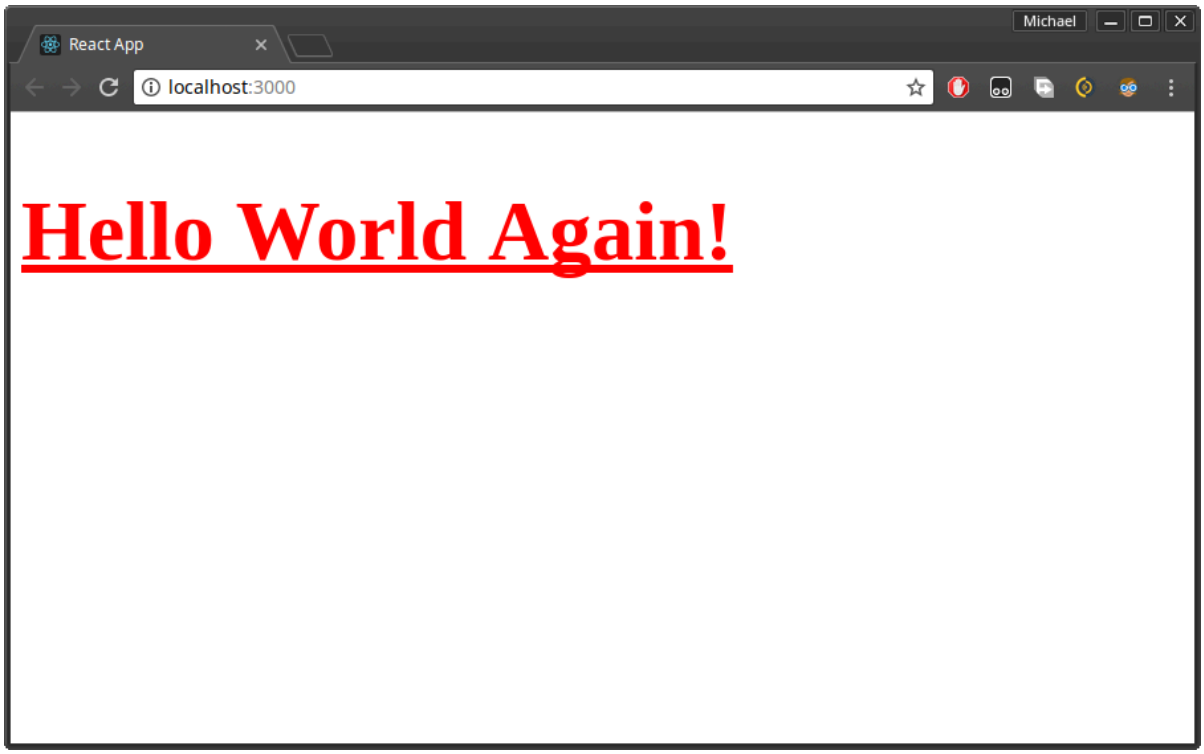
The second method is using external stylesheets. By default, external CSS stylesheets are already supported. If you want to use Sass or Less, please check the reference documentation on how to configure it. Inside the `src` folder, create a file named `App.css` and type the following code:

```
h2 {
    font-size: 4rem;
}
```

Add the following import statement to `src/App.js` on line 2 or 3:

```
// src/App.js
…
import './App.css';
…
```

After saving, you should see the text content on your browser dramatically change in size.



Now that you've learned how to add styling to your React project, let's go ahead and learn about stateless and stateful React components:

Stateless vs Stateful Components

In React, we generally deal with two types of data: **props** and **state**. `Props` are read-only and are set by a parent component. `State` is defined within a component and can change during the lifecyle of a component. Basically, stateless components (also known as **dumb** components) use `props` to store data, while stateful components (also known as **smart** components) use `state`. To gain a better understanding, let's examine the following practical examples. Inside the `src` folder, create a folder and name it `messages`. Inside that folder, create a file named `message-view.js` and type the following code to create a stateless component:

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

class MessageView extends Component {
    render() {
    return(
        <div className="container">
        <div className="from">
            <span className="label">From: </span>
            <span className="value">John Doe</span>
        </div>
        <div className="status">
            <span className="label">Status: </span>
            <span className="value"> Unread</span>
        </div>
        <div className="message">
            <span className="label">Message: </span>
            <span className="value">Have a great day!</span>
        </div>
        </div>
    )
    }
}

export default MessageView;
```

Next, add some basic styling in the `src/App.css` with the following code:

```
container {
    margin-left: 40px;
```

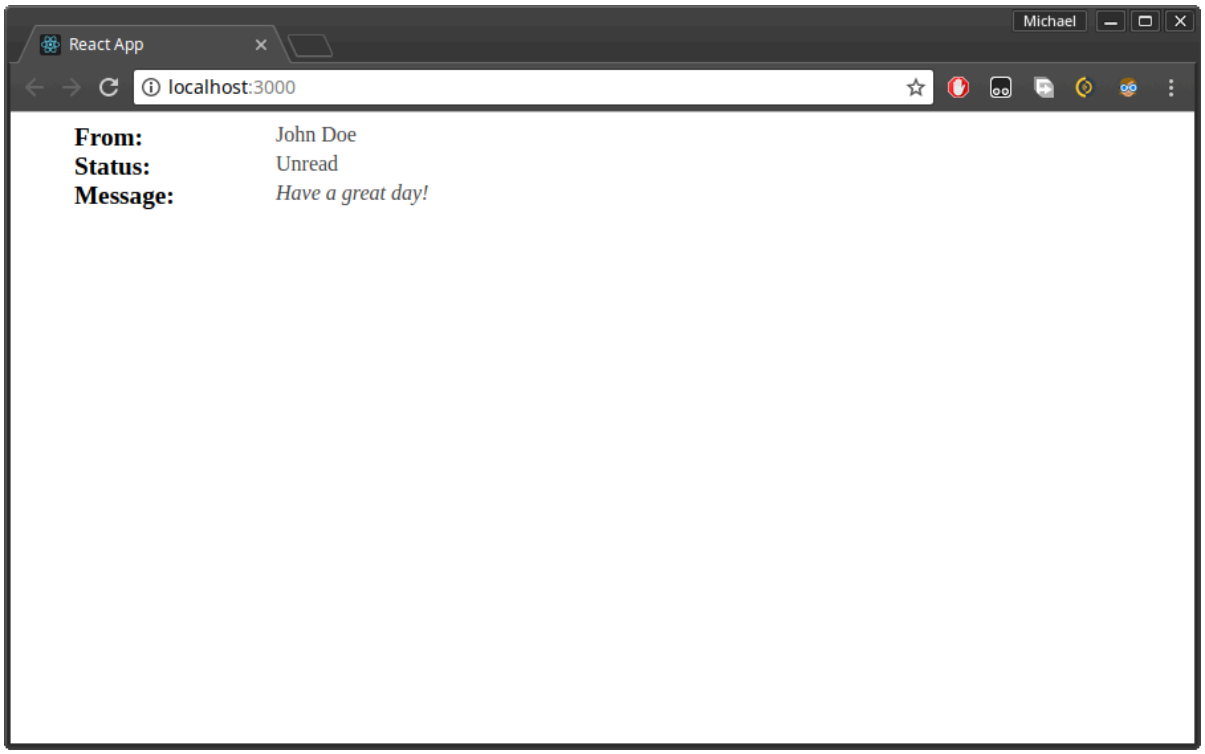
```
}

.label {
    font-weight: bold;
    font-size: 1.2rem;
}

.value {
    color: #474747;
    position: absolute;
    left: 200px;
}

.message .value {
    font-style: italic;
}
```

Finally, modify `src/App.js` so that the entire file looks like this:

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

import './App.css';
import MessageView from './messages/message-view';

class App extends Component {
    render(){
    return (
        <MessageView />
    )
```

```
    }
}

export default App;
```

By now, the code should be pretty self explanatory, as I've already explained the basic React concepts. Take a look at your browser now, and you should have the following result:



I'd also like to mention that it isn't necessary to use **object-oriented** syntax for stateless components, especially if you aren't defining a lifecycle function. We can rewrite `MessageView` using functional syntax like this:

```
// src/messages/message-view.js

import React from 'react';
import PropTypes from 'prop-types';

export default function MessageView({message}) {
    return (
    <div className="container">
        <div className="from">
        <span className="label">From: </span>
        <span className="value">{message.from}</span>
        </div>
        <div className="status">
        <span className="label">Status: </span>
        <span className="value">{message.status}</span>
        </div>
        <div className="message">
        <span className="label">Message: </span>
        <span className="value">{message.content}</span>
        </div>
    </div>
    );
}

MessageView.propTypes = {
    message: PropTypes.object.isRequired
}
```

Take note that I've removed the `Component` import, as this isn't required in the functional syntax. This style might be confusing at first, but you'll quickly learn it's faster to write React components this way.

You've successfully created a stateless React Component. It's not complete, though, as there's a bit more work that needs to be done for it to be properly integrated with a stateful component or container. Currently, the `MessageView` is displaying static data. We need to modify so that it can accept input parameters. We do this using `this.props`. We're going to assign a variable named `message` to props. We'll also mark the `message` variable as `required` using the `prop-types` package. This is to make it easier to debug our project as it grows. Update `message-view.js` with the following code:

```
// src/messages/message-view.js
…
import PropTypes from 'prop-types';

class MessageView extends Component {
    render() {

    const message = this.props.message;

    return(
        <div className="container">
        <div className="from">
            <span className="label">From: </span>
            <span className="value">{message.from}</span>
        </div>
        <div className="status">
            <span className="label">Status: </span>
            <span className="value">{message.status}</span>
        </div>
        <div className="message">
            <span className="label">Message: </span>
            <span className="value">{message.content}</span>
```

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

    // Mark message input parameter as required
    MessageView.propTypes = {
    message: PropTypes.object.isRequired
    }
}
…
```

Next, we'll create a stateful component which will act as a parent to `MessageView` component. We'll make use of the `state` data type to store a message which we'll pass on to `MessageView`. To do this, create `message-list.js` file inside `src/messages` and type the following code:

```
// src/messages/message-list.js

import React, { Component } from 'react';
import MessageView from './message-view';

class MessageList extends Component {

    state = {
    message: {
        from: 'Martha',
        content: 'I will be traveling soon',
```

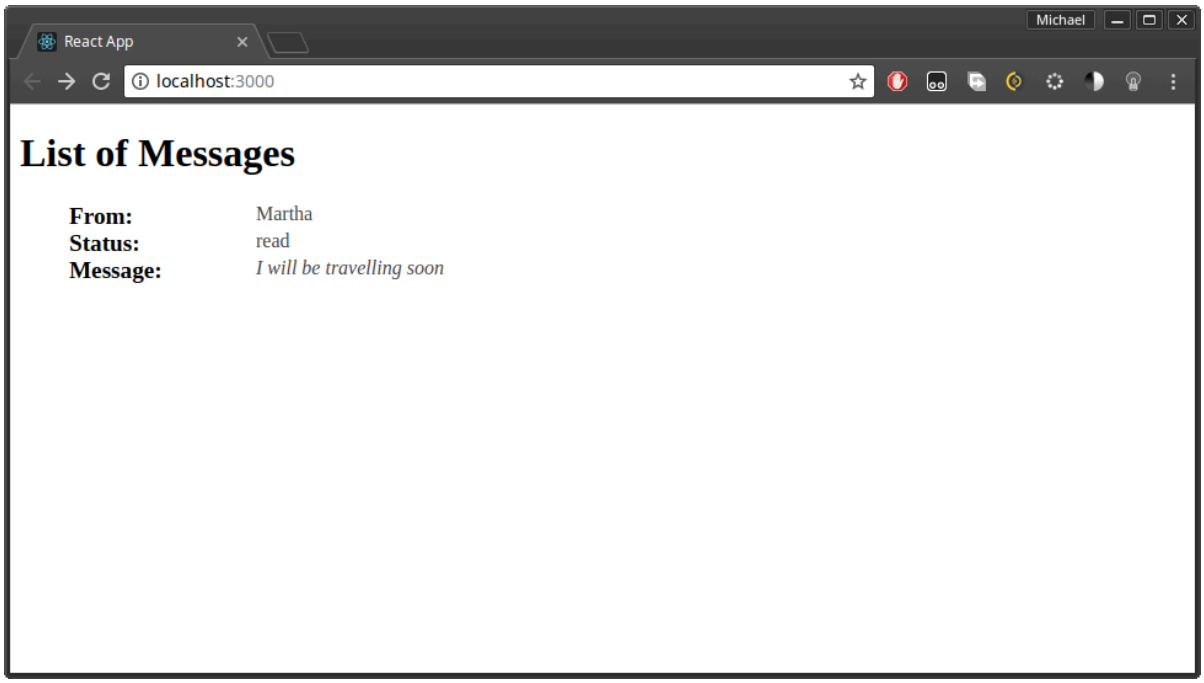
```
        status: 'read'
        }
    }

    render() {
    return(
        <div>
        <h1>List of Messages</h1>
        <MessageView message={this.state.message} />
        </div>
    )
    }
}

export default MessageList;
```

Next, update `src/App.js` such that `MessageList` gets rendered instead (which in turn renders its child component, `MessageView`).

```
// src/App.js
…
import MessageList from './messages/message-list';

class App extends Component {
    render(){
    return (
        <MessageList />
    )
    }
```

```
  }
...
```

After saving the changes, check your browser to see the result.



Now let's see how we can display multiple messages using `MessageView` instances. First, we'll change `state.message` to an array and rename it to `messages`. Then, we'll use the map function to generate multiple instances of `MessageView` each corresponding to a message in the `state.messages` array. We'll also need to populate a special attribute named key with a unique value such as `index`. React needs this in order to keep track of what items in the list have been changed, added or removed. Update `MessageList` code as follows:

```
class MessageList extends Component {

    state = {
    messages:  [
        {
        from: 'John',
        message: 'The event will start next week',
        status: 'unread'
        },
        {
        from: 'Martha',
        message: 'I will be traveling soon',
        status: 'read'
        },
        {
        from: 'Jacob',
        message: 'Talk later. Have a great day!',
        status: 'read'
        }
    ]
    }

    render() {
    const messageViews = this.state.messages.map
    ↪(function(message, index) {
        return(
        <MessageView key={index} message={message} />
        )
    })
    return(
        <div>
        <h1>List of Messages</h1>
```

```
            {messageViews}
            </div>
        )
        }
    }
```

Check your browser to see the results:

As you can see, it's easy to define building blocks to create powerful and complex UI interfaces using React. Feel free to add more styling such as putting spacing and dividers between each `MessageView` instance.

Chapter

4

Getting React Projects Ready Fast with Pre-configured Builds

by Pavels Jelisejevs

Starting a new React project nowadays is not as simple as we'd like it to be. Instead of instantly diving into the code and bringing your application to life, you have to spend time configuring the right build tools to set up a local development environment, unit testing, and a production build. But there are projects where all you need is a simple setup to get things running quickly and with minimal effort.

Create React App provides just that. It's a CLI tool from Facebook that allows you to generate a new React project and use a pre-configured Webpack build for development. Using it, you'll never have to look at the Webpack config again.

How Does Create React App Work?

Create React App is a standalone tool that should be installed globally via npm, and called each time you need to create a new project:

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

To create a new project, run:

```
create-react-app react-app
```

Create React App will set up the following project structure:

```
.
├── .gitignore
├── README.md
```

```
├── package.json
├── node_modules
├── public
│   ├── favicon.ico
│   └── index.html
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    └── logo.svg
```

It will also add a `react-scripts` package to your project that will contain all of the configuration and build scripts. In other words, your project depends `react-scripts`, not on `create-react-app` itself. Once the installation is complete, you can start working on your project.

Starting a Local Development Server

The first thing you'll need is a local development environment. Running `npm start` will fire up a Webpack development server with a watcher that will automatically reload the application once you change something. Hot reloading, however, is only supported for styles.

The application will be generated with a number of features built-in.

ES6 and ES7

The application comes with its own Babel preset, babel-preset-react-app, to

support a set of ES6 and ES7 features. It even supports some of the newer features like async/await, and import/export statements. However, certain features, like decorators, have been intentionally left out.

Asset import

You can also import CSS files from your JS modules that allow you to bundle styles that are only relevant for the modules that you ship. The same thing can be done for images and fonts.

ESLint

During development, your code will also be run through ESLint, a static code analyzer that will help you spot errors during development.

Environment variables

You can use Node environment variables to inject values into your code at built-time. React-scripts will automatically look for any environment variables starting with `REACT_APP_` and make them available under the global `process.env`. These variables can be in a `.env` file for convenience:

```
REACT_APP_BACKEND=http://my-api.com
REACT_APP_BACKEND_USER=root
```

You can then reference them in your code:

```
fetch({process.env.REACT_APP_SECRET_CODE}/endpoint)
```

Proxying to a backend

If your application will be working with a remote backend, you might need to be able to proxy requests during local development to bypass CORS. This can be set up by adding a proxy field to your `package.json` file:

```
"proxy": "http://localhost:4000",
```

This way, the server will forward any request that doesn't point to a static file the given address.

Running Unit Tests

Executing `npm test` will run tests using Jest and start a watcher to re-run them whenever you change something:

```
PASS  src/App.test.js
    ✓ renders without crashing (7ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.123s, estimated 1s
Ran all test suites.

Watch Usage
    › Press p to filter by a filename regex pattern.
    › Press q to quit watch mode.
    › Press Enter to trigger a test run.
```

Jest is a test runner also developed by Facebook as an alternative to Mocha or Karma. It runs the tests on a Node environment instead of a real browser, but provides some of the browser-specific globals using jsdom.

Jest also comes integrated with your VCS and by default will only run tests on files changed since your last commit. For more on this, refer to "How to Test React Components Using Jest".

Creating a Production Bundle

When you finally have something you deploy, you can create a production bundle using `npm run build`. This will generate an optimized build of your application, ready to be deployed to your environment. The generated artifacts will be placed in the build folder:

```
.
├── asset-manifest.json
├── favicon.ico
├── index.html
└── static
    ├── css
    │   ├── main.9a0fe4f1.css
    │   └── main.9a0fe4f1.css.map
    ├── js
    │   ├── main.3b7bfee7.js
    │   └── main.3b7bfee7.js.map
    └── media
        └── logo.5d5d9eef.svg
```

The JavaScript and CSS code will be minified, and CSS will additionally be run through Autoprefixer to enable better cross-browser compatibility.

Deployment

React-scripts provides a way to deploy your application to GitHub pages by simply adding a homepage property to `package.json`. There's also a separate Heroku build pack.

Opting Out

If at some point you feel that the features provided are no longer enough for your project, you can always opt out of using react-scripts by running `npm run eject`. This will copy the Webpack configuration and build scripts from `react-scripts` into your project and remove the dependency. After that, you're free to modify the configuration however you see fit.

In Conclusion

If you're looking to start a new React project look no further. Create React App will allow you to quickly start working on your application instead of writing yet another Webpack config.

Chapter

5

Styling in React: From External CSS to Styled Components

by Chris Laughlin

While many aspects of building applications with React have been standardized to some degree, styling is one area where there are still a lot of competing options. Each has its pros and cons, and there's no clear best choice.

In this article, I'll provide a condensed overview of the progression in web application styling with regards to React components. I'll also give a brief introduction to styled-components.

Evolution of Styling in JavaScript

The initial release of CSS was in 1996, and not much has changed since. In its third major release, and with a fourth on the way, it has continued to grow by adding new features and has maintained its reputation as a fundamental web technology. CSS will always be the gold standard for styling web components, but how it's used is changing every day.

From the days when we could build a website from sliced up images to the times when custom, hand-rolled CSS could reflect the same as an image, the evolution of CSS implementation has grown with the movement of JavaScript and the web as a platform.

Since React's release in 2013, component-built web applications have become the norm. The implementation of CSS has, in turn, been questioned. The main argument against using CSS in-line with React has been the separation of concerns (SoC). SoC is a design principle that describes the division of a program into sections, each of which addresses a different concern. This principle was used mainly when developers kept the three main web technologies in separate files: styles (CSS), markup (HTML) and logic (JavaScript).

This changed with the introduction of JSX in React. The development team argued that what we had been doing was, in fact, *the separation of technologies, not concerns*. One could ask, since JSX moved the markup into the JavaScript code, why should the styles be separate?

As opposed to divorcing styles and logic, different approaches can be used to merge them in-line. An example of this can be seen below:

```
<button style="background: red; border-radius: 8px;
↪color: white;">Click Me</button>
```

In-line styles move the CSS definitions from the CSS file. This thereby removes the need to import the file and saves on bandwidth, but sacrifices readability, maintainability, and style inheritance.

CSS Modules

button.css

```
.button {
    background: red;
    border-radius: 8px;
    color: white;
}
```

button.js

```
import styles from './button.css';
document.body.innerHTML = `<button class="${styles.button}"
↪>test</button>`;
```

As we can see from the code example above, the CSS still lives in its own file. However, when CSS Modules is bundled with Webpack or another modern bundler, the CSS is added as a script tag to the HTML file. The class names are also hashed to provide a more granular style, resolving the problems that come with cascading style sheets.

The process of hashing involves generating a unique string instead of a class name. Having a class name of `btn` will result in a hash of `DhtEg` which prevents styles cascading and applying styles to unwanted elements.

index.html

```
<style>
.DhtEg {
    background: red;
    border-radius: 8px;
    color: white;
}
</style>

…

<button class="DhtEg">test</button>
```

From the example above we can see the style tag element added by CSS Modules, with the hashed class name and the DOM element we have that uses the hash.

Glamor

Glamor is a CSS-in-JS library that allows us to declare our CSS in the same file as

our JavaScript. Glamor, again, hashes the class names but provides a clean syntax to build CSS style sheets via JavaScript.

The style definition is built via a JavaScript variable that describes each of the attributes using camel case syntax. The use of camel case is important as CSS defines all attributes in *train case*. The main difference is the change of the attribute name. This can be an issue when copying and pasting CSS from other parts of our application or CSS examples. For example `overflow-y` would be changed to `overFlowY`. However, with this syntax change, Glamor supports media queries and shadow elements, giving more power to our styles:

button.js

```
import { css } from 'glamor';

const rules = css({
    background: red;
    borderRadius: 8px;
    color: 'white';
});

const button = () => {
    return <button {...rules}>Click Me</button>;
};
```

styled-components

styled-components is a new library that focuses on keeping React components and styles together. Providing a clean and easy-to-use interface for styling both React and React Native, styled-components is changing not only the implementation but the thought process of building styled React components.

styled-components can be installed from npm via:

```
npm install styled-components
```

Imported as any standard npm package:

```
import styled from 'styled-components';
```

Once installed, it's time to start making styled React components easy and enjoyable.

Building Generic Styled React Components

Styled React components can be built in many ways. The styled-components library provides patterns that enable us to build well-structured UI applications. Building from small UI components — such as buttons, inputs, typography and tabs — creates a more unified and coherent application.

Using our button example from before, we can build a generic button using styled-components:

```
const Button = styled.button`
    background: red;
    border-radius: 8px;
    color: white;
`;
```

```
class Application extends React.Component {
    render() {
    return (
        <Button>Click Me</Button>
    )
    }
}
```

Codepen Example

http://codepen.io/SitePoint/pen/bWpoPO

As we can see, we're able to create our generic button while keeping the CSS in-line with the JavaScript. styled-components provides a wide range of elements that we can style. We can do this by using direct element references or by passing strings to the default function.

```
const Button = styled.button`
    background: red;
    border-radius: 8px;
    color: white;
`;

const Paragraph = styled.p`
    background: green;
`;
```

```
const inputBg = 'yellow';
const Input = styled.input'yellow'`
    background: ${inputBg};
    color: black;
`;

const Header = styled('h1')`
    background: #65a9d7;
    font-size: 26px;
`Application extends React.Component {
    render() {
    return (
        <div>
        <Button>Click Me</Button>
        <Paragraph>Read ME</Paragraph>
        <Input
            placeholder="Type in me"
        />
        <Header>I'm a H1</Header>
        </div>
    )
    }
}
                "Type in me"
```

Codepen Example

http://codepen.io/SitePoint/pen/NjNaQo

The main advantage to this styling method is being able to write pure CSS. As seen in the Glamor example, the CSS attribute names have had to be changed to

camel case, as they were attributes of a JavaScript object.

styled-components also produces React friendly primitives that act like the existing elements. Harnessing JavaScript template literals allows us to use the full power of CSS to style components. As seen in the input element example, we can define external JavaScript variables and apply these to our styles.

With these simple components, we can easily build a style guide for our application. But in many cases, we'll also need more complicated components that can change based on external factors.

Customizable Styled React Components

The customizable nature of styled-components is the real strength. This could commonly be applied to a button that needs to change styles based on context. In this case, we have two button sizes — small and large. Below is the pure CSS method:

CSS

```
button {
    background: red;
    border-radius: 8px;
    color: white;
}

.small {
    height: 40px;
    width: 80px;
}

.medium {
```

```
    height: 50px;
    width: 100px;
}

.large {
    height: 60px;
    width: 120px;
}
```

JavaScript

```
class Application extends React.Component {
    render() {
    return (
        <div>
        <button className="small">Click Me</button>
        <button className="large">Click Me</button>
        </div>
    )
    }
}
```

Codepen Example

http://codepen.io/SitePoint/pen/eWZeOp

When we reproduce this using styled-components, we create a Button

component that has the basic default style for the background. As the component acts like a React component, we can make use of props and change the style outcome accordingly:

```
const Button = styled.button`
    background: red;
    border-radius: 8px;
    color: white;
    height: ${props => props.small ? 40 : 60}px;
    width: ${props => props.small ? 60 : 120}px;
`;

class Application extends React.Component {
    render() {
    return (
        <div>
        <Button small>Click Me</Button>
        <Button large>Click Me</Button>
        </div>
    )
    }
}
```

Codepen Example

http://codepen.io/SitePoint/pen/qmZVWm

Advanced Usage

styled-components provides the ability to create complex advanced

components, and we can use existing JavaScript patterns to compose components. The example below demonstrates how components are composed — in the use case of notification messages that all follow a basic style, but each type having a different background color. We can build a basic, styled component and compose on top to create advanced components:

```
const BasicNotification = styled.p`
    background: lightblue;
    padding: 5px;
    margin: 5px;
    color: black;
`;

const SuccessNotification = styled(BasicNotification)`
    background: lightgreen;
`;

const ErrorNotification = styled(BasicNotification)`
    background: lightcoral;
    font-weight: bold;
`;

class Application extends React.Component {
    render() {
    return (
        <div>
        <BasicNotification>Basic Message</BasicNotification>
        <SuccessNotification>Success Message</SuccessNotification>
        <ErrorNotification>Error Message</ErrorNotification>
        </div>
    )
    }
```

```
}
```

Codepen Example

https://codepen.io/SitePoint/pen/JNXOjW

As styled-components allows us to pass standard DOM elements and other components, we can compose advanced features from basic components.

Component Structure

From our advanced and basic example, we can then build a component structure. Most standard React applications have a components directory: we place our styled components in a `styledComponents` directory. Our `styledComponents` directory holds all the basic and composed components. These are then imported into the display components used by our application. The directory layout can be seen below:

```
src/
    components/
    addUser.js
    styledComponents/
        basicNotification.js
        successNotification.js
        errorNotification.js
```

Conclusion

As we've seen in this post, the ways in which we can style our components varies greatly — none of which is a clear winning method. This article has shown that styled-components has pushed the implementation of styling elements forward, and has caused us to question our thought processes with regards to our approach.

Every developer, including myself, has their favorite way of doing things, and it’s good to know the range of different methods out there to use depending on the application we're working on. Styling systems and languages have advanced greatly throughout the years, and there's no question that they are sure to develop further and change more in the future. It’s a very exciting and interesting time in front-end development.

Chapter 6

An Introduction to JSX

by Matt Burnett

When React was first introduced, one of the features that caught most people's attention (and drew the most criticism) was JSX. If you're learning React, or have ever seen any code examples, you probably did a double-take at the syntax. What is this strange amalgamation of HTML and JavaScript? Is this even real code?

Let's take a look at what JSX actually is, how it works, and why the heck we'd want to be mixing HTML and JS in the first place!

What is JSX?

Defined by the React Docs as an "extension to JavaScript" or “syntax sugar for calling `React.createElement(component, props, ...children))`”, JSX is what makes writing your React Components easy.

JSX is considered a domain-specific language (DSL), which can look very similar to a template language, such as Mustache, Thymeleaf, Razor, Twig, or others.

It doesn't render out to HTML directly, but instead renders to React Classes that are consumed by the Virtual DOM. Eventually, through the mysterious magic of the Virtual DOM, it will make its way to the page and be rendered out to HTML.

How Does it Work?

JSX is basically still just JavaScript with some extra functionality. With JSX, you can write code that looks very similar to HTML or XML, but you have the power of seamlessly mixing JavaScript methods and variables into your code. JSX is interpreted by a transpiler, such as Babel, and rendered to JavaScript code that the UI Framework (React, in this case) can understand.

Don't like JSX? That's cool. It's technically not required, and the React Docs actually include a section on using “React Without JSX”. Let me warn you right now, though, it's not pretty. Don't believe me? Take a look.

JSX:

```
class SitePoint extends Component {
    render() {
    return (
        <div>My name is <span>{this.props.myName}</span></div>
    )
    }
}
```

React Sans JSX:

```
class SitePoint extends Component {
    render() {
    return React.createElement(
        "div",
        null,
        "My name is",
        React.createElement(
            "span",
            null,
            this.props.myName
        )
    )
    }
}
```

Sure, looking at those small example pieces of code on that page you might be

thinking, "Oh, that's not so bad, I could do that." But could you imagine writing an entire application like that?

The example is just two simple nested HTML elements, nothing fancy. Basically, just a nested `Hello World`. Trying to write your React application without JSX would be extremely time consuming and, if you're like most of us other developers out here working as characters in DevLand™, it will very likely quickly turn into a convoluted spaghetti code mess. Yuck!

Using frameworks and libraries and things like that are meant to make our lives easier, not harder. I'm sure we've all seen the overuse and abuse of libraries or frameworks in our careers, but using JSX with your React is definitely not one of those cases.

What About Separation of Concerns?

Now, for those of us who learned to not mix our HTML with our JS --- how bad it was, and how if we did, our applications would be haunted by the Bug Gods of the Apocalypse --- mixing your HTML inside of your JavaScript probably seems like all kinds of wrong. After all, you have to maintain a Separation of Concerns at all costs! The world depends on it!

I know, I was there, I was you.

But, as weird as it may seem, it's really *not that bad*, and your concerns are still separated. As someone who's been working with React for long enough to be very comfortable with it, using small pieces of HTML/XML inside your JavaScript codebase is really kinda ... magical. It takes the management of HTML out of the equation, and leaves you with nice, solid code. Just code. It helps to ease the pain of trying to use a markup language --- which was designed for building word documents --- for building applications, and brings it back to the control of the application's code.

Another issue of not separating out your JavaScript from your HTML: what if

your JS fails to download or run, and the user is only left with HTML/CSS to render? If you're building your application the React way, then your HTML (and maybe even your CSS if you're using WebPack), is bundled in with your JavaScript itself. So the user really only has one small HTML file to download, then a large JavaScript payload that contains everything else.

Worrying about search engines and SEO is unfortunately still a legitimate concern. We all know that the major search engines now render JavaScript, but initial rendering with JavaScript can still be slower, which could have an effect on your overall ranking. To mitigate this, it's possible to do an initial render of your React on the server before sending it to the client. This will not only allow search engines to pull your site quicker, but also provide your users with a quicker First Meaningful Paint.

Doing this can become complicated quickly, but at the time of this writing, SitePoint itself actually uses this method. Brad Denver did a fantastic write-up of how it was done here: "Universal Rendering: How We Rebuilt SitePoint".

Rendering server-side is still only going to help with failed loads (which are uncommon) or slow load times (much more common). It won't help with users who completely disable JavaScript. Sorry, but a React app is still a JavaScript-based application. You can do a lot behind the scenes, but once you start mixing in app functionality and state management (e.g. Flux, Redux, or MobX), your time investment vs potential payoff starts going negative.

Not Just for React

JSX with React is pretty great stuff. But what if you're using another framework or library, but still want to use it? Well, you're in luck --- because JSX technically isn't tied to React. It's still just DSL or "syntax sugar", remember? Vue.js actually supports JSX out of the box, and there have been attempts to use it with other frameworks such as Angular 2 and Ember.

I hope this JSX quick introduction helped to give you a better understanding of

just what JSX is, how it can help you, and how it can be used to create your applications. Now get back out there and make some cool stuff!

Chapter

7

Working with Data in React: Properties & State

Eric Greene

Managing data is essential to any application. Orchestrating the flow of data through the user interface (UI) of an application can be challenging. Often, today's web applications have complex UIs such that modifying the data in one area of the UI can directly and indirectly affect other areas of the UI. Two-way data binding via Knockout.js and Angular.js are popular solutions to this problem.

For some applications (especially with a simple data flow), two-way binding can be a sufficient and quick solution. For more complex applications, two-way data binding can prove to be insufficient and a hindrance to effective UI design. React does not solve the larger problem of application data flow (although Flux does), but it does solve the problem of data flow within a single component.

Within the context of a single component, React solves both the problem of data flow, as well as updating the UI to reflect the results of the data flow. The second problem of UI updates is solved using a pattern named Reconciliation which involves innovative ideas such as a Virtual DOM. The next article will examine Reconciliation in detail. This article is focused on the first problem of data flow, and the kinds of data React uses within its components.

Kinds of Component Data

Data within React Components is stored as either properties or state.

Properties are the input values to the component. They are used when rendering the component and initializing state (discussed shortly). After instantiating the component, properties should be considered immutable. Property values can only be set when instantiating the component, then when the component is re-rendered in the DOM, React will compare between the old and new property values to determine what DOM updates are required.

Codepen Example

Here is a CodePen demonstration of setting the property values and updating the DOM in consideration of updated property values.

State data can be changed by the component and is usually wired into the component's event handlers. Typically, updating the state triggers React components re-render themselves. Before a component is initialized, its state must be initialized. The initialized values can include constant values, as well as, property values (as mentioned above).

In comparison with frameworks such as Angular.js, properties can be thought of as one-way bound data, and state as two-way bound data. This is not a perfect analogy since Angular.js uses one kind of data object which is used two different ways, and React is using two data objects, each with their specific usages.

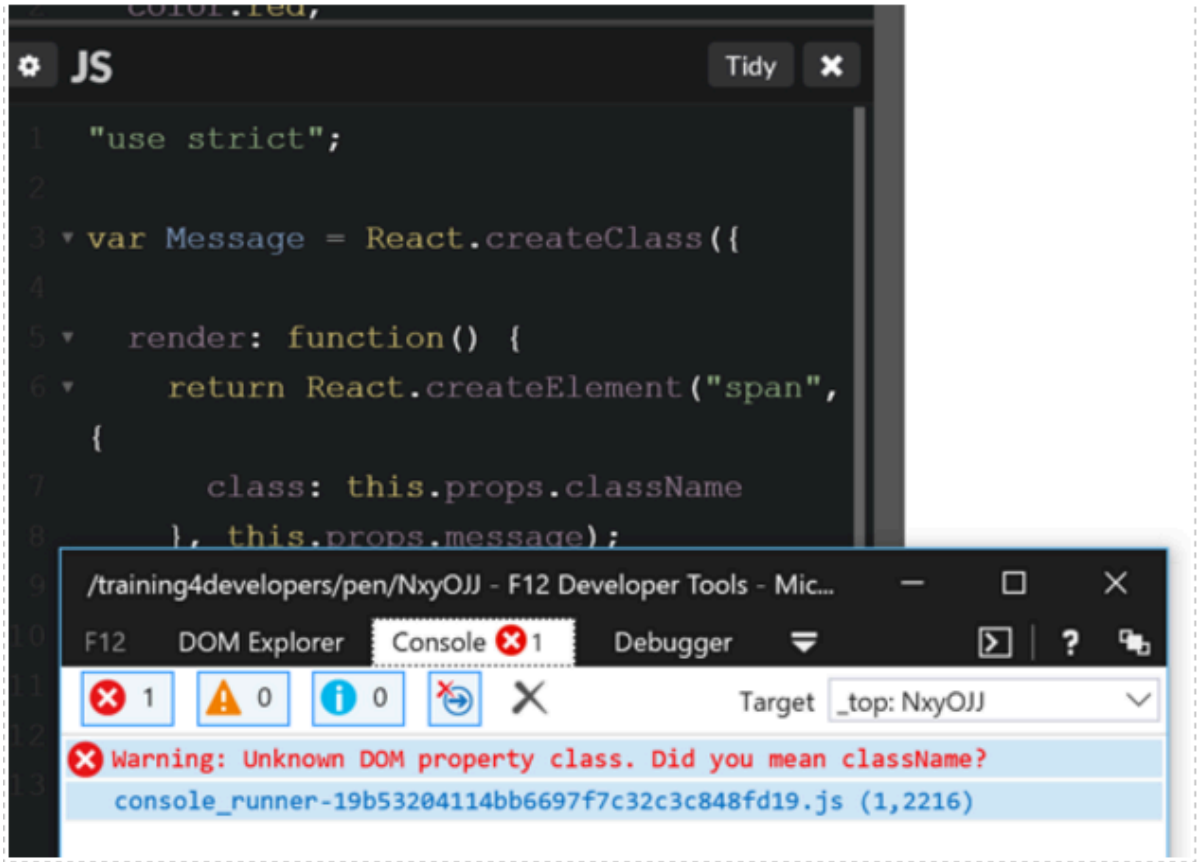
Properties

My previous React article covered the syntax to specify and access properties. The article explored the use of JavaScript and JSX with static as well as dynamic properties in various code demonstrations. Expanding on the earlier exploration, let's look at some interesting details regarding working with properties.

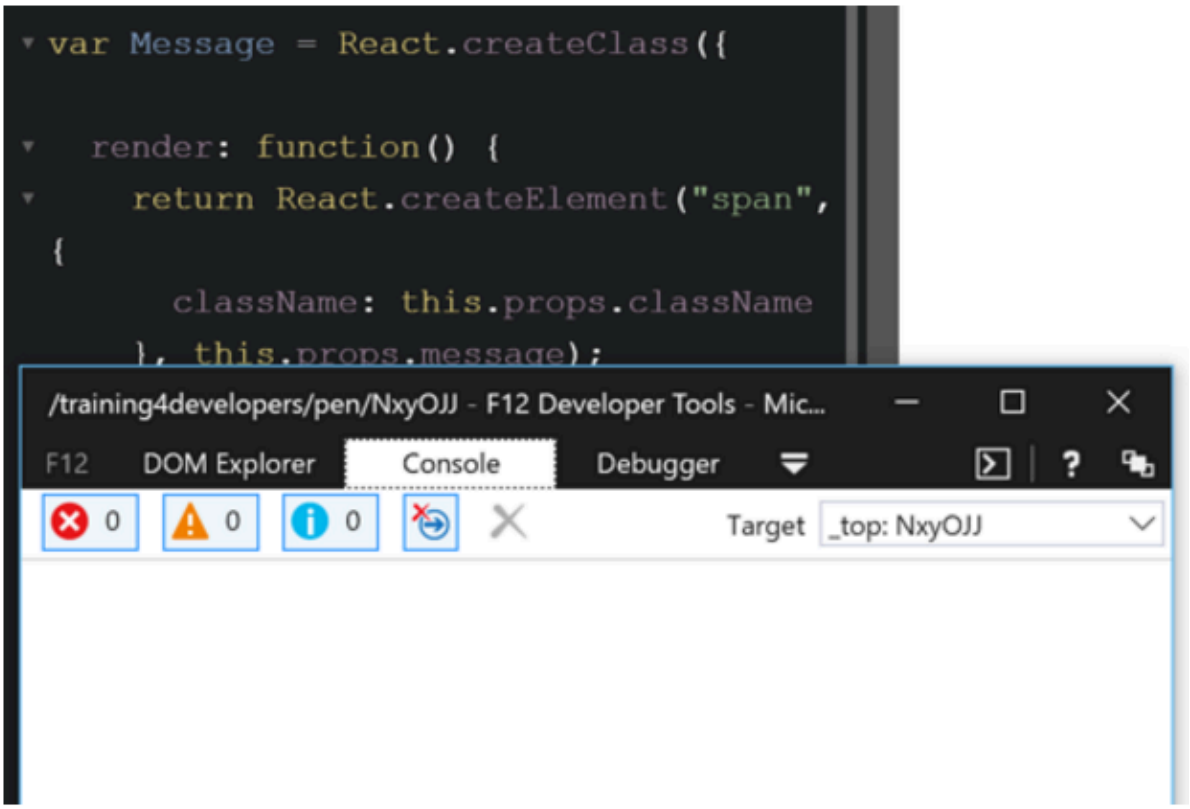
When adding a CSS class name to a component, the property name `className` must be used, rather than `class` must be used. React requires this because ES2015 identifies the word `class` as a reserved keyword and is used for defining objects. To avoid confusion with this keyword, the property name `className` is used. If a property named `class` is used, React will display a helpful console message informing the developer that the property name needs to be changed to `className`.

Observe the incorrect `class` property name, and the helpful warning message

displayed in the Microsoft Edge console window.



Changing the `class` property to `className`, results in the warning message not being displayed.



When the property name is changed from `class` to `className` the warning message does not appear. Check out the complete CodePen demonstration:

Codepen Example

https://codepen.io/SitePoint/pen/akGxGW/.

In addition to property names such as `className`, React properties have other interesting aspects. For example, mutating component properties is an anti-pattern. Properties can be set when instantiating a component, but should not be mutated afterwards. This includes changing properties after instantiating the component, as well as after rendering it. Mutating values within a component are considered state, and tracked with the `state` property rather than the `props` property.

In the following code sample, `SomeComponent` is instantiated with `createElement`, and then the property values are manipulated afterwards.

JavaScript:

