

React

Speed Coding



Code
Along
Edition

ES6 Redux
Webpack Firebase
Enzyme Flexbox

reactspeed.com

Manav Sehgal

React Speed Coding

From concept to coding real world React apps, speedily!

Significantly revised and new code along edition. Speed up your React development workflow using Webpack. Learn to build a custom UI library in React, ES6, Flexbox, and PostCSS. Create single page app using Redux actions, reducers, and store. Apply new ES6 features to make your React code more reliable. Apply Behavior-Driven Development using Mocha, Chai, and Enzyme unit tests. Master the complete component design workflow with several strategies for reusable, reliable, and rapid coding in React.

Manav Sehgal

Manav Sehgal is a strategic thinker, self-starter, solving for market leading technology and business model challenges using design led innovation. Certified in Leading Innovative Change, from UC Berkeley, Haas School of Business. Manav has 20 years of experience in digital, cloud, e-commerce, mobile apps development and design with Global 1000 organizations. Manav has worked at Xerox PARC in the area of information visualization software research. He is currently working at a FinTech market leader. He has also set up and led digital design and innovation labs serving more than 100 organizations, across 200+ engagements, in 11 industries.

Manav is the author of four books in the areas of Java, JavaScript, React, E-commerce, and rapid development. His recent books include [React Speed Coding](#) and [React Eshop](#). He is also an active contributor to popular GitHub repositories in the React and Angular ecosystems. Manav is

among top most viewed writers on Quora for his answers on Firebase and React technologies.



This work is licensed under a [Creative Commons Attribution-NonCommercial3.0 3.0 Unported License](https://creativecommons.org/licenses/by-nc-nd/3.0/)

Shalini, Rama

Table of Contents

1. [Awesome React Ecosystem](#)
 1. [Code Along Edition](#)
 2. [Easy start React](#)
 3. [Who this book is for](#)
 4. [Development environment](#)
 5. [Why React is awesome](#)
 6. [Why read React Speed Coding](#)
 7. [Prior art](#)
 8. [Stakeholder perspectives on speed](#)
 9. [Who is using React](#)
 10. [Technology stack](#)
 11. [Measuring speed](#)
 12. [Why learn React comparing with Angular](#)
 13. [Shared learning path between Angular 2 and React](#)
2. [Setup React Webpack](#)
 1. [Installing starter dependencies](#)
 2. [Configure Webpack in webpack.config.js](#)
 3. [HTML webpack template for index.html](#)
 4. [Configuring startup scripts in package.json](#)
 5. [Run webpack setup](#)
3. [ES6 React Guide](#)
 1. [Hello World React](#)
 2. [Component composition and naming](#)
 3. [Files and folder structure](#)
 4. [Root component index.jsx](#)
 5. [Module import](#)
 6. [World.jsx component](#)
 7. [Class definition](#)
 8. [Constructor](#)
 9. [Event Handlers and setState](#)
 10. [JSX and the render method](#)

11. [Template literals and ternary conditionals](#)
12. [Properties](#)
13. [Event UI binding](#)
14. [Controlled components](#)
15. [PropTypes and defaultProps](#)
16. [ES7 Property Initializers](#)
17. [Complete World.jsx listing](#)
18. [Hello.jsx stateless component](#)
19. [Component styles in world.css](#)
20. [Base styles in element.css](#)
21. [Entry CSS using style.css](#)
22. [Run development server](#)
23. [React Chrome Extension](#)
4. [Production Optimize Webpack](#)
 1. [Add production plugins and supporting dependencies](#)
 2. [HTML minifying](#)
 3. [CSS minifying](#)
 4. [Webpack production webpack.prod.config.js](#)
 5. [Initialization](#)
 6. [Entry points](#)
 7. [Loaders](#)
 8. [PostCSS and plugins](#)
 9. [Plugins](#)
 10. [Profiling Webpack build](#)
5. [ReactSpeed UI](#)
 1. [ReactSpeed UI objectives](#)
 2. [PostCSS processing](#)
 3. [BEM CSS naming method](#)
 4. [Configurable theme using variables.css](#)
 5. [Theme definition in theme.css](#)
 6. [Updating elements.css](#)
 7. [Typography in type.css](#)
 8. [Utilities in util.css](#)
 9. [Image styles in image.css](#)
 10. [Flexbox layout in layout.css](#)

11. [Import styles in style.css](#)
12. [Define card component styles in card.css](#)
13. [Card.jsx component](#)
14. [Home.jsx landing_page component](#)
6. [Start Component Design](#)
 1. [Embed to React](#)
 2. [CSS libraries to React](#)
 3. [API to React](#)
7. [Define Component Internals](#)
 1. [Naming files, folders, and modules](#)
 2. [Imports and exports](#)
 3. [Stateless components and pure functions](#)
 4. [Classes and inheritance](#)
 5. [Constructor and binding](#)
 6. [Properties and property types](#)
 7. [State management](#)
 8. [Lifecycle methods](#)
 9. [Event handlers](#)
 10. [Render and ReactDOM.render methods](#)
 11. [JSX features and syntax](#)
 12. [Workflow.jsx component](#)
8. [Wire Multiple Components](#)
 1. [Render multiple components programmatically](#)
 2. [Composition using parent child node tree](#)
 3. [Presentational and container components](#)
 4. [Reconciliation algorithm and keys for dynamic children](#)
 5. [Integrating vendor components](#)
 6. [Routing to wire component layouts](#)
9. [Route Component Layouts](#)
 1. [Layout strategy](#)
 2. [AboutBook.jsx component](#)
 3. [AboutEmbeds.jsx component](#)
 4. [Configure routes in index.js](#)

5. [Nav.jsx component](#)
 6. [Navigation styles in nav.css](#)
 7. [Footer.jsx component](#)
 8. [Landing.jsx layout component](#)
 9. [NavLink component](#)
 10. [Programmatic routing in Ribbon.jsx](#)
 11. [Refactoring IconText.jsx for ribbon menu](#)
 12. [Search engine friendly URLs](#)
 13. [Handling router exceptions](#)
10. [Lint React Apps](#)
 1. [Browsersync multi-device testing](#)
 2. [Lint config in webpack.lint.config.js](#)
 3. [JavaScript lint using ESLint](#)
 4. [Configuring eslint in eslintrc.js](#)
 5. [Eslint command line interface](#)
 6. [Fixing ESLint reported problems](#)
 7. [ESLint webpack.lint.config.js integration](#)
 8. [StyleLint for CSS](#)
 9. [StyleLint CLI](#)
 10. [Fixing StyleLint reported problems](#)
 11. [Webpack integration for StyleLint](#)
 12. [Complete webpack.lint.config.js listing](#)
 11. [Test App Components](#)
 1. [Mocha Chai Behavior-Driven Development](#)
 2. [Test 01_mocha_timeout.spec.js](#)
 3. [Test 02_mocha_chai.spec.js](#)
 4. [Enzyme React component testing](#)
 5. [JSDOM browser.js helper](#)
 6. [Test 01_workflow.spec.js](#)
 7. [Sinon spy methods and events](#)
 8. [Istanbul code coverage](#)
 12. [Refactor Existing Components](#)
 1. [ES5 to ES6 conversion in TodoList.jsx](#)
 2. [Adding TodoApp to AboutCustom.jsx](#)

3. [Testing and refactoring](#)
4. [Refactoring for converting standard React apps to Redux](#)
5. [Refactoring for optimizing React apps](#)
6. [Refactoring Font Awesome to SVG icons](#)
7. [IconSvg.jsx custom component](#)
8. [Icon data fixture in icons.js](#)

13. [Redux State Container](#)

1. [The Roadmap app](#)
2. [Redux basics](#)
3. [State tree definition](#)
4. [Redux spec in 02_roadmap_redux.spec.js](#)
5. [Actions for Roadmap in /actions/roadmap.js](#)
6. [Reducers in /reducers/roadmap.js](#)
7. [Store in /store/roadmap.js](#)
8. [Test store, actions, and reducers](#)
9. [Optimize Redux app](#)
10. [Component hierarchy spec in 03_roadmap.spec.js](#)
11. [Rapid prototype hierarchy in roadmap.jsx](#)
12. [Data in fixtures/roadmap/features.js](#)
13. [Extract Feature.jsx component](#)
14. [Extract FeatureList.jsx component](#)
15. [Refactoring Roadmap.jsx app](#)

14. [Redux Wiring App](#)

1. [Extract CategoryButton.jsx component](#)
2. [Refactor data fixture features.js](#)
3. [Refactor actions in actions/roadmap.js](#)
4. [Extract SearchFeature.jsx component](#)
5. [Connecting Store in VisibleFeatureList.jsx](#)
6. [FilterCategoryButton.jsx container component](#)
7. [Refactor Roadmap.jsx](#)
8. [Hydrate Redux app using reducers/hydrate.js](#)
9. [Pass store in index.js](#)
10. [Update test suite 03_roadmap.spec.js](#)
11. [Update 02_roadmap_redux.spec.js](#)

15. [Firestore React Integration](#)

1. [Comparing Firestore with Meteor](#)
2. [Firestore Hosting](#)
3. [How Firestore stores files and data](#)
4. [Designing a REST API using Firestore](#)
5. [For what kind of apps is Firestore not ideal](#)
6. [Refactor Workflow component for Firestore integration](#)
7. [Data in fixtures/workflow/steps.json](#)
8. [Refactor Workflow.jsx](#)
9. [Firestore config data in fixtures/rsdb.js](#)
10. [Pass route param using index.js](#)
11. [Refactor AboutWorkflow.jsx](#)
12. [Create WorkflowFire.jsx](#)

16. [React Developer Experience](#)

1. [Redux DevTools](#)
2. [Katana Storybook](#)
3. [Create /.storybook/config.js](#)
4. [Create /app/stories/button.js](#)
5. [Create /.storybook/webpack.config.js](#)

Awesome React Ecosystem

Welcome reader. The aim of this book is to get you from concept to coding real world React apps, as fast as possible.

React ecosystem is constantly evolving and changing at a fast pace. This book equips you to take the right decisions matching your project requirements with best practices, optimized workflows, and powerful tooling.

Code Along Edition

We are significantly revising this edition of React Speed Coding book.

- Adding new [Code Along GitHub repository](#) containing branches for code you complete in each chapter.
- Chapter by chapter demos are available at [new demos](#) website.
- Making each chapter stand on its own so you can complete a significant learning step at the end of each chapter.
- Ensuring that your learning path is as linear as possible, without too many cross-references due to refactoring of code we write in each chapter.
- The new edition also features better typography, color coding, and more screenshots to aid your learning.
- Upgrading to the latest development environment and dependencies as of this writing.

- Adding new sections to make your React journey faster, easier, and better.
- Reducing the code you write to achieve the same goals.
- Redesigning the UI CSS using Block, Element, Modifier method for more scalable, yet less verbose design.
- Several new custom components for Buttons, Forms, Layout, and other features.

Easy start React

If you want to dig into React coding right away, you can fast forward first four chapters in this book.

Getting started with React is now easier than ever. Get your first React app up and running in three easy steps. You may want to read the sample chapter titled [Easy Start React](#) from our new book React Eshop. We use Facebook's **Create React App** scaffold generator to fast start into our first React app.

Go ahead give it a go. You will appreciate the magic behind the scenes as we learn to build a powerful development toolchain on similar lines as Facebook's Create React App in this book. This will result in two benefits for our readers. First, you will learn about important React ecosystem technologies including Webpack, PostCSS, ESLint, and Babel in a step-by-step linear manner. Second, you will be able to easily extend apps generated using Create React App as you mature beyond the scaffolded code and the opinionated defaults used by Create React App.

Who this book is for

The React Speed Coding book assumes basic knowledge of programming in JavaScript, HTML, and CSS.

If you are a complete beginner, there is enough guidance available for you to make this your first programming primer

with suggested additional reading.

Experienced web developers will master React component design workflow using latest ES6 language features. If you already program in React, you can use this book to optimize your development, testing, and production workflow.

Development environment

This book assumes you have access to a Mac, Linux, or a Cloud based editor offering virtual machine hosted development environment in your Web browser.

We will walk you through the entire React development environment setup from scratch.

While it is possible to run samples from this book using Windows, there are [known issues](#) in setting up Node and certain NPM packages.

As a safe alternative you can use any of the Cloud based code editors which offer Linux Development environment within the convenience of your web browser. You do not need to know how to use Linux to operate these web based editors. You can start with a generous free account with basic stack including Node.js already setup for you.

[Cloud9](#) is our favorite web based code editor. Other options include [Nitrous](#).

On Mac or Linux you can use your favorite code editor. This book is written using the open source [Atom](#) editor from the Github team. Atom gets you started coding by just dragging and dropping a folder onto the editor. You can then add power user features as you grow with Atom using custom packages, code snippets, among others.

Why React is awesome

Writing the React Speed Coding book, companion code, and the ReactSpeed.com demo website has been fun and fulfilling at the same time. Thanks to the amazing ecosystem that React and open source community have created in a relatively short span of time.

What we love about React and companion libraries like Redux is how they introduce constraints and flexibility at the same time. A very difficult goal to achieve when writing generic libraries and frameworks. React and Redux seem to have done so elegantly. Growing GitHub stars and cross-industry adoption is proof of this achievement.

To us React is about thinking in design and architectural patterns. It is more than making choices about which framework or library to use, or how to use these. We rapidly raise our thinking to design, requirements, solving real-world problems, that our apps are expected to address.

Learning React is about future proofing our investment more than any other framework or library with similar goals. Thanks to flexibility of integrating with React, even some of the competing frameworks offer integration paths with the React ecosystem. These include Meteor-React integration, Redux use cases with Angular, and TypeScript-React playing well together, just to name a few.

Most awesome aspect of learning React is that it is an ecosystem. It has a life of its own above and beyond Facebook, the original authors of React core. No wonder you see companies like Netflix, Airbnb, Kadir, Khan Academy, and Flipboard contributing their React libraries and tools to the open source.

Successfully navigating this growing ecosystem, making the right technology stack decisions along the way, will make the difference between an average programmer and a world-class designer-developer of the future. We sincerely hope React Speed Coding can contribute to your journey in mastering the React ecosystem of technologies.

Here's to moving from Concept to Code to Cash, speedily!

Why read React Speed Coding

React Speed Coding enables you to optimize your React development workflow and speed up the app design lifecycle.

Setup React Webpack development environment complete with Node and Babel including development, testing, and production workflows. Production optimize Webpack development toolchain for CSS, JS, HTML pre-processing, faster builds, more performant code.

Learn ES6 React features including arrow functions, template literals, variable scoping, immutability, pure functions, among others.

Create complete single page app using Redux store, actions, and reducers.

Create custom React Speed UI library using Flexbox and PostCSS, with goals including responsive design, single page app components, ease of customization, reusable code, and high performance.

Apply Behavior-Driven Development techniques to create a comprehensive testing strategy for your apps. This includes ESLint and StyleLint to provide in-editor coding guidance on industry best practices for JavaScript and CSS. Use Mocha to describe specs. Chai for writing assertions. Sinon to spy on methods and events. We also learn about Enzyme for simple yet powerful React component level testing.

Adopt a comprehensive component design workflow including five strategies for starting component design by creating React components from embeds, REST APIs, samples, and wireframes.

Integrate your apps with serverless architecture using Firebase hosting. Create REST API for component design workflow using Firebase visual tools. Connect custom React components you create in this book with Firebase realtime database.

Run demo app and components live at ReactSpeed.com website.

Visit our popular [GitHub repository](#) to download and reuse source code from this book.

Prior art

The author would love to take the credit for coining the term “Speed Coding”. However, Speed Coding is based on very strong foundations and popular prior art.

Speed of Developer Workflow. Speed Coding follows some of the methods and tools as prescribed by the Lean Startup principles. See [infographic](#) for code faster, measure faster, learn faster.

Code faster principles we cover in this book include writing unit tests, continuous integration, incremental deployment, leveraging open source, cloud computing, refactoring, just-in-time scalability, and developer sandboxing.

Measure faster principles include usability tests, real-time monitoring, and search engine marketing.

Learn faster principles we apply in this book include smoke tests, split testing, and rapid prototyping.

Speed of Design. Speed Coding embraces the **designer-developer** evolution and also bases certain principles on the [Design Thinking](#) methodology and [Visual Design](#) principles.

We use these techniques in designing React Speed UI library using custom React components, PostCSS, Flexbox, and SVG.

Speed of Technology Decisions. Speed Coding technology stack is compared with industry best practice guidance including the awesome [ThoughtWorks Technology Radar](#).

Technology Stack section, part of Introduction chapter, highlights React ecosystem technologies we cover in this book, in line with recommendations from the Radar.

The cover image for our book depicts NASA space shuttle lift-off and is representative of our central theme. The science of speed. We thank [Pixabay](#) for providing this NASA imagery in the Creative Commons.

Stakeholder perspectives on speed

In order to fulfill the promise of Speed Coding, we need to start by establishing some baselines. What are we speeding up? How are we measuring this speed? Why does it matter?

Let us start with the Why. Speed Coding is essential for three stakeholders. The user. The developer. The sponsor.

As app users we define *speed* mostly as performance and reactivity of the app. We even define speed as frequency of timely and desired updates to the apps we are using. Most importantly we define speed by time it takes to get things done.

As developers we define speed in terms of our development workflow. How long does it take to code, build, test, deploy, debug, and reactor. We also define speed of decision making relating to our development and technology stack.

As sponsors for an app project we define speed in terms of time to market. How long does it take to move from **Concept to Code to Cash**. Believe us, you first heard that phrase here, and we truly mean it!

Who is using React

Open sourced, developed, and used by Facebook and Instagram teams. React has also found wide adoption among leading technology, app, and digital companies.

- Airbnb are contributors of the popular React style guide.
- Atlassian redesigned their popular messaging app HipChat in React and Flux. They chose React because it is component based, declarative minus the bloat, uses Virtual DOM, relatively small library as opposed to full framework, simple, offers unidirectional data flow, and easily testable.
- BBC mobile homepage uses React.
- CloudFlare has active React projects on GitHub.
- Flipboard are authors of popular React Canvas.
- Khan Academy has several GitHub projects using React.
- Mapbox React Native module for creating custom maps.
- Netflix chooses React for startup speed of UI, runtime performance, and modularity it offers.
- Uber has several React GitHub projects including react-vis, a charting component library for React.

Technology stack

ThoughtWorks Technology Radar ranks technologies based on Adopt > Trial > Assess > Hold relative ranking. This is based on their own usage of these technologies across projects for leading enterprises globally. In terms of speed of decision making about your own technology stack, this is one tool that proves very helpful.

React Speed Coding will be addressing following technologies, platforms, techniques, frameworks, and tools.



© 2016, Manav Sehgal. ReactSpeed.com

ReactSpeed Technology Stack

ReactSpeed Technology Stack

ES6 (Adopt). JavaScript ECMAScript 6 is right at the top of the Radar list of languages and frameworks. We cover important concepts relevant for coding React in ES6.

React (Adopt). React is a close second on the Radar. Of course this book is all about React so we are well covered.

Redux and Flux (Trial). Redux is a new entrant on the Radar. We are dedicating an entire chapter and a relatively complex app for decent coverage of this important

technology in React ecosystem. Flux is an architectural style recommended for React. Redux evolves ideas of Flux, avoiding its complexity, according to the author of Redux.

React Native (Trial). Another entry high on the Radar from React ecosystem. We are covering Flexbox which is one of the key technologies in React Native stack. Of course React and Redux make up the mix.

GraphQL (Assess). Another up and coming technology in React stack. GraphQL is an alternative to REST protocol. Goes hand in hand with Relay, another technology from the Facebook camp.

Immutable.js (Assess). Yet another Facebook open source project. Goes well with Redux.

Recharts (Assess). Integrates D3 charts and React. We will implement samples using Rumble Charts, a popular alternative, in this book.

Browsersync (Trial). Browsersync is a great time-saver for multi-devices testing of mobile-web hybrid apps. React Speed Coding implements Browsersync + Webpack + Hot Reloading. So if you make any changes in your JSX, these should update on all devices on saving the changes. While maintaining your current UI state. Isn't this awesome!

GitUp (Trial). Graphical tool complementing Git workflow. We are find this tool useful for going back in time and revising commit logs for instance.

Webpack (Trial). We are implementing your React developer workflow using Webpack. Two chapters are dedicated to get you started with Webpack and help you production optimize the workflow.

Serverless Architecture (Assess). We are implementing serverless architecture using Firebase. Another technology worth evaluating is AWS Lambda, though it may not be in scope for this book.

Measuring speed

So it will be nice to define some baseline measurements of speed and see if we can improve these as we go through the book.

Website Performance. Google PageSpeed defines 25+ criteria for website performance as relative measures or percentile scores compared with rest of the Web. For example *Enable Gzip Compression* is 88% as recommended baseline.

As on May 9, 2016 the ReactSpeed.com website is evaluating grade A (93% average) on PageSpeed score, grade B (82% average) on YSlow score, 2.1s page load time, 834KB total page size, with 26 requests. View [GTMetrix ReactSpeed.com report here](#).

As on Aug 9, 2016 while the React Speed app has more than 3,000 lines of code, we improve our page load time to 1.3s, 438KB page size, and only 13 requests back to the servers. Our page speed score improves to 95%, and YSlow score is 86%.

Load Impact (Radar Trial). Online load testing tool. We are using this tool to perform concurrent user load tests on ReactSpeed.com website.

As on May 9, 2016 with 20+ custom React components live on ReactSpeed website, we are recording faster than 200ms load time for our website for 25 concurrent users. That translates to handling approximately 2,50,000 monthly visitors. Excellent! [View results snapshot here](#).

Build and Deploy Time. How long does it take to run the developer workflow.

As on May 9, 2016 our development server continuously builds and updates our app as we save our working code. Production build takes 5.3s with around 250 hidden modules. We deploy 44 files to Firebase several times during a day.

Time to Release. How long does it take to ship new features to production.

Since start of ReactSpeed project we have closed 150 production commits to GitHub over a 30 day period. Our peak is 40 commits during week of April 10.

Production Payload. How optimized are our production assets.

As on May 9, 2016 our CSS library is 4.7KB Gzip, 21KB minified with 25+ style modules. App JS bundle is 42KB minified. Vendor JS bundle is 192KB minified. HTML is 3KB.

Time to fix issues. How long does it take to fix issues in code. Code issues can be of several types including compliance with coding guidelines and best practices, logical bugs, usability issues, performance issues, among others.

As on May 10, 2016 it took us 6 hours to resolve 300+ issues down to 3 open issues using ESLint integration with Atom editor and Webpack. The issues ranged from coding best practices to refactoring requirements as per React coding patterns.

Continuos production build workflow. This can be measured by number of commands or developer actions required to complete one production ready build. Alternatively how automated is this lifecycle.

NPM for all the things. Can be measured based on number of project dependencies that are updated from NPM

or popular managed repositories and CDNs. This is Trial stage at ThoughtWorks Technology Radar.

Static code analytics. This can be measured for number of lint warnings or errors. Complexity analysis of JavaScript code can be included apart from other static code analytics.

Why learn React comparing with Angular

One of the most important decisions modern app developers make is choosing the right front-end framework for their technology stack. Most popular question relates to choosing React over Angular 2.

We recognize that React is at its core a simpler library representing the View pattern, while Angular 2 is a complete framework offering Model, View, and Controller architectural patterns.

React is actually an ecosystem of well designed and highly popular libraries, open sourced mostly by Facebook (React, React Native, Immutable, Flow, Relay, GraphQL), other leading developers (Redux, React Material UI, React Router), and industry leaders (Flipboard React Canvas, Airbnb Enzyme).

React + Redux for instance offer the Model (Store), View (React, Actions), and Controller (Reducers) pattern to compare on equal grounds with the Angular 2 stack.

Why learn Angular 2? It is like learning Yoga, from one Guru, in a large group.

If you are in a **large** team, Angular will be your choice to get everyone on the same page, faster, at scale.

Contrary to popular opinion on the subject, we think Angular 2 is **faster** to learn when compared to React for the same goals, simply because you are making fewer “first-time-learner or developer” decisions along your journey.

- Angular2 and TypeScript are opinionated,

- Most documentation is “single version of truth” from one source (Google, the authors of Angular and Microsoft, the authors of TypeScript),
- There is mostly “one Angular/TypeScript way” of doing things, so fewer decisions to make along your learning and development journey,
- Angular 2 API and TypeScript language are well documented.
- Official samples are up to date with latest changes in the API, well mostly.
- Development boilerplates or starters are fairly mature, some like Angular Universal are backed by Google/Angular core team.
- The development and build tool-chain is mostly addressed by Angular/TypeScript, and few popular starter projects.

Why learn React? It is like doing cross-training, with multiple experts, at multiple locations.

If you are in a **lean** team or a single developer-designer-architect, “the React way” may be more fun. Learning React is more **rewarding** in the long run. As you are making “hopefully informed” decisions all along your learning and development journey, you become a more thorough developer, designer, architect in the long haul. If you make your decisions by evaluating pros-cons of architectural and design patterns, you are becoming a better developer.

React is opinionated for fewer core concepts like one-way-data-binding and offers sensible workarounds even for that.

- There are many ways to develop in React starting from how to define React components, how to create React build pipeline, which frameworks to integrate with, how to wire up a backend, the list goes on.

- You will learn from multiple sources and authors, not just Facebook/Instagram, the authors of React. Having these multiple perspectives will give you stronger real-world decision making muscles!
- Facebook and the React community is very driven by app performance patterns. Most core React concepts are centered around creating high performance code.
- The React community is more component driven. You will most likely find more reusable code.
- The community is also increasingly driven by “Developer Experience”. Writing beautiful code, writing more manageable code, writing readable code, and tools that make developer’s experience more fun and visual. See [kadirahq/react-storybook](https://kadirahq.com/react-storybook) as an example.

In our experience learning both and going back and forth helps. Programming design patterns remain the same. Syntactical sugar changes. Learning one, reinforces the other.

Shared learning path between Angular 2 and React

JavaScript. JavaScript (ES5 and ES6) is fundamental. TypeScript transpiles to JavaScript. React-JSX-Babel tooling transpiles to JavaScript.

CSS3. You cannot do serious front-end coding without it.

HTML5. It is obvious, but extend your knowledge on concepts like Offline Storage and Device Access, best practice starters like HTML5 boilerplate.

Webpack. Modern day packaging, module bundling, build pipeline automation tooling.

Design Patterns and Object Oriented principles. Composition, Inheritance, Singletons, Pure Functions, Immutability, and many others are core concepts helping you in doing good development in general.

Algorithms and data structures. Serious development cannot be done without using these in a good measure.

Backend as a Service. Firebase, AWS Lambda, among others.

Microservices and REST/APIs. No modern app is built in isolation these days.

So, here is a learning path if you want to go beyond React. Learn React first, build some reusable components, learn the component design workflow. Learn Angular 2 next, try reusing your component design and above mentioned shared learning here. Maybe round off your knowledge by learning Meteor (more opinionated with best practice

patterns for speed coding and performance) and integrating React and Angular, replacing Meteor's Blaze.

Setup React Webpack

You will learn in this chapter how to setup React development environment starting from scratch. By the end of this chapter we will have a starter boilerplate to develop React apps.

We will cover following topics in this chapter.

- How to install Node.js and use Node Version Manager
- Setup package.json to manage your NPM dependencies
- Quick access companion code for this book using Github
- Install starter dependencies for React, Webpack, and Babel
- Create Webpack configuration for development pipeline automation
- Write a simple React app to run your Webpack setup

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
```

```
git checkout -b c01 origin/c01-setup-react-webpack  
npm install  
npm start
```

Installing Node.js

You will need Node.js to get started with React. Your Mac OS comes with Node pre-installed. However you may want to use the latest stable release.

Check you node release using `node -v` command.

We recommend installing or upgrading Node using Node Version Manager (NVM). Their [Github repo](#) documents install and usage.

To install NVM:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.0/install.sh | bash
```

Now you can install a Node release by choosing one from Node [releases](#) page.

The command `nvm install 5.10.1` installs a stable release for us.

One of the advantages of using NVM is you can switch between multiple node releases you may have on your system.

Here's what our terminal looks like when using `nvm ls` to list installed node releases.

```
v4.2.3
v5.3.0
v5.10.0
->      v5.10.1
system
default -> 5.3.0 (-> v5.3.0)
node -> stable (-> v5.10.1) (default)
stable -> 5.10 (-> v5.10.1) (default)
iojs -> iojs- (-> system) (default)
```

Using `nvm use x.y.z` command we can switch to `x.y.z` installed node release.

Setting up package.json

You will require `package.json` to manage your NPM dependencies and scripts.

Create a new one using `npm init` command, selecting defaults where uncertain.