```
var someComponent = React.createElement(SomeComponent);

someComponent.props.prop1 = "some value";
someComponent.props.prop2 = "some value";
```

JSX:

```
var someComponent = &lt;SomeComponent /&gt;;

someComponent.props.prop1 = "some value";
someComponent.props.prop2 = "some value";
```

Manipulating the `props` of the instantiated component could result in errors that would be hard to trace. Also, changing the properties does not trigger an update to the component, resulting in the component and the properties could be out of sync.

Instead, properties should be set as part of the component instantiation process, as shown below.

JavaScript:

```
var someComponent = React.createElement(SomeComponent, {
```

```
    prop1: "some value",
    prop2: "some value"
});
```

JSX:

```
var someComponent = <SomeComponent prop1="some value"
↪prop2="some value" />
```

The component can then be re-rendered at which point React will perform its Reconciliation process to determine how the new property values affect the DOM. Then, the DOM is updated with the changes.

See the first CodePen demonstration at the top of this article for a demonstration of the DOM updates.

State

State represents data that is changed by a component, usually through interaction with the user. To facilitate this change, event handlers are registered for the appropriate DOM elements. When the events occur, the updated values are retrieved from the DOM, and notify the component of the new state. Before the component can utilize state, the state must be initialized via the `getInitialState` function. Typically, the `getInitialState` function initializes the state using static values, passed in properties, or another data store.

```
var Message = React.createClass({
```

```
    getInitialState: function() {
        return { message: this.props.message };
    },
```

Once the state is initialized, the state values can be used like property values when rendering the component. To capture the user interactions which update the state, event handlers are registered. To keep the React components self-contained, event handler function objects can be passed in as properties or defined directly on the component object definition itself.

Codepen Example

http://codepen.io/SitePoint/pen/JKvVvy/.

One of the benefits of React is that standard HTML events are used. Included with standard HTML events is the standard HTML Event object. Learning special event libraries, event handlers, or custom event objects is not needed. Because modern browsers are largely compatible, intermediary cross-browser libraries such as jQuery are not needed.

To handle the state changes, the `setState` function is used to set the new value on the appropriate state properties. Calling this function causes the component to re-render itself.

As shown below in the Visual Studio Code editor, the `setState` function is called from the `_messageChange` event handler.

state-updates.jsx blog-post-2/src/www/js

```
(function() {

  "use strict";

  var Message = React.createClass({

    getInitialState: function() {
      return { message: this.props.message };
    },

    _messageChange: function(e) {
      this.setState({ message: e.target.value });
    },
```

Conclusion

React components provide two mechanisms for working with data: properties and state. Dividing data between immutable properties and mutable state more clearly identifies the role of each kind of data, and the component's relationship to it. In general, properties are preferred because they simplify the flow of data. State is useful for capturing data updates resulting from user interactions and other UI events.

The relationship between properties and state facilitate the flow of data through a component. Properties can be used to initialize state, and state values can be used to set properties when instantiating and rendering a component. New values from user interaction are captured via state, and then used to update the properties.

The larger flow of data within an application is accomplished via a pattern named Flux.

Chapter

8

React for Angular Developers

by Mark Brown

This article is for developers who are familiar with Angular 1.x and would like to learn more about React. We'll look at the different approaches they take to building rich web applications, the overlapping functionality and the gaps that React doesn't attempt to fill. Skip it if you're not familiar with Angular.

After reading, you'll have an understanding of the problems that React sets out to solve and how you can use the knowledge you have already to get started using React in your own projects.

Frameworks vs Libraries

Angular is a *framework*, whereas React is a *library* focused only on the view layer. There are costs and benefits associated with both using frameworks and a collection of loosely coupled libraries.

Frameworks try to offer a complete solution, and they may help organize code through patterns and conventions if you're part of a large team. However, having a large API adds cognitive load when you're writing, and you'll spend a lot more time reading documentation and remembering patterns --- especially in the early days when you're still learning.

Using a collection of loosely coupled libraries with small APIs is easier to learn and master, but it means that when you run into problems you'll need to solve them with more code or pull in external libraries as required. This usually results in you having to write *your own* framework to reduce boilerplate.

Out Of The Box

Angular gives you a rich feature set for building web applications. Among its features are:

- HTML templates with dynamic expressions in double curlies `{{ }}`
- built-in directives like `ng-model`, `ng-repeat` and `ng-class` for extending the capability of HTML

- controllers for grouping logic and passing data to the view
- two-way binding as a simple way to keep your view and controller in sync
- a large collection of modules like `$http` for communicating with the server and `ngRoute` for routing
- custom directives for creating your own HTML syntax
- dependency injection for limiting exposure of objects to specific parts of the application
- services for shared business logic
- filters for view formatting helpers.

React, on the other hand, gives you:

- JSX syntax for templates with JavaScript expressions in single curlies { }
- components, which are most like Angular's element directives.

React is unopinionated when it comes to the rest of your application structure and it encourages the use of standard JavaScript APIs over framework abstractions. Rather than providing a wrapper like $http for server communication, you can use fetch() instead. You're free to use constructs like services and filters, but React won't provide an abstraction for them. You can put them in JavaScript modules and require them as needed in your components.

So, while Angular gives you a lot more abstractions for common tasks, React deliberately avoids this to keep you writing standard JavaScript more often and to use external dependencies for everything else.

Bootstrapping

Initializing Angular apps requires a module, a list of dependencies and a root element.

```
let app = angular.module('app', [])
let root = document.querySelector('#root');
angular.element(root).ready(function() {
    angular.bootstrap(root, ['app']);
});
```

The entry point for React is rendering a component into a root node. It's possible to have multiple root components, too:

```
let root = document.querySelector('#root');
ReactDOM.render(<App />, root)
```

Templates

The anatomy of an Angular view is complex and has many responsibilities. Your HTML templates contain a mix of directives and expressions, which tie the view and the associated controllers together. Data flows throughout multiple contexts via `$scope`.

In React, it's components *all the way down*, data flows in one direction from the top of the component tree down to the leaf nodes. JSX is the most common syntax for writing components, transforming a familiar XML structure into JavaScript. Whilst this does *resemble* a template syntax, it compiles into nested function calls.

```
const App = React.createClass({
```

```
    render: function() {
    return (
        <Component>
            <div>{ 2 + 1 }</div>
            <Component prop="value" />
            <Component time={ new Date().getTime() }>
                <Component />
            </Component>
        </Component>
        )
    }
})
```

The compiled code below should help clarify how the JSX expressions above map to `createElement(component, props, children)` function calls:

```
var App = React.createClass({
    render: function render() {
        return React.createElement(
            Component,
                null,
                React.createElement("div", null, 2 + 1),
                React.createElement(Component, { prop: "value" }),
            React.createElement(
            Component,
            { time: new Date().getTime() },
            React.createElement(Component, null)
            )
        );
```

```
    }
});
```

Template Directives

Let's look at how some of Angular's most used template directives would be written in React components. Now, React doesn't have templates, so these examples are JSX code that would sit inside a component's `render` function. For example:

```
class MyComponent extends React.Component {
    render() {
        return (
            // JSX lives here
        )
    }
}
```

ng-repeat

```
<ul>
    <li ng-repeat="word in words">{ word }</li>
</ul>
```

We can use standard JavaScript looping mechanisms such as `map` to get an array

of elements in JSX.

```
<ul>
    { words.map((word)=> <li>{ word }</li> )}
</ul>
```

ng-class

```
<form ng-class="{ active: active, error: error }">
</form>
```

In React, we're left to our own devices to create our space-separated list of classes for the `className` property. It's common to use an existing function such as Jed Watson's classNames for this purpose.

```
<form className={ classNames({active: active, error: error}) }>
</form>
```

The way to think about these attributes in JSX is as if you're setting properties on those nodes directly. That's why it's `className` rather than the `class` attribute name.

```
formNode.className = "active error";
```

ng-if

```
<div>
    <p ng-if="enabled">Yep</p>
</div>
```

`if … else` statements don't work inside JSX, because JSX is just syntactic sugar for function calls and object construction. It's typical to use ternary operators for this or to move conditional logic to the top of the render method, outside of the JSX.

```
// ternary
<div>
    { enabled ? <p>Enabled</p> : null }
</div>

// if/else outside of JSX
let node = null;
if (enabled) {
    node = <p>Enabled</p>;
}
<div>{ node }</div>
```

ng-show / ng-hide

```
<p ng-show="alive">Living</p>
```

```
<p ng-hide="alive">Ghost</p>
```

In React, you can set style properties directly or add a utility class, such as `.hidden { display: none }`, to your CSS for the purpose of hiding your elements (which is how Angular handles it).

```
<p style={ display: alive ? 'block' : 'none' }>Living</p>
<p style={ display: alive ? 'none' : 'block' }>Ghost</p>

<p className={ classNames({ hidden: !alive }) }>Living</p>
<p className={ classNames({ hidden: alive }) }>Ghost</p>
```

You've got the hang of it now. Instead of a special template syntax and attributes, you'll need to use JavaScript to achieve what you want instead.

An Example Component

React's *Components* are most like Angular's *Directives*. They're used primarily to abstract complex DOM structures and behavior into reusable pieces. Below is an example of a slideshow component that accepts an array of slides, renders a list of images with navigational elements and keeps track of its own `activeIndex` state to highlight the active slide.

```
<div ng-controller="SlideShowController">
    <slide-show slides="slides"></slide-show>
</div>
```

```
app.controller("SlideShowController", function($scope) {
    $scope.slides = [{
        imageUrl: "allan-beaver.jpg",
        caption: "Allan Allan Al Al Allan"
    }, {
        imageUrl: "steve-beaver.jpg",
        caption: "Steve Steve Steve"
    }];
});

app.directive("slideShow", function() {
    return {
        restrict: 'E',
        scope: {
            slides: '='
        },
        template: `
            <div class="slideshow">
                <ul class="slideshow-slides">
                <li ng-repeat="slide in slides" ng-class="
                    ↪{ active: $index == activeIndex }">
                    <figure>
                    <img ng-src="{{ slide.imageUrl }}" />
                    <figcaption ng-show="slide.caption"
                        ↪>{{ slide.caption }}
                        ↪</figcaption>
                </figure>
                </li>
                </ul>
                <ul class="slideshow-dots">
                    <li ng-repeat="slide in slides" ng-class
                        ↪="{ active: $index == activeIndex }">
```

```
                        <a ng-click="jumpToSlide($index)">
                        ↪{{ $index + 1 }}</a>
                    </li>
                </ul>
            </div>
        `,
        link: function($scope, element, attrs) {
            $scope.activeIndex = 0;

            $scope.jumpToSlide = function(index) {
                $scope.activeIndex = index;
            };
        }
    };
});
```

The Slideshow Component in Angular

Codepen Example

http://codepen.io/SitePoint/pen/QyNJxO/

This component in React would be rendered inside another component and passed the slides data via props.

```
let _slides = [{
    imageUrl: "allan-beaver.jpg",
    caption: "Allan Allan Al Al Allan"
```

```
}, {
    imageUrl: "steve-beaver.jpg",
    caption: "Steve Steve Steve"
}];

class App extends React.Component {
    render() {
        return <SlideShow slides={ _slides } />
    }
}
```

React components have a local scope in `this.state`, which you can modify by calling `this.setState({ key: value })`. Any changes to state causes the component to re-render itself.

```
class SlideShow extends React.Component {
    constructor() {
            super()
            this.state = { activeIndex: 0 };
        }
        jumpToSlide(index) {
            this.setState({ activeIndex: index });
        }
        render() {
            return (
                <div className="slideshow">
                    <ul className="slideshow-slides">
                        {
                        this.props.slides.map((slide, index) => (
```

```
                        <li className={ classNames
                            ↪({ active: index == this.
                            ↪state.activeIndex }) }>
                            <figure>
                                <img src={ slide.imageUrl } />
                                { slide.caption ?
                                ↪<figcaption>
                                ↪{ slide.caption }
                                ↪</figcaption> : null }
                            </figure>
                        </li>
                        ))
                    }
                </ul>
                <ul className="slideshow-dots">
                    {
                    this.props.slides.map((slide, index) => (
                        <li className={ (index == this.
                            ↪state.activeIndex) ?
                            ↪'active': '' }>
                            <a onClick={ (event)=> this.
                            ↪jumpToSlide(index) }>
                            ↪{ index + 1 }</a>
                        </li>
                        ))
                }
            </ul>
        </div>
    );
  }
}
```

Events in React look like old-school inline event handlers such as `onClick`. Don't feel bad, though: under the hood it does the right thing and creates highly performant delegated event listeners.

The Slideshow Component in React

Codepen Example

http://codepen.io/SitePoint/pen/ZQWmoj/

Two-Way Binding

Angular's trusty `ng-model` and `$scope` form a link where the data flows back and forth between a form element and properties on a JavaScript object in a controller.

```
app.controller("TwoWayController", function($scope) {
    $scope.person = {
        name: 'Bruce'
    };
});
```

```
<div ng-controller="TwoWayController">
    <input ng-model="person.name" />
    <p>Hello {{ person.name }}!</p>
</div>
```

React eschews this pattern in favor of a one-way data flow instead. The same

types of views can be built with both patterns though.

```
class OneWayComponent extends React.Component {
    constructor() {
        super()
        this.state = { name: 'Bruce' }
    }
    change(event) {
        this.setState({ name: event.target.value });
    }
    render() {
        return (
            <div>
                <input value={ this.state.name }
                ↪onChange={ (event)=> this.change(event) } />
                <p>Hello { this.state.name }!</p>
            </div>
        );
    }
}
```

The `<input>` here is called a "controlled input". This means its value is only ever changed when the `render` function is called (on every key stroke in the example above). The component itself is called "stateful" because it manages its own data. This isn't recommended for the majority of components. The ideal is to keep components "stateless" and have data passed to them via `props` instead.

Codepen Example

http://codepen.io/SitePoint/pen/BjKGPW/

Typically, a stateful Container Component or Controller View sits at the top of the tree with many stateless child components underneath. For more information on this, read What Components Should Have State? from the docs.

Call Your Parents

Whilst data flows down in one direction, it's possible to call methods on the parent through callbacks. This is usually done in response to some user input. This flexibility gives you a lot of control when refactoring components to their simplest presentational forms. If the refactored components have no state at all, they can be written as pure functions.

```
// A presentational component written as a pure function
const OneWayComponent = (props)=> (
    <div>
        <input value={ props.name } onChange={ (event)=> props.
            ↪onChange(event.target.value) } />
        <p>Hello { props.name }!</p>
    </div>
);

class ParentComponent extends React.Component {
    constructor() {
        super()
        this.state = { name: 'Bruce' };
    }
    change(value) {
        this.setState({name: value});
    }
    render() {
        return (
            'Bruce'<div>
```

```
                <OneWayComponent name={ this.state.name }
                ↪onChange={ this.change.bind(this) } />
                <p>Hello { this.state.name }!</p>
            </div>
        )
    }
}
```

This might seem like a round-about pattern at first if you're familiar with two-way data binding. The benefit of having a lot of small presentational "dumb" components that just accept data as `props` and render them is that they are simpler by default, and simple components have *far* fewer bugs. This also prevents the UI from being in an inconsistent state, which often occurs if data is in multiple places and needs to be maintained separately.

Dependency Injection, Services, Filters

JavaScript Modules are a much better way to handle dependencies. You can use them today with a tool like Webpack, SystemJS or Browserify.

```
// An Angular directive with dependencies
app.directive('myComponent', ['Notifier', '$filter',
↪function(Notifier, $filter) {
    const formatName = $filter('formatName');

    // use Notifier / formatName

}]
```

```
// ES6 Modules used by a React component
import Notifier from "services/notifier";
import { formatName } from "filters";

class MyComponent extends React.Component {

    "services/notifier"// use Notifier / formatName

}
```

Sounds Great. Can I Use Both!?

Yes! It's possible to render React components inside an existing Angular application. Ben Nadel has put together a good post with screencast on how to render React components inside an Angular directive. There's also ngReact, which provides a `react-component` directive for acting as the glue between React and Angular.

If you've run into rendering performance problems in certain parts of your Angular application, it's possible you'll get a performance boost by delegating some of that rendering to React. That being said, it's not ideal to include two large JavaScript libraries that solve a lot of the same problems. Even though React is just the view layer, it's roughly the same size as Angular, so that weight may be prohibitive based on your use case.

While React and Angular solve some of the same problems, they go about it in very different ways. React favors a functional, declarative approach, where components are pure functions free of side effects. This functional style of programming leads to fewer bugs and is simpler to reason about.

How About Angular 2?

Components in Angular 2 resemble React components in a lot of ways. The example components in the docs have a class and template in close proximity. Events look similar. It explains how to build views using a Component Hierarchy, just as you would if you were building it in React, and it embraces ES6 modules for dependency injection.

```
// Angular 2
@Component({
    selector: 'hello-component',
    template: `
        <h`
        <h4>Give me some keys!</h4>
        <input (keyup)="onKeyUp($event)" />
        <div>{{ values }}</div>
    `lloComponent {
    values='';
    onKeyUp(event) {
        this.values += event.target.value + ' | ';
    }
}

// React
class HelloComponent extends React.Component {
    constructor(props) {
    super()
    this.state = { values: '' };
}
onKeyUp(event) {
    const values = `${this.state.values + event.target.value} | `;
    this.setState({ values: values });
```

```
}
    render() {
        return (
            '''`${this.state.values + event.target.value} | `<div>
                <h4>Give me some keys!</h4>
                <div><input onKeyUp={ this.onKeyUp.bind(this) } />
                </div>
                <div>{ this.state.values }</div>
            </div>
        );
    }
}
```

A lot of the work on Angular 2 has been making it perform DOM updates a lot more efficiently. The previous template syntax and complexities around scopes led to a lot of performance problems in large apps.

A Complete Application

In this article I've focused on templates, directives and forms, but if you're building a complete application, you're going to require other things to help you manage your data model, server communication and routing at a minimum. When I first learned Angular and React, I created an example Gmail application to understand how they worked and to see what the developer experience was like before I started using them in real applications.

You might find it interesting to look through these example apps to compare the differences in React and Angular. The React example is written in CoffeeScript with CJSX, although the React community has since gathered around ES6 with Babel and Webpack, so that's the tooling I would suggest adopting if you're starting today.

- https://github.com/markbrown4/gmail-react
- https://github.com/markbrown4/gmail-angular

There's also the TodoMVC applications you could look at to compare:

- http://todomvc.com/examples/react/
- http://todomvc.com/examples/angularjs/

Chapter

A Guide to Testing React Components

9

by Camile Reyes

React is a framework that has made headway within the JavaScript developer community. React has a powerful composition framework for designing components. React components are bits of reusable code you can wield in your web application.

React components are not tightly coupled from the DOM, but how easy are they to unit test? In this take, let's explore what it takes to unit test React components. I'll show the thought process for making your components testable.

Keep in mind, I'm only talking about **unit tests**, which are a special kind of test. (For more on the different kinds of tests, I recommend you read "JavaScript Testing: Unit vs Functional vs Integration Tests".)

With unit tests, I'm interested in two things: rapid and neck-breaking feedback. With this, I can iterate through changes with a high degree of confidence and code quality. This gives you a level of reassurance that your React components will not land dead on the browser. Being capable of getting good feedback at a rapid rate gives you a competitive edge --- one that you'll want to keep in today's world of agile software development.

For the demo, let's do a list of the great apes, which is filterable through a checkbox. You can find the entire codebase on GitHub. For the sake of brevity, I'll show only the code samples that are of interest. This article assumes a working level of knowledge with React components.

If you go download and run the demo sample code, you'll see a page like this:

Write Testable Components

In React, a good approach is to start with a hierarchy of components. The single responsibility principle comes to mind when building each individual component. React components use object composition and relationships.

For the list of the great apes, for example, I have this approach:

```
FilterableGreatApeList
|_ GreatApeSearchBar
|_ GreatApeList
    |_ GreatApeRow
```

Take a look at how a great ape list has many great ape rows with data. React components make use of this composition data model, and it's also testable.

In React components, avoid using inheritance to build reusable components. If you come from a classic object-oriented programming background, keep this in mind. React components don't know their children ahead of time. Testing components that descend from a long chain of ancestors can be a nightmare.

I'll let you explore the `FilterableGreatApeList` on your own. It's a React component with two separate components that are of interest here. Feel free to explore the unit tests that come with it, too.

To build a testable `GreatApeSearchBar`, for example, do this:

```
class GreatApeSearchBar extends Component {
    constructor(props) {
        super(props);

        this.handleShowExtantOnlyChange = this.
        ↪handleShowExtantOnlyChange.bind(this);
    }

    handleShowExtantOnlyChange(e) {
        this.props.onShowExtantOnlyInput(e.target.checked);
```

```
    }

    render() {
        return(
            <form>
                <input
                    id="GreatApeSearchBar-showExtantOnly"
                    type="checkbox"
                    checked={this.props.showExtantOnly}
                    onChange={this.handleShowExtantOnlyChange}
                />

                <label htmlFor="GreatApeSearchBar-showExtantOnly"
                ↪>Only show extant species</label>
            </form>
        );
    }
}
```

This component has a checkbox with a label and wires up a click event. This approach may already be all too familiar to you, which is a very good thing.

Note that with React, testable components come for free, straight out of the box. There's nothing special here – an event handler, JSX, and a render method.

The next React component in the hierarchy is the `GreatApeList`, and it looks like this:

```
class GreatApeList extends Component {
```

```
    render() {
        let rows = [];

        this.props.apes.forEach((ape) => {
            if (!this.props.showExtantOnly) {
                rows.push(<GreatApeRow key={ape.name} ape={ape} />);

                return;
            }

            if (ape.isExtant) {
                rows.push(<GreatApeRow key={ape.name} ape={ape} />);
            }
        });

        return (
            <div>
                {rows}
            </div>
        );
    }
}
```

It's a React component that has the `GreatApeRow` component and it's using object composition. This is React's most powerful composition model at work. Note the lack of inheritance when you build reusable yet testable components.

In programming, object composition is a design pattern that enables data-driven elements. To think of it another way, a `GreatApeList` has many `GreatApeRow` objects. It's this relationship between UI components that drives the design. React components have this mindset built in. This way of looking at UI elements

allows you to write some nice unit tests.

Here, you check for the `this.props.showExtantOnly` flag that comes from the checkbox. This `showExtantOnly` property gets set through the event handler in `GreatApeSearchBar`.

For unit tests, how do you unit test React components that depend on other components? How about components intertwined with each other? These are great questions to keep in mind as we get into testing soon. React components may yet have secrets one can unlock.

For now, let's look at the `GreatApeRow`, which houses the great ape data:

```
class GreatApeRow extends Component {
    render() {
        return (
            <div>
                <img
                    className="GreatApeRow-image"
                    src={this.props.ape.image}
                    alt={this.props.ape.name}
                />

                <p className="GreatApeRow-detail">
                    <b>Species:</b> {this.props.ape.name}
                </p>

                <p className="GreatApeRow-detail">
                    <b>Age:</b> {this.props.ape.age}
                </p>
            </div>
```

```
        );
    }
}
```

With React components, it's practical to isolate each UI element with a laser focus on a single concern. This has key advantages when it comes to unit testing. As long as you stick to this design pattern, you'll find it seamless to write unit tests.

Test Utilities

Let's recap our biggest concern when it comes to testing React components. How do I unit test a single component in isolation? Well, as it turns out, there's a nifty utility that enables you to do that.

The Shallow Renderer in React allows you to render a component one level deep. From this, you can assert facts about what the render method does. What's remarkable is that it doesn't require a DOM.

Using ES6, you use it like this:

```
import ShallowRenderer from 'react-test-renderer/shallow';
```

In order for unit tests to run fast, you need a way to test components in isolation. This way, you can focus on a single problem, test it, and move on to the next concern. This becomes empowering as the solution grows and you're able to refactor at will --- staying close to the code, making rapid changes, and gaining reassurance it will work in a browser.

One advantage of this approach is you think better about the code. This produces the best solution that deals with the problem at hand. I find it liberating when you're not chained to a ton of distractions. The human brain does a terrible job at dealing with more than one problem at a time.

The only question remaining is, how far can this little utility take us with React components?

Put It All Together

Take a look at `GreatApeList`, for example. What's the main concern you're trying to solve? This component shows you a list of great apes based on a filter.

An effective unit test is to pass in a list and check facts about what this React component does. We want to ensure it filters the great apes based on a flag.

One approach is to do this:

```
import GreatApeList from './GreatApeList';

const APES = [{ name: 'Australopithecus afarensis', isExtant: false },
    { name: 'Orangutan', isExtant: true }];

// Arrange
const renderer = new ShallowRenderer();
renderer.render(<GreatApeList
    apes={APES}
    showExtantOnly={true} />);

// Act
const component = renderer.getRenderOutput();
const rows = component.props.children;
```

```
// Assert
expect(rows.length).toBe(1);
```

Note that I'm testing React components using Jest. For more on this, check out "How to Test React Components Using Jest".

In JSX, take a look at `showExtantOnly={true}`. The JSX syntax allows you to set a state to your React components. This opens up many ways to unit test components given a specific state. JSX understands basic JavaScript types, so a `true` flag gets set as a boolean.

With the list out of the way, how about the `GreatApeSearchBar`? It has this event handler in the `onChange` property that might be of interest.

One good unit test is to do this:

```
import GreatApeSearchBar from './GreatApeSearchBar';

// Arrange
let showExtantOnly = false;
const onChange = (e) => { showExtantOnly = e };

const renderer = new ShallowRenderer();
renderer.render(<GreatApeSearchBar
    showExtantOnly={true}
    onShowExtantOnlyInput={onChange} />);
```

```
// Act
const component = renderer.getRenderOutput();
const checkbox = component.props.children[0];

checkbox.props.onChange({ target: { checked: true } });

// Assert
expect(showExtantOnly).toBe(true);
```

To handle and test events, you use the same shallow rendering method. The `getRenderOutput` method is useful for binding callback functions to components with events. Here, the `onShowExtantOnlyInput` property gets assigned the callback `onChange` function.

On a more trivial unit test, what about the `GreatApeRow` React component? It displays great ape information using HTML tags. Turns out, you can use the shallow renderer to test this component too.

For example, let's ensure we render an image:

```
import GreatApeRow from './GreatApeRow';

const APE = {
    image: 'https://en.wikipedia.org/wiki/File:
    ↪Australopithecus_afarensis.JPG',
    name: 'Australopithecus afarensis'
};
```

```
// Arrange
const renderer = new ShallowRenderer();
renderer.render(<GreatApeRow ape={APE} />);

// Act
const component = renderer.getRenderOutput();
const apeImage = component.props.children[0];

// Assert
expect(apeImage).toBeDefined();
expect(apeImage.props.src).toBe(APE.image);
expect(apeImage.props.alt).toBe(APE.name);
```

With React components, it all centers around the `render` method. This makes it somewhat intuitive to know exactly what you need to test. A shallow renderer makes it so you can laser focus on a single component while eliminating noise.

Conclusion

As shown, React components are very testable. There's no excuse to forgo writing good unit tests for your components.

The nice thing is that JSX works for you in each individual test, not against you. With JSX, you can pass in booleans, callbacks, or whatever else you need. Keep this in mind as you venture out into unit testing React components on your own.

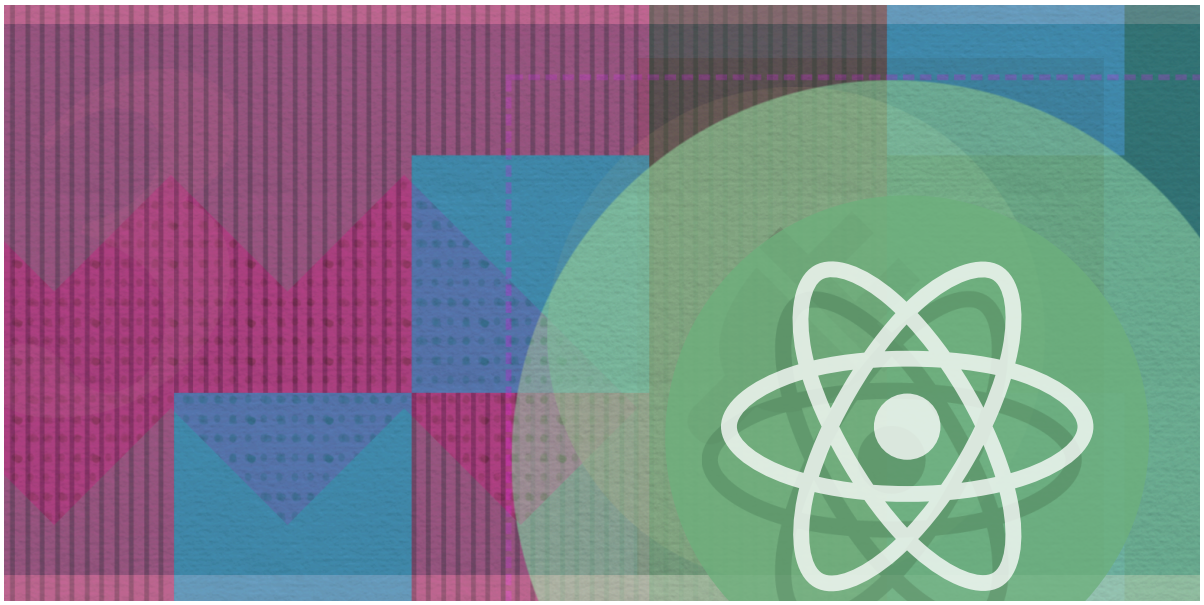
The shallow renderer test utility gives you all you need for good unit tests. It only renders one level deep and allows you to test in isolation. You're not concerned with any arbitrary child in the hierarchy that might break your unit tests.

With the Jest tooling, I like how it gives you feedback only on the specific files

you're changing. This shortens the feedback loop and adds laser focus. I hope you see how valuable this can be when you tackle some tough issues.

Build

5 Practical React Projects



sitepoint

5 Practical React Projects

Copyright © 2017 SitePoint Pty. Ltd.

Product Manager: Simon Mackie **Cover Designer:** Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

Table of Contents

Preface

This book is a collection of in-depth tutorials, selected from SitePoint's React Hub, that will guide you through some fun and practical projects. Along the way, you'll pick up lots of useful development tips.

Who Should Read This Book

This book is for develeopers with some React experience. If you're a novice, please read *Your First Week With React* before tackling this book.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.</p>
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
 function animate() {
  new_variable = "Hello";
}
```

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed:

```
function animate() {
    ⋮
new_variable = "Hello";
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ↪ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-design-real-user-testing/
↪?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter

1

How to Create a Reddit Clone Using React and Firebase

by Nirmalya Ghosh

In this article, we'll be using Firebase along with Create React App to build an app that will function similar to Reddit. It will allow the user to submit a new link that can then be voted on.

Here's a live demo of what we'll be building.

Why Firebase?

Using Firebase will make it very easy for us to show real-time data to the user. Once a user votes on a link, the feedback will be instantaneous. Firebase's Realtime Database will help us in developing this feature. Also, it will help us to understand how to bootstrap a React application with Firebase.

Why React?

React is particularly known for creating user interfaces using a component architecture. Each component can contain internal state or be passed data as props. State and props are the two most important concepts in React. These two things help us determine the state of our application at any point in time. If you're not familiar with these terms, please head over to the React docs first.

Using a State Container

Note: you can also use a state container like Redux or MobX, but for the sake of simplicity, we won't be using one for this tutorial.

The whole project is available on GitHub.

Setting up the Project

Let's walk through the steps to set up our project structure and any necessary dependencies.

Installing create-react-app

If you haven't already, you need to install **create-react-app**. To do so, you can type the following in your terminal:

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

Once you've installed it globally, you can use it to scaffold a React project inside any folder.

Now, let's create a new app and call it **reddit-clone**.