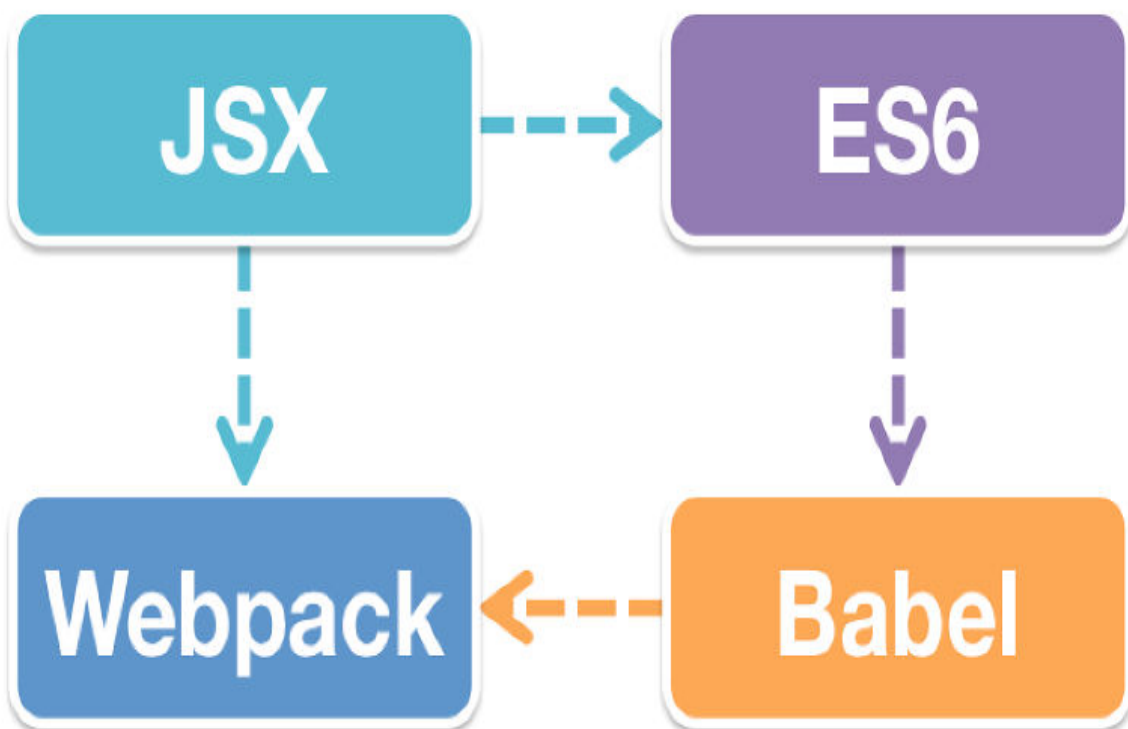
This is what our `package.json` looks like as we start off. Note that we added the `private` flag to avoid accidental publishing of the project to NPM repo, and also to stop any warnings for missing flags like `project repo`.

```
{
  "name": "react-speed-book",
  "version": "1.0.0",
  "description": "Companion code for React Speed Coding book",
  "main": "index.js",
  "private": true,
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/manavseghal/react-speed-book.git"
  },
  "author": "Manav Sehgal",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/manavseghal/react-speed-book/issues"
  },
  "homepage": "https://github.com/manavseghal/react-speed-book#readme"
}
```

The dependencies section will start showing up as we add npm dependencies.

Installing starter dependencies

Before we start writing our React app we need to install starter dependencies for our development environment. React uses JSX as the XML-like syntax extension over JavaScript to specify component tree structure, data flow, and event handlers. JSX is processed by Webpack module bundler using specific loaders or convertors.



React Dependency Stack

React recommends JavaScript ES6 for latest features and best practices. ES6 needs Babel for compiling to ES5 and maintain browser compatibility. Babel integrates with Webpack to stitch it all together for our app.

React and React DOM

React is available via NPM and this is the recommended way of using React in a project. React core is available in the react

package. The `react-dom` package targets browser DOM for rendering React. React enables several targets including iOS, Android for rendering React apps.

```
npm install --save react
npm install --save react-dom
```

Webpack

Webpack is used for module packaging, development, and production pipeline automation. We will use `webpack-dev-server` during development and `webpack` to create production builds.

```
npm install --save-dev webpack
npm install --save-dev webpack-dev-server
```

HTML generation

You can add functionality to Webpack using plugins. We will use automatic HTML generation plugins for creating `index.html` for our app.

The `html-webpack-plugin` webpack plugin will refer to template configured within webpack configuration to generate our `index.html` file.

```
npm install --save-dev html-webpack-plugin
```

Loaders

Webpack requires loaders to process specific file types. CSS loader resolves `@import` interpreting these as `require()` statements. Style loader turns CSS into JS modules that inject `<style>` tags. JSON loader enables us to import local JSON files when using data fixtures during prototyping our app.

```
npm install --save-dev css-loader
npm install --save-dev style-loader
npm install --save-dev json-loader
```

PostCSS and Normalize CSS

PostCSS loader adds CSS post-processing capabilities to our app. This includes Sass like capabilities including CSS variables and mixins using `precss` and automatic vendor prefixes using the `autoprefixer` plugin for PostCSS.

Normalize CSS offers sensible resets for most use cases. Most CSS frameworks include it.

The `postcss-easy-import` plugin enables processing `@import` statements with rules defined in webpack configuration.

```
npm install --save-dev postcss-loader
npm install --save-dev precss
npm install --save-dev autoprefixer
npm install --save-dev normalize.css
npm install --save-dev postcss-easy-import
```

Babel

Babel compiles React JSX and ES6 to ES5 JavaScript. We need `babel-loader` as Webpack Babel loader for JSX file types.

Hot loading using `babel-preset-react-hmre` makes your browser update automatically when there are changes to code, without losing current state of your app.

ES6 support requires `babel-preset-es2015` Babel preset.

```
npm install --save-dev babel-core
npm install --save-dev babel-loader
npm install --save-dev babel-preset-es2015
npm install --save-dev babel-preset-react
npm install --save-dev babel-preset-react-hmre
```

Presets in Babel

Presets in Babel are a collection of plugins as npm dependencies which help transform source to target code using Babel. For [full list of such plugins](#) visit Babel's website.

Configure Babel `.babelrc`

Babel configuration is specified in `.babelrc` file. React Hot Loading is required only during development.

```
{
  "presets": ["react", "es2015"],
  "env": {
    "development": {
      "presets": ["react-hmre"]
    }
  }
}
```


Dependencies in package.json

The dependencies within package.json now lists all the installed dependencies for our app so far.

```
"dependencies": {  
  "react": "^15.3.0",  
  "react-dom": "^15.3.0"  
},  
"devDependencies": {  
  "autoprefixer": "^6.4.0",  
  "babel-core": "^6.13.2",  
  "babel-loader": "^6.2.4",  
  "babel-preset-es2015": "^6.13.2",  
  "babel-preset-react": "^6.11.1",  
  "babel-preset-react-hmre": "^1.1.1",  
  "copy-webpack-plugin": "^3.0.1",  
  "css-loader": "^0.23.1",  
  "html-webpack-plugin": "^2.22.0",  
  "json-loader": "^0.5.4",  
  "normalize.css": "^4.2.0",  
  "postcss-easy-import": "^1.0.1",  
  "postcss-loader": "^0.9.1",  
  "precss": "^1.4.0",  
  "style-loader": "^0.13.1",  
  "webpack": "^1.13.1",  
  "webpack-dev-server": "^1.14.1"  
}
```

Configure Webpack in webpack.config.js

Webpack configuration drives your development pipeline, so this is a really important file to understand. We will split various sections of the config file to aid step-by-step learning.

Initialization

To start off, you need to initialize the config file with dependencies. These include webpack itself, an HTML generation plugin, a webpack plugin to copy folders from development to build target, and Node path library for initializing default paths.

Next we initialize the default paths. We also configure defaults for our HOST and PORT configuration.

```
// Initialization
const webpack = require('webpack');

// File ops
const HtmlWebpackPlugin = require('html-webpack-plugin');

// Folder ops
const CopyWebpackPlugin = require('copy-webpack-plugin');
const path = require('path');

// PostCSS support
const postcssImport = require('postcss-easy-import');
const precss = require('precss');
const autoprefixer = require('autoprefixer');

// Constants
const APP = path.join(__dirname, 'app');
const BUILD = path.join(__dirname, 'build');
const STYLE = path.join(__dirname, 'app/style.css');
const PUBLIC = path.join(__dirname, 'app/public');
const TEMPLATE = path.join(__dirname, 'app/templates/index.html');
const NODE_MODULES = path.join(__dirname, 'node_modules');
const HOST = process.env.HOST || 'localhost';
const PORT = process.env.PORT || 8080;
```

- APP path is used to specify the app entry point, location of the root component

- BUILD path specifies the target folder where production compiled app files will be pushed
- STYLES path indicates the root CSS file that imports other styles
- PUBLIC path is a folder in development that is copied as-is to root of BUILD, used for storing web server root files like robots.txt and favicon.ico, among others
- TEMPLATE specifies the template used by html-webpack-plugin to generate the index.html file
- NODE_MODULES path is required when including assets directly from installed NPM packages

Entry points and extensions

Next section defines your app entry, build output, and the extensions which will resolve automatically.

```
module.exports = {
  // Paths and extensions
  entry: {
    app: APP,
    style: STYLE
  },
  output: {
    path: BUILD,
    filename: '[name].js'
  },
  resolve: {
    extensions: ['', '.js', '.jsx', '.css']
  },
}
```

Loaders

We follow this by defining the loaders for processing various file types used within our app.

React components will use .jsx extension. The React JSX and ES6 is compiled by Babel to ES5 using the babel webpack loader. We use the cacheDirectory parameter to improve repeat development compilation time.

The include key-value pair indicates where webpack will look for these files.

CSS processing is piped among `style` for injecting CSS style tag, `css` for processing import paths, and `postcss` for using various plugins on the CSS itself.

We need JSON processing much later in the book to enable loading of local JSON files storing data fixtures (sample data) for our app.

```
module: {
  loaders: [
    {
      test: /\.jsx?$/,
      loaders: ['babel?cacheDirectory'],
      include: APP
    },
    {
      test: /\.css$/,
      loaders: ['style', 'css', 'postcss'],
      include: [APP, NODE_MODULES]
    },
    {
      test: /\.json$/,
      loader: 'json',
      include: [APP, NODE_MODULES]
    }
  ]
},
```

PostCSS plugins configuration

PostCSS requires further configuration to handle how PostCSS plugins behave.

Now we add `postcssImport` which enables Webpack build to process `@import` even from `node_modules` directory directly.

Sequence matters as this is the order of execution for the PostCSS plugins. Process imports > compile Sass like features to CSS > add vendor prefixes.

```
// Configure PostCSS plugins
postcss: function processPostcss(webpack) { // eslint-disable-line no\
-shadow
  return [
    postcssImport({
      addDependencyTo: webpack
    }),
    precss,
    autoprefixer({ browsers: ['last 2 versions'] })
  ]
}
```

```
    1;  
  },
```

Configure webpack-dev-server

Now that we have loaders configured, let us add settings for our development server.

Source maps are used for debugging information.

The devServer settings are picked up by webpack-dev-server as it runs.

The historyApiFallback enables local browser to be able to handle direct access to route URLs in single page apps.

The hot flag enables hot reloading. Using the inline flag a small webpack-dev-server client entry is added to the bundle which refresh the page on change.

The progress and stats flags indicate how webpack reports compilation progress and errors.

```
// Source maps used for debugging information  
devtool: 'eval-source-map',  
// webpack-dev-server configuration  
devServer: {  
  historyApiFallback: true,  
  hot: true,  
  progress: true,  
  
  stats: 'errors-only',  
  
  host: HOST,  
  port: PORT,  
  
  // CopyWebpackPlugin: This is required for webpack-dev-server.  
  // The path should be an absolute path to your build destination.  
  outputPath: BUILD  
},
```

Plugins

We now wrap up by adding plugins needed during our development.

This section specifies the compilation workflow. First we use the DefinePlugin to define the NODE_ENV variable.

We then activate the Hot Reloading plugin to refresh browser with any app changes.

Then we generate any HTML as configured in the HtmlWebpackPlugin plugin.

```
plugins: [  
  // Required to inject NODE_ENV within React app.  
  new webpack.DefinePlugin({  
    'process.env': {  
      'NODE_ENV': JSON.stringify('development') // eslint-disable-line  
    }  
  },  
  new webpack.HotModuleReplacementPlugin(),  
  new CopyWebpackPlugin([  
    { from: PUBLIC, to: BUILD }  
  ],  
  {  
    ignore: [  
      // Doesn't copy Mac storage system files  
      '.DS_Store'  
    ]  
  })  
  ),  
  new HtmlWebpackPlugin({  
    template: TEMPLATE,  
    // JS placed at the bottom of the body element  
    inject: 'body'  
  })  
];
```

HTML webpack template for index.html

We can now add a custom template to generate index.html using the HtmlWebpackPlugin plugin.

This enables us to add viewport tag to support mobile responsive scaling of our app. We also add icons for mobile devices. Following best practices from HTML5 Boilerplate, we add html5shiv to handle IE9 and upgrade warning for IE8 users.

```
<!DOCTYPE html>  
<html class="no-js" lang="">  
  <head>  
    <meta charset="utf-8">
```

```

<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
<title>React Speed Coding</title>
<meta name="description" content="">
<meta name="viewport" content="width=device-width, initial-scale=1\
">

<link rel="icon" href="/favicon.ico" />

<!--[if lt IE 9]>
  <script src="//html5shiv.googlecode.com/svn/trunk/html5.js"></\
script>
  <script>window.html5 || document.write('<script src="js/html5s\
hiv.js"></script>')</script>
<![endif]-->
</head>
<body>
  <!--[if lt IE 8]>
    <p class="browserupgrade">You are using an <strong>outdated</s\
trong> browser. Please <a href="http://browsehappy.com/">upgrade your \
browser</a> to improve your experience.</p>
  <![endif]-->

  <div id="app"></div>
  <!-- Google Analytics: change UA-XXXXX-X to be your site's ID. -->
</body>
</html>

```

Configuring startup scripts in package.json

We can configure startup scripts in package.json to speed up our development even further.

Before we do that, let us create a copy of webpack.config.js as webpack.prod.config.js intended for production build version of our webpack configuration. We will only make one change in the production configuration. Changing NODE_ENV in plugins section from development to production value.

Now npm start will run webpack-dev-server using inline mode which works well with hot reloading browser when app changes without manually refreshing the browser.

Running npm run build will use webpack to create our production compiled build.

```

"scripts": {
  "start": "NODE_ENV=development webpack-dev-server --inline",
  "build": "NODE_ENV=production webpack --config webpack.prod.config.js"
},

```

The webpack-dev-server will pick up the webpack.config.js file configuration by default.

Run webpack setup

index.js

Add index.js in the root of our development folder.

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

ReactDOM.render(
  <h1>App running in {process.env.NODE_ENV} mode</h1>,
  document.getElementById('app')
);
```

We will learn each aspect of a React component in the next chapter. For right now we are writing a minimal React app to test our Webpack configuration.

style.css

We also import the Normalize CSS library in our main style entry file.

```
@import 'normalize.css';
```

public/ folder

To test how our public folder is copied over to build, let us add favicon.ico file to public folder in root of our development folder.

Running webpack

We can now run the development server using npm start command in the Terminal. Now open your browser to localhost URL that your webpack dev server suggests.

The browser app displays message that you are running in development mode. Try changing the message text and hit save. Your browser will refresh automatically.

You can also build your app to serve it using any web server.

First add a file server like so.

```
npm install -g serve
```

Now build and serve.

```
npm run build  
serve build
```

As you open your localhost URL with port suggested by the file server, you will note that the message now displays that you are running in production mode.

The `build` folder lists following files created in our webpack build.

<code>app.js</code>	<== App entry point
<code>favicon.ico</code>	<== Copied as-is from <code>public/</code> folder
<code>index.html</code>	<== Generated by HTML webpack plugin
<code>style.js</code>	<== Contains our styles

Congratulations... You just built one of the most modern development environments on the planet!

ES6 React Guide

This chapter guides you through important React concepts and ES6 features for speeding up your React learning and development journey.

You will learn following concepts in this chapter.

- How to define a React component using ES6 syntax
- What are modules and how to import and export React components
- Why we need constructors
- How components talk to each other and the UI using events, props, and state
- Importance of stateless components
- Using React Chrome Extension to inspect your component hierarchy at runtime

For Speed Coding in React, ES6 is essential. Not only does it reduce the amount of code you end up writing, ES6 also introduces language patterns for making your app better designed, more stable, and performant.

Let us create a Hello World app to understand the React and ES6 features together. Subsequent chapters introduce more React and ES6 features based on the context of samples written for the chapter. This spreads your learning journey as you apply these concepts.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c02 origin/c02-es6-react-guide
npm install
npm start
```

Hello World React

We will write a Hello World app with three custom components. A component to contain our app, a Hello component to render the greeting message, and a World component to handle user events like changing greeting or adding a name.



Hello World app in your browser

Component composition and naming

React apps follow component based development. So understanding how components are composed, how they relate to each other, simplifies several aspects of your React

learning path. This includes understanding how React app folders and files are organized.



Ownership vs. Parent-Child

According to Facebook React docs, “It’s important to draw a distinction between the owner-ownee relationship and the parent-child relationship. The owner-ownee relationship is specific to React, while the parent-child relationship is simply the one you know and love from the DOM.”

Component Hierarchy. React has owner components which render or set properties on other components. This ownership continues across all components building a tree-like hierarchy.

Root Component. In case of our Hello World ap, `index.jsx` represents the root component (one that does not have an owner). The root component owns and renders `World` component. The `World` component is owner of `Hello` component.

Component File Naming. Root component inside a folder is named `index.jsx` and the component name takes on the name of the folder, `app` in this case. Components other than root are named same as the component class names, including PascalCase. Refer [naming conventions](#) in Airbnb style guide.

Files and folder structure

This is the files and folder hierarchy you will create by the end of this chapter.

.babelrc	<== Babel configuration
package.json	<== NPM configuration
webpack.config.js	<== Webpack development configuration
webpack.prod.config.js	<== Webpack production configuration
app/	<== React app root
index.jsx	<== Root app component, app entry point
style.css	<== Style entry point @import from styles/
styles/	<== Style partials
base/	
elements.css	<== Base HTML element styles
components/	
world.css	<== World component specific styles
public/	<== Copied as-is to root of build folder
favicon.ico	
components/	<== Components folder
Hello.jsx	<== Custom component
World.jsx	<== Custom component
templates/	
index.html	<== Template used by HTML webpack plugin
build/	<== Webpack generated build folder
node_modules/	<== NPM dependencies installed here

Root component index.jsx

We start by writing the entry point to our HelloWorld React app. We replace the `index.js` created in the last chapter with a new one we start from scratch.

Module import

In React each component is typically defined in a single file, also known as a module. Any dependencies are imported using `import` statement. The statement specifies name of the exported component, constant, or function from the dependencies.

Dependencies are then used within a component. Dependencies are also used as rendered components within an owner component. Like in our case `World` component is imported by `app` root component before it is rendered.

```
import React from 'react';
import ReactDOM from 'react-dom';
import World from './components/World.jsx';
```

```
ReactDOM.render(
  <World />,
  document.getElementById('app')
);
```

Thinking in modules is central to how Webpack bundles your code and traces dependencies while creating chunks. However, Webpack 1.x does not natively support ES6 modules, though this is on their 2.x roadmap. This is where Babel steps in. Read more about [Webpack ES6 support](#) on the Webpack official docs.

The `ReactDOM.render` method uses React library targeting DOM output to identify a DOM element id as target for rendering our entire app component hierarchy, starting with the `World` component.

World.jsx component

Now we write the `World` component which renders the `Hello` component with a message and handles user inputs.

Class definition

We start by importing `Hello` component which we will render in `World` component.

A module exports a component using `export default` keywords. There can only be one such export. While importing such components we do not need to use the `{ }` braces.

All React components extend `React.Component`. This enables our components to have access to React lifecycle methods and other features.

```
import React from 'react';
import Hello from './Hello.jsx';

export default class World extends React.Component {
```


Constructor

The constructor of our `World` component highlights three most important features of how components talk to each other and the user.

State. Changing UI or internal state of a component is maintained using `this.state` object. When state changes, rendered markup is updated by re-invoking the `render()` method.

Props. Properties are the mechanism to pass data from owner to rendered components.

Events. Methods or event handlers are bound to UI events (like `onClick`) to perform actions when the event takes place.

Constructor is called when component is created, so it is the right place for the following three objectives.

1. Using `super(props)` keyword to make the props object available to `React.Component` methods.
2. Setting initial component state. Using `this.state` object. One can set state to default values or properties passed from owner component accessing props object.
3. Binding event handlers to `this` context of the component class. Using `bind(this)` method.

```
constructor(props) {  
  super(props);  
  this.state = {  
    currentGreeting: props.greet,  
    value: 'ReactSpeed'  
  };  
  this.slangGreet = this.slangGreet.bind(this);  
  this.hindiGreet = this.hindiGreet.bind(this);  
  this.handleNameChange = this.handleNameChange.bind(this);  
}
```

Event Handlers and setState

Components can exist with no events defined. In our HelloWorld app we define three events.

The slangGreet event handles user click on Slang link to change the greeting message accordingly. Likewise the hindiGreet does so with the Hindi greeting message.

The handleNameChange method handles user input in the input text box to change the name of greeting sender in the message. The `value` state is available to React controlled components as explained few sections later in this chapter.

We use `setState` method to change state within React components. This in turn re-renders the component UI calling the `render` method.

```
slangGreet() {  
  this.setState({ currentGreeting: 'Yo!' });  
}  
  
hindiGreet() {  
  this.setState({ currentGreeting: 'Namaste' });  
}  
  
handleNameChange(event) {  
  this.setState({ value: event.target.value });  
}
```

JSX and the render method

Let us now write the render method for World component. React component is rendered using JSX syntax. JSX is HTML-like syntax where the nodes are actually native React components providing same functionality as the HTML DOM equivalents. So `<div>`, `<h2>`, `<a>`, and others used in the JSX code are all native React components.

```
render() {
  const renderGreeting = this.state.value
    ? `${this.state.value} says ${this.state.currentGreeting}`
    : this.state.currentGreeting;
  return (
    <div className="World-card">
      <Hello greet={renderGreeting} message="World!" />
      <h2>
        <a onClick={this.slangGreet}>Slang</a>
        &nbsp;OR&nbsp;
        <a onClick={this.hindiGreet}>Hindi</a>
      </h2>
      <input
        type="text" value={this.state.value}
        placeholder="Enter a name"
        onChange={this.handleNameChange}
      />
    </div>
  );
}
```

In the next four sections we will break down the render method JSX to understand more ES6 React concepts.

Template literals and ternary conditionals

We are using ES6 [template literals](#) as strings with back-ticks. Template literals allow embedded expressions using the `${expression}` syntax.

We are also using the JavaScript ternary conditional expression to set the JSX value of `renderGreeting` based on `value state`.

```
const renderGreeting = this.state.value
  ? `${this.state.value} says ${this.state.currentGreeting}`
  : this.state.currentGreeting;
```

Properties

Properties are used for passing input data from Root/Owner component to Child. Owner defines a property=value which is used within rendered component. The “Owner-ownee” relationship can exist without property passing, as in owner simply rendering a component.

Treat props as immutable. You can use props to set state within event handlers using `this.setState` method or state definition in constructor using `this.state` object. Props, state, and event bindings can be defined in the constructor.

```
<Hello greet={renderGreeting} message="World!" />
```

Event UI binding

Event UI binding is done by passing a prop named after the event `onClick` in this case and the bound event handler method.

```
<a onClick={this.slangGreet}>Slang</a>
&nbsp;OR&nbsp;
<a onClick={this.hindiGreet}>Hindi</a>
```

Controlled components

Current UI state changes as user clicks on greeting language links. We are also processing input data using `this.state.value` provided by React. The input control used in this example does not maintain its own state. It is known as *Controlled Component* as opposed to *Uncontrolled Component* if the `value` property is not used. Uncontrolled components manage their own state. Read more about [handling forms](#) at Facebook React documentation.

```
<input
  type="text" value={this.state.value}
  placeholder="Enter a name"
  onChange={this.handleChange}
/>
```

PropTypes and defaultProps

At the end of the class definition we define default properties and property types on the component constructor.

The `propTypes` are used in property validation during development to throw warnings in the Browser's JavaScript console, if your code is not meeting the validation criteria. Read more on [Prop Validation](#) in Facebook post that lists various types, including custom validations.

```
World.propTypes = {  
  greet: React.PropTypes.string.isRequired  
};
```

```
World.defaultProps = {  
  greet: 'Hello'  
};
```

So if you change the value of `greet` to any number, the app will run, however you will see following warning in your browser console.

Warning: Failed propType: Invalid prop
expected string.

`greet` of type `number` supplied to `World`,

ES7 Property Initializers

[Property initializers](#) are an ES7 Stage 1 proposed feature.

In the prior section we were using class properties by defining the `World.propTypes` and `World.defaultProps` outside of the class definition on the component constructor.

Now using `babel-plugin-transform-class-properties` we can bring these within the class definition.

Install the Babel plugin supporting this ES7 transform.

```
npm install --save-dev babel-plugin-transform-class-properties
```

Update `.babelrc` with this new plugin.

```
{
  "presets": ["react", "es2015"],
  "env": {
    "development": {
      "presets": ["react-hmre"]
    }
  },
  "plugins": ["transform-class-properties"]
}
```

Next we update our class definition like so.

```
export default class World extends React.Component {
  static propTypes = {
    greet: React.PropTypes.string.isRequired
  }

  static defaultProps = {
    greet: 'Hello'
  }

  constructor(props) {
```

Note the use of `static` statement before initializing `propTypes` and `defaultProps` in our class definition.

Complete World.jsx listing

We can further reduce few lines of code from our World component by importing PropTypes and Component from React core.

Here is the complete World component listing.

```
import React, { PropTypes, Component } from 'react';
import Hello from './Hello.jsx';

export default class World extends Component {
  static propTypes = {
    greet: PropTypes.string.isRequired
  }

  static defaultProps = {
    greet: 'Hello'
  }

  constructor(props) {
    super(props);
    this.state = {
      currentGreeting: props.greet,
      value: 'ReactSpeed'
    };
    this.slangGreet = this.slangGreet.bind(this);
    this.hindiGreet = this.hindiGreet.bind(this);
    this.handleNameChange = this.handleNameChange.bind(this);
  }

  slangGreet() {
    this.setState({ currentGreeting: 'Yo!' });
  }

  hindiGreet() {
    this.setState({ currentGreeting: 'Namaste' });
  }

  handleNameChange(event) {
    this.setState({ value: event.target.value });
  }

  render() {
    const renderGreeting = this.state.value
      ? `${this.state.value} says ${this.state.currentGreeting}`
      : this.state.currentGreeting;
    return (
      <div className="World-card">
        <Hello greet={renderGreeting} message="World!" />
        <h2>
          <a onClick={this.slangGreet}>Slang</a>
          &nbsp;OR&nbsp;
          <a onClick={this.hindiGreet}>Hindi</a>
        </h2>
        <input
```

```
        type="text" value={this.state.value}
        placeholder="Enter a name"
        onChange={this.handleChange}
      />
    </div>
  );
}
```


Hello.jsx stateless component

The `Hello` component renders Hello World message based on how `World` component calls it and current UI state.

Our `Hello` component is stateless. It does not define or change any state.



Why stateless components are important

According to Facebook, “In an ideal world, most of your components would be stateless functions because in the future we’ll also be able to make performance optimizations specific to these components by avoiding unnecessary checks and memory allocations. This is the recommended pattern, when possible.”

We are using [arrow functions](#) which are ES6 shorthand for writing functions.

```
import React from 'react';

const Hello = ({ greet, message }) => (
  <h1>{greet} {message}</h1>
);

Hello.propTypes = {
  greet: React.PropTypes.string.isRequired,
  message: React.PropTypes.string.isRequired
};

export default Hello;
```

Component styles in world.css

We can add some styles to our app. The `/app/styles/components/world.css` path is used to create style for our World component.

```
.World-card {  
  background: white;  
  border: 1px solid lightgrey;  
  border-radius: 3px;  
  box-shadow: 1px 1px 1px 0 darkgrey;  
  padding: 0.8em 1em;  
  width: 300px;  
  margin: 10px;  
  text-align: center;  
}
```

Base styles in element.css

We also add some shared styles in `/app/styles/base/element.css` file.

```
html {  
  height: 100%;  
  color: black;  
  font-family: 'Open Sans', sans-serif;  
  font-size: 18px;  
  font-weight: 400;  
}  
  
h1 {  
  font-weight: 300;  
  font-size: 2em;  
}  
  
h2 {  
  font-size: 1.333em;  
  font-weight: 400;  
}  
  
a {  
  color: steelblue;  
  text-decoration: none;  
}  
  
a:focus,  
a:hover {  
  border-bottom: 1px solid steelblue;  
  cursor: pointer;  
}
```

Entry CSS using style.css

Finally we include our base and component styles, along with Normalize resets from the NPM module we installed earlier in this chapter.

```
@import 'normalize.css';  
@import 'styles/base/elements';  
@import 'styles/components/world';
```

The CSS entry file is stored at the `/app/style.css` path.

Run development server

This completes our first React app. Now run the app using the development server.

```
npm start
```

Once the app runs you should see following message from webpack-dev-server in your terminal window.

```
> react-speed-book@1.0.0 start ...
> NODE_ENV=development webpack-dev-server

http://localhost:8080/
webpack result is served from /
content is served from ...
404s will fallback to /index.html
Child html-webpack-plugin for "index.html":

webpack: bundle is now VALID.
```

Browse to your app on the url mentioned in webpack output. Now try changing some code like the style background or the Hello World message and hit save. Your browser should update the app without refreshing state.