```
create-react-app reddit-clone
```

This will scaffold a new **create-react-app** project inside the **reddit-clone** folder. Once the bootstrapping is done, we can go inside **reddit-clone** directory and fire up the development server:

```
npm start
```

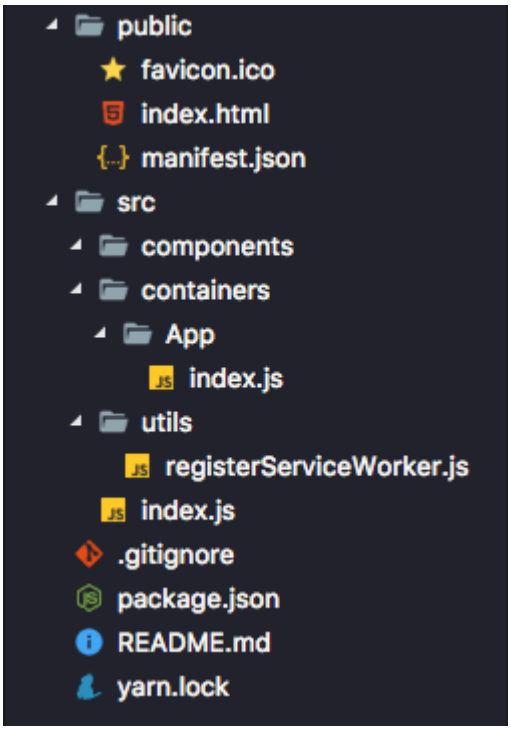
At this point, you can go to http://localhost:3000/ and see your app skeleton up and running.

Structuring the app

For maintenance, I always like to separate my **containers** and **components**. Containers are the smart components that contain the business logic of our application and manage Ajax requests. Components are simply dumb

presentational components. They can have their own internal state, which can be used to control the logic of that component (e.g. showing the current state of a controlled input component).

After removing the unnecessary logo and CSS files, this is how your app should look now. We created a `components` folder and a `containers` folder. Let's move `App.js` inside the `containers/App` folder and create `registerServiceWorker.js` inside the `utils` folder.



Your `src/containers/App/index.js` file should look like this:

```
// src/containers/App/index.js

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

```
class App extends Component {
  render() {
    return (
      <div className="App">
        Hello World
      </div>
    );
  }
}

export default App;
```

Your `src/index.js` file should look like this:

```
// src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import App from './containers/App';
import registerServiceWorker from './utils/registerServiceWorker';

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

Go to your browser, and if everything works fine, you'll see **Hello World** on your screen.

You can check my commit on GitHub.

Adding react-router

React-router will help us define the routes for our app. It's very customizable and very popular in the React ecosystem.

We'll be using version 3.0.0 of react-router.

```
npm install --save react-router@3.0.0
```

Now, add a new file `routes.js` inside the `src` folder with the following code:

```
// routes.js

import React from 'react';
import { Router, Route } from 'react-router';

import App from './containers/App';

const Routes = (props) => (
  <Router {...props}>
    <Route path="/" component={ App }>
    </Route>
  </Router>
);

export default Routes;
```

The `Router` component wraps all the `Route` components. Based on the `path` prop of the `Route` component, the component passed to the `component` prop will be

rendered on the page. Here, we're setting up the root URL (/) to load our `App` component using the `Router` component.

```
<Router {...props}>
  <Route path="/" component={ <div>Hello World!</div> }>
  </Route>
</Router>
```

The above code is also valid. For the path `/`, the `<div>Hello World!</div>` will be mounted.

Now, we need to call our `routes.js` file from our `src/index.js` file. The file should have the following content:

```
// src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import { browserHistory } from 'react-router';

import App from './containers/App';
import Routes from './routes';
import registerServiceWorker from './utils/registerServiceWorker';

ReactDOM.render(
  <Routes history={browserHistory} />,
  document.getElementById('root')
);
```

```
registerServiceWorker();
'root'
```

Basically, we're mounting our `Router` component from our `routes.js` file. We pass in the `history` prop to it so that the routes know how to handle history tracking.

You can check my commit on GitHub.

Adding Firebase

If you don't have a Firebase account, create one now (it's free!) by going to their website. After you're done creating a new account, log in to your account and go to the console page and click on **Add project**.

Enter the name of your project (I'll call mine **reddit-clone**), choose your country, and click on the **Create project** button.

Now, before we proceed, we need to change the **rules** for the database since, by default, Firebase expects the user to be authenticated to be able to read and write data. If you select your project and click on the **Database** tab on the left, you'll be able to see your database. You need to click on the **Rules** tab on the top that will redirect us to a screen which will have the following data:

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}
```

We need to change this to the following:

```
{
  "rules": {
    ".read": "auth === null",
    ".write": "auth === null"
  }
}
```

This will let users update the database without logging in. If we implemented a flow in which we had authentication before making updates to the database, we would need the default rules provided by Firebase. To keep this application simple, we *won't* be doing authentication.

You Must Make This Modification

If you don't make this modification, Firebase won't let you update the database from your app.

Now, let's add the `firebase` npm module to our app by running the following code:

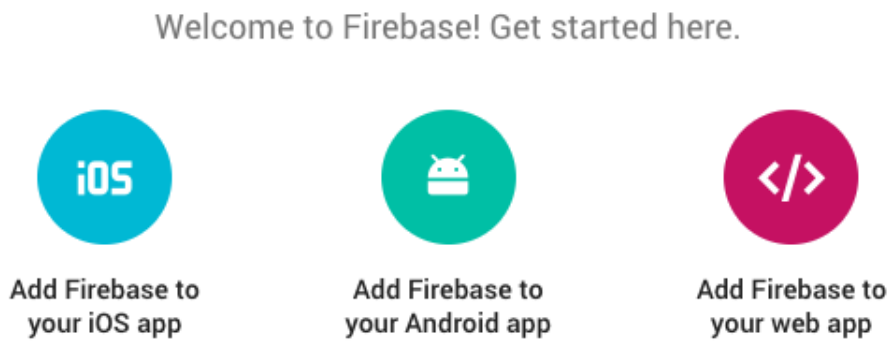
```
npm install --save firebase
```

Next, import that module in your `App/index.js` file as:

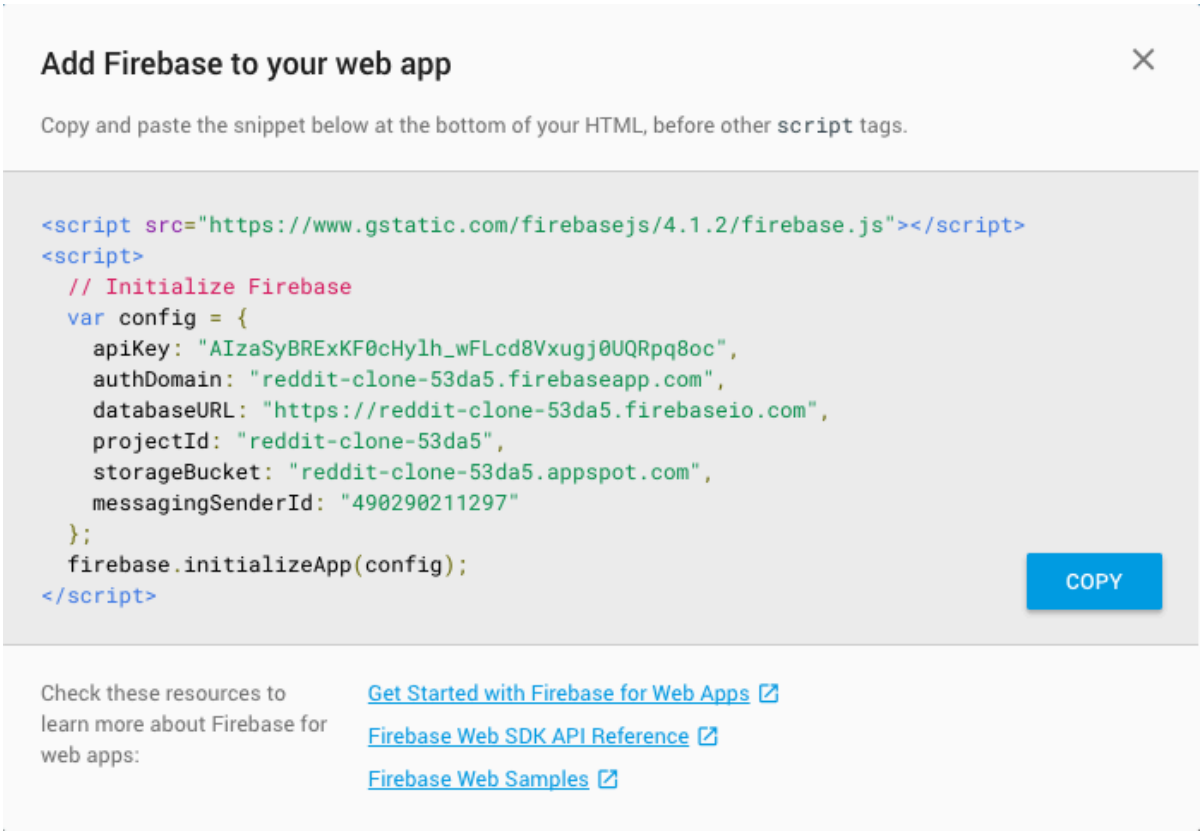
```
// App/index.js
```

```
import * as firebase from "firebase";
```

When we select our project after logging in to Firebase, we'll get an option **Add Firebase to your web app**.



If we click on that option, a modal will appear that will show us the `config` variable which we will use in our `componentWillMount` method.



Let's create the Firebase config file. We'll call this file `firebase-config.js`, and it will contain all the configs necessary to connect our app with Firebase:

```
// App/firebase-config.js

export default {
  apiKey: "AIzaSyBRExKF0cHylh_wFLcd8Vxugj0UQRpq8oc",
  authDomain: "reddit-clone-53da5.firebaseapp.com",
  databaseURL: "https://reddit-clone-53da5.firebaseio.com",
  projectId: "reddit-clone-53da5",
  storageBucket: "reddit-clone-53da5.appspot.com",
  messagingSenderId: "490290211297"
};
```

We'll import our Firebase config into `App/index.js`:

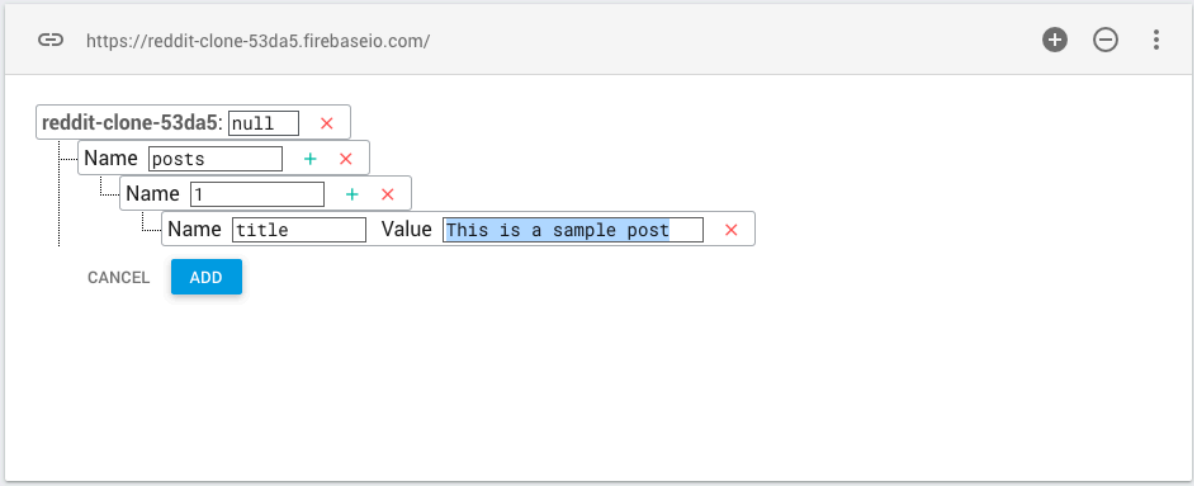
```
// App/index.js

import config from './firebase-config';
```

We'll initialize our Firebase database connection in the `constructor`.

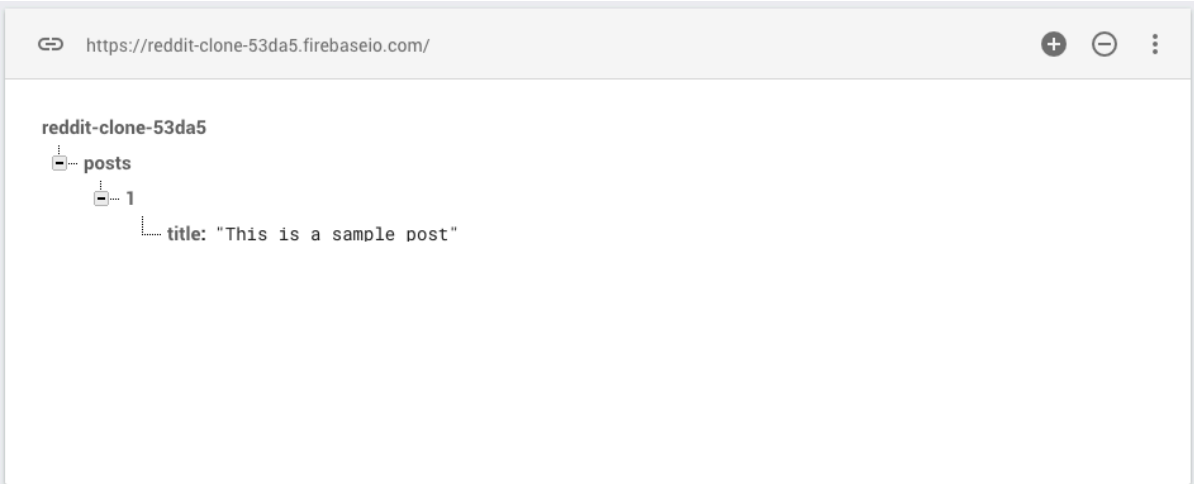
```
// App/index.js

constructor() {
  super();

  // Initialize Firebase
  firebase.initializeApp(config);
}
```

In the `componentWillMount()` lifecycle hook, we use the package `firebase` we just installed and call its `initializeApp` method and passed the `config` variable to it. This object contains all the data about our app. The `initializeApp` method will connect our application to our Firebase database so that we can read and write data.

Let's add some data to Firebase to check if our configuration is correct. Go to the *Database* tab and add the following structure to your database:



Clicking on *Add* will save the data to our database.



Now, let's add some code to our `componentWillMount` method to make the data appear on our screen:

```
// App/index.js

componentWillMount() {
  ...
```

```
let postsRef = firebase.database().ref('posts');

let _this = this;

postsRef.on('value', function(snapshot) {
  console.log(snapshot.val());

  _this.setState({
    posts: snapshot.val(),
    loading: false
    });
    });
  }
```

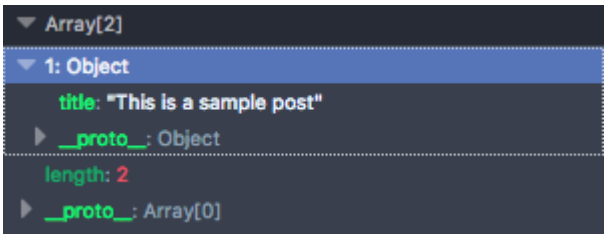
`firebase.database()` gives us a reference to the database service. Using `ref()`, we can get a specific reference from the database. For example, if we call `ref('posts')`, we'll be getting the `posts` reference from our database and storing that reference in `postsRef`.

`postsRef.on('value', ...)` gives us the updated value whenever there's any change in the database. This is very useful when we need a real-time update to our user interface based on any database events.

Using `postsRef.once('value', ...)` will only give us the data once. This is useful for data that only needs to be loaded once and isn't expected to change frequently or require active listening.

After we get the updated value in our `on()` callback, we store the values in our `posts` state.

Now we'll see the data appearing on our console.



Also, we'll be passing this data down to our children. So, we need to modify the `render` function of our `App/index.js` file:

```
// App/index.js

render() {
  return (
    <div className="App">
      {this.props.children && React.cloneElement(
        ↪this.props.children, {
        firebaseRef: firebase.database().ref('posts'),
        posts: this.state.posts,
        loading: this.state.loading
      })}
    </div>
  );
}
```

The main objective here is to make the posts data available in all our children components, which will be passed through `react-router`.

We're checking if `this.props.children` exists or not, and if it exists we clone that element and pass all our props to all our children. This is a very efficient way

of passing props to dynamic children.

Calling cloneElement will shallowly merge the already existing props in `this.props.children` and the props we passed here (`firebaseRef`, `posts` and `loading`).

Using this technique, the `firebaseRef`, `posts` and `loading` props will be available to all routes.

You can check my commit on GitHub.

Connecting the App with Firebase

Firebase can only store data as objects; it doesn't have any native support for arrays. We'll store the data in the following format:

Add the data in the screenshot above manually so that you can test your views.

Add views for all the posts

Now we'll add views to show all the posts. Create a file `src/containers/Posts/index.js` with the following content:

```
// src/containers/Posts/index.js

import React, { Component } from 'react';

class Posts extends Component {
  render() {
    if (this.props.loading) {
      return (
        <div>
          Loading...
        </div>
      );
    }

    return (
      <div className="Posts">
        { this.props.posts.map((post) => {
            return (
              <div>
                { post.title }
              </div>
            );
        })}
      </div>
    );
  }
}

export default Posts;
```

Here, we're just mapping over the data and rendering it to the user interface.

Next, we need to add this to our `routes.js` file:

```
// routes.js

…
<Router {...props}>
  <Route path="/" component={ App }>
    <Route path="/posts" component={ Posts } />
  </Route>
</Router>
…
```

This is because we want the posts to show up only on the `/posts` route. So we just pass the `Posts` component to the `component` prop and `/posts` to the `path` prop of the `Route` component of react-router.

If we go to the URL localhost:3000/posts, we'll see the posts from our Firebase database.

You can check my commit on GitHub.

Add views to write a new post

Now, let's create a view from where we can add a new post. Create a file `src/containers/AddPost/index.js` with the following content:

```
// src/containers/AddPost/index.js

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

```
class AddPost extends Component {
  constructor() {
    super();

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  state = {
    title: ''
  };

  handleChange = (e) => {
    this.setState({
      title: e.target.value
    });
  }

  handleSubmit = (e) => {
    e.preventDefault();

    this.props.firebaseRef.push({
      title: this.state.title
    });

    this.setState({
      title: ''
    });
  }

  render() {
```

```
    return (
      <div className="AddPost">
        <input
          type="text"
          placeholder="Write the title of your post"
          onChange={ this.handleChange }
          value={ this.state.title }
        />
        <button
          type="submit"
          onClick={ this.handleSubmit }
        >
          Submit
        </button>
      </div>
    );
  }
}

export default AddPost;
```

Here, the `handleChange` method updates our state with the value present in the input box. Now, when we click on the button, the `handleSubmit` method is triggered. The `handleSubmit` method is responsible for making the API request to write to our database. We do it using the `firebaseRef` prop that we passed to all the children.

```
this.props.firebaseRef.push({
  title: this.state.title
```

```
});
```

The above block of code sets the current value of the title to our database.

After the new post has been stored in the database, we make the input box empty again, ready to add a new post.

Now we need to add this page to our routes:

```
// routes.js

import React from 'react';
import { Router, Route } from 'react-router';

import App from './containers/App';
import Posts from './containers/Posts';
import AddPost from './containers/AddPost';

const Routes = (props) => (
  <Router {...props}>
    <Route path="/" component={ App }>
      <Route path="/posts" component={ Posts } />
      <Route path="/add-post" component={ AddPost } />
    </Route>
  </Router>
);

export default Routes;
```

Here, we just added the `/add-post` route so that we can add a new post from

that route. Hence, we passed the `AddPost` component to its component prop.

Also, let's modify the `render` method of our `src/containers/Posts/index.js` file so that it can iterate over objects instead of arrays (since Firebase doesn't store arrays).

```
// src/containers/Posts/index.js

render() {
    let posts = this.props.posts;

    if (this.props.loading) {
      return (
        <div>
          Loading...
        </div>
      );
    }

    return (
      <div className="Posts">
        { Object.keys(posts).map(function(key) {
            return (
              <div key={key}>
                { posts[key].title }
              </div>
            );
        })}
      </div>
    );
  }
```

Now, if we go to localhost:3000/add-post, we can add a new post. After clicking on the **submit** button, the new post will appear immediately on the posts page.

You can check my commit on GitHub.

Implement voting

Now we need to allow users to vote on a post. For that, let's modify the `render` method of our `src/containers/App/index.js`:

```
// src/containers/App/index.js

render() {
  return (
    <div className="App">
      {this.props.children && React.cloneElement(this.props.
        ↪children, {
        // https://github.com/ReactTraining/react-router/blob/v3/
        ↪examples/passing-props-to-children/app.js#L56-L58
        firebase: firebase.database(),
        posts: this.state.posts,
        loading: this.state.loading
      })}
    </div>
  );
}
```

We changed the `firebase` prop from `firebaseRef: firebase.database().ref('posts')` to `firebase: firebase.database()` because we'll be using Firebase's `set` method to update our voting count. In this way, if we had more Firebase refs, it would be very easy for us to handle them by

using only the `firebase` prop.

Before proceeding with the voting, let's modify the `handleSubmit` method in our `src/containers/AddPost/index.js` file a little bit:

```
// src/containers/AddPost/index.js

handleSubmit = (e) => {
  …
  this.props.firebase.ref('posts').push({
    title: this.state.title,
    upvote: 0,
    downvote: 0
  });
  …
}
```

We renamed our `firebaseRef` prop to `firebase` prop. So, we change the `this.props.firebaseRef.push` to `this.props.firebase.ref('posts').push`.

Now we need to modify our `src/containers/Posts/index.js` file to accommodate the voting.

The `render` method should be modified to this:

```
// src/containers/Posts/index.js

render() {
  let posts = this.props.posts;
```

```
let _this = this;

if (!posts) {
  return false;
}

if (this.props.loading) {
  return (
    <div>
      Loading...
    </div>
  );
}

return (
  <div className="Posts">
    { Object.keys(posts).map(function(key) {
        return (
          <div key={key}>
            <div>Title: { posts[key].title }</div>
            <div>Upvotes: { posts[key].upvote }</div>
            <div>Downvotes: { posts[key].downvote }</div>
            <div>
              <button
                onClick={ _this.handleUpvote.bind(this,
                  ↪posts[key], key) }
                type="button"
              >
                Upvote
              </button>
              <button
                onClick={ _this.handleDownvote.bind(this,
```

```
                    ↪posts[key], key) }
                  type="button"
                >
                  Downvote
                </button>
              </div>
            </div>
          );
      })}
    </div>
  );
}
```

When the buttons are clicked, the **upvote** or **downvote** count will be incremented in our Firebase DB. To handle that logic, we create two more methods: `handleUpvote()` and `handleDownvote()`:

```
// src/containers/Posts/index.js

handleUpvote = (post, key) => {
  this.props.firebase.ref('posts/' + key).set({
    title: post.title,
    upvote: post.upvote + 1,
    downvote: post.downvote
  });
}

handleDownvote = (post, key) => {
  this.props.firebase.ref('posts/' + key).set({
    title: post.title,
```

```
      upvote: post.upvote,
      downvote: post.downvote + 1
    });
  }
```

In these two methods, whenever a user clicks on either of the buttons, the respective count is incremented in the database and is instantly updated in the browser.

If we open two tabs with localhost:3000/posts and click on the voting buttons of the posts, we'll see each of the tabs get updated almost instantly. This is the magic of using a real-time database like Firebase.

You can check my commit on GitHub.

In the repository, I've added the `/posts` route to the `IndexRoute` of the application just to show the posts on localhost:3000 by default. You can check that commit on GitHub.

Conclusion

The end result is admittedly a bit barebones, as we didn't try to implement any design (although the demo has some basic styles added). We also didn't add any authentication, in order to reduce the complexity and the length of the tutorial, but obviously any real-world application would require it.

Firebase is really useful for places where you don't want to create and maintain a separate back-end application, or where you want real-time data without investing too much time developing your APIs. It plays really well with React, as you can hopefully see from the article.

Further reading

- Getting React Projects Ready Fast with Pre-configured Builds
- Build a React Application with User Login and Authentication
- Firebase Authentication for Web
- Leveling Up With React: React Router

Chapter

2

Build a CRUD App Using React, Redux and FeathersJS

by Michael Wanyoike

Building a modern project requires splitting the logic into front-end and back-end code. The reason behind this move is to promote code re-usability. For example, we may need to build a native mobile application that accesses the back-end API. Or we may be developing a module that will be part of a large modular platform.

The popular way of building a server-side API is to use a library like Express or Restify. These libraries make creating RESTful routes easy. The problem with these libraries is that we'll find ourselves writing a *ton* of **repeating code**. We'll also need to write code for authorization and other middleware logic.

To escape this dilemma, we can use a framework like Loopback or Feathers to help us generate an API.

At the time of writing, Loopback has more GitHub stars and downloads than Feathers. Loopback is a great library for generating RESTful CRUD endpoints in a short period of time. However, it does have a slight learning curve and the documentation is not easy to get along with. It has stringent framework requirements. For example, all models must inherit one of its built-in model class. If you need real-time capabilities in Loopback, be prepared to do some additional coding to make it work.

FeathersJS, on the other hand, is much easier to get started with and has realtime support built-in. Quite recently, the Auk version was released (because Feathers is so modular, they use bird names for version names) which introduced a vast number of changes and improvements in a number of areas. According to a post they published on their blog, they are now the **4th most popular real-time web framework**. It has excellent documentation, and they've covered pretty much any area we can think of on building a real-time API.

What makes Feathers amazing is its simplicity. The entire framework is modular and we only need to install the features we need. Feathers itself is a thin wrapper built on top of Express, where they've added new features – services and hooks. Feathers also allows us to effortlessly send and receive data over WebSockets.

Prerequisites

Before you get started with the tutorial, you'll need to have a solid foundation in the following topics:

- How to write ES6 JavaScript code
- How to create React components
- Immutability in JavaScript
- How to manage state with Redux

On your machine, you'll need to have installed recent versions of:

- NodeJS 6+
- Mongodb 3.4+
- Yarn package manager (optional)
- Chrome browser

If you've never written a database API in JavaScript before, I'd recommend first taking a look at this tutorial on creating RESTful APIs.

Scaffold the App

We'e going to build a CRUD contact manager application using React, Redux, Feathers and MongoDB. You can take a look at the completed project here.

In this tutorial, I'll show you how to build the application from the bottom up. We'll kick-start our project using the create-react-app tool.

```
# scaffold a new react project
create-react-app react-contact-manager
cd react-contact-manager
```

```
# delete unnecessary files
rm src/logo.svg src/App.css
```

Use your favorite code editor and remove all the content in index.css. Open App.js and rewrite the code like this:

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

class App extends Component {
  render() {
    return (
      <div>
        <h1>Contact Manager</h1>
      </div>
    );
  }
}

export default App;
```

Make sure to run `yarn start` to ensure the project is running as expected. Check the console tab to ensure that our project is running cleanly with no warnings or errors. If everything is running smoothly, use `Ctrl+C` to stop the server.

Build the API Server with Feathers

Let's proceed with generating the back-end API for our CRUD project using the `feathers-cli` tool.

```
# Install Feathers command-line tool
npm install -g feathers-cli

# Create directory for the back-end code
mkdir backend
cd backend

# Generate a feathers back-end API server
feathers generate app

? Project name | backend
? Description | contacts API server
? What folder should the source files live in? | src
? Which package manager are you using (has to be installed
↪globally)? | Yarn
? What type of API are you making? | REST, Realtime via Socket.io

# Generate RESTful routes for Contact Model
feathers generate service

? What kind of service is it? | Mongoose
? What is the name of the service? | contact
? Which path should the service be registered on? | /contacts
? What is the database connection string? |
↪mongodb://localhost:27017/backend


# Install email field type
yarn add mongoose-type-email

# Install the nodemon package
yarn add nodemon --dev
```

Open `backend/package.json` and update the start script to use nodemon so that the API server will restart automatically whenever we make changes.

```
// backend/package.json

…
"scripts": {
  ...
  "start": "nodemon src/",
  …
},
…
```

Let's open `backend/config/default.json`. This is where we can configure MongoDB connection parameters and other settings. I've also increased the default paginate value to 50, since in this tutorial we won't write front-end logic to deal with pagination.

```
{
  "host": "localhost",
  "port": 3030,
  "public": "../public/",
  "paginate": {
    "default": 50,
    "max": 50
  },
  "mongodb": "mongodb://localhost:27017/backend"
}
```

Open `backend/src/models/contact.model.js` and update the code as follows:

```
// backend/src/models/contact.model.js

require('mongoose-type-email');

module.exports = function (app) {
  const mongooseClient = app.get('mongooseClient');
  const contact = new mongooseClient.Schema({
    name : {
      first: {
        type: String,
        required: [true, 'First Name is required']
      },
      last: {
        type: String,
        required: false
      }
    },
    email : {
      type: mongooseClient.SchemaTypes.Email,
      required: [true, 'Email is required']
    },
    phone : {
      type: String,
      required: [true, 'Phone is required'],
      validate: {
        validator: function(v) {
          return /^\+(?:[0-9] ?){6,14}[0-9]$/.test(v);
        },
        message: '{VALUE} is not a
        ↪valid international phone number!'
```

```
      }
    },
    createdAt: { type: Date, /^\+(?:[0-9] ?){6,14}[0-9]$/'default': Date.now },
    updatedAt: { type: Date, 'default': Date.now }
  });

  return mongooseClient.model('contact', contact);
};
```

In addition to generating the contact service, Feathers has also generated a test case for us. We need to fix the service name first for it to pass:

```
// backend/test/services/contact.test.js

const assert = require('assert');
const app = require('../../src/app');

describe('\'contact\' service', () => {
  it('registered the service', () => {
    const service = app.service('contacts'); // change
    ↪contact to contacts

    assert.ok(service, 'Registered the service');
  });
});
'Registered the service'
```

Open a new terminal and inside the backend directory, execute `yarn test`. You should have all the tests running successfully. Go ahead and execute `yarn start` to start the backend server. Once the server has finished starting it should print

the line: `'Feathers application started on localhost:3030'`.

Launch your browser and access the url: http://localhost:3030/contacts. You should expect to receive the following JSON response:

```
{"total":0,"limit":50,"skip":0,"data":[]}
```

Now let's use Postman to confirm all CRUD restful routes are working. You can launch Postman using this link.

If you're new to Postman, check out this tutorial. When you hit the SEND button, you should get your data back as the response along with three additional fields – `_id`, `createdAt` and `updatedAt`.

Use the following JSON data to make a POST request using Postman. Paste this in the body and set content-type to `application/json`:

```
{
  "name": {
    "first": "Tony",
    "last": "Stark"
  },
  "phone": "+18138683770",
  "email": "tony@starkenterprises.com"
}
```

Build the UI

Let's start by installing the necessary front-end dependencies. We'll use

semantic-ui css/semantic-ui react to style our pages and react-router-dom to handle route navigation.

Where To Install

Make sure you are installing *outside* the backend directory.

```
// Install semantic-ui
yarn add semantic-ui-css semantic-ui-react

// Install react-router
yarn add react-router-dom
```

Update the project structure by adding the following directories and files:

```
|-- react-contact-manager
    |-- backend
    |-- node_modules
    |-- public
    |-- src
        |-- App.js
        |-- App.test.js
        |-- index.css
        |-- index.js
        |-- components
        |   |-- contact-form.js #(new)
        |   |-- contact-list.js #(new)
        |-- pages
            |-- contact-form-page.js #(new)
```

```
            |-- contact-list-page.js #(new)
```

Let's quickly populate the JS files with some placeholder code.

For the component `contact-list.js`, we'll write it in this syntax since it will be a purely presentational component.

```
// src/components/contact-list.js

import React from 'react';

export default function ContactList(){
  return (
    <div>
      <p>No contacts here</p>
    </div>
  )
}
```

For the top-level containers, I use pages. Let's provide some code for the `contact-list-page.js`

```
// src/pages/contact-list-page.js

import React, { Component} from 'react';
import ContactList from '../components/contact-list';

class ContactListPage extends Component {
```

```
  render() {
    return (
      <div>
        <h1>List of Contacts</h1>
        <ContactList/>
      </div>
    )
  }
}

export default ContactListPage;
```

For the `contact-form` component, it needs to be smart, since it's required to manage its own state, specifically form fields. For now, we'll place this placeholder code.

```
// src/components/contact-form.js
import React, { Component } from 'react';

class ContactForm extends Component {
  render() {
    return (
      <div>
        <p>Form under construction</p>
      </div>
    )
  }
}

export default ContactForm;
```

Populate the `contact-form-page` with this code:

```
// src/pages/contact-form-page.js

import React, { Component} from 'react';
import ContactForm from '../components/contact-form';

class ContactFormPage extends Component {
  render() {
    return (
      <div>
        <ContactForm/>
      </div>
    )
  }
}

export default ContactFormPage;
```

Now, let's create the navigation menu and define the routes for our App. `App.js` is often referred to as the 'layout template' for the Single Page Application.

```
// src/App.js

import React, { Component } from 'react';
import { NavLink, Route } from 'react-router-dom';
import { Container } from 'semantic-ui-react';
import ContactListPage from './pages/contact-list-page';
import ContactFormPage from './pages/contact-form-page';
```

```
class App extends Component {
  render() {
    return (
      <Container>
        <div className="ui two item menu">
          <NavLink className="item" activeClassName="active"
          ↪exact to="/">
            Contacts List
          </NavLink>
          <NavLink className="item" activeClassName="active"
          ↪exact to="/contacts/new">
            Add Contact
          </NavLink>
        </div>
        <Route exact path="/" component={ContactListPage}/>
        <Route path="/contacts/new" component={ContactFormPage}/>
        <Route path="/contacts/edit/:_id"
        ↪component={ContactFormPage}/>
      </Container>
    );
  }
}

export default App;
```

Finally, update the `index.js` file with this code where we import semantic-ui CSS for styling and BrowserRouter for using the HTML5 history API that keeps our app in sync with the URL.

```
// src/index.js
```

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter } from 'react-router-dom';
import App from './App';
import 'semantic-ui-css/semantic.min.css';
import './index.css';

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

Go back to the terminal and execute `yarn start`. You should have a similar view to the screenshot below:

Manage React State with Redux

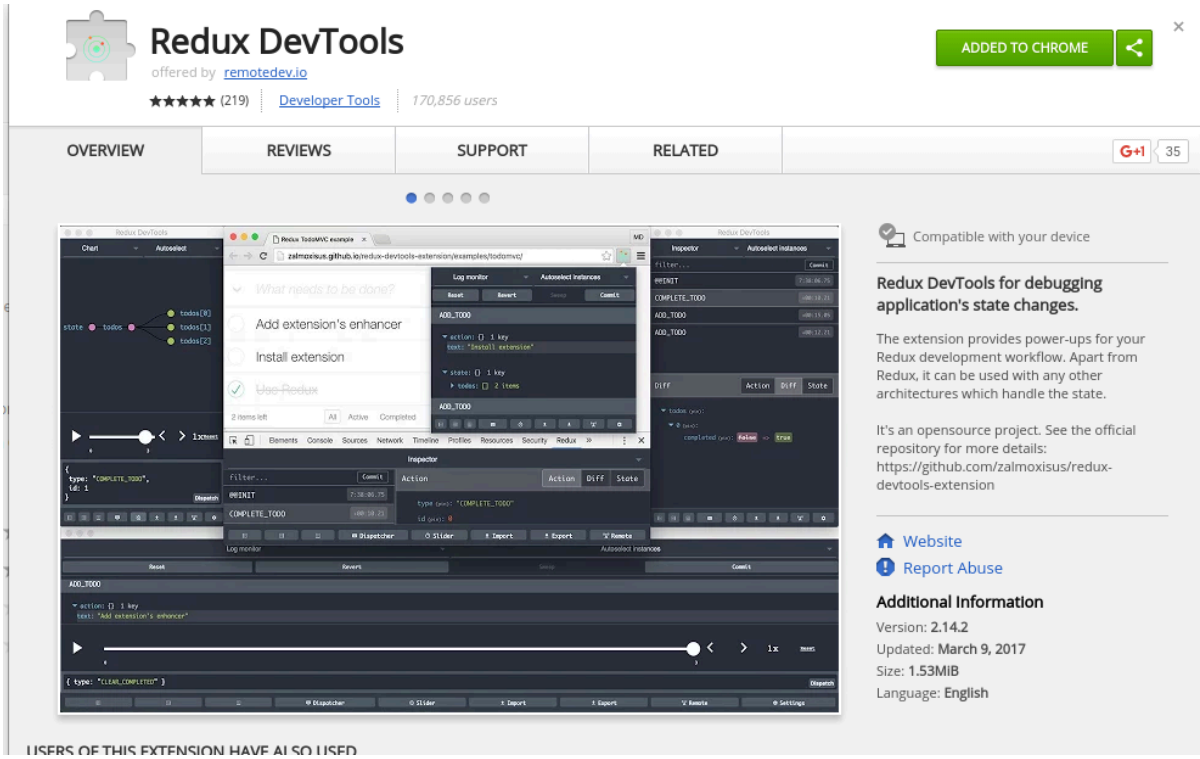
Stop the server with `ctrl+c` and install the following packages using yarn package manager:

```
yarn add redux react-redux redux-promise-middleware redux-thunk
↪redux-devtools-extension axios
```

Phew! That's a whole bunch of packages for setting up Redux. I assume you are already familiar with Redux if you're reading this tutorial. Redux-thunk allows writing action creators as async functions while redux-promise-middleware reduces some Redux boilerplate code for us by handling dispatching of pending, fulfilled, and rejected actions on our behalf.