When this hot loading update happens you will see following output in the browser console.

```
[HMR] App is up to date.
[React Transform HMR] Patching Hello
[HMR] Updated modules:
[HMR]   - 379
[HMR] App is up to date.
```

Now we build and serve our HelloWorld app.

```
npm run build
serve build
```

You will see a different webpack output on your terminal this time.

```
[ReactSpeed] react-speed-book: $ npm run build

> react-speed-book@1.0.0 build ../react-speed-book
> NODE_ENV=production webpack --config webpack.prod.config.js

Hash: 98bcd5d896e89e0c987a
Version: webpack 1.13.1
```

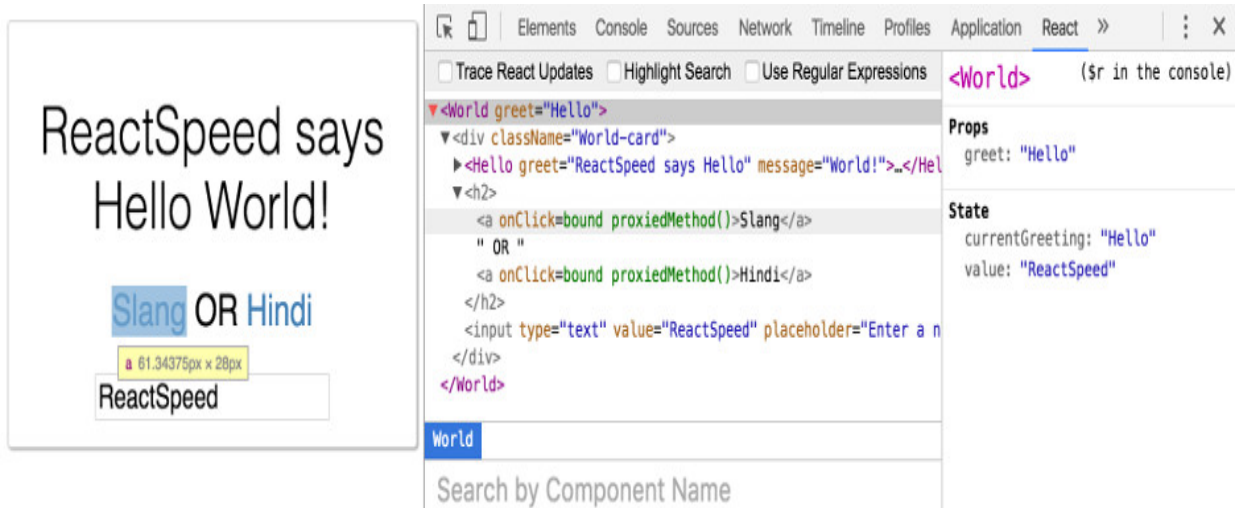
```
Time: 2534ms
  Asset      Size  Chunks             Chunk Names
  app.js    1.85 MB          0  [emitted]  app
  style.js   74.1 kB          1  [emitted]  style
  favicon.ico 1.15 kB          [emitted]
  index.html 1.03 kB          [emitted]
    + 171 hidden modules
Child html-webpack-plugin for "index.html":
    + 3 hidden modules
```

```
[ReactSpeed] react-speed-book: $ serve build
serving ../react-speed-book/build on port 3000
GET / 200 8ms - 1.01kb
GET / 304 2ms
GET /style.js 200 4ms - 72.35kb
GET /app.js 200 69ms - 1.85mb
GET / 304 1ms
GET /style.js 304 0ms
GET /app.js 304 1ms
```

As you may have noticed the build is not highly optimized. The app.js file is a huge ~2MB and css turned into JavaScript! In the chapter **Production Optimize Webpack** we will discuss various techniques to optimize for a production environment.

React Chrome Extension

In case you want to inspect how your components pass properties and how they are organized at runtime, you can install [React Chrome Extension](#).



React Chrome Extension in your browser

You can then select the code responsible for component UI and see the rendered UI highlighted. The extension will also update properties as they are passed along if you turn on the trace feature.

Production Optimize Webpack

If you recall from the **ES6 React Guide** chapter, our build size is approaching 2MB. This is not viable for a small Hello World app. In this chapter we will optimize our Webpack configuration for production use case.

We will cover following topics in this chapter.

- Optimize React code for production bundle.
- Separate CSS for static or CDN serving.
- Bundle dependencies separately.
- Minify JavaScript code.
- Profiling Webpack generated build.
- Adding public assets for your app.
- Creating custom index template.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c03 origin/c03-production-optimize-webpack
```

```
npm install  
npm start
```


Add production plugins and supporting dependencies

For production following plugins add functionality to Webpack. The `clean-webpack-plugin` helps clear build folder before each build. The `extract-text-webpack-plugin` extracts CSS and bundles it into a single file to reduce server requests, enable CDN serving, and browser caching.

```
npm install --save-dev clean-webpack-plugin
npm install --save-dev extract-text-webpack-plugin
```

HTML minifying

The `html-webpack-plugin` recommends using `html-minifier` to minify HTML.

```
npm install --save-dev html-minifier
```

CSS minifying

Our build already takes care of CSS minifying. The `css-loader` uses `cssnano` as a dependency which in turn uses `postcss-discard-comments` plugin to minify CSS output.

Read more about configuring the `cssnano` within `css-loader` [here](#).

The `cssnano` minifying options are documented [here](#).

Webpack production webpack.prod.config.js

We have already created a production version of webpack.prod.config.js so that we can pass this to webpack in the npm run build script.

Initialization

Initialization section adds the plugins for cleaning build directory for repeat builds and extracting CSS for separate file in production.

We also parse the package.json dependencies for creating separate vendor and manifest bundles.

```
// Initialization
const webpack = require('webpack');

// File ops
const HtmlWebpackPlugin = require('html-webpack-plugin');
const ExtractTextPlugin = require('extract-text-webpack-plugin');

// Folder ops
const CleanPlugin = require('clean-webpack-plugin');
const CopyWebpackPlugin = require('copy-webpack-plugin');
const path = require('path');

// PostCSS support
const postcssImport = require('postcss-easy-import');
const precss = require('precss');
const autoprefixer = require('autoprefixer');

// Constants
const APP = path.join(__dirname, 'app');
const BUILD = path.join(__dirname, 'build');
const STYLE = path.join(__dirname, 'app/style.css');
const PUBLIC = path.join(__dirname, 'app/public');
const TEMPLATE = path.join(__dirname, 'app/templates/index.html');
const NODE_MODULES = path.join(__dirname, 'node_modules');

const PACKAGE = Object.keys(
  require('./package.json').dependencies
);
```

Entry points

Our next section is similar to development config. For production use case it adds chunkhash to file names. This optimizes browser cache and CDN storage/retrieval as only

updated files are downloaded as you apply patches to your production code.

```
module.exports = {
  entry: {
    app: APP,
    style: STYLE,
    vendor: PACKAGE
  },
  resolve: {
    extensions: ['', '.js', '.jsx', '.css']
  },
  output: {
    path: BUILD,
    filename: '[name].[chunkhash].js',
    chunkFilename: '[chunkhash].js',
    publicPath: '/'
  },
}
```

Loaders

We continue using Babel loader for JSX files. For CSS we use extract text plugin so we can generate separate CSS file. This helps in browser caching as well as avoids FOUC or Flash Of Unstyled Content during initial app loading within the browser.

```
module: {
  loaders: [
    {
      test: /\.jsx?$/,
      loaders: ['babel?cacheDirectory'],
      include: APP
    },
    // Extract CSS during build
    {
      test: /\.css$/,
      loader: ExtractTextPlugin.extract('style', 'css!postcss'),
      include: [APP, NODE_MODULES]
    },
    // Process JSON data fixtures
    {
      test: /\.json$/,
      loader: 'json',
      include: [APP, NODE_MODULES]
    }
  ]
},
```

PostCSS and plugins

Next section defining PostCSS plugins is same as development webpack config.

```
// Configure PostCSS plugins
postcss: function processPostcss(webpack) { // eslint-disable-line no\
-shadow
  return [
    postcssImport({
      addDependencyTo: webpack
    }),
    precss,
    autoprefixer({ browsers: ['last 2 versions'] })
  ];
},
```

Plugins

In the next and final section on adding plugins is where our production pipeline actually kicks in.

Clean build. We start by cleaning our build directory of any past builds.

Optimize HTML. Then we continue with HTML webpack plugin to generate `index.html`. We add HTML minifying capability.

Extract CSS. Next the CSS is extracted as a separate file.

Optionally Dedupe. Larger projects may have dependency trees with duplicate files. Dedupe plugin removes such duplications. This option is commented out in the starter project as it has few dependencies and has no affect on the build size.

Separate JS bundles. Vendor and manifest JavaScript files are bundled separately. This also helps in browser/CDN caching as you may only do a release where one of them is updated, the other does not need to be downloaded again to the client.

React for production. Facebook recommends React production settings to optimize the React library payload.

Minify JS. Finally we minify the JavaScript.

```

// Remove comment if you require sourcemaps for your production code
// devtool: 'cheap-module-source-map',
plugins: [
  // Required to inject NODE_ENV within React app.
  // Redundant package.json script entry does not do that, but require\
d for .babelrc
  // Optimizes React for use in production mode
  new webpack.DefinePlugin({
    'process.env': {
      'NODE_ENV': JSON.stringify('production') // eslint-disable-line \
quote-props
    }
  }),
  // Clean build directory
  new CleanPlugin([BUILD]),
  new CopyWebpackPlugin([
    { from: PUBLIC, to: BUILD }
  ],
  {
    ignore: [
      // Doesn't copy Mac storage system files
      '.DS_Store'
    ]
  })
),
// Auto generate index.html
new HtmlWebpackPlugin({
  template: TEMPLATE,
  // JS placed at the bottom of the body element
  inject: 'body',
  // Use html-minifier
  minify: {
    collapseWhitespace: true
  }
}),

// Extract CSS to a separate file
new ExtractTextPlugin('[name].[chunkhash].css'),

// Remove comment to dedupe duplicating dependencies for larger proj\
ects
// new webpack.optimize.DedupePlugin(),

// Separate vendor and manifest files
new webpack.optimize.CommonsChunkPlugin({
  names: ['vendor', 'manifest']
}),

// Minify JavaScript
new webpack.optimize.UglifyJsPlugin({
  compress: {
    warnings: false
  }
})
});

```

Now we are ready to run the build using `npm run build` command.

npm run build terminal output

```
[ReactSpeed] react-speed-book: $ npm run build

> react-speed-book@1.0.0 build .../react-speed-book
> NODE_ENV=production webpack --config webpack.prod.config.js

clean-webpack-plugin: .../react-speed-book/build has been removed.
Hash: b97369be6954f5bcf05b
Version: webpack 1.13.1
Time: 3460ms
```

Names	Asset	Size	Chunks	Chunk \
app.769127474b5d32ae9832.js	2.83 kB	0, 3	[emitted]	app
style.86216d360cef01a5f839.js	38 bytes	1, 3	[emitted]	style
vendor.e8fb09ea31e2e586b857.js	146 kB	2, 3	[emitted]	vendor
manifest.6e7f9bc045eff3678cef.js	783 bytes	3	[emitted]	manife\
st				
style.86216d360cef01a5f839.css	2.68 kB	1, 3	[emitted]	style
favicon.ico	1.15 kB		[emitted]	
index.html	1.22 kB		[emitted]	
[0] multi vendor 40 bytes {2} [built]				
+ 171 hidden modules				
Child html-webpack-plugin for "index.html":				
+ 3 hidden modules				
Child extract-text-webpack-plugin:				
+ 2 hidden modules				

You will notice in the terminal window that the overall size of your build is reduced significantly when compared with the development release. The total size of optimized build is ~150 KB when compared with ~1,900 KB of development build. That is nearly 90% reduction in size!

Your app also benefits from caching (separate files) and much smaller initial load time payload.

Wow! That is a lot of work. Fortunately you will not change this pipeline too often. Worthwhile as a one-off investment as you build your React app to conquer the world!

Profiling Webpack build

Webpack has developed a powerful tool to profile and analyze your build for further optimization.

Add following run script to package.json file.

```
"profile": "NODE_ENV=production webpack --config webpack.prod.config.js --profile --json > profile.json"
```

Now npm run profile to generate profile.json file with data on your build. You can pass this file to [webpack analysis tool](#) for analyzing your build.

webpack.github.io/analyse/#module/38



36

./~/react/lib/ReactDOMInjection.js

time

257ms finished @ 795ms

size

4 KiB

flags

built

chunks

2

issuer

32

./~/react/lib/ReactDOM.js

reasons

type	module	user request	location
cjs require	32 ./~/react/lib/ReactDOM.js	./ReactDOMInjection	17:28-62

dependencies

type	module	user request	location
cjs require	37 ./~/react/lib/BeforeInputEventPlugin.js	./BeforeInputEventPlugin	14:29-64

Webpack module analysis

Using the analysis tool you can drill down into warnings, errors, hints on how to improve your code, and analyze module chunks to further optimize your code.

ReactSpeed UI

Now that we are comfortable with basic React, ES6 concepts and have a production ready Webpack build pipeline, it is time to do some serious app development.

In this chapter we will start designing our very own ReactSpeed UI component library for your apps. We will do so using Flexbox, PostCSS, and custom React components.

You will learn following concepts in this chapter.

- ReactSpeed UI objectives
- PostCSS Processing
- BEM CSS naming method
- Configurable themes
- Flexbox layouts
- Card custom component
- Home custom layout component

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c04 origin/c04-react-speed-ui
npm install
npm start
```

ReactSpeed UI objectives

Designing a custom UI library is an ambitious undertaking. Our task becomes achievable if we scope our design goals upfront.

Speed. Like the name suggests, our UI library is built for speed of development and creating performant apps.

Single Page App. We will design various UI components required for a single page app including landing page, buttons, forms, navigation menu, interactive content cards, and footer.

Responsive. Our app will be responsive and components will render according to target screen size.

Customizable. We want our UI library to be easily customizable using custom color themes.

Reusable. The ReactSpeed UI library will be reusable across multiple apps.

Simple. We will keep our UI library simple to understand, extend, and reuse.

Optimized. ReactSpeed UI library will be production ready and optimized for light payloads.

Expressive. Our UI library will be reusable in expressive, English like statements.

PostCSS processing

To help us along in these design goals, PostCSS offers some great features.

As a simple analogy, [PostCSS](#) does for CSS what Babel does for JS.

Autoprefixer. You can write latest CSS rules and PostCSS Autoprefixer will add vendor prefixes based on current browser popularity and support. Your post-processed CSS always stays up-to-date with latest browser compatibility.

CSSnext. Future-proofing your code, use latest CSS features today. PostCSS converts to CSS current browsers support.

CSS Modules. Make your CSS code more modular, reusable, and maintainable.

Speed. PostCSS is faster than Less, Sass, others, when it concerns your development pipeline. See [benchmarks here](#).

Plugins. PostCSS ecosystem has several plugins including Global CSS fixes, CSS readability, future CSS, images and fonts, linters, and syntaxes similar to SCSS, SASS, LESS. See [list of plugins](#) here. You can search plugins in a catalog at [postcss.parts](#).

We use following plugins for building Speed UI.

- The [Autoprefixer plugin](#), so that we don't need to worry about browser prefixes anymore.
- The [PreCSS plugin](#) enables Sass-like markup within CSS.

BEM CSS naming method

Large CSS implementations are infamous for difficulty in maintainability. You can do four things to make your CSS code as readable, reusable, and robust as your React code.

1. Follow component oriented design principles we follow in our React code
2. Use naming conventions based on well established methodologies
3. Incorporate CSS lint tests
4. Use CSS post processing automation in your build pipeline

While we have added PostCSS and other CSS optimization tools to our build pipeline, we will add lint tools in the chapter **Test App Components**.

This chapter focuses on 1 and 2. Following component oriented design principles. Developing our style methodology.

Detailed CSS organization and naming best practices include [SMACSS](#), [BEM](#), and [OOCSS](#), among many others.

We will use Block Element Modifier or BEM CSS naming methodology when creating React Speed UI. Popular websites like Shopify use BEM method to make CSS naming relatively easy and manageable for thousands of designers.

BEM follows a simple method which can be described in just one selector `block__element--modifier` example. We can relate blocks to React components, elements to component children, and modifiers to props which may modify block or element look and feel.

An example is `card__title--active` selector name which conveys style for Card component or block, title child node or element, when active prop or modifier is set to true.

Of course one can have `block--modifier` use cases, however always prefix element with a block to avoid namespace collisions in larger projects.

The rationale for double dash (-) is to distinguish among single dash use case for splitting multi-world selector names like we see in CSS property names.

Configurable theme using variables.css

Let us start organizing our styles. We will extend the styles folder we created in prior chapters to add more base styles. First, let us create a `variables.css` file to specify reusable theme variables. That way our css can refer to these variables. Changes in one file will impact our entire UI library. We are using Sass-like variables in our CSS thanks to PostCSS plugins which will process these variables for us.

```
/* Dark color palette */
$default: slategray;
$primary: steelblue;
$secondary: teal;
$danger: crimson;
$warning: tomato;
$success: mediumseagreen;

/* Light color palette */
$default-light: color(whitesmoke tint(50%));
$primary-light: color(lightsteelblue tint(70%));
$secondary-light: color(lightcyan tint(50%));
$danger-light: lightpink;
$warning-light: moccasin;
$success-light: honeydew;
```

As you can see we have also started using Sass-like calculations within CSS. PostCSS will process these and convert to normal CSS.

Theme definition in theme.css

Using the variables we just defined, we can now start creating our theme defining background and text styles based on our palette.

```
/* Light background theme. */
.back--default { background-color: $default-light; }
.back--primary { background-color: $primary-light; }
.back--secondary { background-color: $secondary-light; }
.back--danger { background-color: $danger-light; }
.back--success { background-color: $success-light; }
.back--warning { background-color: $warning-light; }
.back--white { background-color: white; }

/* Dark background theme */
.back--gray { background-color: slategray; }

/* Light text theme */
.default { color: $default-light; }
.primary { color: $primary-light; }
.secondary { color: $secondary-light; }
.danger { color: $danger-light; }
.success { color: $success-light; }
.warning { color: $warning-light; }
.white { color: white; }
```


Updating elements.css

We add custom styles for `input` element. We also make sure that any color values are now referring to our configurable theme variables.

```
input {
  border: 1px solid color($default-light shade(10%));
  padding: 0.3em 0.5em;
  transition: background 0.5s;
}

input:focus,
input:hover {
  outline: none;
  text-decoration: none;
  background: color(white shade(2%));
}
```

Typography in type.css

Now let us define the base typography for React Speed UI. For now we are only defining one custom selector for handling subtext to our headings.

```
.subtext {
  font-weight: 300;
  font-size: 1.5em;
  color: $default;
}
```

Utilities in util.css

We also define some utility selectors.

```
.text--center { text-align: center; }
.text--left { text-align: left; }
.text--right { text-align: right; }
```

Image styles in image.css

We define `image.css` within `/app/styles/base/` folder for now. Once we see the need to create an `Image` component, we can move the file to `/app/styles/components` folder.

```
.image__link {
  transition: 0.5s opacity;
}
```

```
.image__link:hover {  
  opacity: 0.8;  
  border-bottom: 0;  
}
```

Flexbox layout in layout.css

Flexbox takes care of some of the most important concerns in designing UI - layouts, alignment, and grids.

Flexbox is a CSS standard. It is also much less code to create layouts using Flexbox, as the task of calculating grid size, placement, alignment, number of fitting elements is left to the browser, instead of the designer. You can say Flexbox is more “developer friendly” in that sense.

Flexbox also lends itself very well to thinking in components and composition. It has a concept of containers and contained elements. Similar to React way of containers and presentational components our owners and children.

Let us understand Flexbox as defined for the `.landing` container in three easy concepts.

Display. First statement sets the display to `flex` instead of `block` or other options.

Flow. Next `flex-flow` sets the flow of contained elements by row or column, wrapping or nowrap, forward direction or reverse.

Justify. Finally, `justify-content` decides how contained elements will be placed in relation to each other. Options include space around, align with start of container, align with end, center, and space in between.

Also note that Flexbox styling only needs to be specified within the container.

```
/* Striped page layout */  
.landing {  
  display: flex;  
  flex-flow: column;
```

```
    justify-content: flex-start;
}
```

Let us continue defining our landing page layout by adding styles for horizontal striped sections.

Note that we are using `@extend` to use inheritance with CSS styles. PostCSS will compile this to standard CSS.

```
/* Stripes are horizontal sections */
.stripe {
    display: flex;
    flex-flow: row wrap;
    justify-content: space-around;
    padding: 40px 0;
}

.stripe--slim {
    @extend .stripe;
    padding: 10px 0;
}
```

We now define our column grid. Using verbose percentage size based column grid is better than using fixed number of columns based grid.

- You can mix and match multiple sizes in one horizontal section
- You do not need to ensure that all sizes add up to 100%, Flexbox takes care of the gaps
- As sizes are percentages, the grid is inherently responsive based on screen size

```
/* Column grid */
.col--fifth { flex: 0 0 20%; }
.col--quarter { flex: 0 0 25%; }
.col--one-third { flex: 0 0 33.3333%; }
.col--half { flex: 0 0 50%; }
.col--two-thirds { flex: 0 0 66.6666%; }
.col--three-quarters { flex: 0 0 75%; }
.col--full { flex: 0 0 100%; }
```

Let us wrap up our layout theme for now by defining simple navigation styles. We are taking care of main navigation bar styling using the `nav` block. Menu links are styled using `nav__link` block element. Brand link is a special case of menu link, which we are addressing using `@extend` inheritance.

```
/* nav */
.nav {
  display: flex;
  flex-flow: row nowrap;
  justify-content: flex-start;
  list-style: none;
  background: black;
  margin: 0;
  padding: 10px;
}

.nav__link {
  text-decoration: none;
  border-bottom: none !important;
  color: white;
  font-size: 16px;
  padding-left: 10px;
  transition: 0.5s color;
}

.nav__link:hover,
.nav__link:active {
  color: color(white shade(20%));
}

.nav__brand {
  @extend .nav__link;
  font-weight: bold;
}
```

Import styles in style.css

Now all that is left is to include all the partials or CSS modules we just created within `style.css`.

```
@import 'normalize.css';
@import 'styles/base/variables';
@import 'styles/base/elements';
@import 'styles/base/util';
@import 'styles/base/theme';
@import 'styles/base/type';
@import 'styles/base/layout';
@import 'styles/base/image';
@import 'styles/components/card';
```

With this we have defined our generic styles for Speed UI. These styles will apply across components. For component specific styles we follow the same principles as we learnt here.

- Create a partial or CSS module named after the component.
- Reuse variables from base theme.
- Inherit from other selectors where it makes sense.
- Import partial in `style.css`.
- Design the component using the styles as HTML elements or using `className`.

Note that while we are importing our components/card styles we have not defined these yet.

Define card component styles in card.css

Now we replace the `world.css` styles with more appropriate `/app/styles/components/card.css` styles.

```
/* Plain card */
.card {
  padding: 1em;
  margin: 0.25em;
}

/* Box card */
.card--box {
  @extend .card;
  border: 1px solid lightgrey;
  border-radius: 1px;
  box-shadow: 0 2px 4px lightgrey;
}
```

We also make a minor change in the World component, removing the `className` prop in the root `div` component. We are doing this as styles will now be handled by the Card component.

Card.jsx component

Card component will provide a bounding box to its children. Only in cases where we pass a prop `plain` we do not draw the bounding box.

You will note that the props we pass to Card component are interesting. They include `className`, `onClick`, and `children` as explicit props. We need to do this to pass on these props to the components that the Card renders.

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

const Card = (props) => {
  const boxStyle = props.plain ? 'card' : 'card--box';
  const cardStyle = props.className
    ? `${boxStyle} ${props.className}`
    : boxStyle;
  return (
    <div onClick={props.onClick} className={cardStyle}>
      {props.children}
    </div>
  );
};
```

```
Card.propTypes = {  
  className: PropTypes.string,  
  children: PropTypes.node,  
  onClick: PropTypes.func,  
  plain: PropTypes.bool  
};
```

```
Card.defaultProps = {  
  className: '',  
  children: null,  
  onClick: null,  
  plain: false  
}
```

```
export default Card;
```


Home.jsx landing page component

Now we use many of the styles defined in our UI library to create our home landing page. We start by importing the World and Card component which we will use within our landing page.

Next let us create our top navigation bar simply using React native component `nav` for now. We style the component using Flexbox styled `nav` selector from our `layout.css`. Links are also styled accordingly.

```
import React, { PropTypes, Component } from 'react';
import World from './World';
import Card from './Card';

export default class Home extends Component {
  render() {
    return(
      <section className="landing">
        <nav className="nav">
          <a className="nav__brand" href="/">
            ReactSpeed
          </a>
          <a className="nav__link"
            href="https://leanpub.com/reactspeedcoding">
            Book
          </a>
          <a className="nav__link"
            href="https://github.com/manavsehal/react-speed-book">
            Code
          </a>
          <a className="nav__link"
            href="https://manavsehal.github.io/react-speed-demos/">
            Demos
          </a>
          <a className="nav__link" href="https://reactspeed.com">
            Website
          </a>
        </nav>
```

Next we create section to be styled as `stripe` as defined in our layout. We use our grid column styles to layout two Card components responsively. Note how we intermix custom React components using our custom styles and native React components.

This intermixing is good iterative and rapid design method. We maximize utility of native React components, unless we see benefits in composing multiple native components into one custom component. We may also consider creating a custom component as a specialization of a native component when we are extending the native functionality.

```
<section className="stripe">
  <Card plain className="text--center">
    <a href="https://leanpub.com/reactspeedcoding"
      className="image__link">
      
    </a>
  </Card>
  <Card plain className="col--half text--center">
    <h1>Develop Awesome Apps</h1>
    <p className="subtext">
      Join 100s of readers learning
      latest React ES6 concepts.
    </p>
  </Card>
</section>
```

We wrap up our Home component by adding two more sections.

```
<section className="stripe back--default">
  <Card plain className="col--half text--center">
    <h1>Custom React Component</h1>
    <p className="subtext">
      This custom component demonstrates props, state,
      and ES6 classes.
    </p>
  </Card>
  <Card className="col--one-third text--center back--white">
    <World />
  </Card>
</section>
<section className="stripe--slim back--gray">
  <Card plain className="col--full text--center white">
    <p>
      Copyright (c) 2016, Manav Sehgal.
      All rights reserved.
    </p>
  </Card>
</section>
</section>
);}}
```

As we compile and run our app, we notice how our new UI library impacts the look and feel of our app, while our build still contains one bundled css.



Develop Awesome Apps

Join 100s of readers learning latest
React ES6 concepts.

Custom React Component

This custom component demonstrates
props, state, and ES6 classes.

ReactSpeed says
Hello World!

[Slang](#) OR [Hindi](#)

ReactSpeed

Start Component Design

If the blank, dark code editor window is staring back at you, do not despair. We will run through several ways you can get started building your components in React!

angular/angular

★ 14862 stars

🔥 1175 issues

facebook/react

★ 47217 stars

🔥 649 issues

meteor/meteor

★ 35049 stars

🔥 1037 issues



Nine Component Creation Strategies

📁 Modular Architecture

☁ Leverages Cloud

⚙ 30 Custom Components

🎯 Goal Oriented Design

Infographic Components

Custom React components easily reusable to create variety of Infographic elements.



Embed React Components

This custom component demonstrates media embed within custom React component.

Rapid prototyping custom components

The objective at this stage is to speedily “prototype” new features and code within your React project. Subsequent chapters will go over best practices to create performant, reusable, and maintainable code.

Here is what you will learn in this chapter along with sample code.

- Using Web Embeds like YouTube, Flickr, Twitter, to create React components.

- Converting CSS libraries into React components.
- Integrating third-party REST APIs to start creating React components.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c05 origin/c05-start-component-design
npm install
npm start
```

Embed to React

Many web platforms like Youtube, Flickr, and Twitter offer embed APIs to easily integrate their platform features into your app.

Strategy for converting embed code to React component is straightforward.

1. Customize the embed code within target platform to suit your site or app styles and placement
2. Optionally, parametrize the embed code attributes using React props
3. Use stateless component as you will most likely not maintain embed UI state locally within your React component. Embed code will handle its own UI state

Let us create a reusable component which takes a YouTube video id and renders a YouTube video embed.

Step 1: This is what a custom YouTube embed code looks like. We have reduced size to fit our Card component and customized controls, and other attributes on YouTube platform.

```
<iframe width="560" height="315" src="https://www.youtube.com/embed/MG\uKhcnrqGA?rel=0&controls=0&showinfo=0"
frameborder="0" allowfullscreen></iframe>
```

Step 2: Note that the video id is part of URL following /embed/ part. This turns into our only property we pass on to the component.

YouTube.jsx custom component

Step 3: We create a stateless component using pure function definition. The component takes one property videoid and renders the embed code.


```
import React from 'react';

const YouTube = ({ videoId }) => (
  <iframe
    className="youtube"
    width="560"
    height="315"
    src={`https://www.youtube.com/embed/${videoId}?
    rel=0&controls=0&showinfo=0`}
    frameBorder="0"
    allowFullScreen
  >
  </iframe>
);

YouTube.propTypes
  = { videoId: React.PropTypes.string };

export default YouTube;
```

Add Home.jsx section to render YouTube

Now all we need to do is render this new component within our Home, just like we did for the World component.

```
<section className="stripe">
  <Card plain className="col--one-third text--center back--white">
    <YouTube videoId="MGuKhcnrqGA" />
  </Card>
  <Card plain className="col--half text--center">
    <h1>Embed React Components</h1>
    <p className="subtext">
      This custom component demonstrates media
      embed within custom React component.
    </p>
  </Card>
</section>
```

The component is reusable, so if we want to add multiple videos we just render more components and pass a different videoId.

Our app remains relatively responsive as we reduce the screen size (or resize browser), we notice the description heading and subtext slides below our YouTube component.

CSS libraries to React

There are some very good CSS libraries and frameworks available which can speed up our React development.

Normally you would bring these CSS libraries into your React project using one of the following ways.

- Use `<link rel="stylesheet" href="//url/to/library.min.css"/>` to reference the library if it is hosted on a CDN.
- Add a popular library as an NPM dependency and `@import` it within `style.css`.
- Add a library that does not change frequently as a partial and `@import` it within `style.css`.

Next you would refer to the library classes within your HTML code.

Using React components you can further speed up your usage of such libraries. If the CSS libraries themselves have UI components or controls, these can be represented as React components. This enables you to create shortcuts to using the library code. You can even do custom processing, apply your own styles, before rendering the CSS library code.

We follow this strategy to convert CSS libraries to React components.

1. Decide how to import or integrate the library within your React app.
2. Identify component(s) from within the library to determine equivalent React components.
3. Optionally, parametrize any style attributes and repeating elements within the CSS library as React

component properties.

4. Reuse the React component in place of CSS library elements within your app.

We can follow this strategy to convert many popular CSS libraries to React components. For this sample, we have chosen [Font Awesome](#), one of the most popular iconic font and CSS toolkit.

Add CSS library to index.html template

Step 1: As Font Awesome is a popular library and is available over a world class CDN (BootstrapCDN by MaxCDN), easiest way to integrate this CSS library into our app is to add it to `/app/templates/index.html` template.