Feathers does include a light-weight client package that helps communicate with the API, but it's also really easy to use other client packages. For this tutorial, we'll use the Axios HTTP client.

The redux-devtools-extension an amazing tool that keeps track of dispatched actions and state changes. You'll need to install its chrome extension for it to work.



Next, let's setup our Redux directory structure as follows:

```
|-- react-contact-manager
    |-- backend
    |-- node_modules
    |-- public
    |-- src
        |-- App.js
        |-- App.test.js
        |-- index.css
        |-- index.js
        |-- contact-data.js #new
        |-- store.js #new
        |-- actions #new
            |-- contact-actions.js #new
```

```
            |-- index.js #new
        |-- components
        |-- pages
        |-- reducers #new
            |-- contact-reducer.js #new
            |-- index.js #new
```

Let's start by populating `contacts-data.js` with some test data:

```
// src/contact-data.js

export const contacts = [
  {
    _id: "1",
    name: {
      first:"John",
      last:"Doe"
    },
    phone:"555",
    email:"john@gmail.com"
  },
  {
    _id: "2",
    name: {
      first:"Bruce",
      last:"Wayne"
    },
    phone:"777",
    email:"bruce.wayne@gmail.com"
  }
```

```
];
```

Define `contact-actions.js` with the following code. For now, we'll fetch data from the `contacts-data.js` file.

```
// src/actions/contact-actions.js

import { contacts } from '../contacts-data';

export function fetchContacts(){
  return dispatch => {
    dispatch({
      type: 'FETCH_CONTACTS',
      payload: contacts
    })
  }
}
```

In `contact-reducer.js`, let's write our handler for the `'FETCH_CONTACT'` action. We'll store the contacts data in an array called `'contacts'`.

```
// src/reducers/contact-reducer.js

const defaultState = {
  contacts: []
}
```

```
export default (state=defaultState, action={}) => {
  switch (action.type) {
    case 'FETCH_CONTACTS': {
      return {
        ...state,
        contacts: action.payload
      }
    }
    default:
      return state;
  }
}
```

In `reducers/index.js`, we'll combine all reducers here for easy export to our Redux store.

```
// src/reducers/index.js

import { combineReducers } from 'redux';
import ContactReducer from './contact-reducer';

const reducers = {
  contactStore: ContactReducer
}

const rootReducer = combineReducers(reducers);

export default rootReducer;
```

In `store.js`, we'll import the necessary dependencies to construct our Redux

store. We'll also set up the `redux-devtools-extension` here to enable us to monitor the Redux store using the Chrome extension.

```
// src/store.js

import { applyMiddleware, createStore } from "redux";
import thunk from "redux-thunk";
import promise from "redux-promise-middleware";
import { composeWithDevTools } from 'redux-devtools-extension';
import rootReducer from "./reducers";

const middleware = composeWithDevTools(applyMiddleware(promise(),
↪ thunk));

export default createStore(rootReducer, middleware);
```

Open `index.js` and update the render method where we inject the store using Redux's Provider class.

```
// src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter } from 'react-router-dom';
import { Provider } from 'react-redux';
import App from './App';
import store from "./store"
import 'semantic-ui-css/semantic.min.css';
import './index.css';
```

```
ReactDOM.render(
  <BrowserRouter>
    <Provider store={store}>
      <App />
    </Provider>
  </BrowserRouter>,
  document.getElementById('root')
);
'root'
```

Let's run `yarn start` to make sure everything is running so far.

Next, we'll **connect** our component `contact-list` with the Redux store we just created. Open `contact-list-page` and update the code as follows:

```
// src/pages/contact-list-page

import React, { Component} from 'react';
import { connect } from 'react-redux';
import ContactList from '../components/contact-list';
import { fetchContacts } from '../actions/contact-actions';

class ContactListPage extends Component {

  componentDidMount() {
    this.props.fetchContacts();
  }

  render() {
```

```
    return (
      <div>
        <h1>List of Contacts</h1>
        <ContactList contacts={this.props.contacts}/>
      </div>
    )
  }
}

// Make contacts  array available in  props
function mapStateToProps(state) {
  return {
      contacts : state.contactStore.contacts
  }
}

export default connect(mapStateToProps, {fetchContacts})
↪(ContactListPage);
```

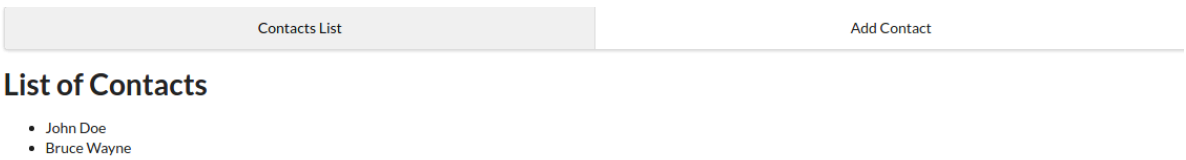
We've made the contacts array in store and the `fetchContacts` function available to `ContactListPage` component via `this.props` variable. We can now pass the contacts array down to the `ContactList` component.

For now, let's update the code such that we can display a list of contacts.

```
// src/components/contact-list

import React from 'react';
```

```
export default function ContactList({contacts}){

  const list = () => {
    return contacts.map(contact => {
      return (
        <li key={contact._id}>{contact.name.first}
        ↪{contact.name.last}</li>
      )
    })
  }

  return (
    <div>
      <ul>
        { list() }
      </ul>
    </div>
  )
}
```

If you go back to the browser, you should have something like this:



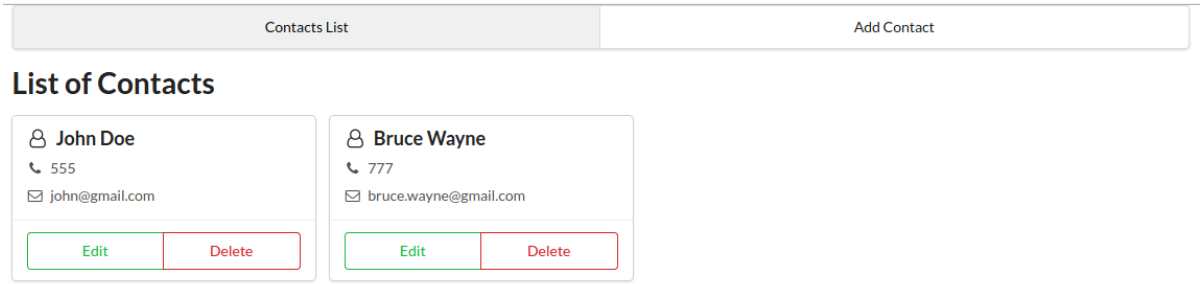
Let's make the list UI look more attractive by using semantic-ui's *Card* component. In the components folder, create a new file `contact-card.js` and paste this code:

```
// src/components/contact-card.js

import React from 'react';
import { Card, Button, Icon } from 'semantic-ui-react'

export default function ContactCard({contact, deleteContact}) {
  return (
    <Card>
      <Card.Content>
        <Card.Header>
          <Icon name='user outline'/> {contact.name.first}
          ↪ {contact.name.last}
        </Card.Header>
        <Card.Description>
          <p><Icon name='phone'/> {contact.phone}</p>
          <p><Icon name='mail outline'/> {contact.email}</p>
        </Card.Description>
      </Card.Content>
      <Card.Content extra>
        <div className="ui two buttons">
          <Button basic color="green">Edit</Button>
          <Button basic color="red">Delete</Button>
        </div>
      </Card.Content>
    </Card>
  )
}

ContactCard.propTypes = {
  contact: React.PropTypes.object.isRequired
}
```

Update `contact-list` component to use the new `ContactCard` component

```
// src/components/contact-list.js

import React from 'react';
import { Card } from 'semantic-ui-react';
import ContactCard from './contact-card';

export default function ContactList({contacts}){

  const cards = () => {
    return contacts.map(contact => {
      return (
        <ContactCard key={contact._id} contact={contact}/>
      )
    })
  }

  return (
    <Card.Group>
      { cards() }
    </Card.Group>
  )
}
```

The list page should now look like this:



Server-side Validation with Redux-Form

Now that we know the Redux store is properly linked up with the React components, we can now make a real fetch request to the database and use the data populate our contact list page. There are several ways to do this, but the way I'll show is surprisingly simple.

First, we need to configure an Axios client that can connect to the back-end server.

```
// src/actions/index.js
import axios from "axios";

export const client = axios.create({
  baseURL: "http://localhost:3030",
  headers: {
    "Content-Type": "application/json"
  }
})
```

Next, we'll update the `contact-actions.js` code to fetch contacts from the database via a GET request using the Axios client.

```
// src/actions/contact-actions.js

import { client } from './';

const url = '/contacts';

export function fetchContacts(){
  return dispatch => {
    dispatch({
      type: 'FETCH_CONTACTS',
      payload: client.get(url)
    })
  }
}
```

Update `contact-reducer.js` as well since the action and the payload being dispatched is now different.

```
// src/reducers/contact-reducer.js

…
    case "FETCH_CONTACTS_FULFILLED": {
      return {
        ...state,
        contacts: action.payload.data.data || action.payload.data
        ↪ // in case pagination is disabled
      }
    }
…
```

After saving, refresh your browser, and ensure the back-end server is running at `localhost:3030`. The contact list page should now be displaying data from the database.

Handle Create and Update Requests using Redux-Form

Next, let's look at how to add new contacts, and to do that we need forms. At first, building a form looks quite easy. But when we start thinking about client-side validation and controlling when errors should be displayed, it becomes tricky. In addition, the back-end server does its own validation, which we also need to display its errors on the form.

Rather than implement all the form functionality ourselves, we'll enlist the help of a library called Redux-Form. We'll also use a nifty package called Classnames that will help us highlight fields with validation errors.

We need to stop the server with `ctrl+c` before installing the following packages:

```
yarn add redux-form classnames
```

We can now start the server after the packages have finished installing.

Let's first quickly add this css class to the `index.css` file to style the form errors:

```
/* src/index.css */

.error {
```

```
  color: #9f3a38;
}
```

Then let's add redux-form's reducer to the `combineReducers` function in `reducers/index.js`

```
// src/reducers/index.js

…
import { reducer as formReducer } from 'redux-form';

const reducers = {
  contactStore: ContactReducer,
  form: formReducer
}
…
```

Next, open `contact-form.js` and build the form UI with this code:

```
// src/components/contact-form

import React, { Component } from 'react';
import { Form, Grid, Button } from 'semantic-ui-react';
import { Field, reduxForm } from 'redux-form';
import classnames from 'classnames';

class ContactForm extends Component {
```

```
renderField = ({ input, label, type, meta: { touched, error }
↪ }) => (
  <Form.Field className={classnames({error:touched && error})}>
    <label>{label}</label>
    <input {...input} placeholder={label} type={type}/>
    {touched && error && <span className="error">
    ↪{error.message}</span>}
  </Form.Field>
)

render() {
  const { handleSubmit, pristine, submitting, loading } = this.
  ↪props;

  return (
    <Grid centered columns={2}>
      <Grid.Column>
        <h1 style={{marginTop:"1em"}}>Add New Contact</h1>
        <Form onSubmit={handleSubmit} loading={loading}>
          <Form.Group widths='equal'>
            <Field name="name.first" type="text" component=
            ↪{this.renderField} label="First Name"/>
            <Field name="name.last" type="text" component=
            ↪{this.renderField} label="Last Name"/>
          </Form.Group>
          <Field name="phone" type="text" component=
          ↪{this.renderField} label="Phone"/>
          <Field name="email" type="text" component=
          ↪{this.renderField} label="Email"/>
          <Button primary type='submit' disabled={pristine ||
          ↪submitting}>Save</Button>
```

```
          </Form>
        </Grid.Column>
      </Grid>
    )
  }
}

export default reduxForm({form: 'contact'})(ContactForm);
'contact'
```

Take the time to examine the code; there's a lot going on in there. See the reference guide to understand how redux-form works. Also, take a look at semantic-ui-react documentation and read about its elements to understand how they are used in this context.

Next, we'll define the actions necessary for adding a new contact to the database. The first action will provide a new `contact` object to the Redux form. While the second action will post the `contact` data to the API server.

Append the following code to `contact-actions.js`

```
// src/actions/contact-actions.js

…

export function newContact() {
  return dispatch => {
    dispatch({
      type: 'NEW_CONTACT'
```

```
    })
  }
}

export function saveContact(contact) {
  return dispatch => {
    return dispatch({
      type: 'SAVE_CONTACT',
      payload: client.post(url, contact)
    })
  }
}
```

In the `contact-reducer`, we need to handle actions for `'NEW_CONTACT'`, `'SAVE_CONTACT_PENDING'`, `'SAVE_CONTACT_FULFILLED'`, and `'SAVE_CONTACT_REJECTED'`. We need to declare the following variables:

- contact - initialize empty object
- loading - update ui with progress info
- errors - store server validation errors in case something goes wrong

Add this code inside `contact-reducer`'s switch statement:

```
// src/reducers/contact-reducer.js

…
const defaultState = {
  contacts: [],
  contact: {name:{}},
```

```
    loading: false,
    errors: {}
  }
  …
  case 'NEW_CONTACT': {
        return {
          ...state,
          contact: {name:{}}
        }
      }

      case 'SAVE_CONTACT_PENDING': {
        return {
          ...state,
          loading: true
        }
      }

      case 'SAVE_CONTACT_FULFILLED': {
        return {
          ...state,
          contacts: [...state.contacts, action.payload.data],
          errors: {},
          loading: false
        }
      }

      case 'SAVE_CONTACT_REJECTED': {
        const data = action.payload.response.data;
        // convert feathers error formatting to match client-side
        ↪ error formatting
        const { "name.first":first, "name.last":last, phone, email
```

```
      ↪} = data.errors;
      const errors = { global: data.message, name: { first,last },
      ↪ phone, email };
      return {
        ...state,
        errors: errors,
        loading: false
      }
    }
  …
"name.first"
```

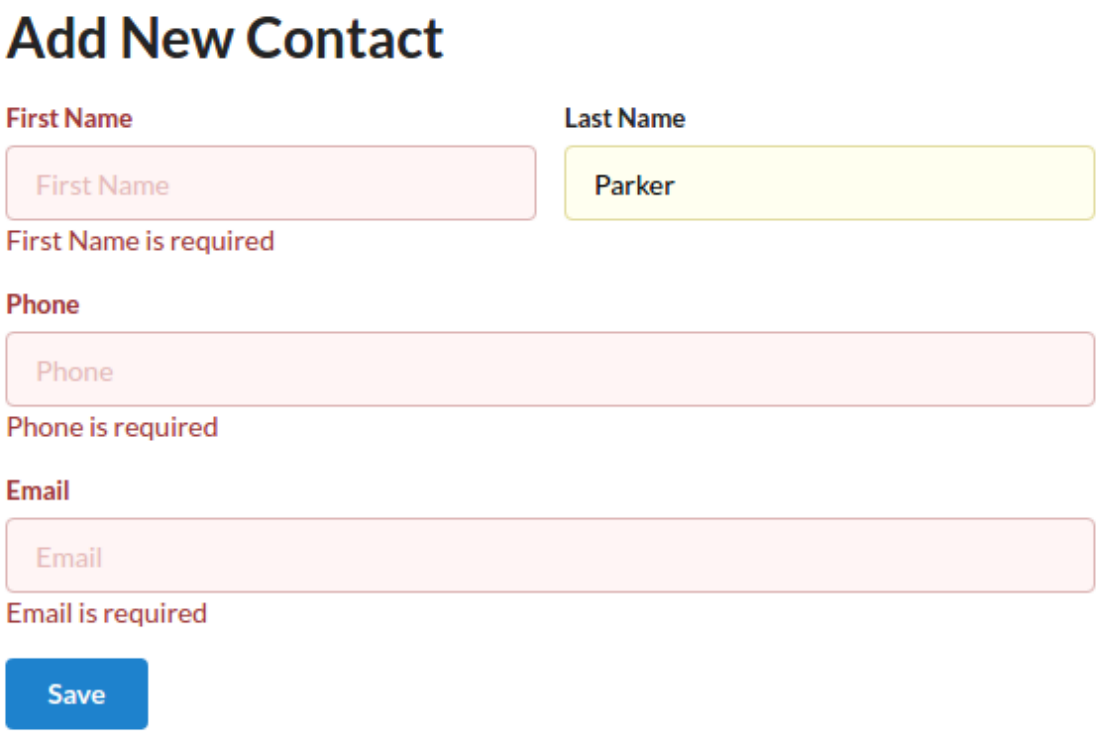
Open `contact-form-page.js` and update the code as follows:

```
// src/pages/contact-form-page

import React, { Component} from 'react';
import { Redirect } from 'react-router';
import { SubmissionError } from 'redux-form';
import { connect } from 'react-redux';
import { newContact, saveContact } from '../actions/
↪contact-actions';
import ContactForm from /actions/'../components/contact-form';


class ContactFormPage extends Component {

  state = {
    redirect: false
  }
```

```
  componentDidMount() {
    this.props.newContact();
  }

  submit = (contact) => {
    return this.props.saveContact(contact)
      .then(response => this.setState({ redirect:true }))
      .catch(err => {
         throw new SubmissionError(this.props.errors)
       })
  }

  render() {
    return (
      <div>
        {
          this.state.redirect ?
          <Redirect to="/" /> :
          <ContactForm contact={this.props.contact} loading=
          ↪{this.props.loading} onSubmit={this.submit} />
        }
      </div>
    )
  }
}

function mapStateToProps(state) {
  return {
    contact: state.contactStore.contact,
    errors: state.contactStore.errors
  }
```

```
}

export default connect(mapStateToProps, {newContact,
↪ saveContact})(ContactFormPage);
```

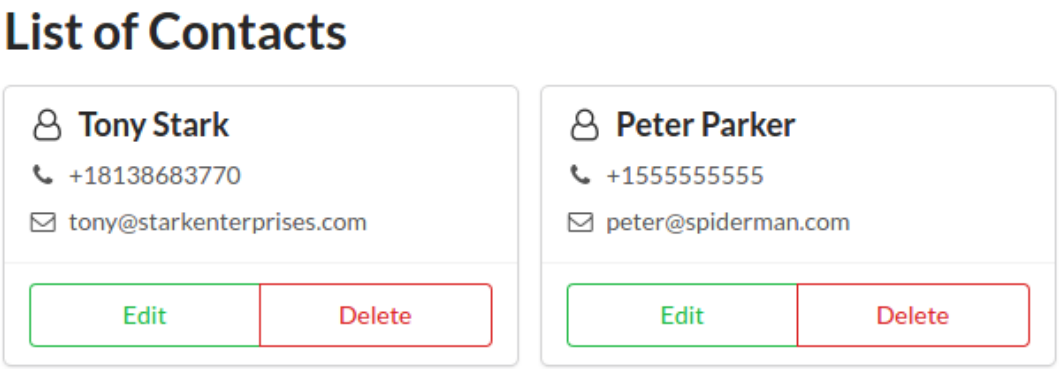
Let's now go back to the browser and try to intentionally save an incomplete form



As you can see, server-side validation prevents us from saving an incomplete contact. We're using the SubmissionError class to pass `this.props.errors` to the form, just in case you're wondering.

Now, finish filling in the form completely. After clicking save, we should be

directed to the list page.



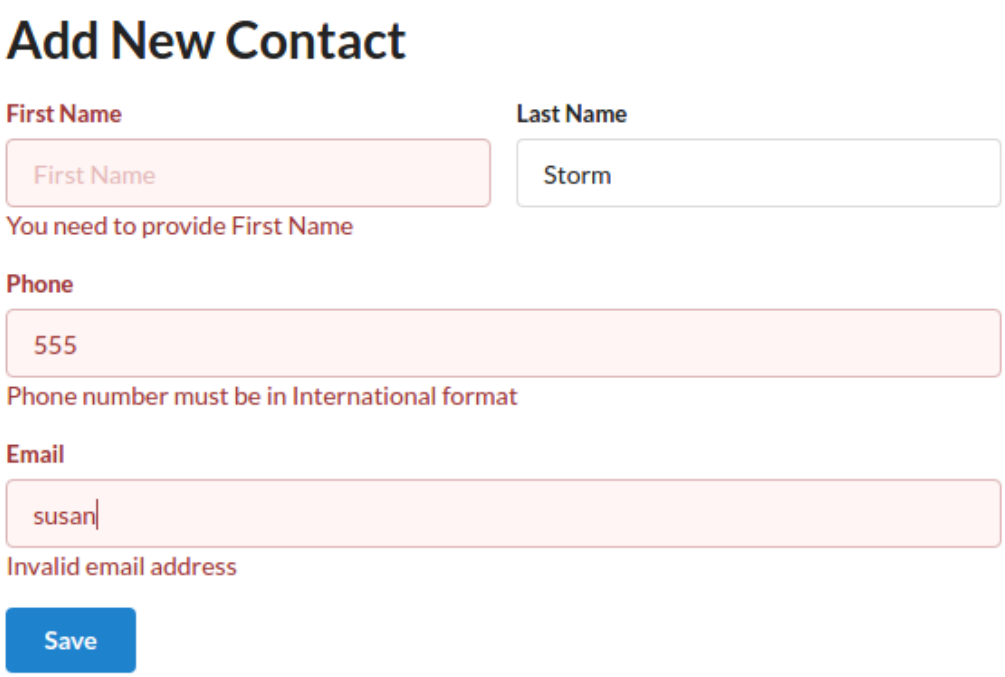
Client-side Validation with Redux Form

Let's take a look at how client-side validation can be implemented. Open `contact-form` and paste this code outside the ContactForm class. Also, update the default export as shown:

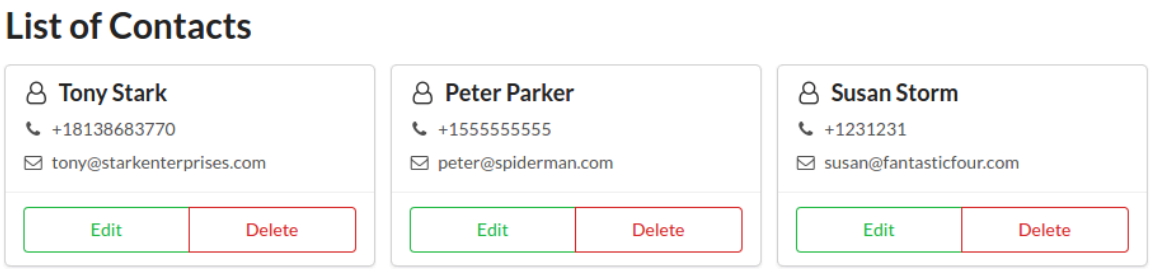
```
// src/components/contact-form.js

…
const validate = (values) => {
  const errors = {name:{}};
  if(!values.name || !values.name.first) {
    errors.name.first = {
      message: 'You need to provide First Name'
    }
  }
  if(!values.phone) {
    errors.phone = {
      message: 'You need to provide a Phone number'
```

```
    }
  } else if(!/^\+(?:[0-9] ?){6,14}[0-9]$/.test(values.phone)) {
    errors.phone = {
      message: 'Phone number must be
      ↪ in International format'
    }
  }
  if(!values.email) {
    errors.email = {
      message: /^\+(?:[0-9] ?){6,14}[0-9]$/'You need to provide an Email address'
    }
  } else if (!/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/i.
  ↪test(values.email)) {
    errors.email = {
      message: 'Invalid email address'
    }
  }
  return errors;
}
…

export default reduxForm({form: 'contact', validate})
↪(ContactForm);
```

After saving the file, go back to the browser and try adding invalid data. This time, the client side validation blocks submitting of data to the server.



Now, go ahead and input valid data. We should have at least three new contacts by now.



Implement Contact Updates

Now that we can add new contacts, let's see how we can update existing contacts. We'll start with the `contact-actions.js` file, where we need to define two actions – one for fetching a single contact, and another for updating the

contact.

```
// src/actions/contact-actions.js

…
export function fetchContact(_id) {
  return dispatch => {
    return dispatch({
      type: 'FETCH_CONTACT',
      payload: client.get(`${url}/${_id}`)
    })
  }
}

export function updateContact(contact) {
  return dispatch => {
    return dispatch({
      type: 'UPDATE_CONTACT',
      payload: client.put('UPDATE_CONTACT'`${url}/${contact._id}`
      ↪, contact)
    })
  }
}
```

Let's add the following cases to `contact-reducer` to update state when a contact is being fetched from the database and when it's being updated.

```
// src/reducers/contact-reducer.js

…
```

```
case 'FETCH_CONTACT_PENDING': {
  return {
    ...state,
    loading: true,
    contact: {name:{}}
  }
}

case 'FETCH_CONTACT_FULFILLED': {
  return {
    ...state,
    contact: action.payload.data,
    errors: {},
    loading: false
  }
}

case 'UPDATE_CONTACT_PENDING': {
  return {
    ...state,
    loading: true
  }
}

case 'UPDATE_CONTACT_FULFILLED': {
  const contact = action.payload.data;
  return {
    ...state,
    contacts: state.contacts.map(item => item._id ===
    ↪ contact._id ? contact : item),
    errors: {},
    loading: false
```

```
    }
  }

  case 'UPDATE_CONTACT_REJECTED': {
    const data = action.payload.response.data;
    const { "name.first":first, "name.last":last, phone, email }
    ↪ = data.errors;
    const errors = { global: data.message, name: { first,last },
    ↪ phone, email };
    return {
      ...state,
      errors: errors,
      loading: false
    }
  }
  …
```

Next, let's pass the new fetch and save actions to the `contact-form-page.js`. We'll also change the `componentDidMount()` and `submit()` logic to handle both create and update scenarios. Be sure to update each section of code as indicated below.

```
// src/pages/contact-form-page.js

…
import { newContact, saveContact, fetchContact, updateContact }
↪ from '../actions/contact-actions';

…
```

```
componentDidMount = () => {
  const { _id } = this.props.match.params;
  if(_id){
    this.props.fetchContact(_id)
  } else {
    this.props.newContact();
  }
}

submit = (contact) => {
  if(!contact._id) {
    return this.props.saveContact(contact)
      .then(response => this.setState({ redirect:true }))
      .catch(err => {
        throw new SubmissionError(this.props.errors)
       })
  } else {
    return this.props.updateContact(contact)
      .then(response => this.setState({ redirect:true }))
      .catch(err => {
        throw new SubmissionError(this.props.errors)
       })
  }
}

…

export default connect(
  mapStateToProps, {newContact, saveContact, fetchContact,
    ↪ updateContact})(ContactFormPage);
```

We'll enable `contact-form` to asynchronously receive data from the

`fetchContact()` action. To populate a Redux Form, we use its initialize function that's been made available to us via the `props`. We'll also update the page title with a script to reflect whether we are editing or adding new a contact.

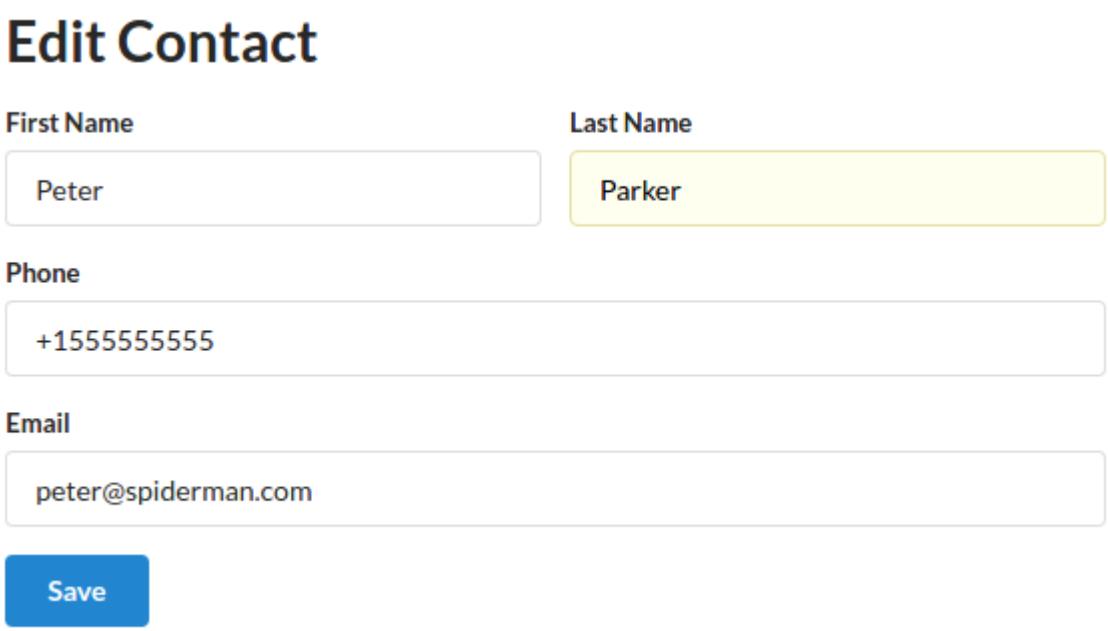
```
// src/components/contact-form.js

…
componentWillReceiveProps = (nextProps) => { // Receive Contact
↪ data Asynchronously
  const { contact } = nextProps;
  if(contact._id !== this.props.contact._id) { // Initialize
  ↪ form only once
    this.props.initialize(contact)
  }
}
…

  <h1 style={{marginTop:"1em"}}>{this.props.contact._id ?
    ↪ 'Edit Contact' : 'Add New Contact'}</h1>

…
```

Now, let's convert the **Edit** button in `contact-card.js` to a link that will direct the user to the form.

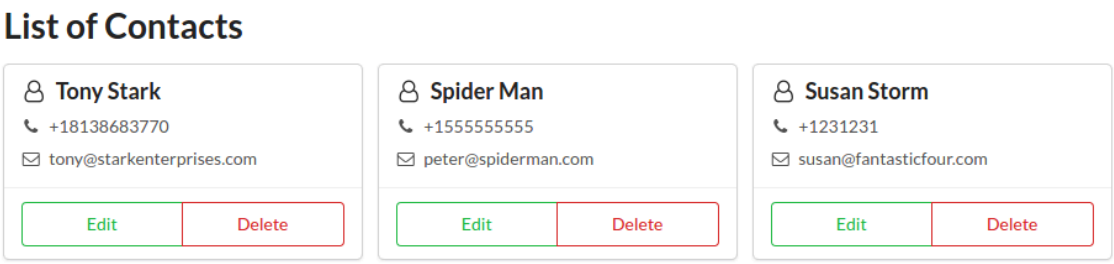
```
// src/components/contact-card.js

…
import { Link } from 'react-router-dom';
```

```
…
  <div className="ui two buttons">
    <Link to={`/contacts/edit/${contact._id}`} className="ui
    ↪ basic button green">Edit</Link>
    <Button basic color="red">Delete</Button>
  </div>
…
```

Once the list page has finished refreshing, choose any contact and hit the Edit button.



Finish making your changes and hit save.



By now, your application should be able to allow users to add new contacts and update existing ones.

Implement Delete Request

Let's now look at the final CRUD operation: delete. This one is much simpler to code. We start at the `contact-actions.js` file.

```
// src/actions/contact-actions.js

…
export function deleteContact(_id) {
  return dispatch => {
    return dispatch({
      type: 'DELETE_CONTACT',
      payload: client.delete(`${url}/${_id}`)
    })
  }
}
```

By now, you should have gotten the drill. Define a case for the `deleteContact()` action in `contact-reducer.js`.

```
// src/reducers/contact-reducer.js

…
case 'DELETE_CONTACT_FULFILLED': {
    const _id = action.payload.data._id;
    return {
      ...state,
      contacts: state.contacts.filter(item => item._id !== _id)
    }
  }
…
```

Next, we import the `deleteContact()` action to `contact-list-page.js` and pass it to the `ContactList` component.

```
// src/pages/contact-list-page.js

…
import { fetchContacts, deleteContact } from
↪ '../actions/contact-actions';
…

<ContactList contacts={this.props.contacts}
↪ deleteContact={this.props.deleteContact}/>

…

export default connect(mapStateToProps, {fetchContacts,
↪ deleteContact})(ContactListPage);
```

The `ContactList` component, in turn, passes the `deleteContact()` action to the `ContactCard` component

```
// src/components/contact-list.js

…
export default function ContactList({contacts, deleteContact}){
↪ // replace this line

const cards = () => {
  return contacts.map(contact => {
    return (
      <ContactCard
      key={contact._id}
      contact={contact}
      deleteContact={deleteContact} /> // and this one
    )
  })
}
…
```

Finally, we update **Delete** button in `ContactCard` to execute the `deleteContact()` action, via the `onClick` attribute.

```
// src/components/contact-card.js

…
<Button basic color="red" onClick={() =>
↪ deleteContact(contact._id)} >Delete</Button>
```

```
...
```

Wait for the browser to refresh, then try to delete one or more contacts. The delete button should work as expected.

Conclusion

By now, you should have learned the basics of creating a CRUD web app in JavaScript. It may seem we've written quite a lot of code to manage only one model. We could have done less work if we had used an MVC framework. The problem with these frameworks is that they become harder to maintain as the code grows.

A Flux-based framework, such as Redux, allows us to build large complex projects that are easy to manage. If you don't like the verbose code that Redux requires you to write, then you could also look at Mobx as an alternative.

At least I hope you now have a good impression of FeathersJS. With little effort, we were able to generate a database API with only a few commands and a bit of coding. Although we have only scratched the surface in exploring its capabilities, you will at least agree with me that it is a robust solution for creating APIs.

Chapter

3

How to Build a Todo App Using React, Redux, and Immutable.js

by Dan Prince

The way React uses components and a one-way data flow makes it ideal for describing the structure of user interfaces. However, its tools for working with state are kept deliberately simple – to help remind us that React is just the View in the traditional Model-View-Controller architecture.

There's nothing to stop us from building large applications with just React, but we would quickly discover that to keep our code simple, we'd need to manage our state elsewhere.

Whilst there's no *official* solution for dealing with application state, there are some libraries that align particularly well with React's paradigm. In this post, we'll pair React with two such libraries and use them to build a simple application.

Redux

Redux is a tiny library that acts as a container for our application state, by combining ideas from Flux and Elm. We can use Redux to manage any kind of application state, providing we stick to the following guidelines:

1. our state is kept in a single store
2. changes come from *actions* and not *mutations*

At the core of a Redux store is a function that takes the current application state and an action and combines them to create a new application state. We call this function a **reducer**.

Our React components will be responsible for sending actions to our store, and in turn our store will tell the components when they need to re-render.

ImmutableJS

Because Redux doesn't allow us to mutate the application state, it can be helpful to enforce this by modeling application state with immutable data structures.

ImmutableJS offers us a number of immutable data structures with mutative interfaces, and they're implemented in an efficient way, inspired by the implementations in Clojure and Scala.

Demo

We're going to use React with Redux and ImmutableJS to build a simple todo list that allows us to add todos and toggle them between complete and incomplete. You can find a CodePen demo here, and the che code is available in a repository on GitHub.

Setup

We'll get started by creating a project folder and initializing a `package.json` file with `npm init`. Then we'll install the dependencies we're going to need.

```
npm install --save react react-dom redux react-redux immutable
npm install --save-dev webpack babel-core babel-loader babel-preset-es2015
↪ babel-preset-react
```

We'll be using JSX and ES2015, so we'll compile our code with Babel, and we're going to do this as part of the module bundling process with Webpack.

First, we'll create our Webpack configuration in `webpack.config.js`:

```
module.exports = {
  entry: './src/app.js',
  output: {
    path: __dirname,
```

```
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: /\.js$/'babel-loader',
        query: { presets: [ 'es2015', 'react' ] }
      }
    ]
  }
};
```

Finally, we'll extend our `package.json` by adding an npm script to compile our code with source maps:

```
"script": {
  "build": "webpack --debug"
}
```

We'll need to run `npm run build` each time we want to compile our code.

React and Components

Before we implement any components, it can be helpful to create some dummy data. This helps us get a feel for what we're going to need our components to render:

```
const dummyTodos = [
  { id: 0, isDone: true,  text: 'make components' },
  { id: 1, isDone: false, text: 'design actions' },
  { id: 2, isDone: false, text: 'implement reducer' },
  { id: 3, isDone: false, text: 'connect components' }
];
```

For this application, we're only going to need two React components, `<Todo />` and `<TodoList />`.

```
// src/components.js

import React from 'react';

export function Todo(props) {
  const { todo } = props;
  if(todo.isDone) {
    return <strike>{todo.text}</strike>;
  } else {
    return <span>{todo.text}</span>;
  }
}

export function TodoList(props) {
  const { todos } = props;
  return (
    <div className='todo'>
      <input type='text' placeholder='Add todo' />
      <ul className='todo__list'>
        {todos.map(t => (
```

```
          <li key={t.id} className='todo__item'>
            <Todo todo={t} />
          </li>
        ))}
      </ul>
    </div>
  );
}
```

At this point, we can test these components by creating an `index.html` file in the project folder and populating it with the following markup. (You can find a simple stylesheet on GitHub):

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="style.css">
    <title>Immutable Todo</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="bundle.js"></script>
  </body>
</html>
```

We'll also need an application entry point at `src/app.js`.

```
// src/app.js
```

```
import React from 'react';
import { render } from 'react-dom';
import { TodoList } from './components';

const dummyTodos = [
  { id: 0, isDone: true,  text: 'make components' },
  { id: 1, isDone: false, text: 'design actions' },
  { id: 2, isDone: false, text: 'implement reducer' },
  { id: 3, isDone: false, text: 'connect components' }
];

render(
  <TodoList todos={dummyTodos} />,
  document.getElementById('app')
);
```

Compile the code with `npm run build`, then navigate your browser to the `index.html` file and make sure that it's working.

Redux and Immutable

Now that we're happy with the user interface, we can start to think about the state behind it. Our dummy data is a great place to start from, and we can easily translate it into ImmutableJS collections:

```
import { List, Map } from 'immutable';

const dummyTodos = List([
  Map({ id: 0, isDone: true,  text: 'make components' }),
```

```
  Map({ id: 1, isDone: false, text: 'design actions' }),
  Map({ id: 2, isDone: false, text: 'implement reducer' }),
  Map({ id: 3, isDone: false, text: 'connect components' })
]);
```

ImmutableJS maps don't work in the same way as JavaScript's objects, so we'll need to make some slight tweaks to our components. Anywhere there was a property access before (e.g. `todo.id`) needs to become a method call instead (`todo.get('id')`).

Designing Actions

Now that we've got the shape and structure figured out, we can start thinking about the actions that will update it. In this case, we'll only need two actions, one to add a new todo and the other to toggle an existing one.

Let's define some functions to create these actions:

```
// src/actions.js

// succinct hack for generating passable unique ids
const uid = () => Math.random().toString(34).slice(2);

export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    payload: {
      id: uid(),
      isDone: false,
     'ADD_TODO't
```

```
    }
  };
}

export function toggleTodo(id) {
  return {
    type: 'TOGGLE_TODO',
    payload: id
  }
}
```

Each action is just a JavaScript object with a type and payload properties. The type property helps us decide what to do with the payload when we process the action later.

Designing a Reducer

Now that we know the shape of our state and the actions that update it, we can build our reducer. Just as a reminder, the reducer is a function that takes a state and an action, then uses them to compute a new state.

Here's the initial structure for our reducer:

```
// src/reducer.js

import { List, Map } from 'immutable';

const init = List([]);

export default function(todos=init, action) {
```

```
  switch(action.type) {
    case 'ADD_TODO':
      // …
    case 'TOGGLE_TODO':
      'TOGGLE_TODO'// …
    default:
      return todos;
  }
}
```

Handling the `ADD_TODO` action is quite simple, as we can use the .push() method, which will return a new list with the todo appended at the end:

```
case 'ADD_TODO':
  return todos.push(Map(action.payload));
```

Notice that we're also converting the todo object into an immutable map before it's pushed onto the list.

The more complex action we need to handle is `TOGGLE_TODO`:

```
case 'TOGGLE_TODO':
  return todos.map(t => {
    if(t.get('id') === action.payload) {
      return t.update('isDone', isDone => !isDone);
    } else {
      return t;
    }
```

```
  });
```

We're using .map() to iterate over the list and find the todo whose `id` matches the action. Then we call .update(), which takes a key and a function, then it returns a new copy of the map, with the value at the key replaced with the result of passing the initial value to the update function.

It might help to see the literal version:

```
const todo = Map({ id: 0, text: 'foo', isDone: false });
todo.update('isDone', isDone => !isDone);
// => { id: 0, text: 'foo', isDone: true }
```

Connecting Everything

Now we've got our actions and reducer ready, we can create a store and connect it to our React components:

```
// src/app.js

import React from 'react';
import { render } from 'react-dom';
import { createStore } from 'redux';
import { TodoList } from './components';
import reducer from './reducer';

const store = createStore(reducer);
```

```
render(
  <TodoList todos={store.getState()} />,
  document.getElementById('app')
);
```

We'll need to make our components aware of this store. We'll use the react-redux to help simplify this process. It allows us to create store-aware containers that wrap around our components, so that we don't have to change our original implementations.

We're going to need a container around our `<TodoList />` component. Let's see what this looks like:

```
// src/containers.js

import { connect } from 'react-redux';
import * as components from './components';
import { addTodo, toggleTodo } from './actions';

export const TodoList = connect(
  function mapStateToProps(state) {
    // …
  },
  function mapDispatchToProps(dispatch) {
    // …
  }
)(components.TodoList);
```

We create containers with the connect function. When we call `connect()`, we

pass two functions, `mapStateToProps()` and `mapDispatchToProps()`.

The `mapStateToProps` function takes the store's current state as an argument (in our case, a list of todos), then it expects the return value to be an object that describes a mapping from that state to props for our wrapped component:

```
function mapStateToProps(state) {
  return { todos: state };
}
```

It might help to visualize this on an instance of the wrapped React component:

```
<TodoList todos={state} />
```

We'll also need to supply a `mapDispatchToProps` function, which is passed the store's `dispatch` method, so that we can use it to dispatch the actions from our action creators:

```
function mapDispatchToProps(dispatch) {
  return {
    addTodo: text => dispatch(addTodo(text)),
    toggleTodo: id => dispatch(toggleTodo(id))
  };
}
```

Again, it might help to visualize all these props together on an instance of our wrapped React component:

```
<TodoList todos={state}
  addTodo={text => dispatch(addTodo(text))}
  toggleTodo={id => dispatch(toggleTodo(id))} />
```

Now that we've mapped our component to the action creators, we can call them from event listeners:

```
export function TodoList(props) {
  const { todos, toggleTodo, addTodo } = props;

  const onSubmit = (event) => {
    const input = event.target;
    const text = input.value;
    const isEnterKey = (event.which == 13);
    const isLongEnough = text.length > 0;

    if(isEnterKey && isLongEnough) {
      input.value = '';
      addTodo(text);
    }
  };

  const toggleClick = id => event => toggleTodo(id);

  return (
    <div className='todo'>
      <input type='text'
         className='todo__entry'
         placeholder='Add todo'
         onKeyDown={onSubmit} />
```

```
      <ul className='todo__list'>
        {todos.map(t => (
          <li key={t.get('id')}
            className='todo__item'
            onClick={toggleClick(t.get('id'))}>
            <Todo todo={t.toJS()} />
          </li>
        ))}
      </ul>
    </div>
  );
}
```

The containers will automatically subscribe to changes in the store, and they'll re-render the wrapped components whenever their mapped props change.

Finally, we need to make the containers aware of the store, using the `<Provider />` component:

```
// src/app.js

import React from 'react';
import { render } from 'react-dom';
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import reducer from './reducer';
import { TodoList } from './containers';
//                          ^^^^^^^^^^

const store = createStore(reducer);
```

```
render(
  <Provider store={store}>
    <TodoList />
  </Provider>,
  document.getElementById('app')
);
'app'
```

Conclusion

There's no denying that the ecosystem around React and Redux can be quite complex and intimidating for beginners, but the good news is that almost all of these concepts are transferable. We've barely touched the surface of Redux's architecture, but already we've seen enough to help us start learning about The Elm Architecture, or pick up a ClojureScript library like Om or Re-frame. Likewise, we've only seen a fraction of the possibilities with immutable data, but now we're better equipped to start learning a language like Clojure or Haskell.

Whether you're just exploring the state of web application development, or you spend all day writing JavaScript, experience with action-based architectures and immutable data is already becoming a vital skill for developers, and *right now* is a great time to be learning the essentials.

Chapter

4

Building a Game with Three.js, React and WebGL

by Andrew Ray

I'm making a game titled "Charisma The Chameleon." It's built with Three.js, React and WebGL. This is an introduction to how these technologies work together using react-three-renderer (abbreviated R3R).

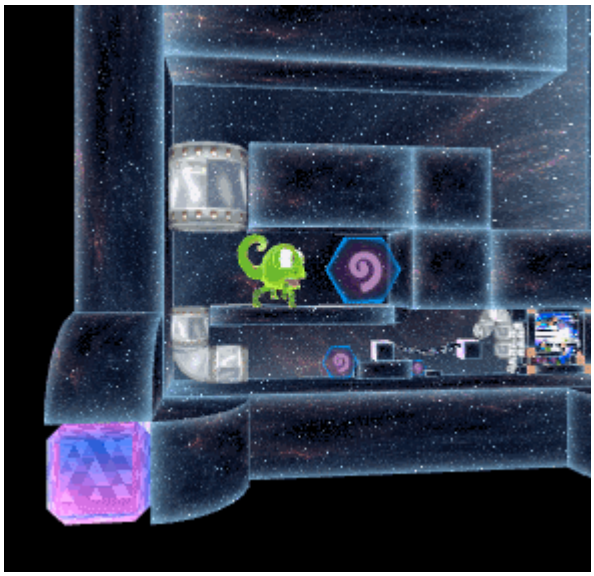
How It All Began

Some time ago, Pete Hunt made a joke about building a game using React in the #reactjs IRC channel:

> I bet we could make a first person shooter with React! Enemy has `<Head /> <Body> <Legs>` etc.

I laughed. He laughed. Everyone had a great time. "Who on earth would do that?" I wondered.

Years later, that's exactly what I'm doing.



Charisma The Chameleon is a game where you collect power-ups that make you shrink to solve an infinite fractal maze. I've been a React developer for a few years, and I was curious if there was a way to drive Three.js using React. That's when R3R caught my eye.

Why React?

I know what you're thinking: **why?** Humor me for a moment. Here's some reasons to consider using React to drive your 3D scene:

- "Declarative" views let you cleanly separate your scene rendering from your game logic.
- Design easy to reason about components, like `<Player />`, `<Wall />`, `<Level />`, etc.
- "Hot" (live) reloading of game assets. Change textures and models and see them update live in your scene!
- Inspect and debug your 3D scene as markup with native browser tools, like the Chrome inspector.
- Manage game assets in a dependency graph using Webpack, eg `<Texture src={ require('../assets/image.png') } />`

Let's set up a scene to get an understanding of how this all works.

React and WebGL

I've created a sample GitHub repository to accompany this article. Clone the repository and follow the instructions in the README to run the code and follow along. It stars SitePointy the 3D Robot!

Watch Out!

R3R is still in beta. Its API is volatile and may change in the future. It only handles a subset of Three.js at the moment. I've found it complete enough to build a full game, but your mileage may vary.

Organizing view code

The main benefit of using React to drive WebGL is our view code is **decoupled** from our game logic. That means our rendered entities are small components that are easy to reason about.

R3R exposes a declarative API that wraps Three.js. For example, we can write:

```
<scene>
  <perspectiveCamera
    position={ new THREE.Vector3( 1, 1, 1 )
  />
</scene>
```

Now we have an empty 3D scene with a camera. Adding a mesh to the scene is as simple as including a `<mesh />` component, and giving it `<geometry />` and a `<material />`.

```
<scene>
  …
  <mesh>
    <boxGeometry
      width={ 1 }
```

```
      height={ 1 }
      depth={ 1 }
    />
    <meshBasicMaterial
      color={ 0x00ff00 }
    />
</mesh>
```

Under the hood, this creates a THREE.Scene and automatically adds a mesh with THREE.BoxGeometry. R3R handles diffing the old scene with any changes. If you add a new mesh to the scene, the original mesh won't be recreated. Just as with vanilla React and the DOM, the 3D scene is **only updated with the differences.**

Because we're working in React, we can separate game entities into component files. The Robot.js file in the example repository demonstrates how to represent the main character with pure React view code. It's a "stateless functional" component, meaning it doesn't hold any local state:

```
const Robot = ({ position, rotation }) => <group
  position={ position }
  rotation={ rotation }
>
  <mesh rotation={ localRotation }>
    <geometryResource
      resourceId="robotGeometry"
    />
    <materialResource
      resourceId="robotTexture"
    />
  </mesh>
```

```
</group>;
```

And now we include the `<Robot />` in our 3D scene!

```
<scene>
  …
  <mesh>…</mesh>
  <Robot
    position={…}
    rotation={…}
  />
</scene>
```

You can see more examples of the API on the R3R GitHub repository, or view the complete example setup in the accompanying project.

Organizing Game Logic

The second half of the equation is handling game logic. Let's give SitePointy, our robot, some simple animation.

How do game loops traditionally work? They accept user input, analyze the old "state of the world," and return the new state of the world for rendering. For convenience, let's store our "game state" object in component state. In a more mature project, you could move the game state into a Redux or Flux store.

We'll use the browser's `requestAnimationFrame` API callback to drive our game loop, and run the loop in GameContainer.js. To animate the robot, let's calculate a new position based on the timestamp passed to `requestAnimationFrame`, then store the new position in state.

```
// …
gameLoop( time ) {
  this.setState({
    robotPosition: new THREE.Vector3(
      Math.sin( time * 0.01 ), 0, 0
    )
  });
}
```

Calling `setState()` triggers a re-render of the child components, and the 3D

scene updates. We pass the state down from the container component to the presentational `<Game />` component:

```
render() {
  const { robotPosition } = this.state;
  return <Game
    robotPosition={ robotPosition }
  />;
}
```

There's a useful pattern we can apply to help organize this code. Updating the robot position is a simple time-based calculation. In the future, it might also take into account the previous robot position from the previous game state. A function that takes in some data, processes it, and returns new data, is often referred to as a **reducer**. We can abstract out the movement code into a reducer function!

Now we can write a clean, simple game loop that only has function calls in it:

```
import robotMovementReducer from './game-reducers/
↪robotMovementReducer.js';

// …

gameLoop() {
  const oldState = this.state;
  const newState = robotMovementReducer(
  ↪ oldState );
  this.setState( newState );
}
```

To add more logic to the game loop, such as processing physics, create another reducer function and pass it the result of the previous reducer:

```
const newState = physicsReducer( robotMovementReducer( oldState ) );
```

As your game engine grows, organizing game logic into separate functions becomes critical. This organization is straightforward with the reducer pattern.

Asset management

This is still an evolving area of R3R. For textures, you specify a `url` attribute on the JSX tag. Using Webpack, you can require the local path to the image:

```
<texture url={ require( '../local/image/path.png' ) } />
```

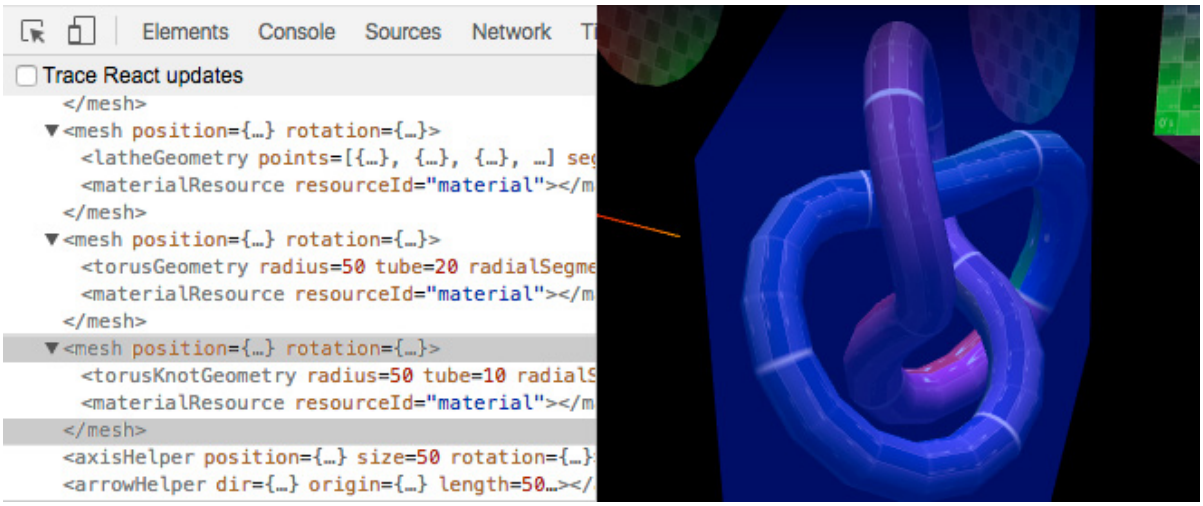
With this setup, if you change the image on disk, your 3D scene will live update! This is invaluable for rapidly iterating game design and content.

For other assets like 3D models, you still have to process them using the built-in loaders from Three.js, like the JSONLoader. I experimented with using a custom Webpack loader for loading 3D model files, but in the end it was too much work for no benefit. It's easier to treat the model as binary data and load them with the file-loader. This still affords live reloading of model data. You can see this in action in the example code.

Debugging

R3R supports the React developer tools extension for both Chrome and Firefox. You can inspect your scene as if it were the vanilla DOM! Hovering over elements in the inspector shows their bounding box in the scene. You can also hover over

texture definitions to see which objects in the scene use those textures.



You can also join us in the react-three-renderer Gitter chat room for help debugging your applications.

Performance Considerations

While building Charisma The Chameleon, I've run into several performance issues that are unique to this workflow.

- My **hot reload time** with Webpack was as long as thirty seconds! This is because large assets have to be re-written to the bundle on every reload. The solution was to implement Webpack's DLLPlugin, which cut down reload times to below five seconds.
- Ideally your scene should only call **one `setState()`** per frame render. After profiling my game, React itself is the main bottleneck. Calling `setState()` more than once per frame can cause double renders and reduce performance.
- Past a certain number of objects, **R3R will perform worse** than vanilla Three.js code. For me this was around 1,000 objects. You can compare R3R to Three.js under "Benchmarks" in the examples.

The Chrome DevTools Timeline feature is an amazing tool for debugging

performance. It's easy to visually inspect your game loop, and it's more readable than the "Profile" feature of the DevTools.

That's It!

Check out Charisma The Chameleon to see what's possible using this setup. While this toolchain is still quite young, I've found React with R3R to be integral to organizing my WebGL game code cleanly. You can also check out the small but growing R3R examples page to see some well organized code samples.

Chapter

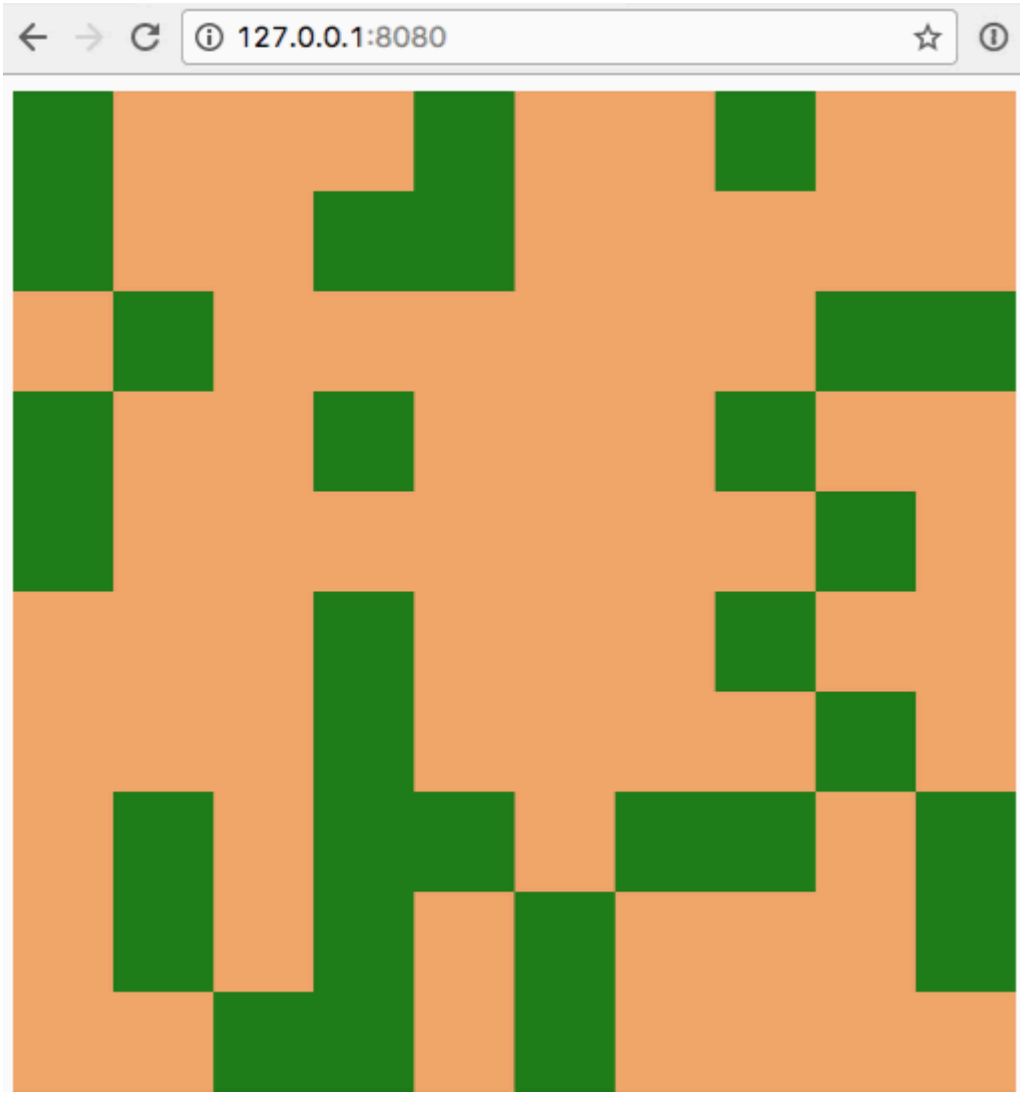
5

Procedurally Generated Game Terrain with React, PHP, and WebSockets

by Christopher Pitt

Last time, I began telling you the story of how I wanted to make a game. I described how I set up the async PHP server, the Laravel Mix build chain, the React front end, and WebSockets connecting all this together. Now, let me tell you about what happened when I starting building the game mechanics with this mix of React, PHP, and WebSockets…

he code for this part can be found at github.com/assertchris-tutorials/sitepoint-making-games/tree/part-2. I've tested it with PHP `7.1`, in a recent version of Google Chrome.



Making a Farm

"Let's start simple. We have a 10 by 10 grid of tiles, filled with randomly generated stuff."

I decided to represent the farm as a `Farm`, and each tile as a `Patch`. From `app/Model/FarmModel.pre`:

```
namespace App\Model;

class Farm
{
  private $width
  {
    get { return $this->width; }
  }

  private $height
  {
    get { return $this->height; }
  }

  public function __construct(int $width = 10,
    int $height = 10)
  {
    $this->width = $width;
    $this->height = $height;
  }
}
```

I thought it would be a fun time to try out the class accessors macro by declaring

private properties with public getters. For this I had to install `pre/class-accessors` (via `composer require`).

I then changed the socket code to allow for new farms to be created on request. From `app/Socket/GameSocket.pre`:

```
namespace App\Socket;

use Aerys\Request;
use Aerys\Response;
use Aerys\Websocket;
use Aerys\Websocket\Endpoint;
use Aerys\Websocket\Message;
use App\Model\FarmModel;

class GameSocket implements Websocket
{
  private $farms = [];

  public function onData(int $clientId,
    Message $message)
  {
    $body = yield $message;

    if ($body === "new-farm") {
      $farm = new FarmModel();

      $payload = json_encode([
        "farm" => [
          "width" => $farm->width,
          "height" => $farm->height,
```

```
          ],
        ]);

        yield $this->endpoint->send(
          $payload, $clientId
        );

        $this->farms[$clientId] = $farm;
      }
    }

    public function onClose(int $clientId,
      int $code, string $reason)
    {
      unset($this->connections[$clientId]);
      unset($this->farms[$clientId]);
    }

    // …
  }
```

I noticed how similar this `GameSocket` was to the previous one I had – except, instead of broadcasting an echo, I was checking for `new-farm` and sending a message back only to the client that had asked.

"Perhaps it's a good time to get less generic with the React code. I'm going to rename `component.jsx` to `farm.jsx`."

From `assets/js/farm.jsx`:

```
import React from "react"


class Farm extends React.Component
{
  componentWillMount()
  {
    this.socket = new WebSocket(
      "ws://127.0.0.1:8080/ws"
    )

    this.socket.addEventListener(
      "message", this.onMessage
    )

    // DEBUG

    this.socket.addEventListener("open", () => {
      this.socket.send("new-farm")
    })
  }
}

export default Farm
"open"
```

In fact, the only other thing I changed was sending `new-farm` instead of `hello world`. Everything else was the same. I did have to change the `app.jsx` code though. From `assets/js/app.jsx`:

```
import React from "react"
import ReactDOM from "react-dom"
import Farm from "./farm"

ReactDOM.render(
  <Farm />,
  document.querySelector(".app")
)ument.querySelector(".app")
)
```

It was far from where I needed to be, but using these changes I could see the class accessors in action, as well as prototype a kind of request/response pattern for future WebSocket interactions. I opened the console, and saw `{"farm":{"width":10,"height":10}}`.

"Great!"

Then I created a `Patch` class to represent each tile. I figured this was where a lot of the game's logic would happen. From `app/Model/PatchModel.pre`:

```
namespace App\Model;

  private $x
  {
    get { return $this->x; }
  }

  private $y
  {
```

```
    get { return $this->y; }
  }

  public function __construct(int $x, int $y)
  {
    $this->x = $x;
    $this->y = $y;
  }
}
```

I'd need to create as many patches as there are spaces in a new `Farm`. I could do this as part of `FarmModel` construction. From `app/Model/FarmModel.pre`:

```
namespace App\Model;

class FarmModel
{
  private $width
  {
    get { return $this->width; }
  }

  private $height
  {
    get { return $this->height; }
  }

  private $patches
  {
    get { return $this->patches; }
```

```
  }

  public function __construct($width = 10, $height = 10)
  {
    $this->width = $width;
    $this->height = $height;

    $this->createPatches();
  }

  private function createPatches()
  {
    for ($i = 0; $i < $this->width; $i++) {
      $this->patches[$i] = [];

      for ($j = 0; $j < $this->height; $j++) {
        $this->patches[$i][$j] =
        new PatchModel($i, $j);
      }
    }
  }
}
```

For each cell, I created a new `PatchModel` object. These were pretty simple to begin with, but they needed an element of randomness – a way to grow trees, weeds, flowers ... at least to begin with. From `app/Model/PatchModel.pre`:

```
public function start(int $width, int $height,
array $patches)
{
```

```
    if (!$this->started && random_int(0, 10) > 7) {
      $this->started = true;
      return true;
    }

    return false;
  }
```

I thought I'd begin just by randomly growing a patch. This didn't change the external state of the patch, but it did give me a way to test how they were started by the farm. From `app/Model/FarmModel.pre`:

```
namespace App\Model;

use Amp;
use Amp\Coroutine;
use Closure;

class FarmModel
{
  private $onGrowth
  {
    get { return $this->onGrowth; }
  }

  private $patches
  {
    get { return $this->patches; }
  }
```

```
public function __construct(int $width = 10,
int $height = 10, Closure $onGrowth)
{
  $this->width = $width;
  $this->height = $height;
  $this->onGrowth = $onGrowth;
}

public async function createPatches()
{
  $patches = [];

  for ($i = 0; $i < $this->width; $i++) {
    $this->patches[$i] = [];

    for ($j = 0; $j < $this->height; $j++) {
      $this->patches[$i][$j] = $patches[] =
      new PatchModel($i, $j);
    }
  }

  foreach ($patches as $patch) {
    $growth = $patch->start(
      $this->width,
      $this->height,
      $this->patches
    );

    if ($growth) {
      $closure = $this->onGrowth;
      $result = $closure($patch);
```

```
        if ($result instanceof Coroutine) {
          yield $result;
        }
      }
    }
  }

  // …
}
```

There was a lot going on here. For starters, I introduced an `async` function keyword using a macro. You see, Amp handles the `yield` keyword by resolving Promises. More to the point: when Amp sees the `yield` keyword, it assumes what is being yielded is a Coroutine (in most cases).

I could have made the `createPatches` function a normal function, and just returned a Coroutine from it, but that was such a common piece of code that I might as well have created a special macro for it. At the same time, I could replace code I had made in the previous part. From `helpers.pre`:

```
async function mix($path) {
  $manifest = yield Amp\File\get(
    .."/public/mix-manifest.json"
  );

  $manifest = json_decode($manifest, true);

  if (isset($manifest[$path])) {
    return $manifest[$path];
```

```
  }

  throw new Exception("{$path} not found");
}
```

Previously, I had to make a generator, and then wrap it in a new `Coroutine`:

```
use Amp\Coroutine;

function mix($path) {
  $generator = () => {
    $manifest = yield Amp\File\get(
      ."/public/mix-manifest.json"
    );

    $manifest = json_decode($manifest, true);

    if (isset($manifest[$path])) {
      return $manifest[$path];
    }

    throw new Exception("{$path} not found");
  };

  return new Coroutine($generator());
}
```

I began the `createPatches` method as before, creating new `PatchModel` objects for each x and y in the grid. Then I started another loop, to call the `start` method

on each patch. I would have done these in the same step, but I wanted my `start` method to be able to inspect the surrounding patches. That meant I would have to create all of them first, before working out which patches were around each other.

I also changed `FarmModel` to accept an `onGrowth` closure. The idea was that I could call that closure if a patch grew (even during the bootstrapping phase).

Each time a patch grew, I reset the `$changes` variable. This ensured the patches would keep growing until an entire pass of the farm yielded no changes. I also invoked the `onGrowth` closure. I wanted to allow `onGrowth` to be a normal closure, or even to return a `Coroutine`. That's why I needed to make `createPatches` an `async` function.

A Note on the `onGrowth` Coroutines

Admittedly, allowing `onGrowth` coroutines complicated things a bit, but I saw it as essential for allowing other async actions when a patch grew. Perhaps later I'd want to send a socket message, and I could only do that if `yield` worked inside `onGrowth`. I could only yield `onGrowth` if `createPatches` was an `async` function. And because `createPatches` was an `async` function, I would need to yield it inside `GameSocket`.

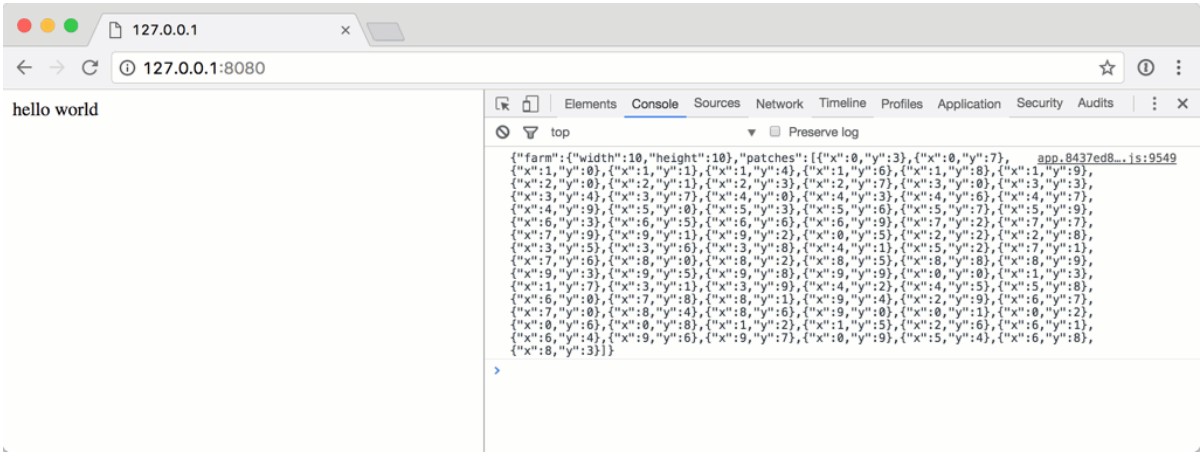
"It's easy to get turned off by all the things that need learning when making one's first async PHP application. Don't give up too soon!"

The last bit of code I needed to write to check that this was all working was in `GameSocket`. From `app/Socket/GameSocket.pre`:

```
if ($body === "new-farm") {
  $patches = [];

  $farm = new FarmModel(10, 10,
  function (PatchModel $patch) use (&$patches) {
    array_push($patches, [
      "x" => $patch->x,
      "y" => $patch->y,
    ]);
  }
);

yield $farm->createPatches();

$payload = json_encode([
  "farm" => [
    "width" => $farm->width,
    "height" => $farm->height,
  ],
  "patches" => $patches,
]);

yield $this->endpoint->send(
  $payload, $clientId
);

$this->farms[$clientId] = $farm;
}
```

This was only slightly more complex than the previous code I had. I needed to provide a third parameter to the `FarmModel` constructor, and yield `$farm->createPatches()` so that each could have a chance to randomize. After

that, I just needed to pass a snapshot of the patches to the socket payload.



"What if I start each patch as dry dirt? Then I could make some patches have weeds, and others have trees ..."

I set about customizing the patches. From `app/Model/PatchModel.pre`:

```
private $started = false;

private $wet {
  get { return $this->wet ?: false; }
};

private $type {
  get { return $this->type ?: "dirt"; }
};

public function start(int $width, int $height,
array $patches)
{
  if ($this->started) {
    return false;
```

```
    }

    if (random_int(0, 100) < 90) {
      return false;
    }

    $this->started = true;
    $this->type = "weed";

    return true;
  }
```

I changed the order of logic around a bit, exiting early if the patch had already been started. I also reduced the chance of growth. If neither of these early exits happened, the patch type would be changed to weed.

I could then use this type as part of the socket message payload. From `app/Socket/GameSocket.pre`:

```
$farm = new FarmModel(10, 10,
function (PatchModel $patch) use (&$patches) {
  array_push($patches, [
    "x" => $patch->x,
    "y" => $patch->y,
    "wet" => $patch->wet,
    "type" => $patch->type,
  ]);
}
);
```

Rendering the Farm

It was time to show the farm, using the React workflow I had setup previously. I was already getting the `width` and `height` of the farm, so I could make every block dry dirt (unless it was supposed to grow a weed). From `assets/js/app.jsx`:

```
import React from "react"

class Farm extends React.Component
{
  constructor()
  {
    super()

    this.onMessage = this.onMessage.bind(this)

    this.state = {
      "farm": {
        "width": 0,
        "height": 0,
      },
      "patches": [],
    };
  }

  componentWillMount()
  {
    this.socket = new WebSocket(
      "ws://127.0.0.1:8080/ws"
    )
```

```
    this.socket.addEventListener(
      "message", this.onMessage
    )

    // DEBUG

    this.socket.addEventListener("open", () => {
      this.socket.send("new-farm")
    })
  }

  onMessage(e)
  {
    let data = JSON.parse(e.data);

    if (data.farm) {
      this.setState({"farm": data.farm})
    }

    if (data.patches) {
      this.setState({"patches": data.patches})
    }
  }

  componentWillUnmount()
  {
    this.socket.removeEventListener(this.onMessage)
    this.socket = null
  }

  render() {
```

```
let rows = []
let farm = this.state.farm
le"open"ePatches = this.state.patches

for (let y = 0; y < farm.height; y++) {
  let patches = []

  for (let x = 0; x < farm.width; x++) {
    let className = "patch"

    statePatches.forEach((patch) => {
      if (patch.x === x && patch.y === y) {
        className += " " + patch.type

        if (patch.wet) {
          className += " " + wet
        }
      }
    })

    patches.push(
      <div className={className}
      key={x + "x" + y} />
    )
  }

  rows.push(
    <div className="row" key={y}>
    {patches}
    </div>
  )
}
```

```
    return (
      <div className="farm">{rows}</div>
    )
  }
}

export default Farm
```

I had forgotten to explain much of what the previous `Farm` component was doing. React components were a different way of thinking about how to build interfaces. They changed one's thought process from "How do I interact with the DOM when I want to change something?" to "What should the DOM look like with any given context?"

I was meant to think about the `render` method as only executing once, and that everything it produced would be dumped into the DOM. I could use methods like `componentWillMount` and `componentWillUnmount` as ways to hook into other data points (like WebSockets). And as I received updates through the WebSocket, I could update the component's state, so long as I had set the initial state in the constructor.

This resulted in an ugly, albeit functional set of divs. I set about adding some styling. From `app/Action/HomeAction.pre`:

```
namespace App\Action;

use Aerys\Request;
use Aerys\Response;
```

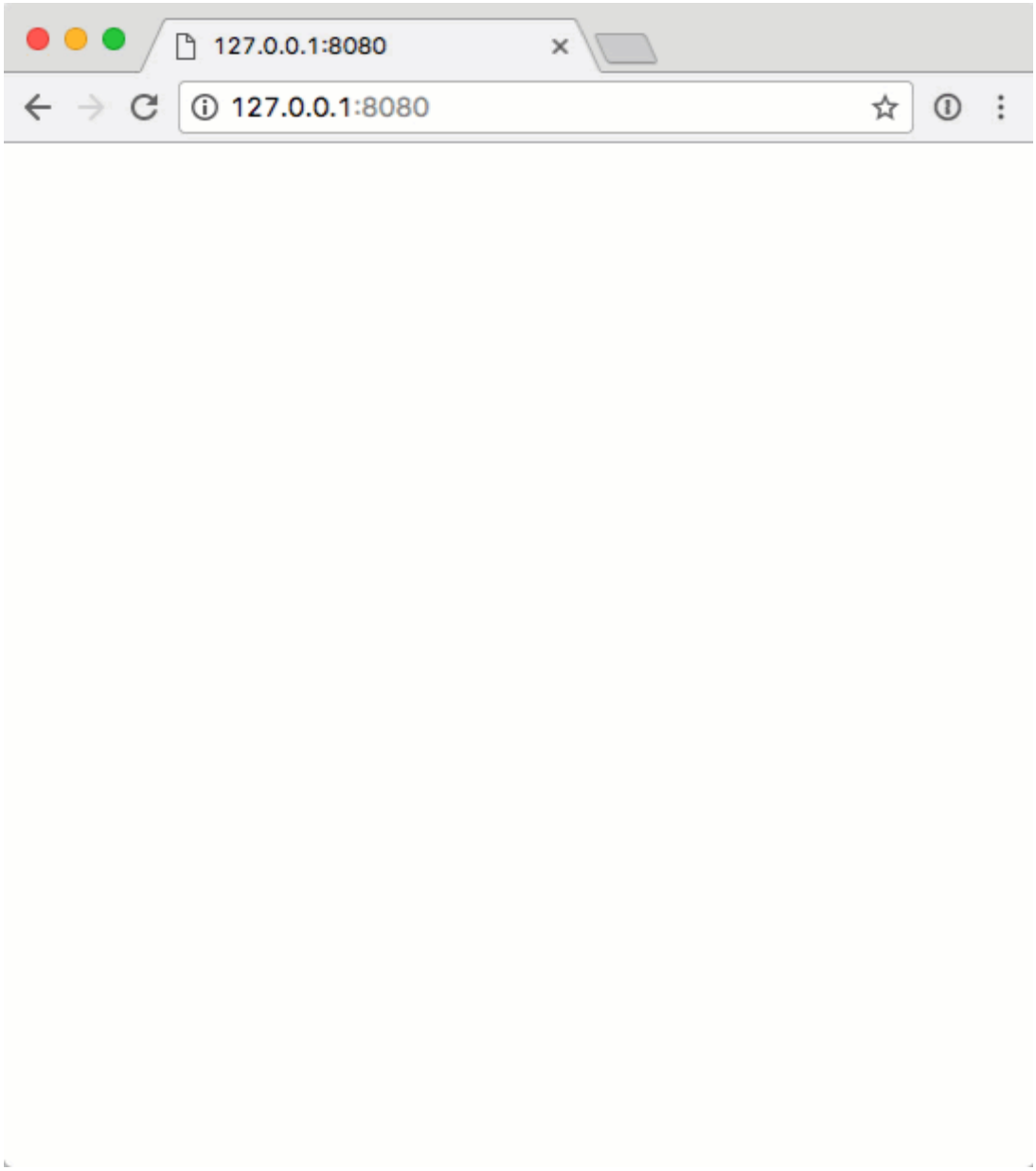
```
class HomeAction
{
  public function __invoke(Request $request,
  Response $response)
  {
    $js = yield mix("/js/app.js");
    $css = yield mix("/css/app.css");

    $response->end("
    <link rel='stylesheet' href='{$css}' />
    <div class='app'></div>
    <script src='{$js}'></script>
    ");
  }
}
```

From `assets/scss/app.scss`:

```
.row {
  width: 100%;
  height: 50px;

  .patch {
    width: 50px;
    height: 50px;
    display: inline-block;
    background-color: sandybrown;

    &.weed {
```

```
        background-color: green;
      }
    }
  }
```

The generated farms now had a bit of color to them:



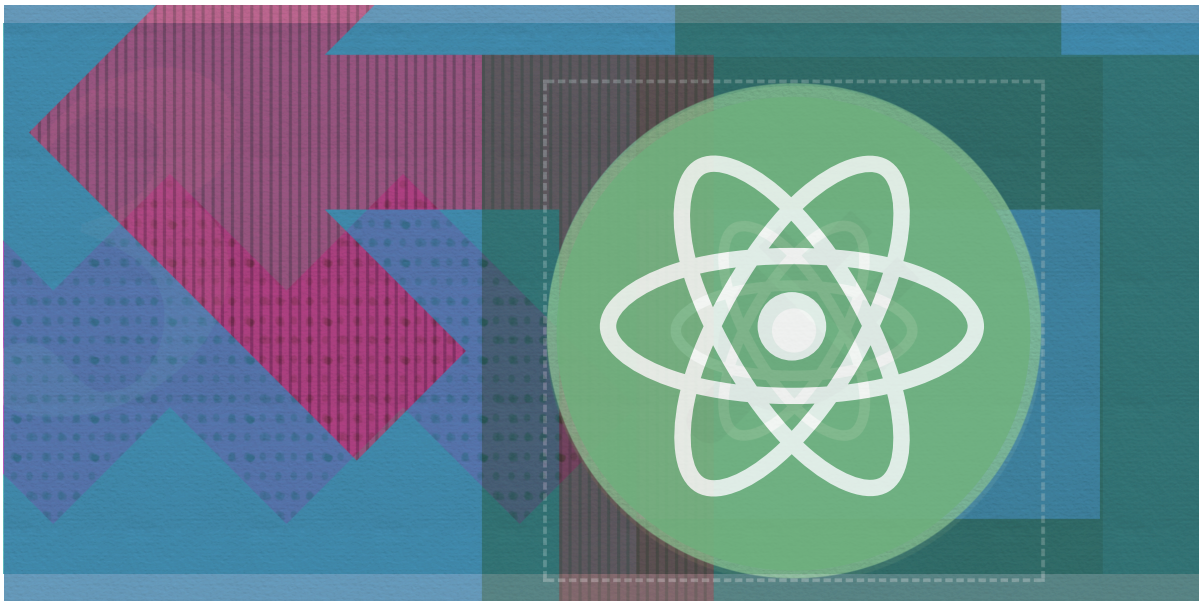
Summary

This was by no means a complete game. It lacked vital things like player input and player characters. It wasn't very multiplayer. But this session resulted in a deeper understanding of React components, WebSocket communication, and preprocessor macros.

I was looking forward to the next part, wherein I could start taking player input, and changing the farm. Perhaps I'd even start on the player login system. Maybe one day!

Tools

React: Tools & Resources



sitepoint

React: Tools & Resources

Copyright © 2017 SitePoint Pty. Ltd.

Product Manager: Simon Mackie

Cover Designer: Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

Table of Contents

Preface

This book is a collection of in-depth guides to some some of the tools and resources most used with React, such as Jest and React Router, as well as a discussion about how React works well with D3, and a look at Preact, a lightweight React alternative. These tutorials were all selected from SitePoint's React Hub.

Who Should Read This Book

This book is for front-end developers with some React experience. If you're a novice, please read *Your First Week With React* before tackling this book.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.</p>
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ↪ indicates a line break that exists for

formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-design-real-user-testing
↪/?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter

1

Getting Started with Redux

by Michael Wanyoike

A typical web application is usually composed of several UI components that share data. Often, multiple components are tasked with the responsibility of displaying different properties of the same object. This object represents state which can change at any time. Keeping state consistent among multiple components can be a nightmare, especially if there are multiple channels being used to update the same object.

Take, for example, a site with a shopping cart. At the top we have a UI component showing the number of items in the cart. We could also have another UI component that displays the total cost of items in the cart. If a user clicks the **Add to Cart** button, both of these components should update immediately with the correct figures. If the user decides to remove an item from the cart, change quantity, add a protection plan, use a coupon or change shipping location, then the relevant UI components should update to display the correct information. As you can see, a simple shopping cart can quickly become *difficult to keep in sync* as the scope of its features grows.

In this guide, I'll introduce you to a framework known as Redux, which can help you build complex projects in way that's easy to scale and maintain. To make learning easier, we'll use a simplified **shopping cart project** to learn how Redux works. You'll need to be at least familiar with the React library, as you'll later need to integrate it with Redux.

Prerequisites

Before we get started, make sure you're familiar with the following topics:

- Functional JavaScript
- Object-oriented JavaScript
- ES6 JavaScript Syntax

Also, ensure you have the following setup on your machine:

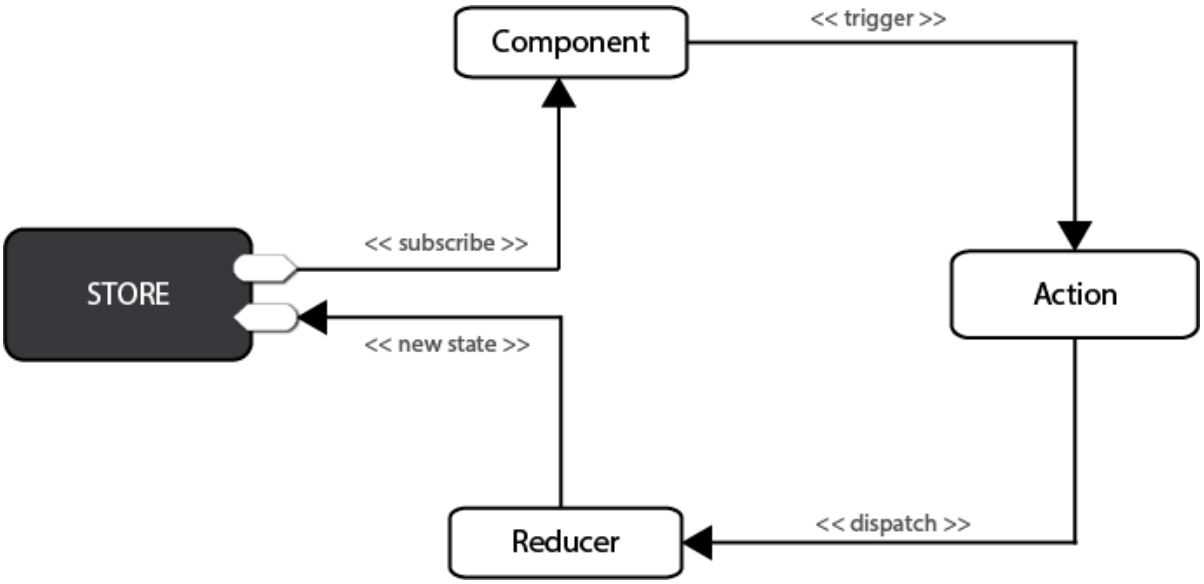
- a NodeJS environment

- a Yarn setup (recommended)

You can access the entire code used in this tutorial on GitHub.

What is Redux?

Redux is a popular JavaScript framework that provides a predictable state container for applications. Redux is based on a simplified version of Flux, a framework developed by Facebook. Unlike standard MVC frameworks, where data can flow between UI components and storage in both directions, Redux strictly allows data to flow in one direction only. See the below illustration:



1-1. Redux Flow Chart

In Redux, all data – i.e. **state** – is held in a container known as the **store**. There can only be one of these within an application. The store is essentially a state tree where states for all objects are kept. Any UI component can access the state of a particular object directly from the store. To change a state from a local or remote component, an **action** needs to be dispatched. **Dispatch** in this context means sending actionable information to the store. When a store receives an `action`, it delegates it to the relevant **reducer**. A `reducer` is simply a pure function that looks at the previous state, performs an action and returns a new state. To see all

this in action, we need to start coding.

Understand Immutability First

Before we start, I need you to first understand what **immutability** means in JavaScript. According to the Oxford English Dictionary, immutability means being **unchangeable**. In programming, we write code that changes the values of variables all the time. This is referred to as **mutability**. The way we do this can often cause unexpected bugs in our projects. If your code only deals with primitive data types (numbers, strings, booleans), then you don't need to worry. However, if you're working with Arrays and Objects, performing **mutable** operations on them can create unexpected bugs. To demonstrate this, open your terminal and launch the Node interactive shell:

```
node
```

Next, let's create an array, then later assign it to another variable:

```
> let a = [1,2,3]
> let b = a
> b.push(9)
> console.log(b)
[ 1, 2, 3, 9 ] // b output
> console.log(a)
[ 1, 2, 3, 9 ] // a output
```

As you can see, updating `array b` caused `array a` to change as well. This happens because Objects and Arrays are known **referential data types** – meaning that such data types don't actually hold values themselves, but are pointers to a memory location where the values are stored. By assigning `a` to `b`,

we merely created a second pointer that references the same location. To fix this, we need to copy the referenced values to a new location. In JavaScript, there are three different ways of achieving this:

1. using immutable data structures created by Immutable.js
2. using JavaScript libraries such as Underscore and Lodash to execute immutable operations
3. using native ES6 functions to execute immutable operations.

For this article, we'll use the ES6 way, since it's already available in the NodeJS environment. Inside your `NodeJS` terminal, execute the following:

```
> a = [1,2,3] // reset a
[ 1, 2, 3 ]
> b = Object.assign([],a) // copy array a to b
[ 1, 2, 3 ]
> b.push(8)
> console.log(b)
[ 1, 2, 3, 8 ] // b output
> console.log(a)
[ 1, 2, 3 ] // a output
```

In the above code example, array `b` can now be modified without affecting array `a`. We've used Object.assign() to create a new copy of values that variable `b` will now point to. We can also use the `rest operator(...)` to perform an immutable operation like this:

```
> a = [1,2,3]
[ 1, 2, 3 ]
> b = [...a, 4, 5, 6]
[ 1, 2, 3, 4, 5, 6 ]
> a
[ 1, 2, 3 ]
```

The rest operator works with object literals too! I won't go deep into this subject, but here are some additional ES6 functions that we'll use to perform immutable operations:

- spread syntax – useful in append operations
- map function – useful in an update operation
- filter function – useful in a delete operation

In case the documentation I've linked isn't useful, don't worry, as you'll see how they're used in practice. Let's start coding!

Setting up Redux

The fastest way to set up a Redux development environment is to use the `create-react-app` tool. Before we begin, make sure you've installed and updated `nodejs`, `npm` and `yarn`. Let's set up a Redux project by generating a `redux-shopping-cart` project and installing the Redux package:

```
create-react-app redux-shopping-cart

cd redux-shopping-cart
yarn add redux # or npm install redux
```

Delete all files inside the `src` folder except `index.js`. Open the file and clear out all existing code. Type the following:

```
import { createStore } from "redux";

const reducer = function(state, action) {
  return state;
}

const store = createStore(reducer);
```

Let me explain what the above piece of code does:

- 1st statement. We import a `createStore()` function from the Redux package.
- 2nd statement. We create an empty function known as a **reducer**. The first argument, `state`, is current data held in the store. The second argument, `action`, is a container for:
  - type – a simple string constant e.g. `ADD`, `UPDATE`, `DELETE` etc.
  - payload – data for updating state
- 3rd statement. We create a Redux store, which can only be constructed using a reducer as a parameter. The data kept in the Redux store can be accessed directly, but can only be updated via the supplied reducer.

You may have noticed I mentioned current data as if it already exists. Currently, our `state` is undefined or null. To remedy this, just assign a default value to state like this to make it an empty array:

```
const reducer = function(state=[], action) {
  return state;
```

```
}
```

Now, let's get practical. The reducer we created is generic. Its name doesn't describe what it's for. Then there's the issue of how we work with multiple reducers. The answer is to use a `combineReducers` function that's supplied by the Redux package. Update your code as follows:

```
// src/index.js
…
import { combineReducers } from 'redux';

const productsReducer = function(state=[], action) {
  return state;
}

const cartReducer = function(state=[], action) {
  return state;
}

const allReducers = {
  products: productsReducer,
  shoppingCart: cartReducer
}

const rootReducer = combineReducers(allReducers);

let store = createStore(rootReducer);
```

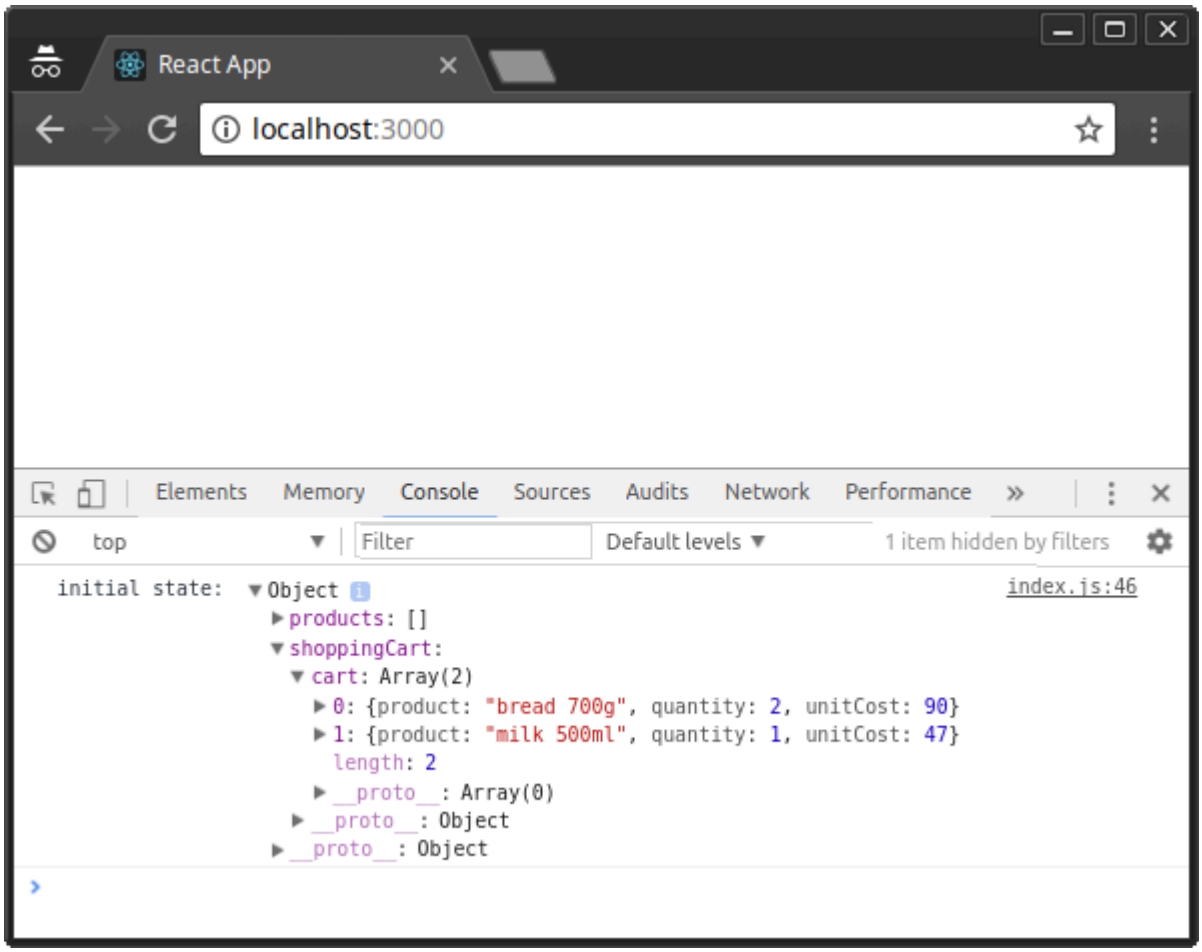
In the code above, we've renamed the generic reducer to `cartReducer`. There's

also a new empty reducer named `productsReducer` that I've created just to show you how to combine multiple reducers within a single store using the `combineReducers` function.

Next, we'll look at how we can define some test data for our reducers. Update the code as follows:

```
// src/index.js
…
const initialState = {
  cart: [
    {
      product: 'bread 700g',
      quantity: 2,
      unitCost: 90
    },
    {
      product: 'milk 500ml',
      quantity: 1,
      unitCost: 47
    }
  ]
}

const cartReducer = function(state=initialState, action) {
  return state;
}
…
let store = createStore(rootReducer);

console.log("initial state: ", store.getState());
```

Just to confirm that the store has some initial data, we use `store.getState()` to print out the current state in the console. You can run the dev server by executing `npm start` or `yarn start` in the console. Then press `Ctrl+Shift+I` to open the inspector tab in Chrome in order to view the console tab.



1-2. Redux Initial State

Currently, our `cartReducer` does nothing, yet it's supposed to manage the state of our shopping cart items within the Redux store. We need to define actions for adding, updating and deleting shopping cart items. Let's start by defining logic for a `ADD_TO_CART` action:

```
// src/index.js
…
const ADD_TO_CART = 'ADD_TO_CART';

const cartReducer = function(state=initialState, action) {
  switch (action.type) {
    case ADD_TO_CART: {
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
    }

    default:
      return state;
  }
}
…
```

Take your time to analyze and understand the code. A reducer is expected to handle different action types, hence the need for a `SWITCH` statement. When an action of type `ADD_TO_CART` is dispatched anywhere in the application, the code defined here will handle it. As you can see, we're using the information provided in `action.payload` to combine to an existing state in order to create a new state.

Next, we'll define an `action`, which is needed as a parameter for `store.dispatch()`. Actions are simply JavaScript objects that must have `type` and an optional payload. Let's go ahead and define one right after the `cartReducer` function:

```
…
function addToCart(product, quantity, unitCost) {
  return {
    type: ADD_TO_CART,
    payload: { product, quantity, unitCost }
  }
}
…
```

Here, we've defined a function that returns a plain JavaScript object. Nothing fancy. Before we dispatch, let's add some code that will allow us to listen to store event changes. Place this code right after the `console.log()` statement:

```
…
let unsubscribe = store.subscribe(() =>
  console.log(store.getState())
);

unsubscribe();
```

Next, let's add several items to the cart by dispatching actions to the store. Place this code before `unsubscribe()`:

```
…
store.dispatch(addToCart('Coffee 500gm', 1, 250));
store.dispatch(addToCart('Flour 1kg', 2, 110));
store.dispatch(addToCart('Juice 2L', 1, 250));
```

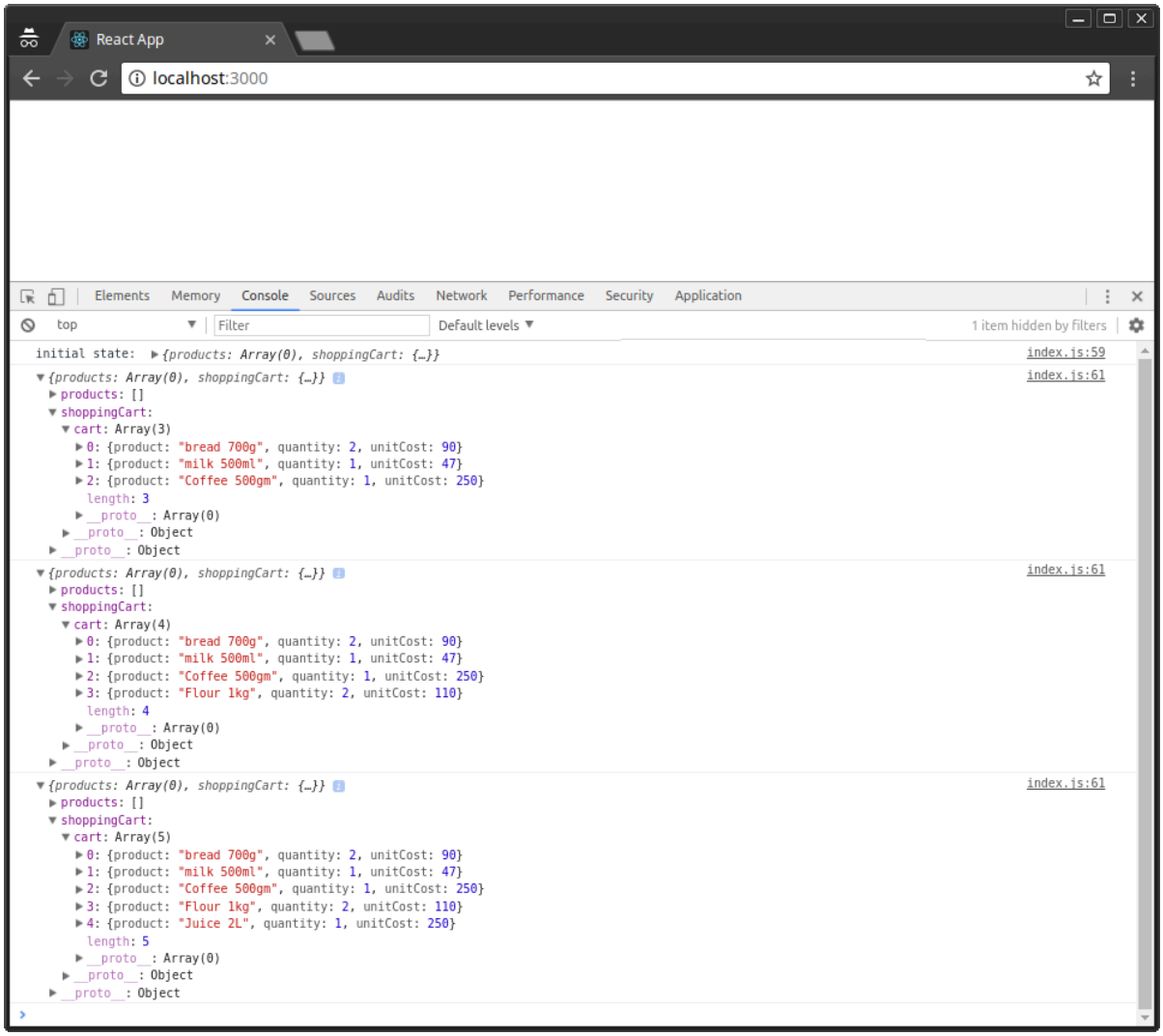
For clarification purposes, I'll illustrate below how the entire code should look after making all the above changes:

```
// src/index.js

import { createStore } from "redux";
import { combineReducers } from 'redux';

const productsReducer = function(state=[], action) {
  return state;
}

const initialState = {
  cart: [
    {
      product: 'bread 700g',
      quantity: 2,
      unitCost: 90
    },
    {
      product: 'milk 500ml',
      quantity: 1,
      unitCost: 47
    }
  ]
}

const ADD_TO_CART = 'ADD_TO_CART';

const cartReducer = function(state=initialState, action) {
  switch (action.type) {
    case ADD_TO_CART: {
```

```
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
    }

    default:
      return state;
  }
}

function addToCart(product, quantity, unitCost) {
  return {
    type: ADD_TO_CART,
    payload: {
      product,
      quantity,
      unitCost
    }
  }
}

const allReducers = {
  products: productsReducer,
  shoppingCart: cartReducer
}

const rootReducer = combineReducers(allReducers);

let store = createStore(rootReducer);

console.log("initial state: ", store.getState());
```

```
let unsubscribe = store.subscribe(() =>
  console.log(store.getState())
);

store.dispatch(addToCart('Coffee 500gm', 1, 250));
store.dispatch(addToCart('Flour 1kg', 2, 110));
store.dispatch(addToCart('Juice 2L', 1, 250));

unsubscribe();
```

After you've saved your code, Chrome should automatically refresh. Check the console tab to confirm that the new items have been added:



1-3. Redux Actions Dispatched

Organizing Redux Code

The `index.js` file has quickly grown large. This is not how Redux code is written. I've only done this to show you how simple Redux is. Let's look at how a Redux project should be organized. First, create the following folders and files within the `src` folder, as illustrated below:

```
src/
├── actions
│   └── cart-actions.js
├── index.js
├── reducers
│   ├── cart-reducer.js
│   ├── index.js
│   └── products-reducer.js
└── store.js
```

Next, let's start moving code from `index.js` to the relevant files:

```
// src/actions/cart-actions.js

export const ADD_TO_CART = 'ADD_TO_CART';

export function addToCart(product, quantity, unitCost) {
  return {
    type: ADD_TO_CART,
    payload: { product, quantity, unitCost }
  }
}
```

```
// src/reducers/products-reducer.js

export default function(state=[], action) {
  return state;
}
```

```
// src/reducers/cart-reducer.js

import  { ADD_TO_CART }  from '../actions/cart-actions';

const initialState = {
  cart: [
    {
      product: 'bread 700g',
      quantity: 2,
      unitCost: 90
    },
    {
      product: 'milk 500ml',
      quantity: 1,
      unitCost: 47
    }
  ]
}

export default function(state=initialState, action) {
  switch (action.type) {
    case ADD_TO_CART: {
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
    }

    default:
      return state;
  }
}
```

```
// src/reducers/index.js

import { combineReducers } from 'redux';
import productsReducer from './products-reducer';
import cartReducer from './cart-reducer';

const allReducers = {
  products: productsReducer,
  shoppingCart: cartReducer
}

const rootReducer = combineReducers(allReducers);

export default rootReducer;
```

```
// src/store.js

import { createStore } from "redux";
import rootReducer from './reducers';

let store = createStore(rootReducer);

export default store;
```

```
// src/index.js

import store from './store.js';
import { addToCart }  from './actions/cart-actions';
```

```
console.log("initial state: ", store.getState());

let unsubscribe = store.subscribe(() =>
  console.log(store.getState())
);

store.dispatch(addToCart('Coffee 500gm', 1, 250));
store.dispatch(addToCart('Flour 1kg', 2, 110));
store.dispatch(addToCart('Juice 2L', 1, 250));

unsubscribe();
```

After you've finished updating the code, the application should run as before now that it's better organized. Let's now look at how we can update and delete items from the shopping cart. Open `cart-reducer.js` and update the code as follows:

```
// src/reducers/cart-actions.js
…
export const UPDATE_CART = 'UPDATE_CART';
export const DELETE_FROM_CART = 'DELETE_FROM_CART';
…
export function updateCart(product, quantity, unitCost) {
  return {
    type: UPDATE_CART,
    payload: {
      product,
      quantity,
      unitCost
```

```
    }
  }
}

export function deleteFromCart(product) {
  return {
    type: DELETE_FROM_CART,
    payload: {
      product
    }
  }
}
```

Next, update `cart-reducer.js` as follows:

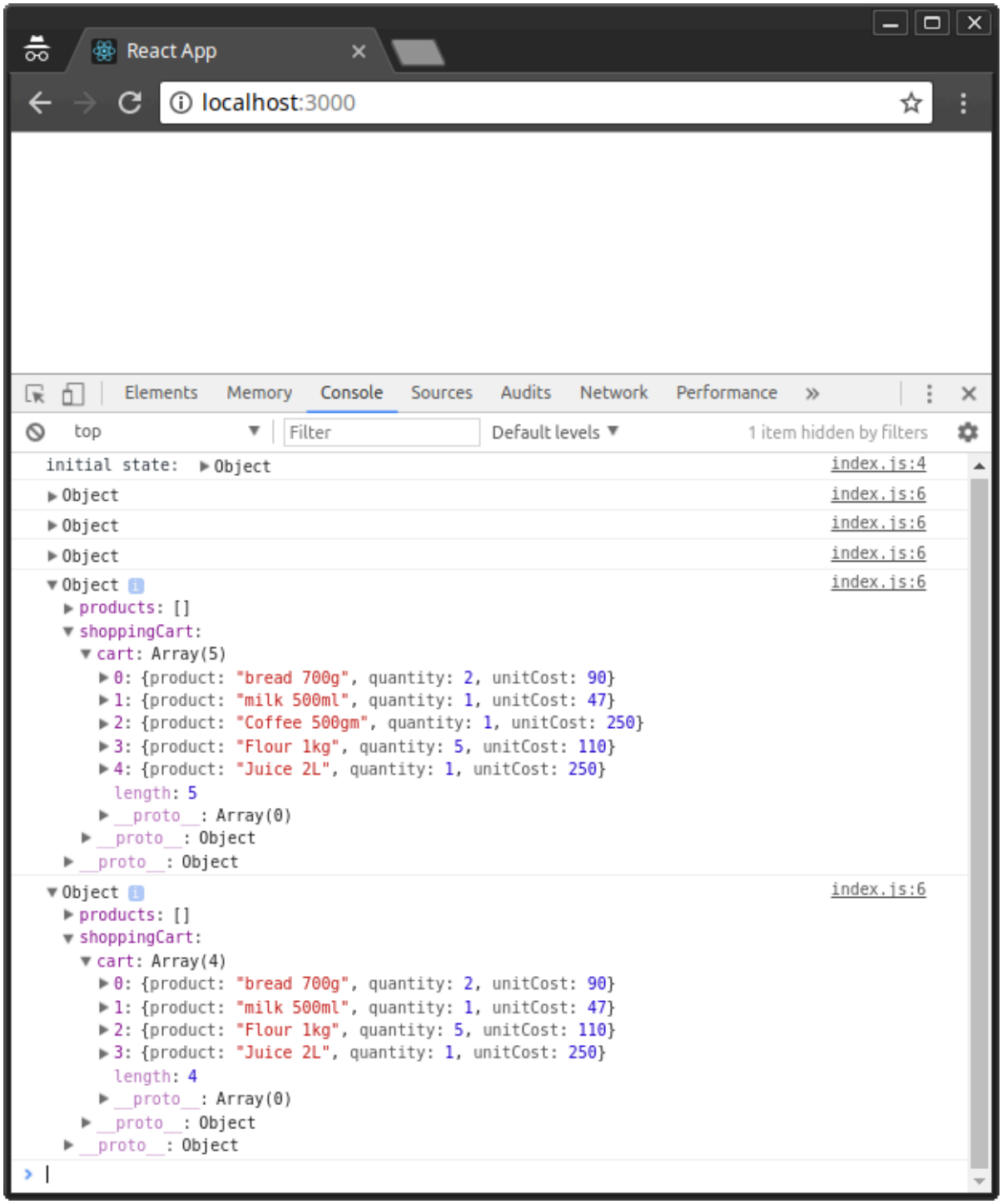
```
// src/reducers/cart-reducer.js
…
export default function(state=initialState, action) {
  switch (action.type) {
    case ADD_TO_CART: {
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
    }

    case UPDATE_CART: {
      return {
        ...state,
        cart: state.cart.map(item => item.product === action.payload.product ?
```

```
        ↪action.payload : item)
      }
    }

    case DELETE_FROM_CART: {
      return {
        ...state,
        cart: state.cart.filter(item => item.product !== action.payload.product)
      }
    }

    default:
      return state;
  }
}
```

Finally, let's dispatch the `UPDATE_CART` and `DELETE_FROM_CART` actions in `index.js`:

```
// src/index.js
…
// Update Cart
store.dispatch(updateCart('Flour 1kg', 5, 110));