This has the added benefit of browser caching for performance. Client browsers that already have Font Awesome in their cache, do not need to download it again when they load our app. Another benefit is that we are always using the latest version of the CSS library without having to update our app.

```
<link rel="stylesheet"
      href="//maxcdn.bootstrapcdn.com/font-awesome/4.6.3/css/font-awesome.\
min.css">
```

IconText.jsx custom component

Step 2: Next we identify which elements of Font Awesome library are candidates for converting to React components. This is a relatively straightforward decision as Font Awesome is all about icons. So we can create a reusable React component called Icon. Let us go a step further into the requirements of our app. We would like to display stats and infographics in our cards, so our icons will also have an associated text message. So let us call our React Component IconText and move on to the next step.

Step 3: Now we identify repeating elements and configurable attributes of the CSS library, in order to turn these into our component properties. Font Awesome icons can be configured using a rich combination of classes to determine size, orientation, icon graphic, among other features. All these attributes are candidates for converting to React component properties.

Let us see how our new React component is shaping up.

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

const IconText = ({
  icon, text, className,
  size, rotate, flip,
  inverse, slim }) => {
  let variation = '';

  variation += size ? ` fa-${size}` : '';
  variation += rotate ? ` fa-rotate-${rotate}` : '';
  variation += flip ? ` fa-flip-${flip}` : '';
  variation += inverse ? ` fa-inverse` : '';

  const iconClass = `fa fa-${icon}${variation}`;

  return (
    slim
    ? <div className={className}>
      <i className={iconClass}></i> {text}
    </div>
    : <div className={className}>
      <i className={iconClass}></i>
      <h1>{text}</h1>
    </div>
  );
};

IconText.propTypes = {
  icon: PropTypes.string.isRequired,
  text: PropTypes.string.isRequired,
  className: PropTypes.string,
  size: PropTypes.oneOf(['lg', '2x', '3x', '4x', '5x']),
  rotate: PropTypes.number,
  flip: PropTypes.oneOf(['horizontal', 'vertical']),
  inverse: PropTypes.bool,
  slim: PropTypes.bool // draw slim single-line InfoText
};

IconText.defaultProps = {
  icon: '',
  text: '',
  className: '',
  size: '',
  rotate: null,
  flip: '',
}
```

```

    inverse: false,
    slim: false // draw slim single-line InfoText
  };

```

```

export default IconText;

```

ES5/ES7 Destructuring Assignment. We are introducing another ES5/ES7 feature in this code to extract properties from `props` object. Read more about [destructuring assignment at MDN](#). As this feature was standard in ES5 we do not need any new Babel transform plugins. This way we do not need to prefix the `props.` object to the property references in our code.

Our component can draw multiple variants of Font Awesome Icon and related text. These include a slim version of `IconText` for single-line render. Other variants are based on Font Awesome selectors for size, rotation, flip, and inverse modifiers.

Note the use of `let` for mutable and `const` keyword for immutable values within our code.

Add Home.jsx section to render IconText

Step 4: Now we reuse our new component within `Home` to render some new `IconText` component variants.

Do not forget to import the new component in `Home` before using it.

```

<section className="stripe back--default">
  <Card className="col--one-third text--center back--white">
    <IconText className="primary" icon="globe" size="5x"
      text="Nine Component Creation Strategies" />
  </Card>
  <Card className="col--one-fourth back--white">
    <h3>
      <IconText slim className="danger" icon="building"
        text="Modular Architecture" />
    </h3>
    <h3>
      <IconText slim className="default" icon="cloud"
        text="Leverages Cloud" />
    </h3>
  </h3>

```

```

    <IconText slim className="secondary" icon="cog"
      text="30 Custom Components" />
  </h3>
  <h3>
    <IconText slim className="warning" icon="bullseye"
      text="Goal Oriented Design" />
  </h3>
</Card>
<Card plain className="col--one-third text--center">
  <h1>Infographic Components</h1>
  <p className="subtext">
    Custom React components easily reusable to
    create variety of Infographic elements.
  </p>
</Card>
</section>

```

Note the advantages of converting the Font Awesome CSS library to React components. Our component can be reused with multiple variations in optional properties. We also process custom rendering of the CSS library elements within component code.

API to React

You may want to integrate with an API from the multitude of web service providers. Normally API integration is left to the backend code for authentication, caching, and performance. However, there are real-time public APIs which do provide rate limited, minimal data suitable for client-side calls.

There are benefits of turning an API access, processing, and data rendering code into a React component.

- You can map API endpoints to React properties.
- You can provide reusable UI components wrapping the API functionality.

Strategy for designing a React component from an external API is as follows.

1. Identify a public API which can be accessed without authentication requirements and supporting JSON as results data format.
2. Optionally, parametrize API endpoints to React properties.
3. API resulting JSON can be parametrized as component state. As the results change, we want the state to change, and the component `render()` to be invoked.
4. Render the new component within your app.

Step 1: For our new component we decide to use the GitHub API. Type this in the terminal to make a REST GET call to the GitHub API and observe the resulting JSON. This particular API endpoint sends back results with GitHub repository details including number of stars and open issues.

```
curl -i https://api.github.com/repos/facebook/react
```

Step 2: The GitHub API endpoint follows `/repos/:owner/:repository` format. Owner and repository can vary for our app, and these are good candidates to be combined as a component property.

Add jQuery to template index.html

Step 3: We want to extract number of stars, open issues, and formal name of the repository from the resulting JSON. These will be part of our component state as using AJAX GET request we want our state to be updated when the results are fetched to the browser.

For the AJAX functionality we integrate jQuery with our app. We follow the easiest strategy, same as we followed for Font Awesome integration. We add jQuery to `/app/templates/index.html` like so.

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/2.2.2/jquery.min.js"></script>
```

GitHub.jsx custom component

Now we are ready to create our component wrapper for the API. At this stage we also introduce React component lifecycle methods `componentDidMount` and `componentWillUnmount` added to our code.

The `componentDidMount` method is invoked once, only on the client, immediately after the initial rendering occurs. This is the ideal method to integrate other JavaScript frameworks like jQuery and make AJAX requests.

The `componentWillUnmount` method is invoked immediately before a component is unmounted from the DOM. We can perform any necessary cleanup in this method, such as cleaning up any DOM elements that were created in `componentDidMount` method.


```

import React, {PropTypes} from 'react';
import IconText from './IconText';

export default class GitHub extends React.Component {
  static propTypes = {
    repo: PropTypes.string.isRequired
  }

  constructor (props) {
    super (props);
    this.state = {
      full_name: '',
      stargazers_count: 0,
      open_issues: 0
    };
  }

  componentDidMount() {
    const sourceRepo =
      `https://api.github.com/repos/${this.props.repo}`;

    this.serverRequest = $.get(sourceRepo, function (result) {
      this.setState({
        full_name: result.full_name,
        stargazers_count: result.stargazers_count,
        open_issues: result.open_issues
      });
    }).bind(this);
  }

  componentWillUnmount() {
    this.serverRequest.abort();
  }

  render() {
    return (
      this.state.full_name
      ? <div>
        <h1>
          <IconText slim className="primary" icon="github"
            text={this.state.full_name} />
        </h1>
        <h2><IconText slim className="secondary" icon="star"
          text={` ${this.state.stargazers_count} stars` } /></h2>
        <h2><IconText slim className="danger" icon="bug"
          text={` ${this.state.open_issues} issues` } /></h2>
        </div>
      : <p>Loading Live Stats...</p>
    );
  }
};

```

Note that the `this.serverRequest` is bound to the component using the `bind(this)` method. This is required to integrate AJAX call with React state management. So, state updates when the AJAX call returns with results. This in turn updates the component UI.

The `render()` method uses JavaScript conditional (ternary) operator as that is allowed here, when deciding what to render based on component state. This is required here as the component may render on the browser faster as the AJAX results take longer showing up.

We are also reusing `IconText` to add some infographics to our results.

Add `Home.jsx` section to render GitHub

Now all that remains is to use this component and render within our `Home` component.

```
<section className="stripe">
  <Card className="col--quarter text--center back--default">
    <GitHub repo="angular/angular" />
  </Card>
  <Card className="col--quarter text--center back--default">
    <GitHub repo="facebook/react" />
  </Card>
  <Card className="col--quarter text--center back--default">
    <GitHub repo="meteor/meteor" />
  </Card>
</section>
```

We can of course reuse this component sparingly (due to rate limits and user wait time) to add results from other repositories by just changing the `repo owner/name` within the `repo` property.

Define Component Internals

This chapter does a deep dive into best practices for defining your React component internals. We will add naming conventions to our best practice guidance, mostly following three sources.

- The official [Facebook React docs and tutorials](#)
- [Airbnb React style guide](#)
- We will also add ES6 features and best practices which make your component design more readable, reusable, and robust.

To maintain this chapter as an easy to follow *cheatsheet* format, we will only list minimal code snippets from samples discussed in other chapters and focus on the guidelines and strategies for designing component internals.

You will learn following strategies in this chapter.

- Naming files, folders, and modules.
- Imports and exports.
- Stateless components and pure functions.
- Classes and inheritance.
- Constructor and binding.
- Properties and property types.
- State management.
- Lifecycle methods.
- Event handlers.

- Render and ReactDOM.render methods.
- JSX features and syntax.

We will wrap-up by creating the Workflow component, our most complex custom React component so far, applying most of these strategies.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c06 origin/c06-define-component-internals
npm install
npm start
```

Naming files, folders, and modules

This section outlines strategies for organizing and naming files, folders, and modules, within your React app.

- Use `.jsx` extension for React components.
- Entry point for app is `/app/index.jsx` file.
- Root component name `app` is picked up from folder containing `index.jsx` file.
- PascalCase for component file names. Like `GitHub.jsx` component.
- Use PascalCase for referencing imported React components.
- Use camelCase for variable names representing instances of React components.
- Entry point for styles is `/app/style.css` file which imports the partials.

Imports and exports

Note that import and export of modules is an ES6 feature.

You will start React component definition with a set of import statements referencing modules which are your component dependencies. You will also export the component you are defining in a `ComponentName.jsx` file.

- Use `import ComponentName from 'library';` statement to import modules.
- App entry point `index.jsx` requires `React` and `ReactDOM` modules.
- `ReactDOM` module exposes DOM-specific methods.
- `React` module has the core tools shared by React on different platforms like, React Native.

- You can use `import {Module1, Module2} from 'library';` to import specific modules directly. This way you can reduce some coding by referencing these modules directly.

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

- Alias an imported member name and use the alias within your code.

```
import React, {VeryLongModuleName as shortName} from 'react';
```

- Use `export default` statements to prefix component name declaration for exporting.

```
export default class Card extends React.Component {...
```

Stateless components and pure functions

- If there are no states or refs then prefer normal functions over classes.
- Multiple stateless components are allowed per file.
- You can declare a stateless component function with props.
- Declare a stateless component function with specific properties using destructuring assignment.

```
const Hello = ({ greet, message }) => (...  
export default function YouTube({ videoid }) {...
```

Classes and inheritance

Classes are introduced in ES6 syntax for defining React components.

- Only one component using class definition per file.
- React ES6 classes do not support Read Mixins.
- Methods do not automatically bind `this` to the class instance. Explicitly use bind statement.

```
constructor(props) {  
  super(props);  
  // some code...  
  this.handleChange = this.handleChange.bind(this);  
}
```

- Custom React components often extend `React.Component`.
- `React.Component` API provides `setState()` to perform a shallow merge of `nextState` into current state. This is the primary method you use to trigger UI updates from event handlers and server request callbacks.

```
handleNameChange(event) {  
  this.setState({value: event.target.value});  
}
```

- Treat `this.state` as immutable. Do not change its value directly. Use `setState` method instead.
- Avoid using `forceUpdate` method from `React.Component` API. Instead use `render` method to read from props and state. This makes your component “pure” in the sense that its output render is predictable based on input props.
- ES6 class components that extend `React.Component` do not have `isMounted`, `replaceProps`, `setProps`, `replaceState` and `getDOMNode` methods. These may be removed from React API in future releases.

Constructor and binding

Constructors are a feature of ES6 classes. Constructor methods are called once per instance of a component.

- First statement in a construction is `super(props)` which passes the props within the inheritance tree.
- Use constructor to bind methods. This is better for performance as it is bound once and also when using `shouldComponentUpdate()` method for shallow comparison in the child components.

```
constructor(props) {  
  super(props);  
  // some code...  
  this.handleNameChange = this.handleNameChange.bind(this);  
}
```

- Use constructor to declare `propTypes` and set `defaultProps`.
- Setting default state can be done within the constructor.

Properties and property types

Defining property types makes your code more reliable.

- Properties in ES6 classes are available using `this.props` object.
- Consider `this.props` as immutable within the component. Never change `this.props` directly. Only use JSX attributes to pass props.
- Import `{PropTypes}` from react dependency.
- Define property types using `static propTypes = {}` statement in ES6.

```
static propTypes = {  
  icon: PropTypes.string.isRequired,  
  text: PropTypes.string.isRequired,  
  className: PropTypes.string,  
  size: PropTypes.oneOf(['lg', '2x', '3x', '4x', '5x']),  
  rotate: PropTypes.number,  
  flip: PropTypes.oneOf(['horizontal', 'vertical']),  
  inverse: PropTypes.bool  
}
```

- Set default properties using `static defaultProps = {}` statement in ES6.

```
static defaultProps = { size: '', message: false }
```

- Property type violations show up as warnings in browser console.
- You can define property types for array, string, number, and bool.
- Define property type for methods, like event handling methods propagating from owner component, using `PropTypes.func` statement.
- You can define a property type as required using `.isRequired` statement.

- When using `this.props.children` you can validate that only a single child is passed using `PropTypes.element.isRequired` statement.
- You can define custom property validator with associated Error message.

```
customProp: function(props, propName, componentName) {  
  if (!/matchme/.test(props[propName])) {  
    return new Error(  
      'Invalid prop `' + propName + '` supplied to' +  
      '`' + componentName + `'. Validation failed.'  
    );  
  }  
}
```

- Use JSX spread syntax `{...this.props}` to pass on props as-is from owner to extended component. Read more at Facebook React docs about [transferring_properties](#) using JSX spread syntax.

Using `propTypes` and `defaultProps` is essential for defining robust and reliable React components.

State management

This section discusses when to use state and how.

- State should contain data that a component's event handlers may change to trigger a UI update.
- Owner component where state is defined usually also defines event handlers manipulating this state.
- Always define state at the highest level in a component hierarchy.
- If you want to know prior value of a prop within your app, state could be used to store prop history.
- Define minimal state for a component.
- Most components in your library should be stateless components.
- If the component data is passed from an owner component via props, it probably isn't state.
- If the component data does not change over time, it probably isn't state.
- Do not define state to replace computed properties or computed states.
- For controlled components in forms, like input, use `this.state.value` and `value` prop to manage state outside of the component.

```
<input
  type="text" value={this.state.value}
  placeholder="Enter a name"
  onChange={this.handleChange}
/>
```

- For uncontrolled components which do not use `value` prop, manage state within the component.
- Set default state in constructor after `super(props)` statement using `this.state = {}` statement.

```
constructor(props) {
  super(props);
```

```
this.state = {  
  currentGreeting: props.greet,  
  value: 'ReactSpeed'  
};  
// some code...  
}
```

- Manipulate state in event handler methods using `this.setState({})` method.

```
handleNameChange(event) {  
  this.setState({value: event.target.value});  
}
```

State management is one of the most powerful React features. Please use it responsibly.

Lifecycle methods

This section explains how to decide which lifecycle methods to use and why.

- `componentWillMount` is invoked once. It is invoked both on client and server. It is invoked immediately *before* the initial rendering.
- `componentDidMount` is invoked once, only on the client, immediately *after* initial rendering.
- `componentDidMount` for child components is invoked before parent components.
- Use `componentDidMount` for sending AJAX requests.
- Use `componentDidMount` to integrate with other JavaScript frameworks like jQuery.

```
componentDidMount() {  
  const sourceRepo =  
    `https://api.github.com/repos/${this.props.repo}`;  
  
  this.serverRequest = $.get(sourceRepo, function (result) {  
    this.setState({  
      full_name: result.full_name,  
      stargazers_count: result.stargazers_count,  
      open_issues: result.open_issues  
    });  
  }).bind(this);  
}
```

- `componentWillReceiveProps` is invoked when component is receiving new props. It is not invoked during initial render. It is invoked before render method.
- Use `componentWillReceiveProps` for processing prop transitions and updating state before render method is called. Old props can be accessed using `this.props` object. The method provides a parameter with new props. Updating state using `this.setState()` in this method will not call render method again.

- `shouldComponentUpdate` is invoked before rendering when new props or state are being received. This method is not called for the initial render.
- Use `shouldComponentUpdate` to return `false` when you're certain that the transition to the new props and state will not require a component render update.
- Use `shouldComponentUpdate` to improve app performance by only allowing re-rendering for components when necessary.
- `componentWillUpdate` is invoked immediately before rendering when new props or state are being received. This method is not called for the initial render.
- You cannot use `this.setState()` in `componentWillUpdate` method.
- Use `componentWillUpdate`, if you need to perform operations in response to a state change.
- `componentDidUpdate` method is invoked immediately after the component's updates are flushed to the DOM. This method is not called for the initial render.
- Use `componentDidUpdate` to operate on the DOM when the component has been updated.
- `componentWillUnmount` is invoked immediately before a component is unmounted from the DOM.
- Perform any necessary cleanup in `componentWillUnmount` method, such as aborting AJAX requests or cleaning up any DOM elements that were created in `componentDidMount`.

```
componentWillUnmount() {  
  this.serverRequest.abort();  
}
```

Event handlers

Event handler methods bind to React ES6 components for manipulating component state.

- Always bind event handler methods within constructor. This is more performant.

```
constructor(props) {  
  super(props);  
  // some code...  
  this.handleClick = this.handleClick.bind(this);  
}
```

- Bind event handler method within `on<Event>` statement when passing custom parameters to the event handler.

```
handleButtonClick(color) {  
  this.setState({demoMessage: `Button ${color} clicked.`});  
}
```

- Define event handler methods at highest level owner component in a component hierarchy.
- Define event handlers in components where you define state.

Render and ReactDOM.render methods

Important things to remember about render method and ReactDOM.render.

- ReactDOM.render method should only be called after the composite components have been defined.
- ReactDOM.render instantiates the root component, starts the react framework, and injects the markup into a raw DOM element, provided as the second argument.

JSX features and syntax

JSX is what you write within `render()` `return()` method. JSX gets transpiled to JS by Babel in our build environment.

- JSX HTML-like tags are React framework native components representing HTML tags and attributes.
- JSX components use `className` instead of `class` when specifying CSS classes.
- Attribute values in JSX use `{}` curly braces to wrap JavaScript expressions.
- Boolean attributes can be specified without true value. Not specifying a boolean attribute implies false value.
- JavaScript expressions can be used to specify children of a JSX component.
- Comments in JSX need `{ }` curly braces to wrap JavaScript comments when in children section of a tag.
- HTML entities like ` ` for space can be used within literal text in JSX.
- You can use mixed arrays with strings and JSX elements within JSX tags.

Workflow.jsx component

Let us apply these strategies in creating a relatively complex custom component.

Sometimes you want to start your component design journey with a basic wireframe. A “boxes and arrows” wireframe is a good start for thinking about React component composition. The boxes depict element hierarchy, relationship to each other, and layout. The arrows depict properties and state.

Let us design a new component using this technique. We will design a `Workflow` component to depict component design workflow described in this book.

As you may have noticed so far, we are following a repeating structure for the content of this chapter.

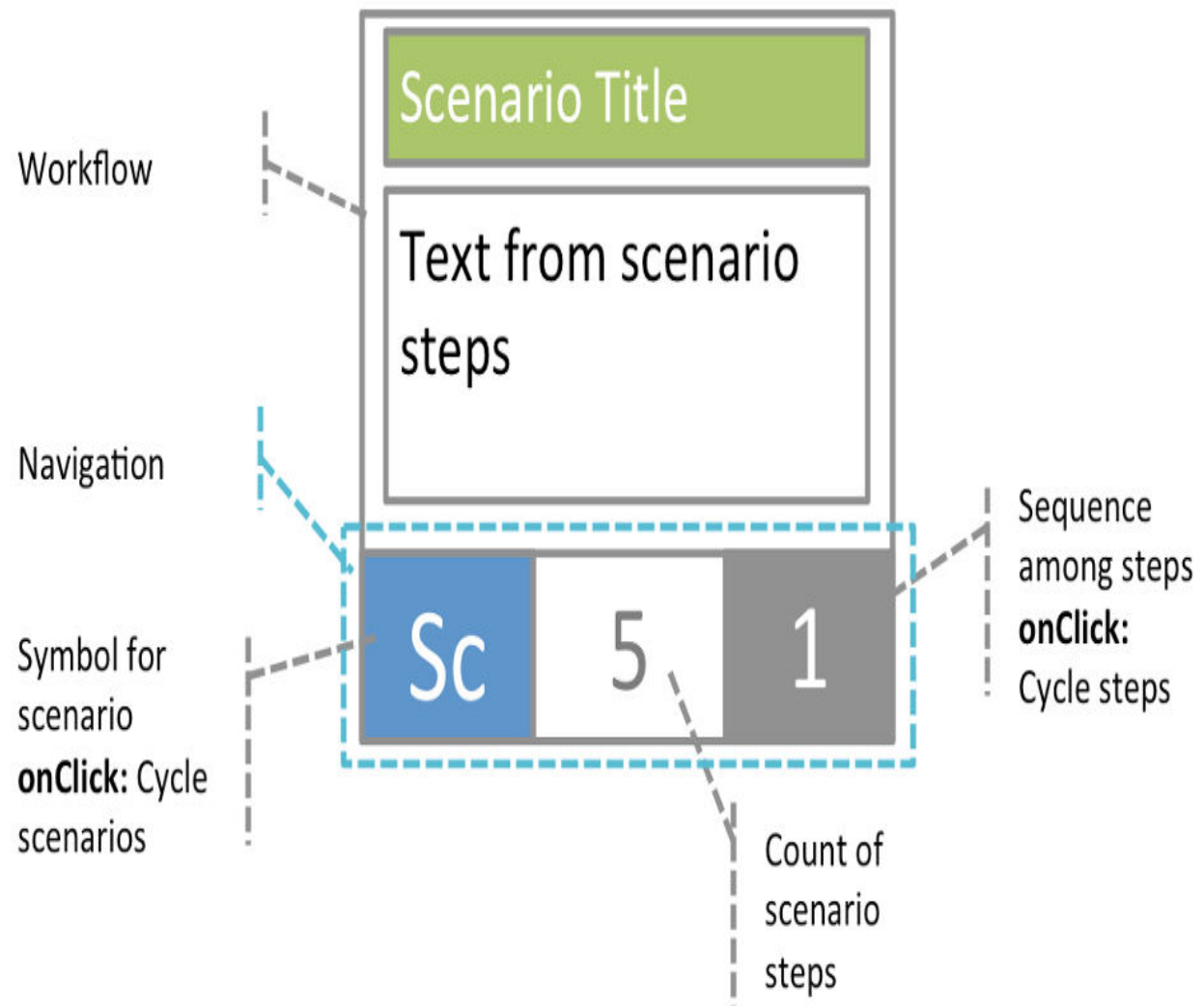
We have outlined several **scenarios** for defining React component internals, like the prior section **JSX features and syntax**. We may use a unique **symbol** (Dj) to identify the scenario, normally following the first character from high level workflow **(D)efine Component Internals** and second letter from scenario name like **(J)SX features...** in this case.

We are also outlining a **sequence** of steps where the **text** defines the action you need to perform to achieve the given scenario.

For our new component we want to visualize it like an element in the Periodic Table of elements. We want to depict the name of the scenario, text from the sequence of steps, symbol depicting the scenario, count of steps available in the scenario, and current sequence for the step displayed.

We also want the component to cycle through scenarios on clicking the symbol and cycle through workflow steps when

clicking on the sequence.



Workflow component wireframe

Notice in the wireframe above how the act of drawing out our component in boxes and arrows provides us the required next steps in our design journey. We know we need a Workflow containing element. We also require a navigation element. Symbol, count, and sequence are contained within the navigation element, and so on.

This information helps us in our next step. Creating the stylesheet partial for Workflow component like so.

Notice how the wireframe guides us in deciding how to use Flexbox layouts to order and style the elements hierarchy.

```
.workflow {
  display: flex;
  flex-flow: column nowrap;
  justify-content: space-between;
  height: 100%;
}

.workflow__scenario {
  color: white;
  background: $secondary;
  font-size: 1.2em;
  font-weight: 300;
  line-height: 25px;
  text-align: center;
  padding: 2px;
}

.workflow__text {
  color: $black;
  font-size: 1em;
  text-align: left;
  padding: 5px;
}

.workflow__nav {
  display: flex;
  flex-flow: row wrap;
  justify-content: space-between;
  margin-top: 5px;
  font-size: 1.2em;
  line-height: 30px;
}

.workflow__symbol {
  background: $primary;
  color: white;
  flex: 1 33%;
  order: 1;
  text-align: center;
  cursor: pointer;
  user-select: none;
  transition: background 1s;
}

.workflow__symbol:hover {
  background: color($primary tint(30%));
}

.workflow__steps {
  color: black;
  flex: 1 34%;
  order: 2;
  text-align: center;
}

.workflow__sequence {
  background: $primary;
}
```

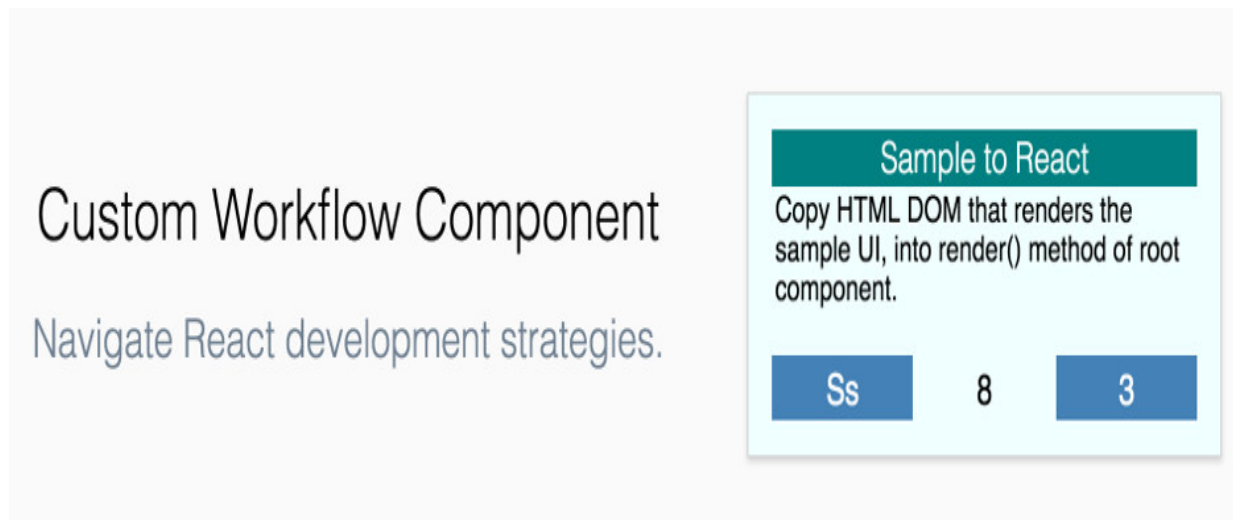
```

    color: white;
    flex: 1 33%;
    order: 3;
    text-align: center;
    cursor: pointer;
    user-select: none;
    transition: background 1s;
  }

  .workflow_sequence:hover {
    background: color($primary tint(30%));
  }
}

```

Here is the listing for the complete Workflow component. Notice how we are passing the component design workflow data into the component. For demo purpose this is fine, however we want something that can scale to cover the entire contents of this book.



Workflow component demo

For this objective we will refactor the component when we wire it up with a database backend using Firebase in a later chapter. We will also reduce the amount of code written to achieve the same results.

```

import React, {PropTypes} from 'react';

export default class Workflow extends React.Component {
  static propTypes = {
    steps: PropTypes.array.isRequired
  }

  constructor(props) {
    super(props);
    this.state = {stepsIndex: 0}
  }
}

```

```

    this.cycleSequence = this.cycleSequence.bind(this);
    this.cycleScenario = this.cycleScenario.bind(this);
  }

  static defaultProps = {
    steps: [
      {symbol: 'Se', scenario: 'Embed to React', sequence: 1,
        text: `Customize the embed code within target platform to suit
        your site or app styles and placement.`},
      {symbol: 'Se', scenario: 'Embed to React', sequence: 2,
        text: `Optionally, parametrize the embed code attributes using\
React props.`},
      {symbol: 'Se', scenario: 'Embed to React', sequence: 3,
        text: `Use stateless component as you will most likely not mai\
ntain embed UI state
        locally in your component.`},
      {symbol: 'Ss', scenario: 'Sample to React', sequence: 1,
        text: `Identify root level component name that represents your\
sample.
        Define component.`},
      {symbol: 'Ss', scenario: 'Sample to React', sequence: 2,
        text: `Split sample code HTML, CSS, JavaScript into separate f\
iles.`},
      {symbol: 'Ss', scenario: 'Sample to React', sequence: 3,
        text: `Copy HTML DOM that renders the sample UI, into render()\
method
        of root component.`},
      {symbol: 'Ss', scenario: 'Sample to React', sequence: 4,
        text: `Optionally, replace some of the HTML with existing reus\
able
        components in your app.`},
      {symbol: 'Ss', scenario: 'Sample to React', sequence: 5,
        text: `Copy CSS into new or existing partial.`},
      {symbol: 'Ss', scenario: 'Sample to React', sequence: 6,
        text: `Copy JS to /app/public/js folder.`},
      {symbol: 'Ss', scenario: 'Sample to React', sequence: 7,
        text: `JS over CDN is referred in <script> tag from /app/templ\
ates/*.html.`},
      {symbol: 'Ss', scenario: 'Sample to React', sequence: 8,
        text: `Import the new component into your index.jsx and
        render in render() method.`},
    ]
  }

  cycleSequence() {
    let nextIndex =
      this.state.stepsIndex === (this.props.steps.length - 1)
      ? 0
      : this.state.stepsIndex + 1;

    this.setState({stepsIndex: nextIndex});
  }

  cycleScenario() {
    const steps = this.props.steps;
    const currentStep = steps[this.state.stepsIndex];
    let stepsCount = 0;
    for(let i = 0; i < steps.length; ++i){
      if(steps[i].symbol === currentStep.symbol) stepsCount++;
    }
    let currentScenario = currentStep.scenario;
    const loopStart =

```

```

        (this.state.stepsIndex + stepsCount) >= steps.length
        ? 0
        : this.state.stepsIndex + 1;
    for(let i = loopStart; i < steps.length; ++i){
        if(steps[i].scenario !== currentScenario) {

            this.setState({stepsIndex: i});

            break;
        }
    }
}

render () {
    const steps = this.props.steps;
    const currentStep = steps[this.state.stepsIndex];
    let stepsCount = 0;
    for(let i = 0; i < steps.length; ++i){
        if(steps[i].symbol === currentStep.symbol) stepsCount++;
    }

    return (
        <div className="workflow">
            <div className="workflow__scenario">
                {currentStep.scenario}
            </div>
            <div className="workflow__text">
                {currentStep.text}
            </div>
            <div className="workflow__nav">
                <div onClick={this.cycleScenario} className="workflow__symbo\
l">
                    {currentStep.symbol}
                </div>
                <div className="workflow__steps">
                    {stepsCount}
                </div>
                <div onClick={this.cycleSequence} className="workflow__seque\
nce">
                    {currentStep.sequence}
                </div>
            </div>
        </div>
    );
}

```

As we illustrated in prior sections, we now import and create and render this component within Home to view it in our app.

The steps followed to design React components from a wireframe can be outlined like so.

1. Scenario ideal for creating custom, non-standard, or complex UI controls.

2. Create wireframe using boxes for elements, layout, composition, and arrows for properties, state, and events.
3. Split individual elements of the wireframe into own style classes. Follow styles structure from wireframe.
4. Use Flexbox to order elements, align element layout, compose element hierarchy.
5. Create HTML/DOM render matching your wireframe, using the new styles.
6. Define properties to handle component data required for each element.
7. Add default property data fixtures if creating a demo component.
8. Create event handler methods and bind these in constructor. Call these methods from element event handlers.
9. Specify component state based on event driven UI state change.
10. Import and render component in an owner component.

Our component does plenty of things within a relatively small screen real-state. One advantage of this size and using Flexbox is how our component experience scales well across mobile and desktop screen size. Another advantage is user perceived performance, as transitions across workflow steps are instantaneous.

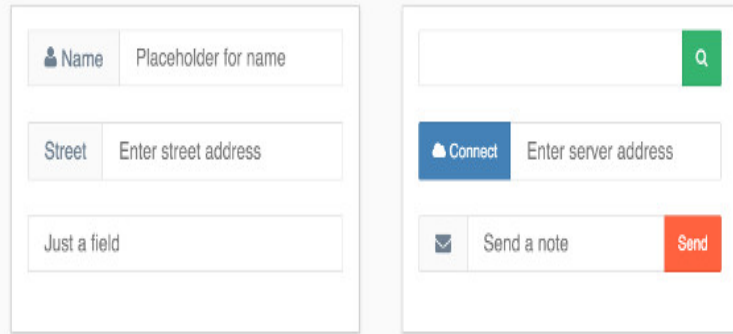
You also learnt that it is relatively straightforward to create a custom UI control, where the look and feel does not need to match with controls in conventional UI libraries or that which HTML enables.

Wire Multiple Components

React is all about composition of multiple components. This chapter highlights best practices in deciding how to design for inter-related multiple components in your project.

Beautiful Responsive Forms

Create beautiful forms using several variations of input controls and buttons.



ReactSpeed says Hello
World!

Slang OR Hindi

Name	ReactSpeed
------	------------

Custom React Component

This custom component demonstrates props, state, and ES6 classes.





Click does not do much...

Default	Primary	Secondary	Warning	Success
Danger				

Click any button...

Primary	Success	Danger	Warning
---------	---------	--------	---------

Click any button...

			
---	---	---	---

Forms and Button Components

You will learn following concepts in this important chapter.

- Render multiple components programmatically.
- Composition using parent child node tree.
- When to use presentational verses container components.
- Reconciliation algorithm and keys for dynamic children.
- Integrating vendor components.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c07 origin/c07-wire-multiple-components
npm install
npm start
```

Render multiple components programmatically

Many times you will require to render multiple components programmatically. Use cases include rendering list items, stream messages, or listing blog posts.

We will continue extending ReactSpeed UI with theme based buttons. We will render demos for these buttons using manual and programmatic listing.

Style button.css

Let us begin by firstly creating some reusable components extending our ReactSpeed UI library. Let us create a button CSS module.

We want our button to render in multiple colors defined in our `variables.css` theme. We also want buttons in various sizes, like, large, medium, small, and default.

This file goes in the `/app/styles/components` folder.

```
.button {
  transition: background-color 0.2s;
  padding: 0.6em 1em;
  border: 0;
  border-radius: 2px;
  margin: 3px;
  cursor: pointer;
  font-size: 0.8125em;
  font-weight: normal;
  line-height: normal;
  text-decoration: none;
  white-space: nowrap;
  user-select: none;
}

.button:focus {
  outline: none;
  text-decoration: none;
}

.button--default { @mixin colorize-button $default; }
.button--primary { @mixin colorize-button $primary; }
.button--secondary { @mixin colorize-button $secondary; }
.button--warning { @mixin colorize-button $warning; }
.button--danger { @mixin colorize-button $danger; }
.button--success { @mixin colorize-button $success; }
```

```

.button--large {
  padding: 0.8em 1.2em;
  font-size: 1.2em;
}

.button--medium {
  padding: .6em 1em;
  font-size: 0.9em;
}

.button--small {
  padding: 0.3em 0.7em;
  font-size: 0.65em;
}

```

Colorize button in mixins.css

Notice we are using CSS mixins to style our button class based on modifiers for different colors. This file goes in the /app/styles/base folder.

```

@define-mixin colorize-button $bg {
  background: $(bg);
  color: white;
  border: 1px solid color($(bg) shade(10%));
  &:active,
  &:focus,
  &:hover {
    background: color($(bg) tint(20%));
    border: 1px solid color($(bg) shade(10%));
    text-decoration:none;
  }
}

```

Update style.css

Next we import the mixins.css and button.css into our style.css file.

```

@import 'normalize.css';

@import 'styles/base/variables';
@import 'styles/base/mixins';

@import 'styles/base/elements';
@import 'styles/base/util';
@import 'styles/base/theme';
@import 'styles/base/type';
@import 'styles/base/layout';

@import 'styles/base/image';

@import 'styles/components/button';
@import 'styles/components/card';
@import 'styles/components/workflow';

```

Button.jsx custom component

We could continue using `<button>` native component in our code however the variations we want on our buttons including icons only, icon plus label, and colors, require us to write more code every time we process button rendering. We can simplify our button rendering code by writing this logic in our custom component just once.

```
import React from 'react';

const Button = (props) => {
  const renderLabel = props.icon
    ? <span>
      <i className={`fa fa-${props.icon}`}></i>
      &nbsp;
      {props.label}
    </span>
    : props.label;

  let renderClass = props.size
    ? `button button--${props.color} button--${props.size}`
    : `button button--${props.color}`;

  // pass on className if any as defined by owner
  renderClass = props.className
    ? `${renderClass} ${props.className}`
    : renderClass;

  return (
    <button {...props} className={renderClass}>
      {renderLabel}
    </button>
  );
}

Button.propTypes = {
  label: React.PropTypes.string,
  icon: React.PropTypes.string,
  size: React.PropTypes.string,
  color: React.PropTypes.string
};

export default Button;
```

Our Button component does not manage its own state so we are using stateless pure function to write the component.

We are also using JSX spread attributes `{...props}` to pass on any props, to the `<button>` native component, that we do not

consume directly within our Button component. This helps in passing on event handlers and className for example.

ButtonDemo.jsx for programmatic rendering

The ButtonDemo component programmatically creates the Button component.

We start by defining properties for receiving an array of button colors, an array of icon names if we are rendering icon buttons, an array of button sizes, and a flag to indicate if we are rendering icon only buttons.

Notice how we bind the event handler to Button custom component and pass a parameter to the method.

We are now rendering the JSX for programmatic generation of multiple demo buttons based on the array props passed by owner component. Using map statement we are mapping colors array to render buttons by color names as labels and className containing color from the primary palette.

```
import React, { PropTypes, Component } from 'react';
import Button from './Button';

export default class ButtonDemo extends Component {
  static propTypes = {
    colors: PropTypes.array.isRequired,
    icons: PropTypes.array,
    sizes: PropTypes.array,
    iconOnly: PropTypes.bool
  }
  static defaultProps = {icons: [], sizes: [], iconOnly: false}

  constructor(props) {
    super(props);
    this.state = {demoMessage: 'Click any button...'};
  }

  handleButtonClick(color) {
    this.setState({demoMessage: `Button ${color} clicked.`});
  }

  render () {
    const renderButtons = this.props.colors.map((color, i) => {
      const renderIcon =
        (this.props.icons === undefined || this.props.icons.length ===
== 0)
        ? null : this.props.icons[i];
      const renderSize =
        (this.props.sizes === undefined || this.props.sizes.length ===
== 0)
```

```

        ? null
        : this.props.sizes[i];
const renderLabel = this.props.iconOnly ? null : color;
return(
  <Button
    key={color}
    label={renderLabel}
    size={renderSize}
    color={color.toLowerCase()}
    icon={renderIcon}
    onClick={this.handleClick.bind(this, color)}
  />
);
});

return (
  <div>
    <p>{this.state.demoMessage}</p>
    {renderButtons}
  </div>
);
}
}

```

Our render method is relatively simple thanks to Button component taking over the render logic. The render method is creating multiple variations of Button custom component based on style parameters passed as properties. As a result, the Home render is relatively simpler, with fewer lines of code when compared with directly rendering instances of the Button custom component variations. This is a good strategy for creating visual test pages for your components.

Render button demos in Home.jsx

The following code shows two variations of how we create instances of Button component. Firstly we render the component directly. Next we parametrize the style information and pass these as properties to ButtonDemo for programmatic rendering.

```
<section className="stripe">
  <Card className="col--one-third text--center">
    <p>Click does not do much...</p>
    <Button label="Default" color="default" />
    <Button label="Primary" color="primary" />
    <Button label="Secondary" color="secondary" />
    <Button label="Warning" color="warning" />
    <Button label="Success" color="success" />
    <Button label="Danger" color="danger" />
  </Card>
  <Card className="col--quarter text--center">
    <ButtonDemo
      colors={['Primary', 'Success', 'Danger', 'Warning']}
    />
  </Card>
  <Card className="text--center">
    <ButtonDemo
      colors={['Secondary', 'Success', 'Danger', 'Warning']}
      sizes={['large', 'medium', 'medium', 'small']}
      icons={['coffee', 'cloud', 'flash', 'plug']}
      iconOnly
    />
  </Card>
</section>
```

Composition using parent child node tree

Once you create reusable components in React, you should be able to compose more complex, feature rich components by just building a component tree, just like you do with HTML nodes.

Let us create another component hierarchy to demonstrate this. This time we are creating a form using React native input components. We want input control to come in several variations. Simple input box. Label and input box. Icon label and input box. And, icon label, input box, and a button. We want to design our input box variations within JSX by just combining the required components together.

Home.jsx section with forms composition

Here is what our Home component rendering of the forms looks like. This is based on input control styles and Button custom component we reused earlier. Notice how Home is rendering multiple instances of Card component, which in turn renders several child nodes including Button custom component and input native component that React provides.

```
<section className="stripe back--default">
  <Card plain className="col--one-third text--center">
    <h1>Beautiful Responsive Forms</h1>
    <p className="subtext">
      Create beautiful forms using several variations
      of input controls and buttons.
    </p>
  </Card>
  <Card className="col--quarter back--white">
    <div className="input">
      <span className="input__label">Name</span>
      <input className="input__field" placeholder="Placeholder for nam\
e" />
    </div>
    <div className="input">
      <input className="input__field" placeholder="Just a field" />
    </div>
  </Card>
  <Card className="back--white">
    <div className="input">
      <button className="button button--success"><i className="fa fa-s\
earch"></i></button>
```

```
    <input className="input__field" placeholder="Search something" />
  </div>
  <div className="input">
    <span className="input__label"><span className="fa fa-envelope">\
</span></span>
    <input className="input__field" placeholder="Send another one" />
    <button className="button button--warning">Send</button>
  </div>
</Card>
</section>
```

Style in input.css

We have CSS modules styling default components React provides out of the box. This is a fair iterative design strategy. We create React components as we need them. If HTML nodes + styles serve the purpose, we continue moving forward in our app design.

```
.input {
  display: flex;
  margin-bottom: 1.5em;
}

.input--slim {
  margin-bottom: 0.5em;
}

.input__field {
  flex: 1;
  transition: background 2s;
}

.input__field:focus {
  outline: none;
  text-decoration: none;
  background: color(white shade(2%));
}

.input__field:not(:first-child) {
  border-left: 0;
}

.input__field:not(:last-child) {
  border-right: 0;
}

.input__label {
  background: $default-light;
  color: $default;
  font: inherit;
  font-weight: normal;
}

.input__field,
.input__label {
  border: 1px solid color($default-light shade(10%));
  padding: 0.6em 1em;
}

.input > .button {
  margin: 0;
}

.input > .button:first-child,
.input__field:first-child,
.input__label:first-child {
```

```
border-radius: 2px 0 0 2px;  
}
```

```
.input > .button:last-child,  
.input__field:last-child,  
.input__label:last-child {  
border-radius: 0 2px 2px 0;  
}
```

Creating InputLabel.jsx custom component

However, we see an opportunity to make our code simpler and more reusable. If we create custom components for Input, InputLabel, and InputField, we will not only simplify rendering our forms in the Home component, we can also refactor our World component to make the input text box look consistent with our theme.

We are processing several variants of InputLabel where we provide icon and label, label only, or icon only.

We also take care of defining the style just once within our custom component.

```
import React from 'react';

const InputLabel = (props) => {
  let renderLabel = null;
  if (props.icon && props.label) {
    renderLabel =
      <span>
        <span className={`fa fa-${props.icon}`} />
        &nbsp;
        {props.label}
      </span>;
  }

  if (!props.icon && props.label) {
    renderLabel = props.label;
  }

  if (props.icon && !props.label) {
    renderLabel =
      <span className={`fa fa-${props.icon}`} />;
  }

  // pass on className if any as defined by owner
  const renderClass = props.className
    ? `input__label ${props.className}`
    : 'input__label';

  return (
    <span {...props} className={renderClass}>
      {renderLabel}
    </span>
  );
}

InputLabel.propTypes = {
  label: React.PropTypes.string,
  icon: React.PropTypes.string
}
```

```
};
```

```
export default InputLabel;
```

InputField.jsx custom component

Next we create InputField custom component in a similar manner. For now we decide to pass on most props as-is using the JSX spread operator.

```
import React from 'react';
```

```
const InputField = (props) => (  
  <input  
    {...props}  
    className={props.className  
      ? `input__field ${props.className}`  
      : 'input__field'}  
    placeholder={props.placeholder}  
  />  
);
```

```
InputField.propTypes = {  
  placeholder: React.PropTypes.string  
};
```

```
export default InputField;
```

Input.jsx custom component

Finally we create a custom component for the input group. We expect this component to render its children which could be either InputField or InputLabel variants.

```
import React from 'react';
```

```
const Input = (props) => (  
  <div  
    {...props}  
    className={props.className  
      ? `input ${props.className}`  
      : 'input'}>  
    {props.children}  
  </div>  
);
```

```
Input.propTypes = {  
  children: React.PropTypes.node  
};
```

```
export default Input;
```

Refactor Home.jsx section

Now that we have defined our custom components let us refactor Home.jsx to reuse these. After importing the required components, you will notice our render code it relatively simplified.

```
<section className="stripe back--default">
  <Card plain className="col--one-third text--center">
    <h1>Beautiful Responsive Forms</h1>
    <p className="subtext">
      Create beautiful forms using several variations
      of input controls and buttons.
    </p>
  </Card>
  <Card className="col--quarter back--white">
    <Input>
      <InputLabel label="Name" icon="user" />
      <InputField placeholder="Placeholder for name" />
    </Input>
    <Input>
      <InputLabel label="Street"/>
      <InputField placeholder="Enter street address" />
    </Input>
    <Input>
      <InputField placeholder="Just a field" />
    </Input>
  </Card>
  <Card className="back--white">
    <Input>
      <InputField />
      <Button color="success" icon="search" />
    </Input>
    <Input>
      <Button color="primary" icon="cloud" label="Connect" />
      <InputField placeholder="Enter server address" />
    </Input>
    <Input>
      <InputLabel icon="envelope" />
      <InputField placeholder="Send a note" />
      <Button color="warning" label="Send" />
    </Input>
  </Card>
</section>
```


Refactor World.jsx

We can also refactor World.jsx to reuse the Input, InputField, and InputLabel components. This may seem as somewhat additional code to do the same thing as we could with the native React component. Benefit of such reuse shows up when we start extending and modifying our custom components. Let us consider when we add in-place validations for our input controls. Modifying the behavior of our UI everywhere in our app can now be controlled from one custom component definition file.

```
render() {
  const renderGreeting = this.state.value
    ? `${this.state.value} says ${this.state.currentGreeting}`
    : this.state.currentGreeting;
  return (
    <div>
      <Hello greet={renderGreeting} message="World!" />
      <h2>
        <a onClick={this.slangGreet}>Slang</a>
        &nbsp;OR&nbsp;
        <a onClick={this.hindiGreet}>Hindi</a>
      </h2>
      <Input>
        <InputLabel label="Name" />
        <InputField
          type="text"
          value={this.state.value}
          placeholder="Enter a name"
          onChange={this.handleNameChange}
        />
      </Input>
    </div>
  );
}
```

Parent-child tree composition strategies are summarized here.

- Compose components like HTML/DOM tree to render complex variations of reusable components.
- Composition in a node tree creates parent-child relationship between components.
- Parent-child relationship is easier to create than Owner-owned relationship.

- Parent-child relationship is relatively decoupled when compared to Owner-owned relationship.
- Consume child nodes within parent render method using `this.props.children` property.

Presentational and container components

Our apps will use two kind of components. Presentational and container components.

Here are some guidelines to decide presentational and container components as suggested by Dan Abramov in his article on [Presentational and Container Components](#).

How do you decide that you are writing presentational components.

- Examples: YouTube, LeanPub, Hello, Card, Button.
- Presentational components are concerned about how things look.
- May contain both presentational and container components inside.
- Usually have some DOM markup.
- Have styles associated with the component.
- Often allow containment via `this.props.children` within the container component.
- Have no dependencies on the rest of the app.
- Don't specify how the data is loaded or mutated.
- Receive data and callbacks exclusively via props.
- Rarely have their own state (when they do, it's UI state rather than data).
- Are written as functional components unless they need state, lifecycle hooks, or performance optimizations.

How to decide that you are writing container components.

- Examples: World, ButtonDemo.
- Are concerned with how things work.
- May contain both presentational and container components inside.

- Usually don't have any DOM markup of their own except for some wrapping divs.
- Never have any styles.
- Provide the data and behavior to presentational or other container components.
- Call Flux actions and provide these as callbacks to the presentational components.
- Are often stateful, as they tend to serve as data sources.
- Are usually generated using higher order components such as `connect()` from React Redux, `createContainer()` from Relay, or `Container.create()` from Flux Utils, rather than written by hand.

Reconciliation algorithm and keys for dynamic children

This section is important when you are designing component hierarchies with multiple, repeating components.

One of the ways React apps remain performant is by doing DOM diff while calculating which nodes need to be updated during state changes and re-rendering.

This strategy applies to [Dynamic Children](#), where you are creating component nodes using an iterator or array.

As the Facebook documentation article on [reconciliation algorithm](#) explains, React uses unique keys to make this algorithm performant.

- **Unique.** The keys need to be unique only within the parent tree.
- **Stable.** Using random generators for keys is not advised as every time the key for same node may be different, resulting in re-rendering of whole tree. This also leads to unpredictable state.
- **Predictable.** Do not use array ids as keys as elements may get removed, added, or reordered, which in turn will change the id to element mapping.
- **Outermost Component.** Specify keys for the outermost component that is part of a dynamic list, not within the component's render method.

There are several strategies for using unique keys.

- **Unique content as key.** If you have a list of finite, pre-defined, unique content items, where you are certain

that the content items will not change, you could reuse the content items as the key.

```
const renderMessages = messages.map(message =>  
  <Card key={message} message><h3>{message}</h3></Card>  
);
```

If you do not supply the key in this piece of code, following warning results.

React keys warning in browser console

Warning: Each child in an array or iterator should have a unique “key” prop. Check the render method of `Card`. See <https://fb.me/react-warning-keys> for more information.

- **Database id as key.** When using a database to create dynamic elements, you may want to reuse any unique ids maintained by your database as key. This transfers the key stability and uniqueness burden to the underlying database. You also have a consistent way of managing unique keys for the data documents or records that the specific component is rendering.
- **GUID generators.** When you have user generated dynamic list of nodes in your app, like a set of comments in a discussion forum, you may want to explore trade-offs between using a GUID generator library or using the underlying database for the unique ids. We prefer the latter approach as it creates less performance burden for our front-end and avoids data redundancy in maintaining two sets of uniques per record or document.

Integrating vendor components

There are times when you may not want to develop your own custom component as you prefer a vendor written React component. We use the following broad strategies to help us decide for a vendor component over creating our own or using another alternative.

Do not reinvent the wheel. We prefer focusing on our domain specific problem, custom components that solve this problem. If there is better, reusable code out there, we prefer integrating it.

NPM for all the things. If the vendor component is not available on NPM we do not use it. NPM is well maintained repository of versioned dependencies and reusable code. Going outside of NPM when our development environment is Node.js really does not make sense.

GitHub. Source on GitHub is another essential passing criteria for us to select a vendor component.

Tests. Does the component source include tests or demos. Always prefer code with tests included.

Documentation. We consider level of source documentation, sample integration code, demos. Better documented alternatives win.

Popularity. Number of stars on GitHub. Number of downloads on NPM. Higher is obviously better.

Payload. Adding vendor code has a cost. It costs additional KB in our overall app payload. We prefer vendor components that do not exhibit feature bloat. Follow single responsibility principle.

Owner Props Match. Our owner component consuming the vendor component integration needs to have the required props to pass on to the vendor component.

So, following our strategies we select one of our first React vendor components to integrate with ReactSpeed website.

We decide to include [Rumble Charts](#) as a vendor component matching the selection criteria we just defined. They offer one of the best documented React component libraries for adding charting using another best of breed library [D3.js](#).

We start by installing the vendor component library using NPM.

```
npm install --save rumble-charts
```


Chart fixture in charts.js

We are picking up the chart data from fixtures we create like so. We place this file in a new `/app/fixtures/charts.js` folder.

```
export const singleSeries = [{
  data: [1, 2, 4]
}];

export const series = [{
  data: [1, 2, 3]
}, {
  data: [5, 7, 11]
}, {
  data: [13, 17, 19]
}];

export const cloudSeries = [{
  data: [
    { label: 'Highcharts', y: 30 },
    { label: 'amCharts', y: 13 },
    { label: 'Google Charts', y: 31 },
    { label: 'ChartJS', y: 15 },
    { label: 'TauCharts', y: 8 },
    { label: 'FusionCharts', y: 2 },
    { label: 'ZingChart', y: 2 },
    { label: 'uvCharts', y: 1 },
    { label: 'jQuery Sparklines', y: 1 },
    { label: 'Ember Charts', y: 2 },
    { label: 'Canvas.js', y: 16 },
    { label: 'Flot', y: 1 },
    { label: 'D3.js', y: 27 },
    { label: 'n3-charts', y: 3 },
    { label: 'NVD3', y: 3 },
    { label: 'Chartist.js', y: 3 },
    { label: 'C3.js', y: 14 },
    { label: 'Cubism.js', y: 1 },
    { label: 'Rickshaw', y: 2 }
  ]
}];
```

Imports for adding charts in Home.jsx

We import the fixtures data and chart components required to render our sample charts.

```
import { series, singleSeries, cloudSeries } from '../fixtures/charts';

const {
  // main component
  Chart,
  // graphs
  Bars, Cloud, Labels, Lines, Pies, RadialLines, Ticks,
  // wrappers
  Layer, Animate, Transform
} = require('rumble-charts');
```

Constructor and event handler in Home.jsx

Next we add a constructor and an event handler. The constructor defines a set of state variables to track how the chart data changes on user interaction.

The event handler generates random data on user click to update state. This enables us to demonstrate animated charts on user interaction.

```
constructor() {
  super();
  this.state = { series, cloudSeries, singleSeries, hovered: {} };
  this.updateSeries = this.updateSeries.bind(this);
}
updateSeries = () => {
  /* eslint-disable no-shadow, no-undef */
  const singleSeries = [{
    data: _.map(_.range(3), () => Math.random() * 100)
  }];
  const series = _.map(_.range(3), () => ({
    data: _.map(_.range(3), () => Math.random() * 100)
  }));
  this.setState({ series, cloudSeries, singleSeries, hovered: {} });
  /* eslint-enable no-shadow, no-undef */
}
```

Home.jsx update to render sample charts

We add two new sections to Home.jsx for rendering six sample charts.

```
<section className="stripe back--default">
  <Card className="back--white">
    <Chart onClick={this.updateSeries} width={300} height={200} series\
    ={this.state.series}>
      <Transform method={['transpose', 'stackNormalized']>
        <Pies
          colors="category10"
          combined
          innerRadius="33%"
          padAngle={0.025}
          cornerRadius={5}
          innerPadding={2}
          pieAttributes={{
            /* eslint-disable no-return-assign, no-param-reassign */
            onMouseMove: (e) => e.target.style.opacity = 1,
            onMouseLeave: (e) => e.target.style.opacity = 0.5
            /* eslint-enable no-return-assign, no-param-reassign */
          }}
          pieStyle={{ opacity: 0.5 }}
        />
      </Transform>
    </Chart>
  </Card>
  <Card className="back--white">
    <Chart
      onClick={this.updateSeries}
      width={300}
      height={200}
      series={this.state.series}
      minY={0}
    >
      <Animate _ease="bounce" _ease="elastic">
        <Layer width="80%" height="80%" position="middle center">
          <Ticks
            axis="y"
            ticks={{ maxTicks: 4 }}
            tickVisible={({ tick }) => tick.y > 0}
            lineLength="100%"
            lineVisible
            lineStyle={{ stroke: 'lightgray' }}
            labelStyle={{
              textAnchor: 'end',
              alignmentBaseline: 'middle',
              fontSize: '0.5em',
              fill: 'lightgray'
            }}
            labelAttributes={{ x: -5 }}
          />
          <Ticks
            axis="x"
            label={({ tick }) => tick.x + 1}
            labelStyle={{
```

```

        textAnchor: 'middle',
        alignmentBaseline: 'before-edge',
        fontSize: '0.5em',
        fill: 'lightgray'
    }}
    labelAttributes={{ y: 3 }}
  />
  <Bars
    groupPadding="3%"
    innerPadding="0.5%"
    barAttributes={{
      /* eslint-disable no-return-assign, no-param-reassign */
      onMouseMove: e => e.target.style.fillOpacity = 1,
      onMouseLeave: e => e.target.style.fillOpacity = 0.5
      /* eslint-enable no-return-assign, no-param-reassign */
    }}
    barStyle={{
      fillOpacity: 0.5
    }}
  />
  <Lines lineWidth={2} />
  <Labels
    label={{({ point }) => (`y=${Math.round(point.y)}`)}}
    dotStyle={{
      textAnchor: 'middle',
      dominantBaseline: 'text-after-edge',
      fontFamily: 'sans-serif',
      fontSize: '0.65em'
    }}
    labelAttributes={{
      y: -4
    }}
  />
</Layer>
</Animate>
</Chart>
</Card>
<Card className="back--white">
  <Chart
    onClick={this.updateSeries}
    width={300}
    height={200}
    series={this.state.series}
    minY={0}
  >
    <RadialLines />
  </Chart>
</Card>
</section>
<section className="stripe">
  <Card>
    <Chart
      onClick={this.updateSeries}
      width={300}
      height={200}
      series={this.state.singleSeries}
      minY={0}
    >
      <Transform method={['transpose']}>
        <Layer width="80%" height="80%">
          <Bars />
        </Layer>
      </Transform>
    </Chart>
  </Card>
</section>

```

```

    <Layer width="25%" height="25%" position="right bottom">
      <Transform method="stack">
        <Pies
          combined
          colors="category10"
          pieStyle={{ opacity: 0.8 }}
        />
      </Transform>
    </Layer>
  </Transform>
</Chart>
</Card>
<Card>
  <Chart
    onClick={this.updateSeries}
    width={300}
    height={200}
    series={this.state.series}
    minY={0}
  >
    <Transform method={['transpose', 'stack']}>
      <Pies
        combined
        innerPadding="3%"
        innerRadius="20%"
      />
    </Transform>
  </Chart>
</Card>
<Card>
  <Chart
    onClick={this.updateSeries}
    width={300}
    height={200}
    series={this.state.cloudSeries}
    minY={0}
  >
    <Transform method="transpose">
      <Cloud
        font="Arial"
        minFontSize={12}
        maxFontSize={36}
        padding={2}
      />
    </Transform>
  </Chart>
</Card>
</section>

```

This is how our new charts look like.

Chart Components



Chart Components

As you can see, this is one of the fastest ways to add functionality by wiring our app with vendor components.

Routing to wire component layouts

We will cover routing in great detail in the next chapter on **Route Component Layouts** creating more than 10 new components and wiring these together using React Router.