'Flour 1kg'// Delete from Cart
store.dispatch(deleteFromCart('Coffee 500gm'));
…
```

Your browser should automatically refresh once you've saved all the changes. Check the console tab to confirm the results:



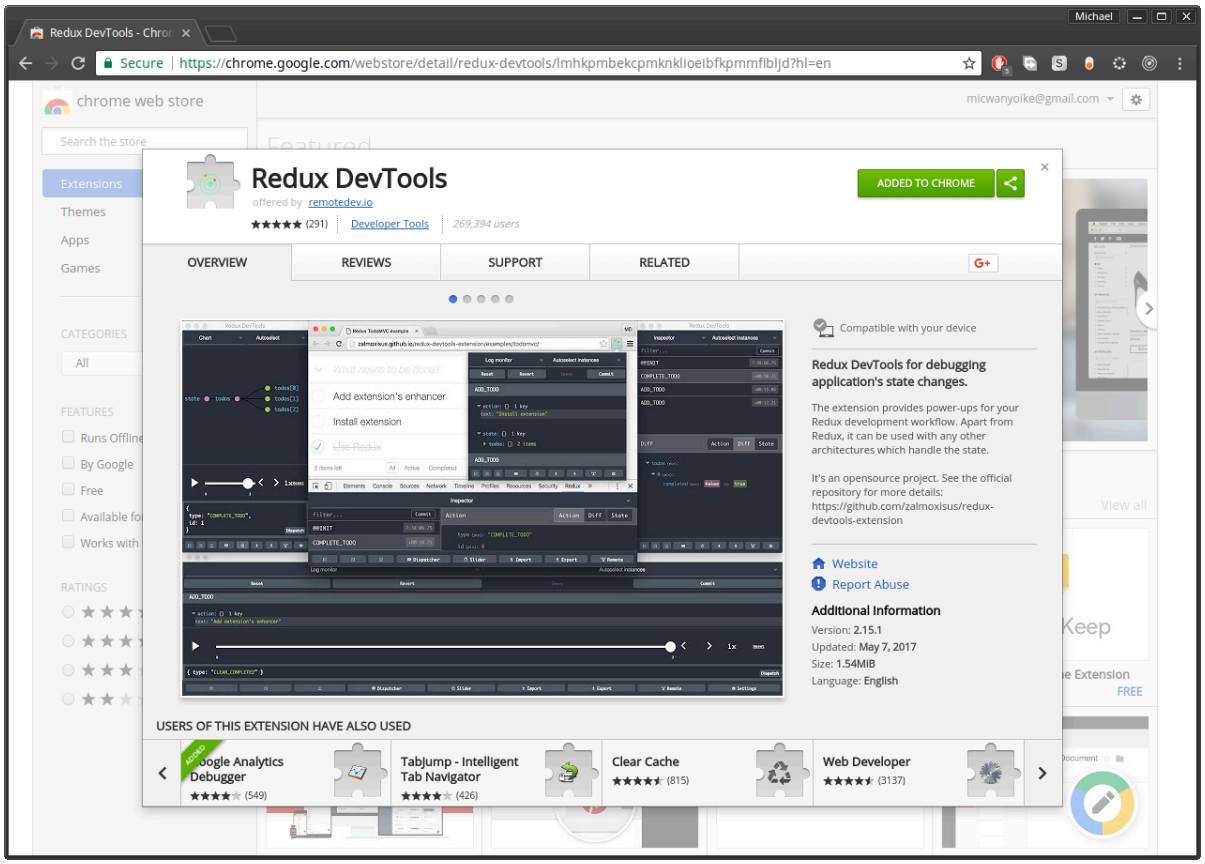
1-4. Redux Update and Delete Actions

As confirmed, the quantity for 1kg of flour is updated from 2 to 5, while the the 500gm of coffee gets deleted from cart.

Debugging with Redux tools

Now, if we've made a mistake in our code, how do we debug a Redux project?

Redux comes with a lot of third-party debugging tools we can use to analyze code behavior and fix bugs. Probably the most popular one is the **time-travelling tool**, otherwise known as redux-devtools-extension. Setting it up is a 3-step process. First, go to your Chrome browser and install the Redux Devtools extension.



1-5. Redux DevTools Chrome Extensions

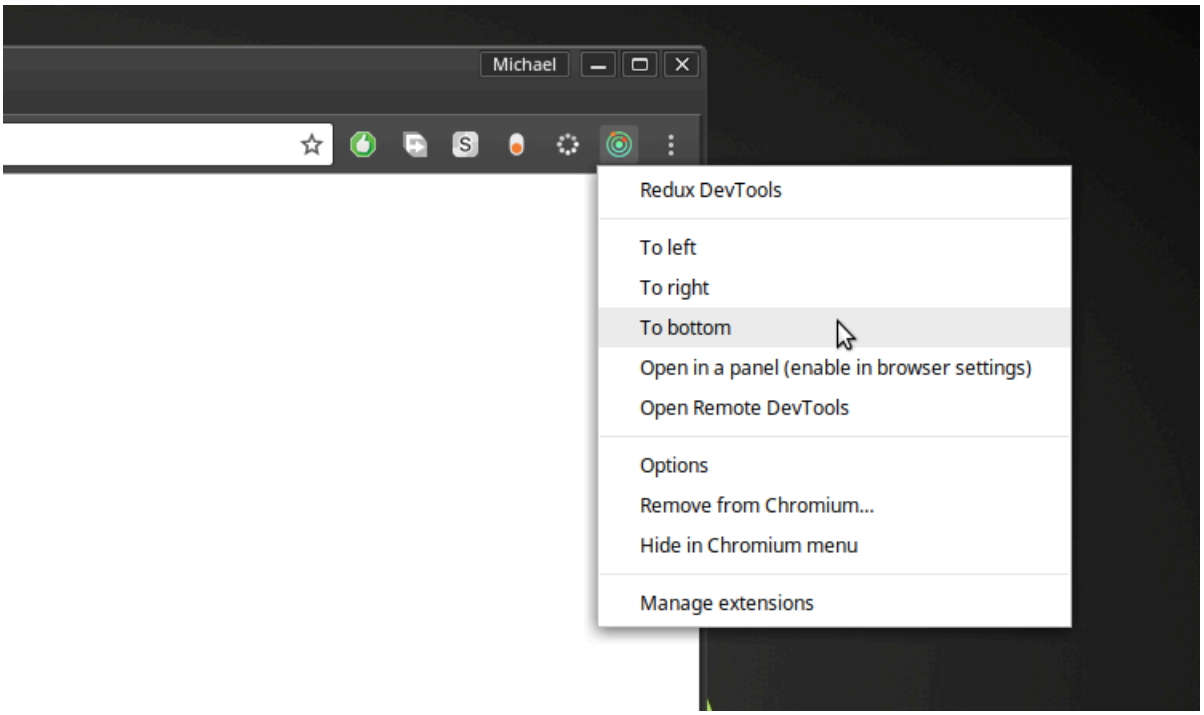
Next, go to your terminal where your Redux application is running and press `Ctrl+C` to stop the development server. Next, use npm or yarn to install the redux-devtools-extension package. Personally, I prefer Yarn, since there's a `yarn.lock` file that I'd like to keep updated.

```
yarn add redux-devtools-extension
```

Once installation is complete, you can start the development server as we implement the final step of implementing the tool. Open `store.js` and replace the existing code as follows:

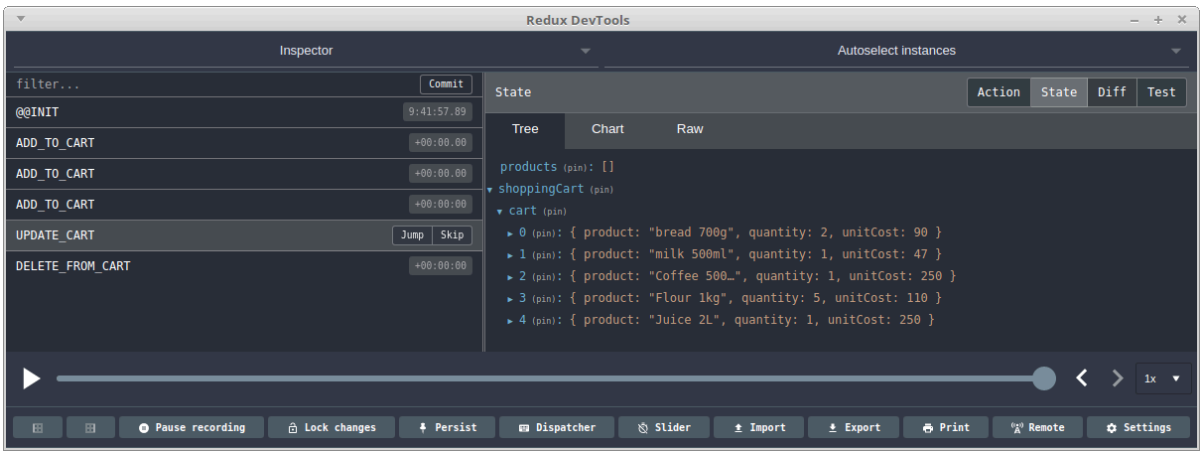
```
// src/store.js
import { createStore } from "redux";
import { composeWithDevTools } from 'redux-devtools-extension';
import rootReducer from './reducers';

const store = createStore(rootReducer, composeWithDevTools());

export default store;
```

Feel free to update `src/index.js` and remove all code related with logging to the console and subscribing to the store. This is no longer needed. Now, go back to Chrome and open the Redux DevTools panel by right-clicking the tool's icon:



1-6. Redux DevTools Menu

In my case, I've selected to **To Bottom** option. Feel free to try out other options.



1-7. Redux DevTools Panel

As you can see, the Redux Devtool is quite amazing. You can toggle between action, state and diff methods. Select actions on the left panel and observe how the state tree changes. You can also use the slider to play back the sequence of actions. You can even dispatch directly from the tool! Do check out the

documentation to learn more on how you can further customize the tool to your needs.

Integration with React

At the beginning of this tutorial, I mentioned Redux really pairs well with React. Well, you only need a few steps to setup the integration. Firstly, stop the development server, as we'll need to install the react-redux package, the official Redux bindings for React:

```
yarn add react-redux
```

Next, update `index.js` to include some React code. We'll also use the `Provider` class to wrap the React application within the Redux container:

```
// src/index.js
…
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';

const App = <h1>Redux Shopping Cart</h1>;

ReactDOM.render(
  <Provider store={store}>
    { App }
  </Provider> ,
  document.getElementById('root')
);
…
```

```
'root'
```

Just like that, we've completed the first part of the integration. You can now start the server to see the result. The second part involves linking React's components with the Redux store and actions using a couple of functions provided by the `react-redux` package that we just installed. In addition, you'll need to set up an API using Express or a framework like Feathers. The API will provide our application with access to a database service.

In Redux, we'll also need to install further packages such as `axios` to perform API requests via Redux actions. Our React components state will then be handled by Redux, making sure that all components are in sync with the database API. To learn more on how to accomplish all this, do take a look at my other tutorial, "Build a CRUD App Using React, Redux and FeathersJS".

Summary

I hope this guide has given you a useful introduction to Redux. There's still quite a bit more for you to learn, though. For example, you need to learn how to deal with async actions, authentication, logging, handling forms and so on. Now that you know what Redux is all about, you'll find it easier to try out other similar frameworks, such as Flux, Alt.js or Mobx. If you feel Redux is right for you, I highly recommend the following tutorials that will help you gain even more experience in Redux:

- Redux State Management in Vanilla JavaScript
- Redux Logging in Production with LogRocket
- Build a CRUD App Using React, Redux and FeathersJS
- Dealing with Asynchronous APIs in Server-rendered React

Chapter

2

React Router v4: The Complete Guide

by Manjunath M

React Router is the de facto standard routing library for React. When you need to navigate through a React application with multiple views, you'll need a router to manage the URLs. React Router takes care of that, keeping your application UI and the URL in sync.

This tutorial introduces you to React Router v4 and a whole lot of things you can do with it.

Introduction

React is a popular library for creating single-page applications (SPAs) that are rendered on the client side. An SPA might have multiple **views** (aka **pages**), and unlike the conventional multi-page apps, navigating through these views shouldn't result in the entire page being reloaded. Instead, we want the views to be rendered inline within the current page. The end user, who's accustomed to multi-page apps, expects the following features to be present in an SPA:

- Each view in an application should have a URL that uniquely specifies that view. This is so that the user can bookmark the URL for reference at a later time – e.g. `www.example.com/products`.
- The browser's back and forward button should work as expected.
- The dynamically generated nested views should preferably have a URL of their own too – e.g. `example.com/products/shoes/101`, where 101 is the product id.

Routing is the process of keeping the browser URL in sync with what's being rendered on the page. React Router lets you handle routing **declaratively**. The declarative routing approach allows you to control the data flow in your application, by saying "the route should look like this":

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

You can place your `<Route>` component anywhere that you want your route to be rendered. Since `<Route>`, `<Link>` and all the other React Router API that we'll be dealing with are just components, you can easily get used to routing in React.

React Router is a Third-party Library

A note before getting started. There's a common misconception that React Router is an official routing solution developed by Facebook. In reality, it's a third-party library that's widely popular for its design and simplicity. If your requirements are limited to routers for navigation, you could implement a custom router from scratch without much hassle. However, understanding the basics of React Router will give you better insights into how a router should work.

Overview

This tutorial is divided into different sections. First, we'll be setting up React and React Router using npm. Then we'll jump right into React Router basics. You'll find different code demonstrations of React Router in action. The examples covered in this tutorial include:

1. basic navigational routing
2. nested routing
3. nested routing with path parameters
4. protected routing

All the concepts connected with building these routes will be discussed along the way. The entire code for the project is available on this GitHub repo. Once you're inside a particular demo directory, run `npm install` to install the dependencies. To serve the application on a development server, run `npm start` and head over to `http://localhost:3000/` to see the demo in action.

Let's get started!

Setting up React Router

I assume you already have a development environment up and running. If not, head over to "Getting Started with React and JSX". Alternatively, you can use Create React App to generate the files required for creating a basic React project. This is the default directory structure generated by Create React App:

```
react-routing-demo-v4
    ├── .gitignore
    ├── package.json
    ├── public
    │   ├── favicon.ico
    │   ├── index.html
    │   └── manifest.json
    ├── README.md
    ├── src
    │   ├── App.css
    │   ├── App.js
    │   ├── App.test.js
    │   ├── index.css
    │   ├── index.js
    │   ├── logo.svg
    │   └── registerServiceWorker.js
    └── yarn.lock
```

The React Router library comprises three packages: `react-router`, `react-router-dom`, and `react-router-native`. `react-router` is the core package for the router, whereas the other two are environment specific. You should use `react-router-dom` if you're building a website, and

`react-router-native` if you're on a mobile app development environment using React Native.

Use npm to install `react-router-dom`:

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

React Router Basics

Here's an example of how our routes will look:

```
<Router>
  <Route exact path="/" component={Home}/>
  <Route path="/category" component={Category}/>
  <Route path="/login" component={Login}/>
  <Route path="/products" component={Products}/>
</Router>
```

Router

You need a router component and several route components to set up a basic route as exemplified above. Since we're building a browser-based application, we can use two types of routers from the React Router API:

1. `<BrowserRouter>`
2. `<HashRouter>`

The primary difference between them is evident in the URLs that they create:

```
// <BrowserRouter>
http://example.com/about

// <HashRouter>
http://example.com/#/about
```

The `<BrowserRouter>` is more popular amongst the two because it uses the HTML5 History API to keep track of your router history. The `<HashRouter>`, on the other hand, uses the hash portion of the URL (`window.location.hash`) to remember things. If you intend to support legacy browsers, you should stick with `<HashRouter>`.

Wrap the `<BrowserRouter>` component around the App component.

index.js

```
/* Import statements */
import React from 'react';
import ReactDOM from 'react-dom';

/* App is the entry point to the React code.*/
import App from './App';

'./App'/* import BrowserRouter from 'react-router-dom' */
import { BrowserRouter } from 'react-router-dom';

ReactDOM.render(
    <BrowserRouter>
        <App />
```

```
    </BrowserRouter>document.getElementById('root'));
'root'
```

A Router Component Can Only Have a Single Child Element

A router component can only have a single child element. The child element can be an HTML element – such as a div – or a React component.

For the React Router to work, you need to import the relevant API from the `react-router-dom` library. Here I've imported the `BrowserRouter` into `index.js`. I've also imported the `App` component from `App.js`. `App.js`, as you might have guessed, is the entry point to React components.

The above code creates an instance of history for our entire App component. Let me formally introduce you to history.

history

> `history` is a JavaScript library that lets you easily manage session history anywhere JavaScript runs. history provides a minimal API that lets you manage the history stack, navigate, confirm navigation, and persist state between sessions. – React Training docs

Each router component creates a history object that keeps track of the current location (`history.location`) and also the previous locations in a stack. When the current location changes, the view is re-rendered and you get a sense of navigation. How does the current location change? The history object has methods such as `history.push()` and `history.replace()` to take care of that. `history.push()` is invoked when you click on a `<Link>` component, and

`history.replace()` is called when you use `<Redirect>`. Other methods – such as `history.goBack()` and `history.goForward()` – are used to navigate through the history stack by going back or forward a page.

Moving on, we have Links and Routes.

Links and Routes

The `<Route>` component is the most important component in React router. It renders some UI if the current location matches the route's path. Ideally, a `<Route>` component should have a prop named `path`, and if the pathname is matched with the current location, it gets rendered.

The `<Link>` component, on the other hand, is used to navigate between pages. It's comparable to the HTML anchor element. However, using anchor links would result in a browser refresh, which we don't want. So instead, we can use `<Link>` to navigate to a particular URL and have the view re-rendered without a browser refresh.

We've covered everything you need to know to create a basic router. Let's build one.

Demo 1: Basic Routing

src/App.js

```
/* Import statements */
import React, { Component } from 'react';
import { Link, Route, Switch } from 'react-router-dom';

/* Home component */
```

```
const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
)

/* Category component */
const Category = () => (
  <div>
    <h2>Category</h2>
  </div>
)

/* Products component */
const Products = () => (
  <div>
    <h2>Products</h2>
  </div>
)

/* App component */
class App extends React.Component {
  render() {
    return (
      <div>
        <nav className="navbar navbar-light">
          <ul className="nav navbar-nav">

            /* Link components are used for linking to other views */
            <li><Link to="/">Homes</Link></li>
            <li><Link to="/category">Category</Link></li>
            <li><Link to="/products">Products</Link></li>
```

```
          </ul>
        </nav>

        /* Route components are rendered if the path prop matches the current
         URL */
        <Route path="/" component={Home}/>
        <Route path="/category" component={Category}/>
        <Route path="/products" component={Products}/>

      </div>
    )
  }
}
```

We've declared the components for Home, Category and Products inside `App.js`. Although this is okay for now, when the component starts to grow bigger, it's better to have a separate file for each component. As a rule of thumb, I usually create a new file for a component if it occupies more than 10 lines of code. Starting from the second demo, I'll be creating a separate file for components that have grown too big to fit inside the `App.js` file.

Inside the App component, we've written the logic for routing. The `<Route>`'s path is matched with the current location and a component gets rendered. The component that should be rendered is passed in as a second prop.

Here `/` matches both `/` and `/category`. Therefore, both the routes are matched and rendered. How do we avoid that? You should pass the `exact= {true}` props to the router with `path='/'`:

```
<Route exact={true} path="/" component={Home}/>
```

If you want a route to be rendered only if the paths are exactly the same, you should use the exact props.

Nested Routing

To create nested routes, we need to have a better understanding of how `<Route>` works. Let's do that.

`<Route>` has three props that you can you use to define what gets rendered:

- **component**. We've already seen this in action. When the URL is matched, the router creates a React element from the given component using `React.createElement`.
- **render**. This is handy for inline rendering. The render prop expects a function that returns an element when the location matches the route's path.
- **children**. The children prop is similar to render in that it expects a function that returns a React element. However, children gets rendered regardless of whether the path is matched with the location or not.

Path and match

The *path* is used to identify the portion of the URL that the router should match. It uses the Path-to-RegExp library to turn a path string into a regular expression. It will then be matched against the current location.

If the router's path and the location are successfully matched, an object is created and we call it the **match** object. The match object carries more information about the URL and the path. This information is accessible through its properties, listed below:

- `match.url`. A string that returns the matched portion of the URL. This is particularly useful for building nested `<Link>`s
- `match.path`. A string that returns the route's path string – that is, `<Route path="">`. We'll be using this to build nested `<Route>`s.
- `match.isExact`. A boolean that returns true if the match was exact (without any trailing characters).
- `match.params`. An object containing key/value pairs from the URL parsed by the Path-to-RegExp package.

Now that we know all about `<Route>`s, let's build a router with nested routes.

Switch Component

Before we head for the demo code, I want to introduce you to the `<Switch>` component. When multiple `<Route>`s are used together, all the routes that match are rendered inclusively. Consider this code from demo 1. I've added a new route to demonstrate why `<Switch>` is useful.

```
<Route exact path="/" component={Home}/>
<Route path="/products" component={Products}/>
<Route path="/category" component={Category}/>
<Route path="/:id" render = {()=> (<p> I want this text to show up for all routes
    ↪ other than '/', '/products' and '/category' '/'</p>)}/>
```

If the URL is `/products`, all the routes that match the location `/products` are rendered. So, the `<Route>` with path `:id` gets rendered along with the `Products` component. This is by design. However, if this is not the behavior you're expecting, you should add the `<Switch>` component to your routes. With `<Switch>`, only the first child `<Route>` that matches the location gets rendered.

Demo 2: nested routing

Earlier on, we created routes for `/`, `/category` and `/products`. What if we wanted a URL of the form `/category/shoes`?

src/App.js

```
import React, { Component } from 'react';
import { Link, Route, Switch } from 'react-router-dom';
import Category from './Category';

class App extends Component {
  render() {

    return (
      <div>
        <nav className="navbar navbar-light">
          <ul className="nav navbar-nav">
            <li><Link to="/">Homes</Link></li>
            <li><Link to="/category">Category</Link></li>
            <li><Link to="/products">Products</Link></li>
          </ul>
        </nav>

    <Switch>
      <Route exact path="/" component={Home}/>
      <Route path="/category" component={Category}/>
       <Route path="/products" component={Products}/>
    </Switch>

    </div>
    );
```

```
  }
}
export default App;

/* Code for Home and Products component omitted for brevity */
```

Unlike the earlier version of React Router, in version 4, the nested `<Route>`s should preferably go inside the parent component. That is, the Category component is the parent here, and we'll be declaring the routes for `category/:name` inside the parent component.

src/Category.jsx

```
import React from 'react';
import { Link, Route } from 'react-router-dom';

const Category = ({ match }) => {
return( <div> <ul>
    <li><Link to={`${match.url}/shoes`}>Shoes</Link></li>
    <li><Link to={`${match.url}/boots`}>Boots</Link></li>
    <li><Link to={`${match.url}/footwear`}>Footwear</Link></li>

  </ul>
  <Route path={`${match.path}/:name`} render= {({match}) =>( <div> <h3>
  ↪ {match.params.name} </h3></div>)}/>
  </div>)
}
export default Category;
```

First, we've declared a couple of links for the nested routes. As previously

mentioned, `match.url` will be used for building nested links and `match.path` for nested routes. If you're having trouble understanding the concept of match, `console.log(match)` provides some useful information that might help to clarify it.

```
<Route path={`${match.path}/:name`}
  render= {({match}) =>( <div> <h3> {match.params.name} </h3></div>)}/>
```

This is our first attempt at dynamic routing. Instead of hard-coding the routes, we've used a variable within the pathname. `:name` is a path parameter and catches everything after `category/` until another forward slash is encountered. So, a pathname like `products/running-shoes` will create a `params` object as follows:

```
{
  name: 'running-shoes'
}
```

The captured data should be accessible at `match.params` or `props.match.params` depending on how the props are passed. The other interesting thing is that we've used a `render` prop. `render` props are pretty handy for inline functions that don't require a component of their own.

Demo 3: Nested routing with Path parameters

Let's complicate things a bit more, shall we? A real-world router will have to deal with data and display it dynamically. Assume that we have the product data returned by a server API of the form below.

src/Products.jsx

```
const productData = [
{
  id: 1,
  name: 'NIKE Liteforce Blue Sneakers',
  description: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  ↪Proin molestie.',
  status: 'Available'

},
{
  id: 2,
  name: 'Stylised Flip Flops and Slippers',
  description: 'Mauris finibus, massa eu tempor volutpat, magna dolor euismod
  ↪ dolor.',
  status: 'Out of Stock'

},
{
  id: 3,
  name: 'ADIDAS Adispree Running Shoes',
  description: 'Maecenas condimentum porttitor auctor. Maecenas viverra fringilla
  ↪ felis, eu pretium.',
  status: 'Available'
},
{
  id: 4,
  name: 'ADIDAS Mid Sneakers',
  description: 'Ut hendrerit venenatis lacus, vel lacinia ipsum fermentum vel.
  ↪ Cras.',
  status: 'Out of Stock'
```

```
},

];
```

We need to create routes for the following paths:

- `/products`. This should display a list of products.
- `/products/:productId`. If a product with the `:productId` exists, it should display the product data, and if not, it should display an error message.

src/Products.jsx

```
/* Import statements have been left out for code brevity */

const Products = ({ match }) => {

   const productsData = [
    {
        id: 1,
        name: 'NIKE Liteforce Blue Sneakers',
        description: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        ↪ Proin molestie.',
        status: 'Available'

    },

    //Rest of the data has been left out for code brevity

];
 /* Create an array of `<li>` items for each product */
```

```
  var linkList = productsData.map( (product) => {
    return(
      <li>
        <Link to={`${match.url}/${product.id}`}>
          {product.name}
        </Link>
      </li>
      )

    })

  return(
    <div>
        <div>
         <div>
           <h3> Products</h3>
           <ul> {linkList} </ul>
         </div>
        </div>

        <Route path={`${match.url}/:productId`}
            render={ (props) => <Product data= {productsData} {...props} />}/>
        <Route exact path={match.url}
            render={() => (
            <div>Please select a product.</div>
            )}
        />
    </div>
  )
}
```

First, we created a list of `<Links>`s using the `productsData.id`s and stored it in

`linkList`. The route takes a parameter in the path string which corresponds to that of the product id.

```
<Route path={`${match.url}/:productId`}
  render={ (props) => <Product data= {productsData} {...props} />}/>
```

You may have expected `component = { Product }` instead of the inline render function. The problem is that we need to pass down `productsData` to the Product component along with all the existing props. Although there are other ways you can do this, I find this method to be the easiest. `{...props}` uses the ES6's spread syntax to pass the whole props object to the component.

Here's the code for Product component.

src/Product.jsx

```
/* Import statements have been left out for code brevity */

const Product = ({match,data}) => {
  var product= data.find(p => p.id == match.params.productId);
  var productData;

  if(product)
    productData = <div>
      <h3> {product.name} </h3>
      <p>{product.description}</p>
      <hr/>
      <h4>{product.status}</h4>  </div>;
  else
```

```
    productData = <h2> Sorry. Product doesnt exist </h2>;

  return (
    <div>
      <div>
        {productData}
      </div>
    </div>
  )
}
```

The `find` method is used to search the array for an object with an id property that equals `match.params.productId`. If the product exists, the `productData` is displayed. If not, a "Product doesn't exist" message is rendered.

Protecting Routes

For the final demo, we'll be discussing techniques concerned with protecting routes. So, if someone tries to access `/admin`, they'd be required to log in first. However, there are some things we need to cover before we can protect routes.

Redirect

Like the server-side redirects, `<Redirect>` will replace the current location in the history stack with a new location. The new location is specified by the `to` prop. Here's how we'll be using `<Redirect>`:

```
<Redirect to={{pathname: '/login', state: {from: props.location}}}
```

So, if someone tries to access the `/admin` while logged out, they'll be redirected to the `/login` route. The information about the current location is passed via state, so that if the authentication is successful, the user can be redirected back to the original location. Inside the child component, you can access this information at `this.props.location.state`.

Custom Routes

A custom route is a fancy word for a route nested inside a component. If we need to make a decision whether a route should be rendered or not, writing a custom route is the way to go. Here's the custom route declared among other routes.

src/App.js

```
/* Add the PrivateRoute component to the existing Routes */
<Switch>
  <Route exact path="/" component={Home} data={data}/>
  <Route path="/category" component={Category}/>
  <Route path="/login" component={Login}/>
  <PrivateRoute authed={fakeAuth.isAuthenticated} path='/products' component =
  ↪ {Products} />
</Switch>
```

`fakeAuth.isAuthenticated` returns true if the user is logged in and false otherwise.

Here's the definition for `PrivateRoute`:

src/App.js

```
/* PrivateRoute component definition */
const PrivateRoute = ({component: Component, authed, ...rest}) => {
  return (
    <Route
      {...rest}
      render={(props) => authed === true
        ? <Component {...props} />
        : <Redirect to={{pathname: '/login', state: {from: props.
            ↪location}}} />} />
  )
}
```

The route renders the Admin component if the user is logged in. Otherwise, the user is redirected to `/login`. The good thing about this approach is that it is evidently more declarative and `PrivateRoute` is reusable.

Finally, here's the code for the Login component:

src/Login.jsx

```
import React from 'react';
import { Redirect } from 'react-router-dom';

class Login extends React.Component {

  constructor() {
    super();
```

```
    this.state = {
      redirectToReferrer: false
    }
    // binding 'this'
    this.login = this.login.bind(this);
  }

  login() {

    fakeAuth.authenticate(() => {
      this.setState({ redirectToReferrer: true })
    })
  }

  render() {
    const { from } = this.props.location.state || { from: { pathname: '/' } }
    const { redirectToReferrer } = this.state;

    if (redirectToReferrer) {
      return (
        <Redirect to={from} />
      )
    }

    return (
      <div>
        <p>You must log in to view the page at {from.pathname}</p>
        <button onClick={this.login}>Log in</button>
      </div>
    )
  }
}
```

```
/* A fake authentication function */
export const fakeAuth = {

  isAuthenticated: false,
  authenticate(cb) {
    this.isAuthenticated = true
     setTimeout(cb, 100)
  },
}
```

The line below demonstrates object destructuring, which is a part of the ES6 specification.

```
const { from } = this.props.location.state || { from: { pathname: '/' } }
```

Demo 4: Protecting Routes

Let's fit the puzzle pieces together, shall we? Here's the final demo of the application that we built using React router:

CodeSandbox Example

https://codesandbox.io/embed/nn8x24vm60

Summary

As you've seen in this article, React Router is a powerful library that complements React for building better, declarative routes. Unlike the prior

versions of React Router, in v4, everything is "just components". Moreover, the new design pattern perfectly fits into the React way of doing things.

In this tutorial, we learned:

- how to setup and install React Router
- the basics of routing and some essential components such as `<Router>`, `<Route>` and `<Link>`
- how to create a minimal router for navigation and nested routes
- how to build dynamic routes with path parameters

Finally, we learned some advanced routing techniques for creating the final demo for protected routes.

Chapter

3

How to Test React Components Using Jest

by Jack Franklin

In this tutorial, we'll take a look at using Jest – a testing framework maintained by Facebook – to test our ReactJS components. We'll look at how we can use Jest first on plain JavaScript functions, before looking at some of the features it provides out of the box specifically aimed at making testing React apps easier. It's worth noting that Jest isn't aimed specifically at React: you can use it to test any JavaScript applications. However, a couple of the features it provides come in really handy for testing user interfaces, which is why it's a great fit with React.

Sample Application

Before we can test anything, we need an application to test! Staying true to web development tradition, I've built a small todo application that we'll use as the starting point. You can find it, along with all the tests that we're about to write, on GitHub. If you'd like to play with the application to get a feel for it, you can also find a live demo online.

The application is written in ES2015, compiled using Webpack with the Babel ES2015 and React presets. I won't go into the details of the build set up, but it's all in the GitHub repo if you'd like to check it out. You'll find full instructions in the README on how to get the app running locally. If you'd like to read more, the application is built using Webpack, and I recommend "A Beginner's guide to Webpack" as a good introduction to the tool.

The entry point of the application is `app/index.js`, which just renders the `Todos` component into the HTML:

```
render(
  <Todos />,
  document.getElementById('app')
);
'app'
```

The `Todos` component is the main hub of the application. It contains all the state (hard-coded data for this application, which in reality would likely come from an API or similar), and has code to render the two child components: `Todo`, which is rendered once for each todo in the state, and `AddTodo`, which is rendered once and provides the form for a user to add a new todo.

Because the `Todos` component contains all the state, it needs the `Todo` and `AddTodo` components to notify it whenever anything changes. Therefore, it passes functions down into these components that they can call when some data changes, and `Todos` can update the state accordingly.

Finally, for now, you'll notice that all the business logic is contained in `app/state-functions.js`:

```
export function toggleDone(state, id) {…}

export function addTodo(state, todo) {…}

export function deleteTodo(state, id) {…}
```

These are all pure functions that take the state and some data, and return the new state. If you're unfamiliar with pure functions, they are functions that only reference data they are given and have no side effects. For more, you can read my article on A List Apart on pure functions and my article on SitePoint about pure functions and React.

If you're familiar with Redux, they're fairly similar to what Redux would call a reducer. In fact, if this application got much bigger I would consider moving into Redux for a more explicit, structured approach to data. But for this size application you'll often find that local component state and some well abstracted functions to be more than enough.

To TDD or Not to TDD?

There have been many articles written on the pros and cons of **test-driven development**, where developers are expected to write the tests first, before writing the code to fix the test. The idea behind this is that, by writing the test first, you have to think about the API that you're writing, and it can lead to a better design. For me, I find that this very much comes down to personal preference and also to the sort of thing I'm testing. I've found that, for React components, I like to write the components first and then add tests to the most important bits of functionality. However, if you find that writing tests first for your components fits your workflow, then you should do that. There's no hard rule here; do whatever feels best for you and your team.

This Tutorial Focuses on Testing Front-end Code

Note that this article will focus on testing front-end code. If you're looking for something focused on the back end, be sure to check out SitePoint's course Test-Driven Development in Node.js.

Introducing Jest

Jest was first released in 2014, and although it initially garnered a lot of interest, the project was dormant for a while and not so actively worked on. However, Facebook has invested the last year into improving Jest, and recently published a few releases with impressive changes that make it worth reconsidering. The only resemblance of Jest compared to the initial open-source release is the name and the logo. Everything else has been changed and rewritten. If you'd like to find out more about this, you can read Christoph Pojer's comment, where he discusses the current state of the project.

If you've been frustrated by setting up Babel, React and JSX tests using another framework, then I definitely recommend giving Jest a try. If you've found your existing test setup to be slow, I also highly recommend Jest. It automatically runs

tests in parallel, and its watch mode is able to run only tests relevant to the changed file, which is invaluable when you have a large suite of tests. It comes with JSDom configured, meaning you can write browser tests but run them through Node, can deal with asynchronous tests and has advanced features such as mocking, spies and stubs built in.

Installing and Configuring Jest

To start with, we need to get Jest installed. Because we're also using Babel, we'll install another couple of modules that make Jest and Babel play nicely out of the box:

```
npm install --save-dev babel-jest babel-polyfill babel-preset-es2015
        ↪ babel-preset-react jest
```

You also need to have a `.babelrc` file with Babel configured to use any presets and plugins you need. The sample project already has this file, which looks like so:

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

We won't install any React testing tools yet, because we're not going to start with testing our components, but our state functions.

Jest expects to find our tests in a `__tests__` folder, which has become a popular convention in the JavaScript community, and it's one we're going to stick to here. If you're not a fan of the `__tests__` setup, out of the box Jest also supports

finding any `.test.js` and `.spec.js` files too.

As we'll be testing our state functions, go ahead and create `__tests__/state-functions.test.js`.

We'll write a proper test shortly, but for now, put in this dummy test, which will let us check everything's working correctly and we have Jest configured.

```
describe('Addition', () => {
  it('knows that 2 and 2 make 4', () => {
    expect(2 + 2).toBe(4);
  });
});
```

Now, head into your `package.json`. We need to set up `npm test` so that it runs Jest, and we can do that simply by setting the `test` script to run `jest`.

```
"scripts": {
  "test": "jest"
}
```

If you now run `npm test` locally, you should see your tests run, and pass!