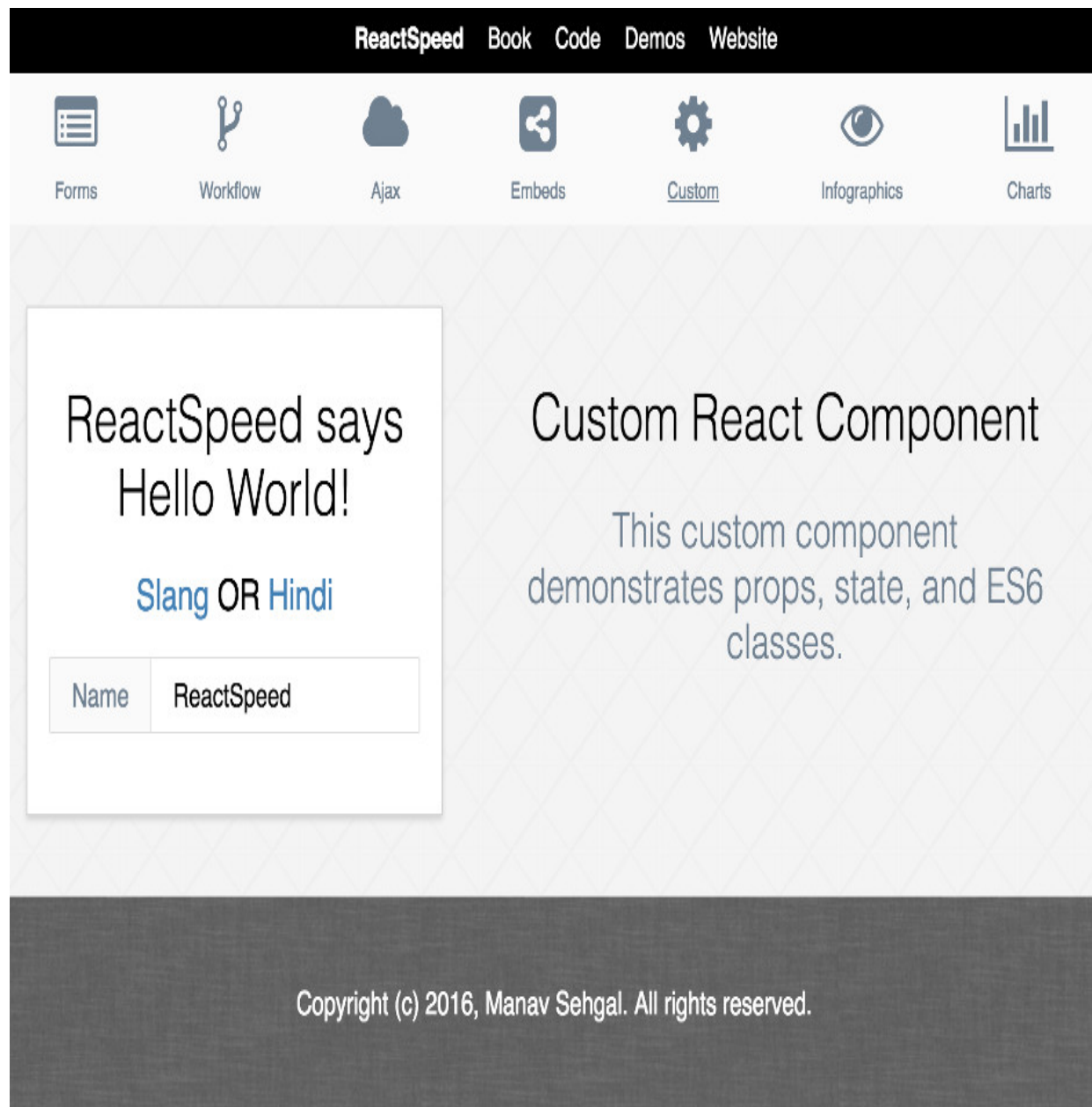
We will explain strategies to refactor the `Home` component to showcase category of UI components together. Like one category could be AJAX components, another could be Media components. We will need a new navigation component as well to switch between these component categories.

General strategies for wiring component layouts include following.

- Identify your navigation hierarchy first.
- Match the navigation hierarchy to component hierarchy required to render when each menu is clicked.
- Reuse layout components as containers.
- Use stateless presentational components to render content on navigation menu endpoints.

Route Component Layouts

So far we have been developing using a single layout. For more complex apps you may want to use multiple layouts. In order to switch from one layout to another using links or page URLs, we will use routing.



Routing Multiple Pages

- Layout strategy for routing.
- AboutBook custom component.
- Router configuration in index.js.
- Nav.jsx component and navigation styles update.
- NavLink component.
- Landing.jsx layout component.
- Programmatic routing in Ribbon.jsx.

- Search engine friendly URLs.
- Handling router exceptions.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c08 origin/c08-route-component-layouts
npm install
npm start
```

Layout strategy

As we start planning our layout strategy we take a few decisions regarding our monolith `Home.jsx` file.

Instead of direct external links from our main navigation (book, code, demons, website) we may want landing pages which describe the external resource that user may decide to visit. We will extract the book section for example as `AboutBook` component to achieve this.

Each section describing specific components should be categorized and reachable using their own navigation path (charts, embeds, workflow, etc.). We will extract the charts section from `Home` as `AboutCharts` component, and repeat same action for all other sections.

We want to display shared sections like `Nav` and `Footer` across all pages of our app. We will extract from `Home` component, new `Nav` and `Footer` components which will be shared across our app.

We will also create a new `Landing.jsx` component to render the shared layout for all our app pages.

Once we are done with these steps, we will no longer need the `Home.jsx` file created in the last chapter.

AboutBook.jsx component

Our first step is to extract the section JSX code from Home component into specific AboutComponent components. Think of these as components displaying the section content describing the components rendered in that section.

Our AboutBook component now renders the book related JSX.

```
import React from 'react';
import Card from './Card';

const AboutBook = () => (
  <section className="stripe">
    <Card plain className="text--center">
      <a href="https://leanpub.com/reactspeedcoding"
        className="image__link">
        
        </a>
      </Card>
    <Card plain className="col--half text--center">
      <h1>Develop Awesome Apps</h1>
      <p className="subtext">
        Join 100s of readers learning
        latest React ES6 concepts.
      </p>
    </Card>
  </section>
);

export default AboutBook;
```

AboutEmbeds.jsx component

Let us consider another example of extracting embed section to AboutEmbeds component.

```
import React from 'react';
import Card from './Card';
import YouTube from './YouTube';

const AboutEmbeds = () => (
  <section className="stripe">
    <Card plain
      className="col--one-third text--center back--white">
      <YouTube videoid="MGuKhcnrqGA" />
    </Card>
    <Card plain className="col--half text--center">
      <h1>Embed React Components</h1>
      <p className="subtext">
        This custom component demonstrates media
        embed within custom React component.
      </p>
    </Card>
  </section>
);

export default AboutEmbeds;
```

We are not listing all the extracted AboutComponents here as they follow the same strategy as shown above. We simply create a stateless component, import the required dependencies like Card component, copy the section JSX from Home component and return this as JSX render from our new component. The GitHub branch contains all the new components we create for this chapter.

Configure routes in index.js

[React Router](#) will help us in achieving our layout strategy. To use React Router we add it using NPM.

```
npm install --save react-router
```

Next we import the required components we want to render in our app.

As we configure `routeConfig` JSX, we note that the Router root component defines the URL strategy for our app pages. In this case we are using the easiest to configure `hashHistory` strategy, which automatically generates unique URLs using hash-bang (#) and a unique string appended to the URL.

The Route component hierarchy represents our navigation hierarchy. The root `/` path specifies the component we want to render when user reaches root of our app. The `IndexRoute` defines component rendered within `this.props.children` of our Landing component when user reaches root of the app. In short we can now change what gets rendered on home page of our app by simply swapping `IndexRoute` component.

Remaining Route definitions specify a path, like `/book` where we want to render the `AboutBook` component.

```
import React from 'react';
import ReactDOM from 'react-dom';
import Landing from './components/Landing.jsx';
import AboutBook from './components/AboutBook';
import AboutCharts from './components/AboutCharts';
import AboutForms from './components/AboutForms';
import AboutCustom from './components/AboutCustom';
import AboutWorkflow from './components/AboutWorkflow';
import AboutButtons from './components/AboutButtons';
import AboutAjax from './components/AboutAjax';
import AboutInfographics from './components/AboutInfographics';
import AboutEmbeds from './components/AboutEmbeds';

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

const routeConfig = (
  <Route path="/" component={Landing}>
```

```

    <IndexRoute component={AboutCharts} />
    <Route path="/book" component={AboutBook} />
    <Route path="/charts" component={AboutCharts} />
    <Route path="/embeds" component={AboutEmbeds} />
    <Route path="/forms" component={AboutForms} />
    <Route path="/custom" component={AboutCustom} />
    <Route path="/workflow" component={AboutWorkflow} />
    <Route path="/buttons" component={AboutButtons} />
    <Route path="/ajax" component={AboutAjax} />
    <Route path="/infographics"
      component={AboutInfographics} />
  </Route>
);

ReactDOM.render(
  <Router
    history={hashHistory}
    routes={routeConfig} />,
  document.getElementById('app')
);

```

Nav.jsx component

Let us extract the components that may repeat across all pages of our app. Good candidates are a Nav component and a Footer component.

In our Nav component we are importing the Link component from React Router. We use this component to replace <a> native components and define to property to map the link to the respective route as configured in the index.jsx file.

Now clicking on ReactSpeed menu and Book menu will bring up the respective components.

Let us also use React Router to indicate active menu. To do this we update the Link component and add activeClassName prop. We add a CSS modifier nav__link--active to indicate that this menu is active.

```

import React from 'react';
import { Link } from 'react-router';

const Nav = () => (
  <nav className="nav">
    <Link className="nav__brand" to="/">
      ReactSpeed
    </Link>
    <Link className="nav__link"
      activeClassName="nav__link--active" to="/book">

```

```

      Book
    </Link>
    <a className="nav__link"
      href="https://github.com/manavseghal/react-speed-book">
      Code
    </a>
    <a className="nav__link"
      href="https://manavseghal.github.io/react-speed-demos/">
      Demos
    </a>
    <a className="nav__link" href="https://reactspeed.com">
      Website
    </a>
  </nav>
);

export default Nav;

```

Navigation styles in nav.css

Now we can move the nav selectors from theme.css to nav.css in the /app/styles/components/ folder and update it like so.

```

.nav {
  display: flex;
  flex-flow: row nowrap;
  justify-content: center;
  list-style: none;
  background: black;
  margin: 0;
  padding: 10px;
}

.nav__link {
  text-decoration: none;
  border-bottom: none !important;
  color: white;
  font-size: 16px;
  padding-left: 20px;
  transition: 0.5s color;
}

.nav__link:hover,
.nav__link:active {
  color: color(white shade(20%));
}

.nav__link--active {
  @extend .nav__link;
  text-decoration: underline;
}

.nav__brand {
  @extend .nav__link;
  font-weight: bold;
}

```


Footer.jsx component

Our footer component is copied as-is from the Home section.

```
import React from 'react';
import Card from './Card';

const Footer = () => (
  <section className="stripe--slim back--gray">
    <Card plain className="col--full text--center white">
      <p>
        Copyright (c) 2016, Manav Sehgal.
        All rights reserved.
      </p>
    </Card>
  </section>
);

export default Footer;
```

Landing.jsx layout component

We can create the new Landing.jsx layout component with Nav and Footer. The `this.props.children` property will be replaced with components as configured in our routes.

```
import React, { Component } from 'react';
import Nav from './Nav';
import Footer from './Footer';

export default class Landing extends Component {
  render() {
    return(
      <section className="landing">
        <Nav />
        {this.props.children}
        <Footer />
      </section>
    );
  }
}
```

At this stage we can test our app by running `npm start` and visiting the localhost URL for our app. We can append each route or path to the hash-bang URL to test each route. Few examples are `localhost:8080/#/book` or `http://localhost:8080/#/custom` to reach different sections of our app. Main navigation links for app root and book also work as intended.

NavLink component

We can extract another component to make our navigation links less verbose. Let us write NavLink component to encapsulate rendering of Link based on associated style modules, modifier (brand link), and props (href or to).

```
import React, {PropTypes} from 'react';
import { Link } from 'react-router';

export default class NavLink extends React.Component {
  static propTypes = {brand: PropTypes.bool}
  static defaultProps = {brand: false}

  render() {
    const renderClass = this.props.brand
      ? "nav__brand" : "nav__link";

    const renderActiveClass = this.props.brand
      ? null : "nav__link--active";

    return (
      <li className="grid-cell">
        {
          this.props.to
          ? <Link
              to={this.props.to}
              className={renderClass}
              activeClassName={renderActiveClass}>
                {this.props.children}
              </Link>
          : <a href={this.props.href} className="nav__link">
                {this.props.children}
              </a>
        }
      </li>
    );
  }
}
```

The behavior we are intending with renderActiveClass is to ensure that when brand link is clicked there is no active link indicator, otherwise we get an active link indicator.

Our Nav links are now much simpler.

```
import React from 'react';
import { Link } from 'react-router';
import NavLink from './NavLink';

const Nav = () => (
  <nav className="nav">
    <NavLink to="/" brand>ReactSpeed</NavLink>
  </nav>
)
```

```
<NavLink to="/book">Book</NavLink>
<NavLink
  href="https://github.com/manavseghal/react-speed-book">
  Code
</NavLink>
<NavLink
  href="https://manavseghal.github.io/react-speed-demos/">
  Demos
</NavLink>
<NavLink href="https://reactspeed.com">
  Website
</NavLink>
</nav>
);

export default Nav;
```

Programmatic routing in Ribbon.jsx

Now that we are rendering components in our Landing page, what we really want is to render categorized components based on user selection. So, when user selects Charts we show Chart components, when they select Infographics, we show the relevant components, and so on.

We need a secondary navigation to achieve this goal. Let us add a ribbon menu below the main navigation, with icons and text representing the menu items.

In order for us to programmatically call our routes, we need access to the router object. We do this by using `contextTypes` definition just before the constructor. When you want to pass data through the component tree without having to pass the props down manually at every level. React's [context feature](#) lets you do this.

Once we have the router context, we use this in the event handler `setRibbonActive` method to programmatically call a route based on ribbon menu clicked by the user.

Ribbon component manages its own state, maintaining `ribbonActive` to store the active ribbon menu's text.

We reuse the `IconText` component by binding `onClick` to our event handler, and passing the menu text when handler is invoked. We then set the state, which triggers re-render of our menu, setting the `active` flag to true for the menu which was last clicked. The ribbon bool flag is consumed by the refactored `IconText` component to render differently for the ribbon menu.

```
import React, { Component, PropTypes } from 'react';
import IconText from './IconText';

export default class Ribbon extends Component {
  static contextTypes = {
```

```

    router: PropTypes.object.isRequired
  }
  constructor(props) {
    super(props);
    this.state = { ribbonActive: '' }
  }
  setRibbonActive(activeText) {
    this.setState({ ribbonActive: activeText });
    this.context.router.push(activeText.toLowerCase());
  }
  render() {
    return (
      <section className="ribbon">
        <IconText
          onClick={this.setRibbonActive.bind(this, "Forms")}
          active={this.state.ribbonActive ===
            "Forms" ? true : false}
          ribbon text="Forms" icon="list-alt"
          size="2x" className="default" />
        <IconText
          onClick={this.setRibbonActive.bind(this, "Workflow")}
          active={this.state.ribbonActive ===
            "Workflow" ? true : false}
          ribbon text="Workflow" icon="code-fork"
          size="2x" className="default" />
        <IconText
          onClick={this.setRibbonActive.bind(this, "Ajax")}
          active={this.state.ribbonActive ===
            "Ajax" ? true : false}
          ribbon text="Ajax" icon="cloud"
          size="2x" className="default" />
        <IconText
          onClick={this.setRibbonActive.bind(this, "Embeds")}
          active={this.state.ribbonActive ===
            "Embeds" ? true : false}
          ribbon text="Embeds" icon="share-alt-square"
          size="2x" className="default" />
        <IconText
          onClick={this.setRibbonActive.bind(this, "Custom")}
          active={this.state.ribbonActive ===
            "Custom" ? true : false}
          ribbon text="Custom" icon="cog"
          size="2x" className="default" />
        <IconText
          onClick={this.setRibbonActive.bind(this, "Infographics")}
          active={this.state.ribbonActive ===
            "Infographics" ? true : false}
          ribbon text="Infographics" icon="eye"
          size="2x" className="default" />
        <IconText
          onClick={this.setRibbonActive.bind(this, "Charts")}
          active={this.state.ribbonActive ===
            "Charts" ? true : false}
          ribbon text="Charts" icon="bar-chart"
          size="2x" className="default" />
      </section>
    );
  }
}

```

Refactoring IconText.jsx for ribbon menu

To handle ribbon flag we will modify IconText. We make a few more modifications as we refactor IconText. We are extracting props using object rest spread destructuring assignment. This enables us to pass on props we do not consume within IconText to the rendered div native component.

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

const IconText = (props) => {
  const {
    icon, text, size, rotate,
    flip, inverse, slim, ribbon,
    active, ...rest } = props;

  let variation = '';

  variation += size ? ` fa-${size}` : '';
  variation += rotate ? ` fa-rotate-${rotate}` : '';
  variation += flip ? ` fa-flip-${flip}` : '';
  variation += inverse ? ` fa-inverse` : '';

  const iconClass = `fa fa-${icon}${variation}`;

  let renderIcon = null;

  if (slim) {
    renderIcon =
      <div {...rest}>
        <i className={iconClass}></i> {text}
      </div>;
  }

  if (ribbon) {
    renderIcon =
      <div {...rest} className=
        `${props.className} ribbon__menu text--center`>
        <i className={iconClass}></i>
        <p className={active
          ? 'ribbon__text--active'
          : 'ribbon__text'}>
          {text}
        </p>
      </div>;
  }

  if (!slim && !ribbon) {
    renderIcon =
      <div {...rest}>
        <i className={iconClass}></i>
        <h1>{text}</h1>
      </div>;
  }
}
```

```

    return (renderIcon);
  };

IconText.propTypes = {
  icon: PropTypes.string.isRequired,
  text: PropTypes.string.isRequired,
  className: PropTypes.string,
  size: PropTypes.oneOf(['lg', '2x', '3x', '4x', '5x', '']),
  rotate: PropTypes.number,
  flip: PropTypes.oneOf(['horizontal', 'vertical', '']),
  inverse: PropTypes.bool,
  slim: PropTypes.bool, // draw slim single-line InfoText
  ribbon: PropTypes.bool
};

IconText.defaultProps = {
  className: '',
  size: '',
  rotate: null,
  flip: '',
  inverse: false,
  slim: false,
  ribbon: false
};

export default IconText;

```

If we do not consume custom props for a component, before passing on the spread operator to rendered components, we get the [unknown props error](#) in our browser console.

We will need Babel Object Rest Spread transform to handle destructuring in this manner.

Installing this dependency and changing `.babelrc` to include [stage-2 preset](#), takes care of both class properties and object rest spread features.

```
npm install --save-dev babel-preset-stage-2
```

And update `.babelrc` accordingly.

```

{
  "presets": ["react", "es2015", "stage-2"],
  "env": {
    "development": {
      "presets": ["react-hmre"]
    }
  }
}

```


Search engine friendly URLs

Right now as you have noticed we are using hash-bang (#) based cryptic URLs. This works fine out of the box on our development environment. However this is not search engine friendly or even user friendly.

React Router's solution requires using [HTML5 pushState](#) browser API for handling URL history. Once you implement search engine friendly URLs, which is a matter of simply replacing `hashHistory` with `browserHistory`, the links will navigate just fine. However, if you load any of the deeper links directly by refreshing or bookmarking these, these will fail.

Direct access, to deeper links, works just fine when running Webpack development server. This is thanks to our webpack configuration where we set `historyApiFallback: true` flag in the `devServer` configuration.

```
// ... some code

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

// ... some code

ReactDOM.render(
  <Router history={browserHistory} routes={routeConfig} />,
  document.getElementById('app')
);
```

What is going on is that the single page app you have just written is handling the URL just fine as long as it is coming from the router, it is redirecting to `index.html`, and the JavaScript modules bundled within `index.html` handles the request correctly. When you call the deeper link urls directly, the HTTP server tries to look for that URL to serve, and does not find it. The request fails as `index.html` is no longer involved.

This according, to React Router documentation, unfortunately requires a server-side solution. We say

“unfortunately” because so far we were just fine building a single page app entirely on the front-end. It will definitely not be an optimum solution to plan a server-side strategy just on the basis of search engine friendly URLs requirement.

There is hope. One of the reasons we are choosing Firebase as our hosting provider is because Firebase knows how to work well with Single Page Apps. Although Firebase is comparable to other static website servers, it does allow for powerful server-side configuration.

Firebase also implements and distinguishes correctly between search engine friendly URLs and clean URLs.

Removing cryptic hash-bang based URLs and following human-readable URLs is **search engine friendly strategy**.

Removing .html at the end of a URL and serving it like you were serving a folder with default index.html is a **strategy called clean URLs**.

Right now we need the former. Here’s how we implement it. Within the `firebase.json` config file we add rules for `rewrites`. The rule in plain English says - “if server gets a request from any URL then treat it as a request meant for `/index.html`”.

```
{
  "firebase": "reactspeed",
  "public": "build",
  "ignore": [
    "firebase.json",
    "**/*.*",
    "**/node_modules/**"
  ],
  "rewrites": [ {
    "source": "**",
    "destination": "/index.html"
  } ]
}
```

That’s it! You will now be serving search engine friendly URLs instead of cryptic hash-bang ones.

If you are not using Firebase hosting then there might still be another solution. The [connect-history-api-fallback](#) NPM package offers a middleware to “proxy requests through a specified index page, useful for Single Page Applications that utilize the HTML5 History API”.

If you are hosting on GitHub, [spa-github-pages](#) may be another solution as GitHub Pages does not directly support Single Page Apps.

We have not tried these solutions, however if you want to give it a go and let us know, we will be happy to add such insights for our readers.

Important. Current [chapter demos site](#) does not support rewrites as it is hosted on GitHub Pages. The [final demo site](#) hosted on Firebase supports rewrites.

Handling router exceptions

We encounter an exception when user types a missing route itself. This can be handled in the routes definition as a general case by adding following route.

```
<Route path="*" component={MissingRoute} />
```

We are using `path="*" and defining a new MissingRoute component to render when the route does not exist.`

```
import React from 'react';
import Card from './Card';

function MissingRoute() {
  return (
    <section className="stripe">
      <Card plain className="col--half text--center">
        <h1>Sorry we could not find that</h1>
        <p className="subtext">
          Please use top navigation to visit what you are looking for.
        </p>
      </Card>
      <Card plain className="text--center">
        <a href="https://leanpub.com/reactspeedcoding"
          className="image__link">
          
        </a>
      </Card>
    </section>
  );
}

export default MissingRoute;
```

To handle cases where user reaches a route that is not defined or directly accesses a route, we also added to our webpack configuration to specify the `publicPath` so that `html-webpack-plugin` inserts our script modules with absolute paths and not relative paths to a missing or active route.

```
output: {
  // some code ...
  publicPath: '/',
},
```



Minimizing Bundle Size

React router documents strategy to minimize bundle size for the JS modules by directly referencing components we need from their respective public API modules. Read more [here](#), but please do NOT follow before reading this.

We tried doing so and compared the bundle sizes before and after. We notice no difference. In fact the overall bundle size of our JS modules is few bytes lighter when just importing directly from the main react router library. This is the **Webpack bundling magic** for you!

Behind the scenes React Router node_modules is >500KB in size. Our overall bundle sizes for app.js and vendor.js which include React, ReactDOM, and React Router modules used by the app are <40 KB and 190KB respectively! Webpack optimally extracts, minifies, and bundles our dependencies and imports, automatically for us, while we focus on writing simple, readable code.

This wraps up advanced usage of React Router. There are few more tricks we may use in other chapters including the following.

- Using ids with routing.
- Passing properties to components from routing configuration.

Now the landing page is wired to display two levels of navigation and our component hierarchy is well represented across easy to navigate categories.

Lint React Apps

This chapter will walk you through multiple lint tools and multi-device testing tool to catch most bugs while coding in editor.

We will learn the following topics in this chapter.

- Browsersync multi-device testing.
- JavaScript lint using eslint.
- Configuring eslint.
- Eslint command line interface.
- Eslint webpack integration.
- Fixing eslint reported problems.
- StyleLint for CSS.
- StyleLint CLI.
- Fixing StyleLint reported problems.
- Webpack integration for StyleLint.
- Complete Webpack lint config listing.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c09 origin/c09-lint-react-apps
npm install
npm start
```

Browsersync multi-device testing

Our single page app is mobile-web friendly. It responds to smaller or larger screen sizes and adapts the UI accordingly. As you continue mobile-web app development, you may want to test your app across multiple devices. [Browsersync](#) is a powerful tool that enables us to do that. It plays nicely with Webpack and Hot Reloading, while adding synchronized browsing of your app across connected devices.

Let us setup Browsersync for our development environment by installing following dependencies.

```
npm install --save-dev browser-sync
npm install --save-dev browser-sync-webpack-plugin
```

Lint config in webpack.lint.config.js

Let us duplicate our webpack.config.js as webpack.lint.config.js and add the following configuration for Browsersync. While Browsersync is not really a lint tool, it aids in visually inspecting the UI as we code it, similar to a lint tool as we will see in following sections. So we decide to carry Browsersync configuration along with other lint tools in the same file.

A benefit of separate lint configuration is on development time performance of running compilation. As we integrate our lint tools with the code editor, we may not want to run compile-time lint every time we update our code.

```
// some code...
const BrowserSyncPlugin = require('browser-sync-webpack-plugin');

// some code...
const HOST = process.env.HOST || 'localhost';
const PORT = process.env.PORT || 8080;
const PROXY = `http://${HOST}:${PORT}`;

// some code...
devServer: {
  historyApiFallback: true,
  hot: true,
```



```

    inline: true,
    progress: true,

    stats: 'errors-only',

    host: HOST,
    port: PORT,

    outputPath: BUILD
  },
  plugins: [
    new BrowserSyncPlugin(
      {
        host: HOST,
        port: PORT,
        proxy: PROXY
      },
      {
        reload: false
      }
    ),
  ],
  // some code...

```

We require the `browser-sync-webpack-plugin` plugin. Initialize few `const` variables for `host`, `port`, and `proxy`. Add the plugin for `BrowserSync`. And we are done!

Now all we need to do is add a script in `package.json` for running lint tests with `Browsersync`.

```

"lint": "NODE_ENV=lint webpack-dev-server --config webpack.lint.config\
.js",

```

When we want to lint our build we simply run `npm run lint` command. You will notice a different output than usual `webpack-dev-server` screen.

```

...
webpack: bundle is now VALID.
[BS] Proxying: http://localhost:8080
[BS] Access URLs:
-----
    Local: http://localhost:8081
    External: http://192.168.1.5:8081
-----
    UI: http://localhost:3001
    UI External: http://192.168.1.5:3001
-----

```

You now get 4 new URLs. Local and external are URLs you can use on multiple local browsers and multiple external devices respectively. UI related URLs enable you to configure

Browsersync runtime. In our setup, which is a Mac with XCode installed, we use iOS Simulator to run virtual device for our tests. Opening Safari browser on this virtual device and browsing to the external URL gives us access to our app. We can also connect to real physical devices on the same network this way.

What the UI configuration links allow you to do is configure if synchronized browsing will capture code sync, clicks, scroll, and form actions. This way you can test on one device and browser and notice synchronized actions automatically on all your other devices. Browsersync also allows other advanced features like simulating network speeds and remote debugging. Go on... explore away.

What is really cool is that hot reloading is still working with Browsersync active. So if you make any changes in your JSX, these should update on all devices on saving the changes. While maintaining your current UI state. Isn't this awesome!

Browsersync is a great time-saver for multi-device testing of mobile-web hybrid apps.

JavaScript lint using ESLint

Now that we are doing multi-device UI testing, let us move to the next stage. Let us test our JavaScript code for good coding practices, getting hints as we code in our editor. Many commercial IDEs like XCode, Visual Studio, and IntelliJ provide this kind of feature while you are coding, without the need to compile first.

We are using [Atom](#) editor from GitHub. Atom can be configured to support such hints as well. All you need to do is install and configure Eslint configuration for Atom to pick up. So let us do that.

```
npm install --save-dev eslint
npm install --save-dev eslint-plugin-react
npm install --save-dev eslint-loader
npm install --save-dev eslint-friendly-formatter
npm install --save-dev eslint-config-airbnb
npm install --save-dev eslint-plugin-import
npm install --save-dev eslint-plugin-jsx-a11y
npm install --save-dev babel-eslint
```

Here is a brief explanation of these dependencies.

- eslint - Core dependency.
- eslint-plugin-react - React specific linting rules for ESLint.
- eslint-loader - Webpack loader for eslint.
- eslint-friendly-formatter - Custom formatter for eslint results. Add a nice summary of errors or warnings in the end, plus highlights statement causing the error or warning.
- eslint-config-airbnb - Airbnb style guide based eslint rules.
- eslint-plugin-import - Required for linting of ES2015+ (ES6+) import/export syntax.
- eslint-plugin-jsx-a11y - Support for accessibility rules on JSX elements.

- babel-eslint - Parser to replace eslint default if we are using class properties, decorators, async/await, types.

Configuring eslint in eslintrc.js

Next we create default configuration of eslint by answering some questions about our development stack and environment. Are we using JSX, React, ES6, among others.

Init eslint configuration

```
node_modules/.bin/eslint --init
```

Resulting in the eslintrc.js configuration file at root of our development folder.

```
module.exports = {
  "extends": "airbnb",
  "plugins": [
    "react",
    "jsx-ally",
    "import"
  ]
};
```

We will make few changes after auto generating the config file.

```
module.exports = {
  "extends": "airbnb",
  "plugins": [
    "react",
    "jsx-ally",
    "import"
  ],
  "parser": "babel-eslint",
  "parserOptions": {
    "sourceType": "module"
  },
  "env": {
    "browser": true,
    "es6": true,
    "jquery": true
  },
  "rules": {
    "strict": 0,
    "no-console": "warn",
    "comma-dangle": [
      "warn",
      "never"
    ],
    "indent": [
      "warn",
      2
    ],
    "linebreak-style": [
      "warn",
      "unix"
    ]
  }
};
```

```

    ],
    "quotes": [
      "error",
      "single"
    ],
    "semi": [
      "error",
      "always"
    ],
    "no-unused-expressions": "warn",
    "no-useless-concat": "warn",
    "block-scoped-var": "error",
    "consistent-return": "error"
  }
};

```

- We are using the babel-eslint parser instead of eslint default parser.
- Changed indent and linebreak-style rules to warn instead of error and changed indent to 2 instead of 4 spaces.
- Add rule for comma-dangle to override Airbnb guidelines and revert to [eslint recommended guidelines](#).
- Add rule for expecting semi-colons for consistent code.
- We explain and add other flags in the following section on fixing eslint errors.

We can also ignore certain files for lint checks using .eslintignore file in the root of our development folder.

```

# /node_modules and /bower_components ignored by default

# ignore vendor js files
app/public/js/*.js

# ignore build files
build/

# coverage reports
coverage/

```

As a final step to getting eslint hints in our Atom editor, let us add an Atom package.

```
apm install linter-eslint
```

That's it. Now if you open a JSX/JS file in the Atom editor, you will notice eslint warnings and errors as you code.

Eslint command line interface

We can take this a step further and add a script command in the package.json to run lint as a separate command as needed.

We will create three commands for three types of result formats. For default stylish results `elint` can be used, for tabular format `elinttable` command, and for using eslint-friendly-formatter we can use the `elintsummary` command.

```
"elint": "eslint . --ext .js --ext .jsx --cache || true",
"elinttable": "eslint . --ext .js --ext .jsx --cache --format table ||\n  true",
"elintsummary": "eslint . --ext .js --ext .jsx --cache --format 'node_\nmodules/eslint-friendly-formatter' || true",
```

Now if you hit `npm run elint` in your terminal, you will notice eslint warnings and errors if any exist in your code.

When we first run `npm run elintsummary` our output looks something like this. Whoa! 245 problems. The `eslint-friendly-formatter` allows us to take a quick look at the summary results and reset any rules on/off/error/warn in the config before proceeding.

```
...
x 245 problems (219 errors, 26 warnings)
```

Errors:

```
54 http://eslint.org/docs/rules/object-curly-spacing
29 http://eslint.org/docs/rules/quotes
14 http://eslint.org/docs/rules/prefer-template
12 https://google.com/#q=react%2Fjsx-first-prop-new-line
11 http://eslint.org/docs/rules/space-before-function-paren
11 https://google.com/#q=react%2Fprop-types
9 http://eslint.org/docs/rules/prefer-const
9 http://eslint.org/docs/rules/keyword-spacing
8 http://eslint.org/docs/rules/semi
7 https://google.com/#q=react%2Fjsx-indent
6 https://google.com/#q=react%2Fjsx-closing-bracket-location
5 https://google.com/#q=react%2Fjsx-indent-props
4 https://google.com/#q=react%2Fprefer-stateless-function
4 https://google.com/#q=react%2Fno-unknown-property
4 http://eslint.org/docs/rules/eqeqeq
4 http://eslint.org/docs/rules/no-multi-spaces
3 http://eslint.org/docs/rules/no-unused-vars
3 http://eslint.org/docs/rules/space-before-blocks
2 https://google.com/#q=react%2Fjsx-space-before-closing
```

- 2 <http://eslint.org/docs/rules/object-shorthand>
- 2 <http://eslint.org/docs/rules/new-cap>
- 2 <http://eslint.org/docs/rules/no-extra-semi>
- 2 <http://eslint.org/docs/rules/space-infix-ops>
- 1 <http://eslint.org/docs/rules/padded-blocks>
- 1 <http://eslint.org/docs/rules/no-spaced-func>
- 1 <http://eslint.org/docs/rules/quote-props>
- 1 <https://google.com/#q=react%2Fsort-comp>
- 1 <https://google.com/#q=react%2Fjsx-no-bind>
- 1 <http://eslint.org/docs/rules/no-nested-ternary>
- 1 <https://google.com/#q=jsx-ally%2Fimg-redundant-alt>
- 1 <http://eslint.org/docs/rules/no-trailing-spaces>
- 1 <http://eslint.org/docs/rules/no-undef>
- 1 <http://eslint.org/docs/rules/prefer-arrow-callback>
- 1 <http://eslint.org/docs/rules/max-len>
- 1 <https://google.com/#q=react%2Fjsx-curly-spacing>

Warnings:

- 16 <http://eslint.org/docs/rules/indent>
- 7 <http://eslint.org/docs/rules/comma-dangle>
- 3 <http://eslint.org/docs/rules/func-names>

Fixing ESLint reported problems

When we first run eslint on ReactSpeed code we got 200+ problems (errors and warnings). Here is the workflow we are following to fix these.

With so many problems there is the temptation to use eslint -fix flag to automatically fix these. However, fixing these manually, we will learn more about our coding practices and correct these for good.

Overview of problems. We first run eslint command line interface using `npm run elintsummary` for eslint-friendly-formatter to tell us a snapshot of types of problems we encounter.

Overriding rules. Next we decide to override certain rules like `comma-dangle` and revert to using eslint recommended rule instead of Airbnb recommendation. This step brings down our problems count drastically. We also switch off some rules for JSX formatting so we balance code readability and verbosity.

```
"react/jsx-indent": 0,  
"react/jsx-first-prop-new-line": 0,  
"react/jsx-closing-bracket-location": 0,  
"react/jsx-no-bind": 0,  
"react/jsx-filename-extension": 0,  
"object-property-newline": 0,
```

Ignore files. We also run `npm run elinttable` for table format report, to take a quick look at all the files generating the problems. We decide to ignore vendor files located in `/app/public/js/` folder of our app and update the `.eslintignore` configuration.

Changing parser. We also change the parser to Babel from eslint default esprez parser to handle ES6/7 features like class properties.

Editor hints. Next we open our JSX files alphabetically in Atom editor to check eslint errors and warnings, while fixing these as per suggestions given by eslint or determining the right fix.

Hot testing. We could keep the webpack-dev-server running on one terminal window, and eslintsummary running on another as we fix the problems and test the app in our browser. Thanks to Hot Reloading, we don't need to refresh our browser or restart development server manually after every fix.

Disable PreLoader. During first run of eslint fixes when you have many problems, you may want to disable the Webpack preLoader to avoid these warnings crowding any app errors you may introduce during fixes.

Defer fix. Sometimes we encounter problems that need more reading for fixing. One such problem we encounter is the jsx-no-bind issue which requires [refactoring suggested here](#).

No-undef errors. We encounter no-undef errors that \$ is not defined at jQuery usage. This can be fixed by adding "jquery" : true within the env section of eslintrc file. Once we change the config file, just close and open the current JSX file to make the issue go away.

Selectively Disable Eslint. We can selectively disable Eslint rules for some of the code. We do this by adding a comment starting with eslint-disable for disabling all rules for code that follows. Enable it back with eslint-enable in comment. Use eslint-disable-line to disable all rules for a line of code. Suffix this with a specific rule name to only disable that rule for specific line of code.

```
'NODE_ENV': JSON.stringify('production') // eslint-disable-line quote-\nprops
```

We are disabling `quote-props` eslint rule check for this line of code which is used as-is from most recommended sources including Facebook React documentation.

Refactor Components. Next set of eslint fixes require refactoring components to Airbnb best practices.

Elegance of React

We have an awesome realization about the simple elegance of React at this point. When using Eslint in Atom editor we see hints for HTML indentation issues. Oh, actually these are hints for HTML-like native components part of JSX. Eslint has rules checking our logic as well as our presentation view at the same time. Of course you also catch subtle errors when you copy-paste HTML and ignore JSX specific camelCase attributes. Eslint is even checking accessibility rules within JSX attribute values. Never made possible before React!

Custom lint rules. We can add more eslint rules to our tests from the [full list of rules here](#). The [complexity rule](#) is an interesting one to add for code readability.

```
module.exports = {  
  // some code...  
  "rules": {  
    "complexity": ["warn", 2],  
    "no-unused-expressions": "warn",  
    "no-useless-concat": "warn",  
    "block-scoped-var": "error",  
    "consistent-return": "error"  
  }  
};
```

Eslint combined with Atom editor package and Webpack is a really powerful first line of defence to make your React code more readable and reliable. This is a really fast workflow, while you code each line! This will save you significant time in downstream testing, team on-boarding, releases, and refactoring.

ESLint webpack.lint.config.js integration

Let us automate our development pipeline even further. Let us add eslint to Webpack lint config.

```
// some code...
const LINT = path.join(__dirname, '.eslintrc.js');
// some code...
module.exports = {
  // some code...
  eslint: {
    configFile: LINT,
    emitError: true
  },
  module: {
    preLoaders: [
      {
        test: /\.jsx?$/,
        loaders: ['eslint'],
        include: APP
      }
    ],
  },
}
```

Adding the eslint preLoader to the Webpack config does the trick. Now every time you run your development server using `npm run lint` you will also run eslint. Your app will load in browser, however terminal window will show any eslint errors or warnings. We suggest using this option when you have run eslint once on your code and removed most errors and warnings. That way you can use this option as a quick check in case you introduce any new errors/warnings which you have missed while getting in-editor hints.

StyleLint for CSS

Just like the awesome ESLint tool for JavaScript, we have StyleLint for CSS.

By now we are familiar with the setup and workflow. We follow similar steps to integrate StyleLint.

Let us add the dependencies.

```
npm install --save-dev stylelint
npm install --save-dev stylelint-config-standard
npm install --save-dev stylelint-webpack-plugin
```

Here is what these dependencies do.

- stylelint - The core library offering lint rules processing and issue reporting for CSS.
- stylelint-config-standard - The standard shareable config for stylelint. Derived from the rules found within: The Idiomatic CSS Principles, Github's PrimerCSS Guidelines, Google's CSS Style Guide, Airbnb's Styleguide, and @mdo's Code Guide.
- stylelint-webpack-plugin - As the name suggest a StyleLint plugin for Webpack. Benefits over stylelint-loader alternative include processing @imports and partials and simpler Webpack setup.

Next let us create the StyleLint configuration in .stylelintrc file within the development folder root.

```
{
  "extends": "stylelint-config-standard"
}
```

That's it. The StyleLint config does not get any simpler. We can add custom rules as we progress in our workflow to fix any warnings. For now we are good to go to next step of creating a command line script.

StyleLint CLI

Let us add the StyleLint CLI shortcut to package.json file.

```
"slint": "stylelint ./app/styles/**/*.css ./app/style.css --syntax scs\
s || true"
```

Now we are ready to run StyleLint for the first time. Run `npm run slint` in your terminal.

```
app/styles/base/layout.css
18:3 * Expected empty line before declaration  declaration-empty-li\
ne-before

app/styles/base/mixins.css
5:3 * Expected empty line before nested      rule-nested-empty-li\
ne-before
rule                                           \

10:21 * Expected single space after ":" with  declaration-colon-sp\
ace-after
a single-line declaration                     \

app/styles/components/button.css
18:19 * Expected single space after ":" with  declaration-colon-sp\
ace-after
a single-line declaration                     \

34:11 * Expected a leading zero              number-leading-zero \

app/styles/components/card.css
10:3 * Expected empty line before declaration  declaration-empty-li\
ne-before

app/styles/components/nav.css
27:3 * Expected empty line before declaration  declaration-empty-li\
ne-before
32:3 * Expected empty line before declaration  declaration-empty-li\
ne-before

app/styles/components/ribbon.css
15:3 * Expected empty line before declaration  declaration-empty-li\
ne-before

app/style.css
3:1 * Unexpected empty line before at-rule    at-rule-empty-line-be\
fore
6:1 * Unexpected empty line before at-rule    at-rule-empty-line-be\
fore
12:1 * Unexpected empty line before at-rule   at-rule-empty-line-be\
fore
14:1 * Unexpected empty line before at-rule   at-rule-empty-line-be\
fore
```

We notice 13 odd problems reported by StyleLint, across 7 CSS files in our project.

Fixing StyleLint reported problems

Here is the workflow used to fix most of the StyleLint reported problems.

Overview of problems. As we ran StyleLint CLI we notice most of our reported problems repeat so Find and Replace in our editor will be our close ally in fixing these.

Editor hints. Let us start by integrating StyleLint with Atom editor and then walking through our CSS files one by one fixing reported problems.

```
apm install linter-stylelint
```

Done! Stepping through each CSS file took us less than 30 minutes to go from 70 problems to zero.

Overriding rules. We have not yet encountered any rule that needs overriding. However, when we do, we can add overrides in the `.stylelintrc` config file like so.

Of course this is just a sample, we are not adding these overrides to our project.

```
{
  "extends": "stylelint-config-standard",
  "rules": {
    "number-leading-zero": null
  }
}
```

Ignore files. The CLI and Webpack config allow us to include a list of files or paths with wildcards.

Hot testing. With the known issue in Hot Reloading not working with PostCSS, unfortunately we cannot do Hot Reloading while we fix the problems.

Webpack integration for StyleLint

Now that we have reached “problems zero”, let us integrate StyleLint with webpack development config so it runs every time we run the webpack-dev-server.

```
// some code...
const StyleLintPlugin = require('stylelint-webpack-plugin');
// some code...
const STYLELINT = ['./app/styles/**/*.css', './app/styles.css'];
// some code...
plugins: [
  new StyleLintPlugin({
    files: STYLELINT,
    syntax: 'scss'
  }),
// some code...
```

That’s all it takes. Adding the stylelint-webpack-plugin into our webpack development config.

Now when we run the `npm run lint` command, StyleLint will report any new issues which we do not catch with editor hints in the first place.

Complete webpack.lint.config.js listing

Here is the complete listing for webpack.lint.config.js as defined in prior sections.

```
/* eslint-disable import/no-extraneous-dependencies */

// Initialization
const webpack = require('webpack');

// Lint and sync
const BrowserSyncPlugin = require('browser-sync-webpack-plugin');
const StyleLintPlugin = require('stylelint-webpack-plugin');

// File ops
const HtmlWebpackPlugin = require('html-webpack-plugin');

// Folder ops
const CopyWebpackPlugin = require('copy-webpack-plugin');
const path = require('path');

// PostCSS support
const postcssImport = require('postcss-easy-import');
const precss = require('precss');
const autoprefixer = require('autoprefixer');

/* eslint-enable import/no-extraneous-dependencies */

// Constants
const APP = path.join(__dirname, 'app');
const BUILD = path.join(__dirname, 'build');
const STYLE = path.join(__dirname, 'app/style.css');
const PUBLIC = path.join(__dirname, 'app/public');
const TEMPLATE = path.join(__dirname, 'app/templates/index.html');
const NODE_MODULES = path.join(__dirname, 'node_modules');
const HOST = process.env.HOST || 'localhost';
const PORT = process.env.PORT || 8080;
const PROXY = `http://${HOST}:${PORT}`;
const LINT = path.join(__dirname, '.eslintrc.js');
const STYLELINT = ['./app/styles/**/*.css', './app/styles.css'];

module.exports = {
  // Paths and extensions
  entry: {
    app: APP,
    style: STYLE
  },
  output: {
    path: BUILD,
    filename: '[name].js',
    publicPath: '/'
  },
  resolve: {
    extensions: ['', '.js', '.jsx', '.css']
  },
  eslint: {
    configFile: LINT,
    emitError: true
  }
}
```

```

},
// Loaders for processing different file types
module: {
  preLoaders: [
    {
      test: /\.jsx?$/,
      loaders: ['eslint'],
      include: APP
    }
  ],
  loaders: [
    {
      test: /\.jsx?$/,
      loaders: ['babel?cacheDirectory'],
      include: APP
    },
    {
      test: /\.css$/,
      loaders: ['style', 'css', 'postcss'],
      include: [APP, NODE_MODULES]
    },
    {
      test: /\.json$/,
      loader: 'json',
      include: [APP, NODE_MODULES]
    }
  ]
},
// Configure PostCSS plugins
postcss: function processPostcss(webpack) { // eslint-disable-line \
no-shadow
  return [
    postcssImport({
      addDependencyTo: webpack
    }),
    precss,
    autoprefixer({ browsers: ['last 2 versions'] })
  ];
},
// Source maps used for debugging information
devtool: 'eval-source-map',
// webpack-dev-server configuration
devServer: {
  historyApiFallback: true,
  hot: true,
  progress: true,

  stats: 'errors-only',

  host: HOST,
  port: PORT,

  // CopyWebpackPlugin: This is required for webpack-dev-server.
  // The path should be an absolute path to your build destination.
  outputPath: BUILD
},
// Webpack plugins
plugins: [
  new StyleLintPlugin({
    files: STYLELINT,
    syntax: 'scss'
  }),

```

```

    new BrowserSyncPlugin(
      {
        host: HOST,
        port: PORT,
        proxy: PROXY
      },
      {
        reload: false
      }
    ),
    // Required to inject NODE_ENV within React app.
    // Redundant package.json script entry does not do that, but require
    for .babelrc
    new webpack.DefinePlugin({
      'process.env': {
        'NODE_ENV': JSON.stringify('development') // eslint-disable-li\
ne quote-props
      }
    }),
    new webpack.HotModuleReplacementPlugin(),
    new CopyWebpackPlugin([
      { from: PUBLIC, to: BUILD }
    ],
    {
      ignore: [
        // Doesn't copy Mac storage system files
        '.DS_Store'
      ]
    })
  ),
  new HtmlWebpackPlugin({
    template: TEMPLATE,
    // JS placed at the bottom of the body element
    inject: 'body'
  })
]
};

```

We now have a complete lint and Browsersync environment which enables fast in-editor pre-testing of our code against lint rules and enables multi-device testing while we develop and compile our app.

Test App Components

This chapter will walk you through multiple testing tools and strategies to make your React app more reliable, robust, and performant.

We will learn the following topics in this chapter.

- Mocha Chai Behavior-Driven Development.
- Enzyme React component testing.
- Sinon spy methods and events.
- Istanbul test coverage.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c10 origin/c10-test-app-components
npm install
npm start
```

Mocha Chai Behavior-Driven Development

With the lint tools we can test our code as we edit it. We can also run lint tests from the command line. Browsersync enables us to do UI testing on multiple devices.

We can now add to our testing strategies with [Behavior-Driven Development](#) (BDD) using Mocha and Chai.

With Mocha and Chai we can add unit tests to cover our app logic or functionality.

```
npm install --save-dev mocha
npm install --save-dev chai
```

Test 01_mocha_timeout.spec.js

BDD describes functionality of our app in the form of descriptive statements informing what the app is expected to do in certain conditions.

Let us write our first test suite using only Mocha BDD interface.

We will write `/test/01_stack/01_mocha_timeout.spec.js` file like so.

```
import { describe, it } from 'mocha';

/* eslint-disable func-names, prefer-arrow-callback */

describe('Mocha Timeout', function () {
  this.timeout(500);

  it('should take around 300ms', function (done) {
    setTimeout(done, 300);
  });

  it('should take around 250ms', function (done) {
    setTimeout(done, 250);
  });
});

/* eslint-enable func-names, prefer-arrow-callback */
```

Mocha supports testing asynchronous code using the callback method, like we are using `done` in this test suite.

Note that we are using ES5 to write our callback functions. Mocha explains why this is so.



Arrow functions and Mocha this context

Passing arrow functions to Mocha is discouraged. Their lexical binding of the `this` value makes them unable to access the Mocha context, and statements like `this.timeout(1000)` will not work inside an arrow function.

ESLint will complain about using ES5 callback functions but we disable it for this test.

Test 02_mocha_chai.spec.js

Let us write our second test using BDD interface provided by Mocha and BDD style assertions provided by Chai. This time we are using ES6 arrow functions as we are not using the Mocha this context.

```
import { expect } from 'chai';
import { describe, it } from 'mocha';

describe('Mocha Chai Demo', () => {
  describe('Array operations', () => {
    describe('#indexOf()', () => {
      it('should return -1 when the value is not present', () => {
        expect([1, 2, 3].indexOf(5)).to.equal(-1);
        expect([1, 2, 3].indexOf(0)).to.equal(-1);
      });
    });

    describe('length', () => {
      it('should return 0 when array is empty', () => {
        expect([]).length.to.equal(0);
      });
    });

    describe('length', () => {
      // Indicate pending test as a TODO for your collaborators
      it('should return number of elements in array');
    });
  });
});
```

Mocha BDD API includes describe, it, before, after, beforeEach, and afterEach. Chai BDD API includes expect and should flavors. The expect flavor is more browser friendly when compared with the should style of assertions. The third API Chai provides uses TDD or Test-Driven Development style assert statements.

We have created another test suite in this example, to demonstrate features of Mocha and Chai. The test suite comprises of three tests. Two of these tests are implemented with callback arrow function returning an assertion result. One of the tests is pending implementation. Just avoiding writing a callback function makes the test as pending.

Before we run these test suites we need to configure our package.json with a test script.

```
"test": "mocha --compilers js:babel-core/register --recursive || true"
```

We do not need to supply the path to our tests as Mocha picks up ./test/*.js by default for tests to run. The --recursive flag enables tests to be located in their own folders with the pattern ./test/**/*.js. Naming the folders and test files with a sequence prefix like 01 is completely a personal preference, not a requirement for Mocha. The sequencing helps run tests in the author intended order instead of alphabetical order.

Now we simply run the test using npm test command.

Mocha Timeout

```
  _/ should take around 300ms (302ms)
  _/ should take around 250ms (256ms)
```

Mocha Chai Demo

Array operations

#indexOf()

```
  _/ should return -1 when the value is not present
```

length

```
  _/ should return 0 when array is empty
```

length

```
    - should return number of elements in array
```

4 passing (612ms)

1 pending

So far our tests are not doing much for our app. In fact writing such tests helps us learn and experiment with the new language features, libraries, and APIs, that we may be using in our stack.

Let us start testing some of our React components.

Enzyme React component testing

Airbnb [Enzyme](#) is very popular tool for React component testing. In fact it is recommended by Facebook React core team and being considered to replace React [Test Utilities](#).

Enzyme implements three strategies for testing React components.

Shallow Rendering. When you want to constrain your testing to a single component and avoid traversing its child tree.

Full DOM Rendering. When you want to test your React code interactions with the DOM APIs or where your React code uses the lifecycle methods.

Static Rendering. When you want to analyze the results of static HTML that your React components render.

Enzyme plays well with Mocha, can be extended using custom Chai assertions and convenience functions, and use JSDOM JavaScript headless browser for creating a realistic testing environment.

Let us install the dependencies to make Enzyme work within our current setup.

```
npm install --save-dev enzyme
npm install --save-dev react-addons-test-utils
npm install --save-dev jsdom
npm install --save-dev babel-preset-airbnb
```

We update the airbnb preset in `.babelrc` config.

```
{
  "presets": ["react", "es2015", "stage-2", "airbnb"],
  "env": {
    "development": {
      "presets": ["react-hmre"]
    }
  },
}
```

```
  "plugins": ["transform-class-properties"]  
}
```

Note that we bring back "plugins": ["transform-class-properties"] as Enzyme does not seem to recognize the stage-2 preset.

JSDOM browser.js helper

For using JSDOM, Enzyme recommends loading a document into the global scope before requiring React for the first time.

It is very important that the below script gets run before React's code is run.

```
const jsdom = require('jsdom').jsdom;

const exposedProperties = ['window', 'navigator', 'document'];

global.document = jsdom('');
global.window = document.defaultView;
Object.keys(document.defaultView).forEach((property) => {
  if (typeof global[property] === 'undefined') {
    exposedProperties.push(property);
    global[property] = document.defaultView[property];
  }
});

global.navigator = {
  userAgent: 'node.js'
};
```

Now we update our test script like so.

```
"test": "NODE_ENV=test mocha test/browser.js test/**/*.spec.js
--compilers js:babel-core/register --recursive || true"
```

We introduce `NODE_ENV=test` so that `.babelrc` ignores the `react_hmre` preset used during development. Otherwise Mocha will complain. Scripts default to the `NODE_ENV=development` setting, if none is provided in the command line.

Test 01_workflow.spec.js

Let us add a simple React component test suite using all three Enzyme testing strategies including shallow, static, and full DOM rendering.

We write this test in `/test/02_components/01_workflow.spec.js` file.

```
import React from 'react';
import { expect } from 'chai';
import { describe, it } from 'mocha';
import { shallow, mount, render } from 'enzyme';
import Workflow from '../../app/components/Workflow.jsx';

describe('<Workflow />', () => {
  it('[Shallow] should render one .workflow component', () => {
    const wrapper = shallow(<Workflow />);
    expect(wrapper.is('.workflow')).to.equal(true);
  });

  it('[Shallow] should define a prop for steps', () => {
    const wrapper = shallow(<Workflow />);
    /* eslint-disable no-unused-expressions */
    expect(wrapper.props().steps).to.be.defined;
    /* eslint-enable no-unused-expressions */
  });

  it('[Static] should render one .workflow__text control', () => {
    const wrapper = render(<Workflow />);
    expect(wrapper.find('.workflow__text')).to.have.length(1);
  });

  it('[Full DOM] should increment state on clicking step button', () => {
    const wrapper = mount(<Workflow />);
    wrapper.find('.workflow__sequence').simulate('click');
    expect(wrapper.state('stepsIndex')).to.equal(1);
  });

  it('[Full DOM] should render new sequence number on clicking step button', () => {
    const wrapper = mount(<Workflow />);
    wrapper.setState({ stepsIndex: 1 });
    wrapper.find('.workflow__sequence').simulate('click'); // stepsIndex = 2
    expect(wrapper.find('.workflow__sequence').text())
      .to.equal('3'); // Sequence = stepsIndex + 1
  });
});
```

Note that Enzyme API offers BDD style traversal of our component hierarchy and internals.

Our first test checks if Workflow component renders correctly with `.workflow` class.

Next test checks to see if the Workflow component defines a property called `steps`.

Third test does static rendering to check if one of the child components are rendered using the `className` associated with this component.

Final two tests are simulating state management, UI interaction (button click), and analyzing the resulting HTML structure using full DOM render option of Enzyme.

We also setup our timeout test to `skip` the test so we don't have to wait for 600ms for test suites to complete. Skipped tests also show up as pending.

```
describe.skip('Mocha Timeout', function () {
```

After running `npm test`, the results appear in our terminal.

```
Mocha Timeout
```

- should take around 300ms
- should take around 250ms

```
Mocha Chai Demo
```

```
  Array operations
```

```
    #indexOf()
```

```
    _/ should return -1 when the value is not present
```

```
    length
```

```
    _/ should return 0 when array is empty
```

```
    length
```

- should return number of elements in array

```
<Workflow />
```

```
  _/ [Shallow] should render one .workflow component
```

```
  _/ [Shallow] should define a prop for steps
```

```
  _/ [Static] should render one .workflow__text control
```

```
  _/ [Full DOM] should increment state on clicking step button
```

```
  _/ [Full DOM] should render new sequence number on clicking step button
```

```
ton
```

```
7 passing (221ms)
```

```
3 pending
```

Sinon spy methods and events

We can extend our test stack with Sinon JS to add spy capabilities for testing events and method calls, among other component internals.

```
npm install --save-dev sinon
```

Let us import sinon to our workflow spec.

```
import sinon from 'sinon';
```

Now we add a test to spy on `cycleScenario` method. Test if this method is called once when we simulate click on the scenario button.

```
it('[Sinon, Full DOM] should call cycleScenario on clicking scenario button', () => {
  sinon.spy(Workflow.prototype, 'cycleScenario');
  const wrapper = mount(<Workflow />);
  wrapper.find('.workflow__symbol').simulate('click');
  expect(Workflow.prototype.cycleScenario.calledOnce).to.equal(true);
});
```

Once we run `npm run test` command we notice our new test is passing.

Istanbul code coverage

One more important part of our test stack is code coverage instrumentation. Knowing what parts of our code are covered by the test suites we have defined is an important step to writing robust apps.

IstanbulJS is a popular instrumentation tool for automatically gathering code coverage report based on Mocha test runs.

First we install the babel version of istanbul and babel-cli to ensure we are able to process babel processed code via istanbul.

```
npm install --save-dev babel-istanbul
npm install --save-dev babel-cli
```

Next we create our package.json script to run code coverage report.

```
"cover": "NODE_ENV=test babel-node
node_modules/.bin/babel-istanbul cover _mocha --
--require test/browser.js test/**/*.spec.js
--reporter dot || true",
```

Note: Please interpret this command as single line of code. We have split it with new lines for ease of readability.

Yes, a very loaded command indeed. It can be interpreted as follows.

- Set `NODE_ENV=test` so that `.babelrc` development configuration is ignored.
- Run `babel-istanbul cover` command using `babel-cli` (`babel-node`).
- Run the `cover` command on results from the `mocha` executable (with underscore).
- Run `mocha` executable requiring `test/browser.js` to run first.
- Run `mocha` on `test/**/*.spec.js` test suites.

- Replace test suites with `dots` for less verbose report.
- Using `|| true` - do not report any error message when tests fail, just report failure message.

Once this is setup we can run the code coverage report using `npm run cover` command.

The code coverage summary shows up in a few seconds on the terminal.

```
===== Coverage summary =====  
Statements   : 100% ( 33/33 )  
Branches     : 80% ( 8/10 )  
Functions    : 100% ( 4/4 )  
Lines        : 100% ( 28/28 )  
=====
```

You can also browse to `./coverage/lcov-report/index.html` to view the HTML report which enables drill-down to tested source files. Lines of code are highlighted based on coverage results.

We have setup a comprehensive command line based test stack using Enzyme for React component testing with Mocha and Chai BDD API for describing, running, reporting, and asserting tests, Sinon for spying methods and events, Istanbul code coverage, complete with JSDOM headless browser testing.

Refactor Existing Components

Refactoring is an essential part of iterative development. React is ideal for iterative refactoring of your code because of component based development.

You will learn following concepts in this chapter.