```
PASS  __tests__/state-functions.test.js
  Addition
    ✓ knows that 2 and 2 make 4 (5ms)
```

```
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 passed, 0 total
Time:        3.11s
```

If you've ever used Jasmine, or most testing frameworks, the above test code itself should be pretty familiar. Jest lets us use `describe` and `it` to nest tests as we need to. How much nesting you use is up to you; I like to nest mine so all the descriptive strings passed to `describe` and `it` read almost as a sentence.

When it comes to making actual assertions, you wrap the thing you want to test within an `expect()` call, before then calling an assertion on it. In this case, we've used `toBe`. You can find a list of all the available assertions in the Jest documentation. `toBe` checks that the given value matches the value under test, using `===` to do so. We'll meet a few of Jest's assertions through this tutorial.

Testing Business Logic

Now we've seen Jest work on a dummy test, let's get it running on a real one! We're going to test the first of our state functions, `toggleDone`. `toggleDone` takes the current state and the ID of a todo that we'd like to toggle. Each todo has a `done` property, and `toggleDone` should swap it from `true` to `false`, or vice-versa.

if You're Following Along

If you're following along with this, make sure you've cloned the repo and have copied the `app` folder to the same directory that contains your `___tests__` folder. You'll also need to install the `shortid` package (`npm install shortid --save`), which is a dependency of the Todo app.

I'll start by importing the function from `app/state-functions.js`, and setting up the test's structure. Whilst Jest allows you to use `describe` and `it` to nest as deeply as you'd like to, you can also use `test`, which will often read better. `test` is just an alias to Jest's `it` function, but can sometimes make tests much easier to read and less nested.

For example, here's how I would write that test with nested `describe` and `it` calls:

```
import { toggleDone } from '../app/state-functions';

describe('toggleDone', () => {
  describe('when given an incomplete todo', () => {
    it('marks the todo as completed', () => {
    });
  });
});
```

And here's how I would do it with `test`:

```
import { toggleDone } from '../app/state-functions';

test('toggleDone completes an incomplete todo', () => {
});
```

The test still reads nicely, but there's less indentation getting in the way now. This one is mainly down to personal preference; choose whichever style you're more comfortable with.

Now we can write the assertion. First we'll create our starting state, before passing it into `toggleDone`, along with the ID of the todo that we want to toggle. `toggleDone` will return our finish state, which we can then assert on:

```
const startState = {
  todos: [{ id: 1, done: false, name: 'Buy Milk' }]
};

const finState = toggleDone(startState, 1);

expect(finState.todos).toEqual([
  { id: 1, done: true, name: 'Buy Milk' }
]);
```

Notice now that I use `toEqual` to make my assertion. You should use `toBe` on primitive values, such as strings and numbers, but `toEqual` on objects and arrays. `toEqual` is built to deal with arrays and objects, and will recursively check each field or item within the object given to ensure that it matches.

With that we can now run `npm test` and see our state function test pass:

```
PASS  __tests__/state-functions.test.js
  ✓ tooggleDone completes an incomplete todo (9ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 passed, 0 total
Time:        3.166s
```

Rerunning Tests on Changes

It's a bit frustrating to make changes to a test file and then have to manually run `npm test` again. One of Jest's best features is its watch mode, which watches for file changes and runs tests accordingly. It can even figure out which subset of tests to run based on the file that changed. It's incredibly powerful and reliable, and you're able to run Jest in watch mode and leave it all day whilst you craft your code.

To run it in watch mode, you can run `npm test -- --watch`. Anything you pass to `npm test` after the first `--` will be passed straight through to the underlying command. This means that these two commands are effectively equivalent:

- `npm test -- --watch`
- `jest --watch`

I would recommend that you leave Jest running in another tab, or terminal window, for the rest of this tutorial.

Before moving onto testing the React components, we'll write one more test on another one of our state functions. In a real application I would write many more tests, but for the sake of the tutorial, I'll skip some of them. For now, let's write a test that ensures that our `deleteTodo` function is working. Before seeing how

I've written it below, try writing it yourself and seeing how your test compares.

Remember that you will have to update the `import` statement at the top to import `deleteTodo` along with `toggleTodo`:

```
import { toggleTodo, deleteTodo } from '../app/state-functions';
```

And here's how I've written the test:

```
test('deleteTodo deletes the todo it is given', () =&gt; {
    const startState = {
      todos: [{ id: 1, done: false, name: 'Buy Milk' }]
    };

    const finState = deleteTodo(startState, 1);

    expect(finState.todos).toEqual([]);
  });
```

The test doesn't vary too much from the first: we set up our initial state, run our function and then assert on the finished state. If you left Jest running in watch mode, notice how it picks up your new test and runs it, and how quick it is to do so! It's a great way to get instant feedback on your tests as you write them.

The tests above also demonstrate the perfect layout for a test, which is:

- set up
- execute the function under test
- assert on the results.

By keeping the tests laid out in this way, you'll find them easier to follow and work with.

Now we're happy testing our state functions, let's move on to React components.

Testing React Components

It's worth noting that, by default, I would actually encourage you to not write too many tests on your React components. Anything that you want to test very thoroughly, such as business logic, should be pulled out of your components and sit in standalone functions, just like the state functions that we tested earlier. That said, it is useful at times to test some React interactions (making sure a specific function is called with the right arguments when the user clicks a button, for example). We'll start by testing that our React components render the right data, and then look at testing interactions. Then we'll move on to snapshots, a feature of Jest that makes testing the output of React components much more convenient.

To do this, we'll need to make use of `react-addons-test-utils`, a library that provides functions for testing React. We'll also install Enzyme, a wrapper library written by AirBnB that makes testing React components much easier. We'll use this API throughout our tests. Enzyme is a fantastic library, and the React team even recommend it as the way to test React components.

```
npm install --save-dev react-addons-test-utils enzyme
```

Let's test that the `Todo` component renders the text of its todo inside a paragraph. First we'll create `__tests__/todo.test.js`, and import our component:

```
import Todo from '../app/todo';
import React from 'react';
import { mount } from 'enzyme';

test('Todo component renders the text of the todo', () => {
});
```

I also import `mount` from Enzyme. The `mount` function is used to render our component and then allow us to inspect the output and make assertions on it. Even though we're running our tests in Node, we can still write tests that require a DOM. This is because Jest configures jsdom, a library that implements the DOM in Node. This is great because we can write DOM based tests without having to fire up a browser each time to test them.

We can use `mount` to create our `Todo`:

```
const todo = { id: 1, done: false, name: 'Buy Milk' };
const wrapper = mount(
  <Todo todo={todo} />
);
```

And then we can call `wrapper.find`, giving it a CSS selector, to find the paragraph that we're expecting to contain the text of the Todo. This API might remind you of jQuery, and that's by design. It's a very intuitive API for searching rendered output to find the matching elements.

```
const p = wrapper.find('.toggle-todo');
```

And finally, we can assert that the text within it is `Buy Milk`:

```
expect(p.text()).toBe('Buy Milk');
```

Which leaves our entire test looking like so:

```
import Todo from '../app/todo';
import React from 'react';
import { mount } from 'enzyme';

test('TodoComponent renders the text inside it', () => {
  const todo = { id: 1, done: false, name: 'Buy Milk' };
  const wrapper = mount(
    <Todo todo={todo} />
  );
  const p = wrapper.find('.toggle-todo');
  expect(p.text()).toBe('Buy Milk');
});
'.toggle-todo'
```

Phew! You might think that was a lot of work and effort to check that "Buy Milk" gets placed onto the screen, and, well ... you'd be correct. Hold your horses for now, though; in the next section we'll look at using Jest's snapshot ability to make this much easier.

In the meantime, let's look at how you can use Jest's spy functionality to assert that functions are called with specific arguments. This is useful in our case, because we have the `Todo` component which is given two functions as properties, which it should call when the user clicks a button or performs an interaction.

In this test we're going to assert that when the todo is clicked, the component will call the `doneChange` prop that it's given.

```
test('Todo calls doneChange when todo is clicked', () => {
});
```

What we want to do is to have a function that we can keep track of its calls, and the arguments that it's called with. Then we can check that when the user clicks the todo, the `doneChange` function is called and also called with the correct arguments. Thankfully, Jest provides this out of the box with spies. A **spy** is a function whose implementation you don't care about; you just care about when and how it's called. Think of it as you spying on the function. To create one, we call `jest.fn()`:

```
const doneChange = jest.fn();
```

This gives a function that we can spy on and make sure it's called correctly. Let's start by rendering our `Todo` with the right props:

```
const todo = { id: 1, done: false, name: 'Buy Milk' };
const doneChange = jest.fn();
const wrapper = mount(
  <Todo todo={todo} doneChange={doneChange} />
);
```

Next, we can find our paragraph again, just like in the previous test:

```
const p = TestUtils.findRenderedDOMComponentWithClass(rendered, 'toggle-todo');
```

And then we can call `simulate` on it to simulate a user event, passing `click` as the argument:

```
p.simulate('click');
```

And all that's left to do is assert that our spy function has been called correctly. In this case, we're expecting it to be called with the ID of the todo, which is `1`. We can use `expect(doneChange).toBeCalledWith(1)` to assert this, and with that we're done with our test!

```
test('TodoComponent calls doneChange when todo is clicked', () => {
  const todo = { id: 1, done: false, name: 'Buy Milk' };
  const doneChange = jest.fn();
  const wrapper = mount(
    <Todo todo={todo} doneChange={doneChange} />
  );

  const p = wrapper.find('.toggle-todo');
  p.simulate('click');
  expect(doneChange).toBeCalledWith(1);
});
'.toggle-todo'
```

Better Component Testing with Snapshots

I mentioned above that this might feel like a lot of work to test React components, especially some of the more mundane functionalities (such as rendering the text). Rather than make a large amount of assertions on React components, Jest lets you run snapshot tests. These are not so useful for interactions (in which case I still prefer a test like we just wrote above), but for testing that the output of your component is correct, they're much easier.

When you run a snapshot test, Jest renders the React component under test and stores the result in a JSON file. Every time the test runs, Jest will check that the React component still renders the same output as the snapshot. Then, when you change a component's behavior, Jest will tell you and either:

- you'll realize you made a mistake, and you can fix the component so it matches the snapshot again
- or, you made that change on purpose, and you can tell Jest to update the snapshot.

This way of testing means that:

- you don't have to write a lot of assertions to ensure your React components are behaving as expected
- you can never accidentally change a component's behavior, because Jest will realize.

You also don't have to snapshot all your components. In fact, I'd actively recommend against it. You should pick components with some functionality that you really need to ensure is working. Snapshotting all your components will just lead to slow tests that aren't useful. Remember, React is a very thoroughly tested framework, so we can be confident that it will behave as expected. Make sure you don't end up testing the framework, rather than your code!

To get started with snapshot testing, we need one more Node package. react-

test-renderer is a package that's able to take a React component and render it as a pure JavaScript object. This means it can then be saved to a file, and this is what Jest uses to keep track of our snapshots.

```
npm install --save-dev react-test-renderer
```

Now, let's rewrite our first Todo component test to use a snapshot. For now, comment out the `TodoComponent calls doneChange when todo is clicked` test as well.

The first thing you need to do is import the `react-test-renderer`, and also remove the import for `mount`. They can't both be used; you either have to use one or the other. This is why we have commented the other test out for now.

```
import renderer from 'react-test-renderer';
```

Now I'll use the renderer we just imported to render the component, and assert that it matches the snapshot:

```
describe('Todo component renders the todo correctly', () => {
  it('renders correctly', () => {
    const todo = { id: 1, done: false, name: 'Buy Milk' };
    const rendered = renderer.create(
      <Todo todo={todo} />
    );
    expect(rendered.toJSON()).toMatchSnapshot();
  });
```

```
});
```

The first time you run this, Jest is clever enough to realize that there's no snapshot for this component, so it creates it. Let's take a look at `__tests__/__snapshots__/todo.test.js.snap`:

```
exports[`Todo component renders the todo correctly renders correctly 1`] = `
<div
  className="todo  todo-1">
  <p
    className="toggle-todo"
   "todo  todo-1"`
<div
  className="todo  todo-1">
  <p
    className="toggle-todo"
    onClick={[Function]}>
    Buy Milk
  </p>
  <a
    className="delete-todo"
    href="#"
    onClick={[Function]}>
    Delete
  </a>
</div>
`
```

You can see that Jest has saved the output for us, and now the next time we run this test it will check that the outputs are the same. To demonstrate this, I'll

break the component by removing the paragraph that renders the text of the todo, meaning that I've removed this line from the `Todo` component:

```
<p className="toggle-todo" onClick={() => this.toggleDone() }>{ todo.name }</p>
```

Let's see what Jest says now:

```
FAIL  __tests__/todo.test.js
 ● Todo component renders the todo correctly › renders correctly

   expect(value).toMatchSnapshot()

   Received value does not match stored snapshot 1.

   - Snapshot
   + Received

     <div
       className="todo  todo-1">
   -   <p
   -     className="toggle-todo"
   -     onClick={[Function]}>
   -     Buy Milk
   -   </p>
       <a
         className="delete-todo"
         href="#"
         onClick={[Function]}>
         Delete
       </a>
```

```
    </div>

    at Object.<anonymous> (__tests__/todo.test.js:21:31)
    at process._tickCallback (internal/process/next_tick.js:103:7)
```

Jest realized that the snapshot doesn't match the new component, and lets us know in the output. If we think this change is correct, we can run jest with the `-u` flag, which will update the snapshot. In this case, though, I'll undo my change and Jest is happy once more.

Next we can look at how we might use snapshot testing to test interactions. You can have multiple snapshots per test, so you can test that the output after an interaction is as expected.

We can't actually test our Todo component interactions through Jest snapshots, because they don't control their own state but call the callback props they are given. What I've done here is move the snapshot test into a new file, todo.snapshot.test.js, and leave our toggling test in todo.test.js. I've found it useful to separate the snapshot tests into a different file; it also means that you don't get conflicts between `react-test-renderer` and `react-addons-test-utils`.

Remember, you'll find all the code that I've written in this tutorial available on GitHub for you to check out and run locally.

Conclusion

Facebook released Jest a long time ago, but in recent times it's been picked up and worked on excessively. It's fast become a favorite for JavaScript developers and it's only going to get better. If you've tried Jest in the past and not liked it, I can't encourage you enough to try it again, because it's practically a different

framework now. It's quick, great at rerunning specs, gives fantastic error messages and tops it all off with its snapshot functionality.

If you have any questions please feel free to raise an issue on GitHub and I'll be happy to help. And please be sure check out Jest on GitHub and star the project; it helps the maintainers.

Chapter

4

Building Animated Components, or How React Makes D3 Better

by Swizec Teller

D3 is great. As the jQuery of the web data visualization world, it can do everything you can think of.

Many of the best data visualizations you've seen online use D3. It's a great library, and with the recent v4 update, it became more robust than ever.

Add React, and you can make D3 even better.

Just like jQuery, D3 is powerful but low level. The bigger your visualization, the harder your code becomes to work with, the more time you spend fixing bugs and pulling your hair out.

React can fix that.

You can read my book React+d3js ES6 for a deep insight, or keep reading for an overview of how to best integrate React and D3. In a practical example, we'll see how to build declarative, transition-based animations.

A version of this article also exists as a D3 meetup talk on YouTube.

Is React Worth It?

OK, React is big. It adds *a ton* of code to your payload, and it increases your dependency footprint. It's yet another library that you have to keep updated.

If you want to use it effectively, you'll need a build step. Something to turn JSX code into pure JavaScript.

Setting up Webpack and Babel is easy these days: just run `create-react-app`. It gives you JSX compilation, modern JavaScript features, linting, hot loading, and code minification for production builds. It's great.

Despite the size and tooling complexity, React *is* worth it, *especially* if you're serious about your visualization. If you're building a one-off that you'll never have

to maintain, debug, or expand, stick to pure D3. If you're building something real, I encourage you to add React to the mix.

To me, the main benefit is that React forces strongly encourages you to componentize your code. The other benefits are either symptoms of componentization, or made possible by it.

The main benefits of using React with your D3 code are:

- componentization
- easier testing and debugging
- smart DOM redraws
- hot loading

Componentization encourages you to build your code as a series of logical units – components. With JSX, you can use them like they were HTML elements: `<Histogram />`, `<Piechart />`, `<MyFancyThingThatIMade />`. We'll dive deeper into that in the next section.

Building your visualization as a series of components makes it easier to test and debug. You can focus on logical units one at a time. If a component works here, it will work over there as well. If it passes tests and looks nice, it will pass tests and look nice no matter how often you render it, no matter where you put it, and no matter who calls it.

React understands the structure of your code, so it knows how to redraw only the components that have changes. There's no more hard work in deciding what to re-render and what to leave alone. Just change and forget. React can figure it out on its own. And yes, if you look at a profiling tool, you'll see that *only* the parts with changes are re-rendered.

Using `create-react-app` to configure your tooling, React can utilize hot loading. Let's say you're building a visualization of 30,000 datapoints. With pure D3, you have to refresh the page for every code change. Load the dataset, parse the

dataset, render the dataset, click around to reach the state you're testing … yawn.

With React -> no reload, no waiting. Just immediate changes on the page. When I first saw it in action, it felt like eating ice cream while the crescendo of *1812 Overture* plays in the background. Mind = blown.

Benefits of Componentization

Components this, components that. Blah blah blah. Why should you care? Your dataviz code already works. You build it, you ship it, you make people happy.

But does the code make *you* happy? With components, it can. Components make your life easier because they make your code:

- declarative
- reusable
- understandable
- organized

It's okay if that sounds like buzzword soup. Let me show you.

For instance, **declarative code** is the kind of code where you say *what* you want, not *how* you want it. Ever written HTML or CSS? You know how to write declarative code! Congratz!

React uses JSX to make your JavaScript look like HTML. But don't worry, it all compiles to pure JavaScript behind the scenes.

Try to guess what this code does:

```
render() {
```

```
// ...
return (
  <g transform={translate}>
    <Histogram data={this.props.data}
       value={(d) => d.base_salary}
       x={0}
       y={0}
       width={400}
       height={200}
       title="All" />
    <His"All"m data={engineerData}
       value={(d) => d.base_salary}
       x={450}
       y={0}
       width={400}
       height={200}
       title="Engineer" />
    <Histogram data={programmerData}
       value={(d) => d.base_salary}
       x={0}
       y={220}
       width={400}
       height={200}
       title="Programmer"/>
    <Histogram data={developerData}
       value={(d) => d.base_salary}
       x={450}
       y={220}
       width={400}
       height={200}
       title="Developer" />
  </g>
```

```
    )
  }
```

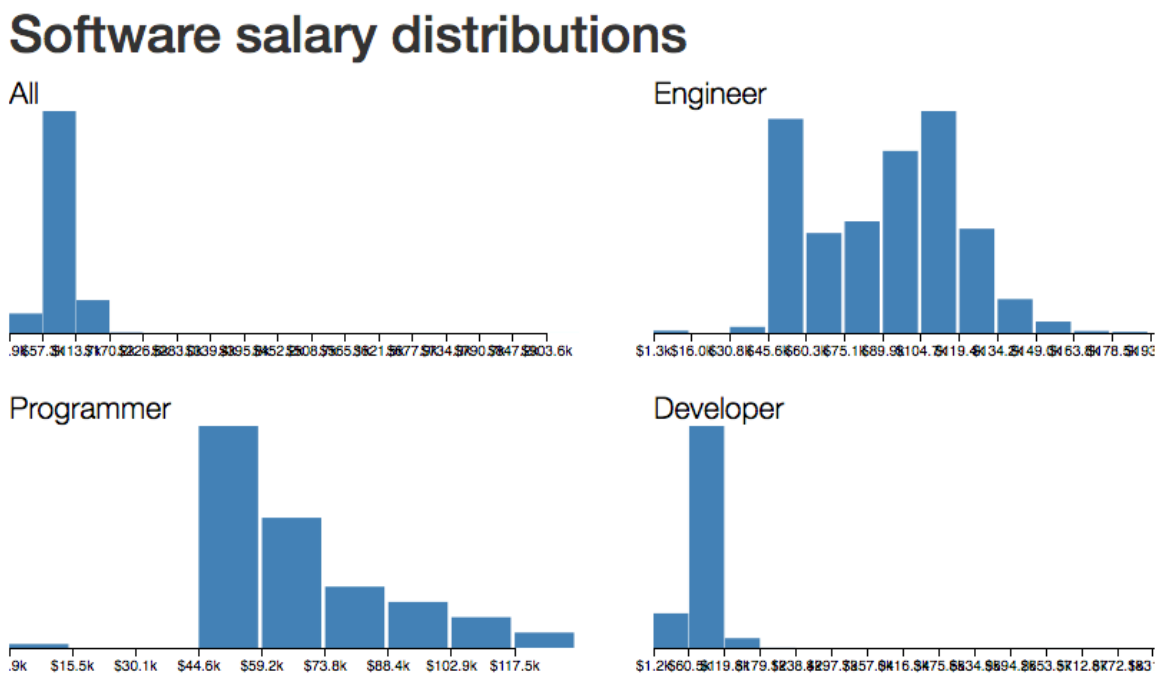
If you guessed *"Renders four histograms"*, you were right. Hooray.

After you create a Histogram component, you can use it like it was a normal piece of HTML. A histogram shows up anywhere you put `<Histogram />` with the right parameters.

In this case, the parameters are `x` and `y` coordinates, `width` and `height` sizing, the `title`, some `data`, and a `value` accessor. They can be anything your component needs.

Parameters look like HTML attributes, but can take any JavaScript object, even functions. It's like HTML on steroids.

With some boilerplate and the right dataset, that code above gives you a picture like this. A comparison of salary distributions for different types of people who write software.



Look at the code again. Notice how **reusable** components are? It's like `<Histogram />` was a function that created a histogram. Behind the scenes it *does* compile into a function call – `(new Histogram()).render()`, or something similar. `Histogram` becomes a class, and you call an instance's render function every time you use `<Histogram />`.

React components should follow the principles of good functional programming. No side effects, statelessness, idempotency, comparability. Unless you really, really want to break the rules.

Unlike JavaScript functions, where following these principles requires deliberate effort, React makes it hard *not* to code that way. That's a win when you work in a team.

Declarativeness and reusability make your code **understandable** by default. If you've ever used HTML, you can read what that code does. You might not understand the details, but if you know some HTML and JavaScript, you know

how to read JSX.

Complex components are made out of simpler components, which are made out of even simpler components, which are eventually made out of pure HTML elements. This keeps your code organized.

When you come back in six months, you can look at your code and think, *"Ah yes, four histograms. To tweak this, I should open the Histogram component and poke around."*

React takes the principles I've always loved about fancy-pants functional programming and makes them practical. I love that.

Let me show you an example – an animated alphabet.

A Practical Example

We're going to build an animated alphabet. Not because it's the simplest example of using React and D3 together, but because it looks cool. When I show this at live talks, people always oooh and aaah, especially when I show proof that only the DOM elements with changes get redrawn.

This is a shortened version of a more in-depth article on React and D3 and transitions that I posted on my blog a few months ago. We're going to gloss over

some details in this version to keep it short. You can dive into the full codebase in the GitHub repository.

The code is based on React 15 and D3 4.0.0. Some of the syntax I use, like class properties, is not in stable ES6 yet, but should work if you use `create-react-app` for your tooling setup.

To make an animated alphabet, we need two components:

- `Alphabet`, which creates random lists of letters every 1.5 seconds, then maps through them to render `Letter` components
- `Letter`, which renders an SVG text element, and takes care of its own enter/update/exit transitions.

We're going to use React to render SVG elements, and we'll use D3 for transitions, intervals, and some maths.

The Alphabet Component

The `Alphabet` component holds the current list of letters in state and renders a collection of `Letter` components in a loop.

We start with a skeleton like this:

```
// src/components/Alphabet/index.jsx
import React, { Component } from 'react';
import ReactTransitionGroup from 'react-addons-transition-group';
import * as d3 from 'd3';

require('./style.css');
```

```
import Letter from './Letter';

class Alphabet extends Component {
  static letters = "abcdefghijklmnopqrstuvwxyz".split('');
  state = {alphabet: []}

  componentWillMount() {
    // starts an interval to update alphabet
  }

  render() {
    // spits out svg elements
  }
}

export default Alphabet;
```

We import our dependencies, add some styling, and define the `Alphabet` component. It holds a list of available letters in a static `letters` property and an empty `alphabet` in component state. We'll need a `componentWillMount` and a `render` method as well.

The best place to create a new alphabet every 1.5 seconds is in `componentWillMount`:

```
// src/components/Alphabet/index.jsx
  componentWillMount() {
    d3.interval(() => this.setState({
       alphabet: d3.shuffle(Alphabet.letters)
```

```
          .slice(0, Math.floor(Math.random() * Alphabet.letters.length))
          .sort()
    }), 1500);
  }
```

We use `d3.interval( //.., 1500)` to call a function every 1.5 seconds. On each period, we shuffle the available letters, slice out a random amount, sort them, and update component state with `setState()`.

This ensures our alphabet is both random and in alphabetical order. `setState()` triggers a re-render.

Our declarative magic starts in the `render` method.

```
// src/components/Alphabet/index.jsx
render() {
  let transform = `translate(${this.props.x}, ${this.props.y})`;

  return (
    <g transform={transform}>
      <ReactTransitionGroup component="g">
        {this.state.alphabet.map((d, i) => (
          <Letter d={d} i={i} key={"g"`letter-${d}`} />
        ))}
      </ReactTransitionGroup>
    </g>
  );
}
```

We use an SVG transformation to move our alphabet into the specified `(x, y)` position, then define a `ReactTransitionGroup` and map through `this.state.alphabet` to render a bunch of `Letter` components with wanton disregard.

Each `Letter` gets its current text, `d`, and index, `i`. The `key` attribute helps React recognize which component is which. Using `ReactTransitionGroup` gives us special component lifecycle methods that help with smooth transitions.

ReactTransitionGroup

In addition to the normal lifecycle hooks that tell us when a component mounts, updates, and unmounts, ReactTransitionGroup gives us access to `componentWillEnter`, `componentWillLeave`, and a few others. Notice something familiar?

`componentWillEnter` is the same as D3's `.enter()`, `componentWillLeave` is the same as D3's `.exit()`, and `componentWillUpdate` is the same as D3's `.update()`.

"The same" is a strong concept; they're analogous. D3's hooks operate on entire selections – groups of components – while React's hooks operate on each component individually. In D3, an overlord is dictating what happens; in React, each component knows what to do.

That makes React code easier to understand. I think. ¯_(ツ)_/¯

`ReactTransitionGroup` gives us even more hooks, but these three are all we need. It's nice that in both `componentWillEnter` and `componentWillLeave`, we can use a callback to explicitly say *"The transition is done. React, back to you"*.

My thanks to Michelle Tilley for writing about `ReactTransitionGroup` on Stack Overflow.

The Letter Component

Now we're ready for the cool stuff – a component that can transition itself into and out of a visualization declaratively.

The basic skeleton for our `Letter` component looks like this:

```
// src/components/Alphabet/Letter.jsx

import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import * as d3 from 'd3';

class Letter extends Component {
    state = {
      y: -60,
      x: 0,
      className: 'enter',
      fillOpacity: 1e-6
    }
    transition = d3.transition()
                   .duration(750)
                   .ease(d3.easeCubicInOut);

    componentWillEnter(callback) {
      // start enter transition, then callback()
    }

    componentWillLeave(callback) {
      // start exit transition, then callback()
    }
```

```
    componentWillReceiveProps(nextProps) {
      if (this.props.i != nextProps.i) {
        // start update transition
      }
    }

    render() {
       // spit out a <text> element
    }
};

export default Letter;
```

We start with some dependencies and define a `Letter` component with a default state and a default transition. In most cases, you'd want to avoid using `state` for coordinates and other transient properties. That's what props are for. With transitions we use state because it helps us keep React's reality in sync with D3's reality.

That said, those magic default values could be default props. That would make our `Alphabet` more flexible.

componentWillEnter

We put the enter transition in `componentWillEnter`.

```
// src/components/Alphabet/Letter.jsx
    componentWillEnter(callback) {
      let node = d3.select(ReactDOM.findDOMNode(this));
```

```
    this.setState({x: this.props.i*32});

    node.transition(this.transition)
      .attr('y', 0)
      .style('fill-opacity', 1)
      .on('end', () => {
          this.setState({y: 0, fillOpacity: 1});
          callback()
      });
  }
```

We use `reactDOM.findDOMNode()` to get our DOM node and use `d3.select()` to turn it into a d3 selection. Now anything D3 can do, our component can do. Yessss!/p>

Then we update `this.state.x` using the current index and letter width. The width is a value that we Just Know™. Putting `x` in state helps us avoid jumpiness: The `i` prop changes on each update, but we want to delay when the `Letter` moves.

When a `Letter` first renders, it's invisible and 60 pixels above the baseline. To animate it moving down and becoming visible, we use a D3 transition.

We use `node.transition(this.transition)` to start a new transition with default settings from earlier. Any `.attr` and `.style` changes that we make happen over time directly on the DOM element itself.

This confuses React, because it assumes it's the lord and master of the DOM. So we have to sync React's reality with actual reality using a callback: `.on('end', ...)`. We use `setState()` to update component state, and trigger the main

`callback`. React now knows this letter is done appearing.

componentWillLeave

The exit transition goes in `componentWillLeave()`. Same concept as above, just in reverse.

```
// src/components/Alphabet/
  componentWillLeave(callback) {
    let node = d3.select(ReactDOM.findDOMNode(this));

    this.setState({className: 'exit'});

    node.transition(this.transition)
      .attr('y', 60)
      .style('fill-opacity', 1e-6)
      .on('end', () => {
          callback()
      });
  }
```

This time, we update state to change the `className` instead of `x`. That's because `x` doesn't change.

The exit transition itself is an inverse of the enter transition: letter moves down and becomes invisible. After the transition, we tell React it's okay to remove the component.

componentWillReceiveProps

The update transition goes into `componentWillReceiveProps()`.

```
// src/components/Alphabet/Letter.jsx
  componentWillReceiveProps(nextProps) {
    if (this.props.i != nextProps.i) {
      let node = d3.select(ReactDOM.findDOMNode(this));

      this.setState({className: 'update'});

      node.transition(this.transition)
        .attr('x', nextProps.i*32)
        .on('end', () => this.setState({x: nextProps.i*32}));
    }
  }
```

You know the pattern by now, don't you? Update state, do transition, sync state with reality after transition.

In this case, we change the `className`, then move the letter into its new horizontal position.

render

After all that transition magic, you might be thinking *"Holy cow, how do I render this!?"*. I don't blame ya!

But we did all the hard work. Rendering is straightforward:

```
// src/components/Alphabet/Letter.jsx
  render() {
    return (
      <text dy=".35em"
```

```
          y={this.state.y}
          x={this.state.x}
          className={this.state.className}
          style={{fillOpacity: this.state.fillOpacity}}>
        {this.props.d}
      </text>
    );
  }
```

We return an SVG `<text>` element rendered at an `(x, y)` position with a `className` and a `fillOpacity`. It shows a single letter given by the `d` prop.

As mentioned: using state for `x`, `y`, `className`, and `fillOpacity` is wrong in theory. You'd normally use props for that. But state is the simplest way I can think of to communicate between the render and lifecycle methods.

You Know the Basics!

Boom. That's it. You know how to build an animated declarative visualization. That's pretty cool if you ask me.

Here is what it looks like in action.

Such nice transitions, and all you had to do was loop through an array and render some `<Letter>` components. How cool is that?

In Conclusion

You now understand React well enough to make technical decisions. You can look at project and decide: *"Yes, this is more than a throwaway toy. Components and debuggability will help me."*

For extra fun, you also know how to use React and D3 together to build declarative animations. A feat most difficult in the olden days.

To learn more about properly integrating React and D3 check out my book, React+d3js ES6.

Chapter

Using Preact as a React Alternative

5

by Ahmed Bouchefra

Preact is an implementation of the virtual DOM component paradigm just like React and many other similar libraries. Unlike React, it's only 3KB in size, and it also outperforms it in terms of speed. It's created by Jason Miller and available under the well-known permissive and open-source MIT license.

Why Use Preact?

Preact is a lightweight version of React. You may prefer to use Preact as a lightweight alternative if you like building views with React but performance, speed and size are a priority for you – for example, in the case of mobile web apps or progressive web apps.

Whether you're starting a new project or developing an existing one, Preact can save you a lot of time. You don't need to reinvent the wheel trying to learn a new library, since it's similar to, and compatible with, React – to the point that you can use existing React packages with it with only some aliasing, thanks to the compatibility layer `preact-compat`.

Pros and Cons

There are many differences between React and Preact that we can summarize in three points:

- Features and API: Preact includes only a subset of the React API, and not all available features in React.
- Size: Preact is much smaller than React.
- Performance: Preact is faster than React.

Every library out there has its own set of pros and cons, and only your priorities can help you decide which library is a good fit for your next project. In this section, I'll try to list the pros and cons of the two libraries.

Preact Pros

- Preact is lightweight, smaller (only 3KB in size when gzipped) and faster than React (see these tests). You can also run performance tests in your browser via this link.
- Preact is largely compatible with React, and has the same ES6 API as React, which makes it dead easy either to adopt Preact as a new library for building user interfaces in your project or to swap React with Preact for an existing project for performance reasons.
- It has good documentation and examples available from the official website.
- It has a powerful and official CLI for quickly creating new Preact projects, without the hassle of Webpack and Babel configuration.
- Many features are inspired by all the work already done on React.
- It has also its own set of advanced features independent from React, like Linked State.

React Pros

- React supports one-way data binding.
- It's backed by a large company, Facebook.
- Good documentation, examples, and tutorials on the official website and the web.
- Large community.
- Used on Facebook's website, which has millions of visitors worldwide.
- Has its own official developer debugging tools extension for Chrome.
- It has the Create React App project boilerplate for quickly creating projects with zero configuration.
- It has a well-architectured and complex codebase.

React Cons

- React has a relatively large size in comparison with Preact or other existing similar libraries. (React minified source file is around 136KB in size, or about 42KB when minified and gzipped.)

- It's slower than Preact.
- As a result of its complex codebase, it's harder for novice developers to contribute.

License Concerns

Another con I listed while writing this article was that React had a grant patent clause paired with the BSD license, making it legally unsuitable for some use cases. However, in September 2017, the React license switched to MIT, which resolved these license concerns.

Preact Cons

- Preact supports only stateless functional components and ES6 class-based component definition, so there's no **createClass**.
- No support for context.
- No support for React propTypes.
- Smaller community than React.

Getting Started with Preact CLI

Preact CLI is a command line tool created by Preact's author, Jason Miller. It makes it very easy to create a new Preact project without getting bogged down with configuration complexities, so let's start by installing it.

Open your terminal (Linux or macOS) or command prompt (Windows), then run the following commands:

```
npm i -g preact-cli@latest
```

This will install the latest version of Preact CLI, assuming you have Node and NPM installed on your local development machine.

You can now create your project with this:

```
preact create my-app
```

Or with this, ff you want to create your app interactively:

```
preact init
```

Next, navigate inside your app's root folder and run this:

```
npm start
```

This will start a live-reload development server.

Finally, when you finish developing your app, you can build a production release using this:

```
npm run build
```

Demystifying Your First Preact App

After successfully installing the Preact CLI and generating an app, let's try to understand the simple app generated with the Preact CLI.

The Preact CLI generates the following directory structure

```
├── node_modules
├── package.json
├── package-lock.json
└── src
    ├── assets
    ├── components
    │   ├── app.js
    │   └── header
    ├── index.js
    ├── lib
    ├── manifest.json
    ├── routes
    │   ├── home
    │   └── profile
    └── style
        └── index.css
```

The `components` folder holds Preact components, and the `routes` folder holds the page components used for each app's route. You can use the `lib` folder for any external libraries, the `style` folder for CSS styles, and the `assets` for icons and other graphics.

Note the `manifest.json` file, which is like `package.json` but for PWAs (progressive web apps). Thanks to the Preact CLI, you can have a perfect-score PWA out of the box.

Now, if you open your project's `package.json` file, you'll see that the main entry point is set to `src/index.js`. Here is the content of this file:

```
import './style';
import App from './components/app';

export default App;
```

As you can see, `index.js` imports styles, and App component from `./components/app**`, and then just exports it as the default.

Now, let's see what's inside `./components/app`:

```
import { h, Component } from 'preact';
import { Router } from 'preact-router';

import Header from './header';
import Home from '../routes/home';
import Profile from '../routes/profile';

export default class App extends Component {
    handleRoute = e => {
        this.currentUrl = e.url;
    };

    render() {
        return (
            <div id="app">
                <Header />
                <Router onChange={this.handleRoute}>
                    <Home path="/" />
                    <Profile path="/profile/" user="me" />
```

```
                    <Profile path="/profile/:user" />
                </Router>
            </div>
        );
    }
}
```

This file exports a default class `App` which extends the `Component` class imported from the `preact` package. Every Preact component needs to extend the `Component` class.

`App` defines a `render` method, which returns a bunch of HTML elements and Preact components that render the app's main user interface.

Inside the `div` element, we have two Preact components, `Header` – which renders the app's header – and a `Router` component.

The Preact Router is similar to the latest version of React Router (version 4). You simply need to wrap the child components with a `<Router>` component, then specify the `path` prop for each component. Then, the router will take care of rendering the component, which has a path prop that matches the current browser's URL.

It's worth mentioning that Preact Router is very simple and, unlike React Router, it doesn't support advanced features such as nested routes and view composition. If you need these features, you have to use either the React Router v3 by aliasing `preact-compat`, or better yet use the latest React Router (version 4) which is more powerful than v3 and doesn't need any compatibility layer, because it works directly with Preact. (See this CodePen demo for an example.)

Preact Compatibility Layer

The `preact-compat` module allows developers to switch from React to Preact without changing imports from React and ReactDOM to Preact, or to use existing React packages with Preact.

Using `preact-compat` is easy. All you have to do is to first install it via npm:

```
npm i -S preact preact-compat
```

Then set up your build system to redirect imports or requires for `react` or `react-dom` to `preact-compat`. For example, in the case of Webpack, you just need to add the following configuration to `webpack.config.js`:

```
{
  "resolve": {
    "alias": {
      "react": "preact-compat",
      "react-dom": "preact-compat"
    }
  }
}
```

Conclusion

Preact is a nice alternative to React. Its community is growing steadily, and more web apps are using it. So if you're building a web app with high-performance requirements, or a mobile app for slow 2G networks, then you should consider Preact – either as the first candidate view library for your project, or as a drop-in

replacement for React.