- ES5 to ES6 React component definition.
- Testing and refactoring.
- Refactoring for converting standard React apps to Redux.
- Refactoring for optimizing React apps.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c11 origin/c11-refactor-existing-components
npm install
npm start
```

ES5 to ES6 conversion in TodoList.jsx

Most React content on the Web still refers to ES5 code. This section is about refactoring existing ES5 React component to ES6 code.

Let us use the TodoList example from Facebook React website homepage and refactor it to ES6 components.

```
var TodoList = React.createClass({
  render: function() {
    var createItem = function(item) {
      return <li key={item.id}>{item.text}</li>;
    };
    return <ul>{this.props.items.map(createItem)}</ul>;
  }
});

var TodoApp = React.createClass({
  getInitialState: function() {
    return {items: [], text: ''};
  },
  onChange: function(e) {
    this.setState({text: e.target.value});
  },
  handleSubmit: function(e) {
    e.preventDefault();
    var nextItems = this.state.items.concat([{text: this.state.text, id: Date.now()}]);
    var nextText = '';
    this.setState({items: nextItems, text: nextText});
  },
  render: function() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={this.handleSubmit}>
          <input onChange={this.onChange} value={this.state.text} />
          <button>'Add #' + (this.state.items.length + 1)</button>
        </form>
      </div>
    );
  }
});

ReactDOM.render(<TodoApp />, mountNode);
```

We will create two new components, TodoList and TodoApp using ES6 code.

Our TodoList component is a pure function or a stateless component.

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

const TodoList = ({ items }) => {
  const createItem = (item) => <div key={item.id}>{item.text}</div>;
  return <div>{items.map(createItem)}</div>;
};

TodoList.propTypes = { items: PropTypes.array.isRequired };

export default TodoList;
```

We are using arrow functions to render createItem with less code. We also make our code safer and easier to read using immutable const instead of var in our ES5 sample.

TodoApp.jsx owner component

For our TodoApp component we use ES6 class definition.

```
import React from 'react';
import TodoList from './TodoList';
import Button from './Button';
import Input from './Input';
import InputField from './InputField';

class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = { items: [], text: '' };
    this.onChange = this.onChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  onChange = (e) => this.setState({ text: e.target.value });
  handleSubmit = (e) => {
    e.preventDefault();
    const nextItems = this.state.items
      .concat([ { text: this.state.text, id: Date.now() } ]);
    this.setState({ items: nextItems, text: '' });
  }
  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={this.handleSubmit}>
          <Input>
            <InputField
              onChange={this.onChange}
              value={this.state.text} />
            <Button color="primary"
              label={`Add # ${this.state.items.length + 1}`} />
          </Input>
        </form>
      </div>
    );
  }
}
```

```
export default TodoApp;
```

We have added a constructor to set state and bind event handler methods. Note that `onChange` and `handleSubmit` elegantly use ES6 arrow functions. Our `render()` code now uses ES6 template literals for dynamic button text. We are also replacing any instances of mutable `var` with safer `const` keyword.

We are also reusing the `Input`, `InputField`, and `Button` components we designed in the earlier chapters.

Adding TodoApp to AboutCustom.jsx

Now we can render our component in the custom components page.

```
import React from 'react';
import Card from './Card';
import World from './World';
import TodoApp from './TodoApp';

const AboutCustom = () => (
  <section>
    <section className="stripe">
      <Card
        className="col--one-third text--center back--white">
        <TodoApp />
      </Card>
      <Card plain className="col--half text--center">
        <h1>ES5 to ES6 Conversion</h1>
        <p className="subtext">
          This custom component is conversion from ES5 to ES6,
          reusing ReactSpeed UI library.
        </p>
      </Card>
    </section>
    <section className="stripe pattern-agsquare">
      <Card plain className="col--half text--center">
        <h1>Custom React Component</h1>
        <p className="subtext">
          This custom component demonstrates props, state,
          and ES6 classes.
        </p>
      </Card>
      <Card
        className="col--one-third text--center back--white">
        <World />
      </Card>
    </section>
  </section>
);
```

```
export default AboutCustom;
```

Done! You just refactored ES5 code to ES6.

Testing and refactoring

Testing and refactoring goes hand in hand. Depending on your choice of TDD or BDD, you will be writing tests along with code to pass these tests, or test covering code that you have already written. During the test workflow refactoring plays a role at various stages.

Spec. When prototyping your component you may just want to start with a spec written in Mocha. As you implement your component, you may refactor the spec based on design decisions.

Prototyping. Using tools like Kadir Storybook you may be isolating visual or usability testing of your component. Results of your tests will enable you to interactively refactor your code and styles.

Lint. ESLint and StyleLint editor integration prompts you in-line while editing your code with suggestions to refactor and improve existing or new code you might be writing.

Enzyme. When doing component testing using a tool like Enzyme, you may want to write tests that need to pass for component rendering in isolation or as an entire hierarchy. This may prompt refactoring your code or your tests may need adjusting to new specs.

Coverage. When running test coverage reports you may encounter code that is not covered by your existing tests. You may refactor your code or add new tests to achieve acceptable levels of code coverage.

Sync and Hot Reloading. Using tools like Browsersync and Hot Reloading, you could be testing your app on multiple devices while adjusting and refactoring styles for responsive design.

Refactoring for converting standard React apps to Redux

Redux is an important and popular part of React stack for relatively complex apps. We cover Redux refactoring strategy in detail within the **Redux state container** chapter.

Broadly we apply following strategies to move from a standard React app to a Redux enabled app.

- Identify state tree definition for UI state of your app components.
- Using Mocha write prototype spec for Redux patterns implementing the state tree.
- Determine Redux actions refactoring state definition within the non-Redux app.
- Implement Reducers refactoring event handlers within the non-Redux app.
- Create the store for managing entire app state in one place.
- Write test suite to cover store, actions, and reducers.
- Apply Redux optimization patterns for ES6 and reducer composition.
- Define component hierarchy specification.
- Prototype component hierarchy reusing existing component library.
- Implement data fixtures for prototype and Redux app.
- Extract presentational components from prototype.
- Extract container components connecting presentation components to Redux.
- Refactor app to replace prototype with presentational and container components.
- Hydrate the Redux store with fixtures data.
- Refactor test suite for Redux patterns.

- Add any new components to Redux app incrementally.

Refactoring for optimizing React apps

During the course of this book we followed several best practices for writing optimized React code. This section summarizes various optimization strategies.

ES6. Arrow functions offer shorthand to writing JS functions. Template literals make string concatenation code more readable and manageable. Spread operators remove significant boilerplate code. Class definitions and property initializes make our component definitions more robust.

Webpack. Webpack runs several optimizations on our app including bundling, packaging, minifying, and auto-generating code based on templates. Refactoring your JS code to use import, export, and modules, around single-area-of-concern per module, helps Webpack deliver optimized payload. Refactor your vendor dependencies suiting Webpack separation of app and vendor JS files for even faster first time loading of your single page app.

PostCSS. Refactoring your CSS styles to follow Airbnb guidelines and PostCSS SASS-like syntax encourages reuse, standardization, and readability of your code.

Inline CSS and SVG. Refactoring to inline CSS strategically is a strategy worth considering. When you combine this strategy with using SVG to replace external libraries like for example icon fonts, this can yield impressive results. You can optimize by reducing DNS lookups, reducing CSS load time, and overall payload of your app. We will explore this strategy in detail in the section **Refactoring Font Awesome to SVG icons** which follows.

Refactoring Font Awesome to SVG icons

So far our app is using Font Awesome font icons library. We used one of the most optimum ways to include this library in our app. Using it over BootstrapCDN and loading the minified version of Font Awesome library in our index.html head section. There are some challenges that remain in this approach.

- While Font Awesome is highly optimized library it still adds around 28.7K payload (7.4K GZip) for first time users.
- It adds an additional DNS lookup to our overall app loading time.
- It exhibits Flash Of Missing Icons when on slower connection.
- We are hardly using 20 odd icons from the large array of Font Awesome icons. There must be a leaner way of loading icons.
- While Font Awesome is very flexible, we are stuck with fixed number of sizing options.
- If we were to load Font Awesome using Webpack we will need additional configuration.
- If we wanted to customize Font Awesome icons we will need to add configuration and build options to Webpack.

There is an elegant solution. Thanks to David Gilbertson's post on [Icons as React Components](#), we are able to create a new custom component, which helps us solve all of the above challenges.

The strategy is simple, building on the ideas from the post.

- Use SVG icons instead of icon fonts.

- Get SVG data for icons from Creative Commons sources like [IcoMoon](#).
- Use this icon data as fixture for our app where we need icons.
- Refactor IconText component to IconSvg component.
- Refactor the app to render the new component instead of using Font Awesome.

IconSvg.jsx custom component

/app/components/IconSvg.jsx refactored component

```
import React from 'react';
const { PropTypes } = React;

const IconSvg = props => {
  const styles = {
    svg: {
      display: 'inline-block',
      verticalAlign: 'baseline'
    },
    path: {
      fill: 'currentColor'
    },
    font: {
      fontSize: props.slim ? props.size * 120 / 100 : props.size * 40 \
/ 100
    }
  };
  const renderText = props.slim
    ? <span className={props.textColor}> {props.text} </span>
    : <div className={props.textColor}>{props.text}</div>;
  const renderClassName = props.className ? props.className : props.co\
lor;

  return (
    <span className={renderClassName} style={styles.font}>
      {props.left ? renderText : ''}
      <svg
        style={styles.svg}
        width={`${props.size}px`}
        height={`${props.size}px`}
        viewBox="0 0 1024 1024"
      >
        <path
          style={styles.path}
          d={props.icon}
        ></path>
      </svg>
      {props.left ? '' : renderText}
    </span>
  );
};

IconSvg.propTypes = {
  icon: PropTypes.string.isRequired,
```

```

    size: PropTypes.number,
    color: PropTypes.string,
    text: PropTypes.string,
    slim: PropTypes.bool,
    textColor: PropTypes.string,
    className: PropTypes.string,
    left: PropTypes.bool
  };

  IconSvg.defaultProps = {
    size: 16,
    color: 'primary-text',
    text: '',
    slim: false,
    textColor: '',
    className: '',
    left: false
  };

```

```
export default IconSvg;
```

There is a lot going on in this component.

- We are using inline styles to enable in-component calculation of style properties like in case of `fontSize` property.
- We are rendering several variations of the icon. Just the icon only. Icon with text inline (slim). Icon with text below it.
- We are also influencing SVG styling based on our CSS variables using `currentColor` property variable.
- Finally we are using SVG to render the icon itself.

Icon data fixture in icons.js

The `/app/fixtures/icons.js` SVG data fixture file is lengthy so does not merit listing in the book. You can access it in the GitHub repo.

Structure of the icons fixture follows `ICON_NAME: 'SVG DATA'` pattern as shown in the example here.

```

const ICONS = {
  ARROW_RIGHT: 'M992 512l-480-480v288h-512v384h512v288z',
  ITALIC: 'M896 64v64h-128l-320 768h128v64h-448v-64h128l320-768h-128v-\
64z',
};

```

```
export default ICONS;
```

We can selectively start replacing IconText like so.

```
import React from 'react';

import IconSvg from './IconSvg.jsx';
import ICONS from '../fixtures/icons.js';

// ... some code

<Card className="col--quarter text--center back--white">
  <IconSvg color="warning" icon={ICONS.DATABASE}
    size={100} text="Firebase React Integration" />
</Card>

// ... some code
```

So, what are the optimization improvements of doing this refactor? Note that these metrics will require completely replacing IconText component with the new IconSvg component. We have not done so in the code along branch, however here are the potential benefits.

- Page size improving from 219KB to 149KB. Reducing our payload by 32% is awesome savings.
- External requests improving from 11 to 9 requests improves our responsiveness.
- No more Flash Of Missing Icons, as our React app now includes the icons as SVG. This improves our user experience.
- Our developer experience also improves on several fronts. We can auto-lookup our icons now from the editor as these are defined as constants. We have more freedom to style the icons and infographics with pixel-level sizing.
- Our designers can also add custom SVG icons in the future.

React app optimization is a growing and interesting topic. We will continue to update this chapter with new ideas and tools as we incorporate these into our workflow.

Redux State Container

When designing React apps, UI state becomes an important concern. How state is managed across your component hierarchy during your app lifecycle gets complex fast as the number of components and user interactions increase.

React is the *View* for your app. We need an equally elegant solution for managing the *State* and *Data Flow* within your app. This becomes even more important concern for Single Page Apps as efficient state management leads to performant user experience.

Fortunately Facebook has already thought this through for us. They have introduced to the open source, [Flux application architecture](#) for building user interfaces.



Flux is the application architecture that Facebook uses for building client-side web applications. It complements React's composable view components by utilizing a unidirectional data flow. It's more of a pattern rather than a formal framework... Flux applications have three major parts: the dispatcher, the stores, and the views (React components). - <http://facebook.github.io/flux/>

Redux is a very popular library in the React ecosystem. It evolves from Flux design patterns and is based on [Elm architecture](#) which is a simple pattern for infinitely nested components.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this

chapter, install and start the app to launch the local version in your default browser.

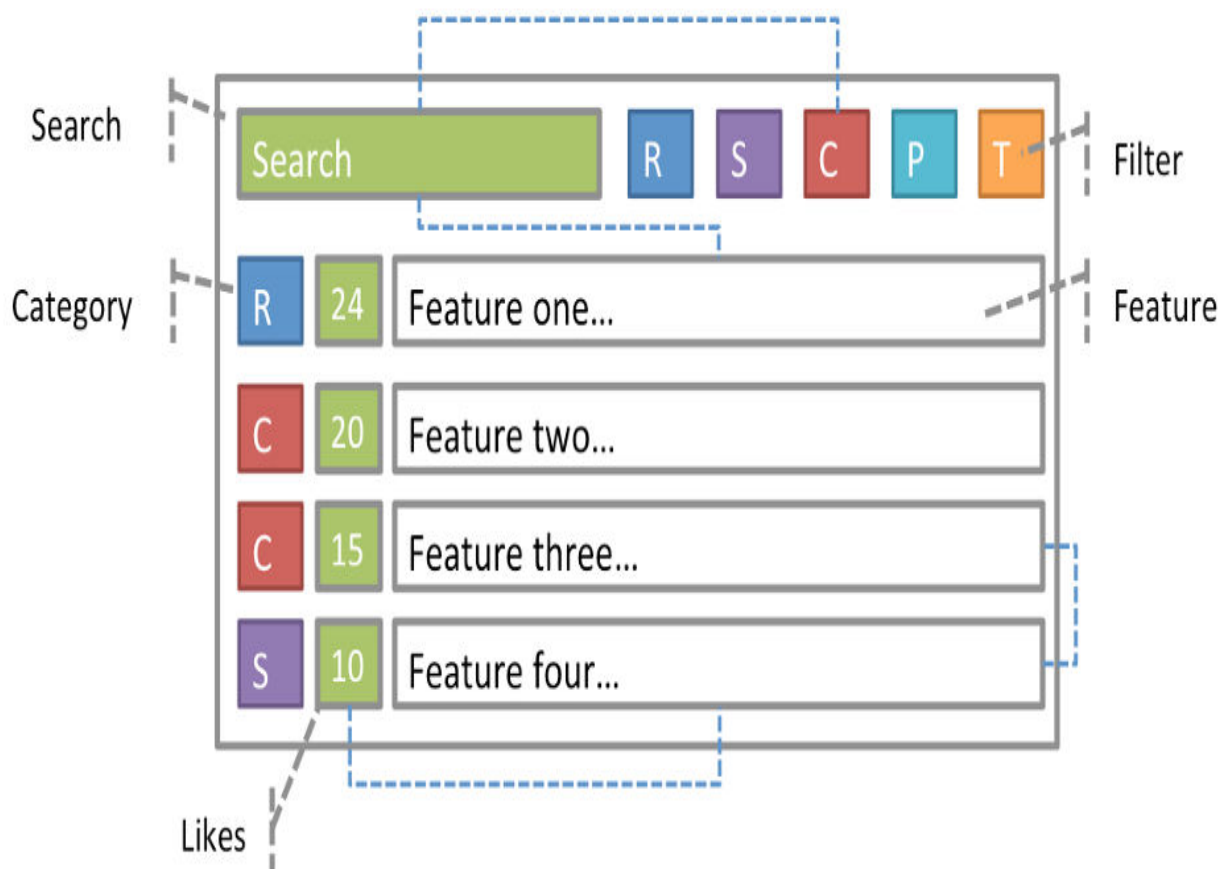
Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c12 origin/c12-redux-state-container
npm install
npm start
```


The Roadmap app

To help understand this important chapter, let us create a relatively complex app to manage the the roadmap for ReactSpeed book and companion code. We want to list upcoming, recent content and code features. Users should have the capability to *Like* features they want to see first or jump to live feature demos and content.



Roadmap app wireframe

Our roadmap app will require a Feature component to render the individual feature. It will also require a FeatureList component to manage list of feature components. We will add a FeatureSearch component to search listed features. A CategoryFilter component will list features by categories like components, styles, chapters, sections, and strategies.

You will note that various components within this app will interact with each other (blue dashed lines in the wireframe). Changing filters will interact with search, reducing the scope of what can be searched. Search will interact with features, showing only features that match the text entered in search. Number of likes will interact with order of features.

Our app will also maintain several UI states. Some candidate states could be, active filter, order of features, search text, and number of *Likes*.

Redux basics

Redux is remarkably simple API and an elegant architectural design pattern at the same time.

Redux introduces three important and inter-related concepts.

Stores for your state data.

- All your app state is a single data structure.
- Think a single JSON document representing the entire state of your app.
- State is immutable.
- Creating new state makes a copy of existing state with the new changes.

Actions for managing data flow.

- Actions usually define functions with two arguments. First, type of state change, and second, change data.
- Type of state change is usually defined as strings or constants representing strings, named after the action performed on the state tree leading to the data change.
- The data change can be represented as a single value, an object with multiple values, or can even be implicitly understood from the action type. Examples: INCREMENT_LIKES, TOGGLE_TODO.
- Actions are the only way to communicate change within the state tree.

Reducer functions.

- Reducers are written as pure functions.
- Reducers manipulate state based on Actions.

- Fixed input to the reducer function are two arguments. First, the existing state tree, and second, the action to be applied on the state tree.
- Reducer returns a predictable output as the new state tree after applying the action.
- As reducer is a pure function, it should not have any unpredictable behavior including data manipulation based on random functions or timestamp. Input should determine the output predictably.

We will discover other advanced concepts that elaborate the three core concepts, as we go through coding the Roadmap app in the Redux way.

State tree definition

Redux is all about managing state and data flow within our app. A good first step is to define the state tree for our app. Redux treats the entire state of your app as single data structure. Think of the state tree as a JSON document describing your app.

Drawing this JSON document for our Roadmap app will make things clearer.

We will maintain state for list of features with feature title, category, and likes count. We will store text entered by user in search box. We will also capture category selected by the user for filtering features.

When our app starts the initial state has defaults set for various states maintained by our app.

Roadmap state tree, initial state

```
{
  features: [],
  searchText: '',
  categoryFilter: 'SHOW_ALL'
}
```

New features can be added to our app, updating the features list within our state tree.

Roadmap state tree, new features

```
{
  features: [
    { title: 'Navigation', likes: 0, category: 'COMPONENT' },
    { title: 'Redux state container', likes: 0, category: 'CHAPTER' },
    { title: 'Roadmap', likes: 0, stage: 'APP' }
  ],
  searchText: '',
  categoryFilter: 'SHOW_ALL'
}
```

As our users like the features the respective likes count increments.

Roadmap state tree, trending

```
{
  features: [
    { title: 'Navigation', likes: 0, category: 'COMPONENT' },
    { title: 'Redux state container', likes: 2, category: 'CHAPTER' },
    { title: 'Roadmap', likes: 1, stage: 'APP' }
  ],
  searchText: '',
  categoryFilter: 'SHOW_ALL'
}
```

As users select a new category filter or enter search text, respective state gets updated.

Roadmap state tree, live

```
{
  features: [
    { title: 'Navigation', likes: 0, category: 'COMPONENT' },
    { title: 'Redux state container', likes: 2, category: 'CHAPTER' },
    { title: 'Roadmap', likes: 1, stage: 'APP' }
  ],
  searchText: 'new search text',
  categoryFilter: 'SHOW_CHAPTERS'
}
```

We can continue to evolve our state tree by adding other states for our app. Ideally you would do this as you evolve the design of your app. For now we can move to the next stage quickly.

Redux spec in 02_roadmap_redux.spec.js

Before jumping into creating our Redux app, we can specify how the Redux stores, actions, and reducers behave as the state tree changes within our app. We can do so using our test environment setup in the **Test App Components** chapter.

```
import { describe, it } from 'mocha';

describe('Roadmap Redux Spec', () => {
  it('should get initial state for store');
  it('should add first feature of COMPONENT category');
  it('should initialize first feature with default state');
  it('should increment likes count for first feature');
  it('should set a new categoryFilter');
  it('should add second feature of CHAPTER category');
  it('should add third feature of APP category');
  it('should set new search text');
});
```

When we run this test using `npm test` we notice following test results.

...

```
Roadmap Redux Spec
  - should get initial state for store
  - should add first feature of COMPONENT category
  - should initialize first feature with default state
  - should increment likes count for first feature
  - should set a new categoryFilter
  - should add second feature of CHAPTER category
  - should add third feature of APP category
  - should set new search text
```

Redux apps are best designed in the order Actions > Reducers > Store > Components. So it is a good idea to develop tests for our Redux app as we evolve the app, through these stages. We do not have to wait for the components to render to see results of our Redux design.

Actions for Roadmap in /actions/roadmap.js

Actions in Redux represent data payloads sent between your app and the Redux store. Actions are the only way you can update your state tree within the store.

We can derive some Redux Actions from the state tree we just defined.

```
/*
 * action types
 */

export const ADD_FEATURE = 'ADD_FEATURE';
export const SET_CATEGORY_FILTER = 'SET_CATEGORY_FILTER';
export const LIKE_FEATURE = 'LIKE_FEATURE';
export const SEARCH_TEXT = 'SEARCH_TEXT';
/*
 * other constants
 */

export const CategoryFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPONENTS: 'SHOW_COMPONENTS',
  SHOW_CHAPTERS: 'SHOW_CHAPTERS'
};

export const Categories = {
  CHAPTER: 'CHAPTER',
  COMPONENT: 'COMPONENT',
  APP: 'APP'
};

/*
 * action creators
 */

export function addFeature(title, category) {
  return {
    type: ADD_FEATURE,
    category,
    title };
}

export function setCategoryFilter(filter) {
  return { type: SET_CATEGORY_FILTER, filter };
}

export function setSearchText(text) {
  return { type: SEARCH_TEXT, text };
}

export function likeFeature(index) {
  return { type: LIKE_FEATURE, index };
}
```


Notice that we have defined constants for our state values including categories and category filters. We are maintaining consistency with definition of our action types as string constants. This is not a Redux requirement.

Reducers in /reducers/roadmap.js

Now that we have some actions defined, we can write what happens to our state tree when these actions are called. We do this writing pure functions called reducers which take two arguments, the current state tree and the action to perform on the state tree. The reducer then returns the new state tree.

```
import * as actions from '../actions/roadmap';

const initialState = {
  searchText: '',
  categoryFilter: actions.CategoryFilters.SHOW_ALL,
  features: []
};

export default function roadmapApp(state = initialState, action) {
  switch (action.type) {
    case actions.LIKE_FEATURE:
      return Object.assign({}, state, {
        features: state.features.map((feature, index) => {
          if (index === action.index) {
            return Object.assign({}, feature, {
              likes: feature.likes + 1
            });
          }
          return feature;
        })
      });
    case actions.SEARCH_TEXT:
      return Object.assign({}, state, {
        searchText: action.text
      });
    case actions.SET_CATEGORY_FILTER:
      return Object.assign({}, state, {
        categoryFilter: action.filter
      });
    case actions.ADD_FEATURE:
      return Object.assign({}, state, {
        features: [
          ...state.features,
          {
            title: action.title,
            category: action.category,
            likes: 0
          }
        ]
      });
    default:
      return state;
  }
}
```

The `Object.assign()` method creates copy of existing state into new state while updating the new changes suggested by the action.

Store in `/store/roadmap.js`

Now that we have defined the Reducer and Action, it is time to create the Store.

Before we do so, we need to install Redux as a run-time dependency.

```
npm install --save redux
```

This is the simplest part of writing a Redux app.

```
import { createStore } from 'redux';  
import roadmapApp from '../reducers/roadmap';  
const store = createStore(roadmapApp);  
export default store;
```

We have exported our store so that we can use it within other parts of our app including our test suite.

Test store, actions, and reducers

Let us elaborate our test spec, as defined in `/app/test/02_components/02_roadmap_redux.spec.js`, with assertions to test our Redux app so far.

We start by importing store and actions. Our first test gets the initial state from our store and checks for default values for our initial state.

The second test adds a new feature with a certain category. We expect the state to change to new features count and match the new category for the feature added. The third test checks if default value is set for the Likes state, where we have not supplied it for this new feature. Fourth test updates this Likes state and checks if the state has changed as expected.

Fifth test sets `categoryFilter` state to a new value and checks for the state change.

Sixth, and seventh tests add two additional features.

The eighth and final test sets `searchText` state.

```
import { describe, it } from 'mocha';
import { expect } from 'chai';
import store from '../../app/store/roadmap';
import * as actions from '../../app/actions/roadmap';

describe('Roadmap Redux', () => {
  it('should get initial state for store', () => {
    expect(store.getState().features.length).to.equal(0);
    expect(store.getState().categoryFilter)
      .to.equal(actions.CategoryFilters.SHOW_ALL);
    expect(store.getState().searchText)
      .to.equal('');
  });
  it('should add first feature of COMPONENT category', () => {
    store.dispatch(
      actions.addFeature('New Component Feature',
        actions.Categories.COMPONENT)
    );
    expect(store.getState().features.length).to.equal(1);
    expect(store.getState().features[0].category)
```

```

        .to.equal(actions.Categories.COMPONENT);
    });
    it('should initialize first feature with default state', () => {
        expect(store.getState().features[0].likes).to.equal(0);
    });
    it('should increment likes count for first feature', () => {
        store.dispatch(actions.likeFeature(0)); // likes = 1
        store.dispatch(actions.likeFeature(0)); // likes = 2
        expect(store.getState().features[0].likes).to.equal(2);
    });
    it('should set a new categoryFilter', () => {
        expect(store.getState().categoryFilter)
            .to.equal(actions.CategoryFilters.SHOW_ALL);
        store.dispatch(actions
            .setCategoryFilter(actions.CategoryFilters.SHOW_COMPONENTS));
        expect(store.getState().categoryFilter)
            .to.equal(actions.CategoryFilters.SHOW_COMPONENTS);
    });
    it('should add second feature of CHAPTER category', () => {
        store.dispatch(
            actions.addFeature('Second Chapter Feature',
                actions.Categories.CHAPTER)
        );
        expect(store.getState().features.length).to.equal(2);
        expect(store.getState().features[1].category)
            .to.equal(actions.Categories.CHAPTER);
    });
    it('should add third feature of APP category', () => {
        store.dispatch(
            actions.addFeature('Third App Feature',
                actions.Categories.APP)
        );
        expect(store.getState().features.length).to.equal(3);
        expect(store.getState().features[2].category)
            .to.equal(actions.Categories.APP);
    });
    it('should set new search text', () => {
        expect(store.getState().searchText)
            .to.equal('');
        store.dispatch(actions
            .setSearchText('new search text'));
        expect(store.getState().searchText)
            .to.equal('new search text');
    });
});

```

When we run this test using `npm test` we notice following test results.

...

Roadmap Redux

- ✓ / should get initial state for store
- ✓ / should add first feature of COMPONENT category
- ✓ / should initialize first feature with default state
- ✓ / should increment likes count for first feature
- ✓ / should set a new categoryFilter
- ✓ / should add second feature of CHAPTER category
- ✓ / should add third feature of APP category
- ✓ / should set new search text

16 passing (259ms)
3 pending

Now as we evolve our Redux app, we can continue adding to our test suite.

Optimize Redux app

There are several ways our basic Redux app can be optimized before we even move onto designing the React components.

These optimizations are important as they improve readability, maintainability, and performance for larger apps.

Object Spread Operator in reducers/roadmap.js

Our `Object.assign()` code in reducers can be further simplified using ES6 stage 2 feature called Object Rest Spread Operator. Using this feature requires installing Babel plugin for [transform-object-rest-spread](#) and making the required changes in the plugins section of `.babelrc` configuration.

Notice how this simplifies our reducers using `...state` object spread operator.

```
import * as actions from '../actions/roadmap';

const initialState = {
  searchText: '',
  categoryFilter: actions.CategoryFilters.SHOW_ALL,
  features: []
};

export default function roadmapApp(state = initialState, action) {
  switch (action.type) {
    case actions.LIKE_FEATURE:
      return { ...state,
        features: state.features.map((feature, index) => {
          if (index === action.index) {
            return { ...feature, likes: feature.likes + 1 };
          }
          return feature;
        })
      };
    case actions.SEARCH_TEXT:
      return { ...state, searchText: action.text };
    case actions.SET_CATEGORY_FILTER:
      return { ...state, categoryFilter: action.filter };
    case actions.ADD_FEATURE:
      return { ...state, features: [...state.features,
        { title: action.title, category: action.category, likes: 0 }] \
      };
    default:
      return state;
  }
}
```

```
}  
}
```

Reducer composition in reducers/roadmap.js

A fundamental pattern of designing Redux apps is to slice the reducer code into separate concerns based on top-level state tree nodes.

We have features, categoryFilter, and searchText as top level nodes within our state tree.

```
import * as actions from '../actions/roadmap';  
  
function features(state = [], action) {  
  switch (action.type) {  
    case actions.LIKE_FEATURE:  
      return state.map((feature, index) => {  
        if (index === action.index) {  
          return { ...feature, likes: feature.likes + 1 };  
        }  
        return feature;  
      });  
    case actions.ADD_FEATURE:  
      return [...state, { title: action.title, category: action.category\  
, likes: 0 }];  
    default:  
      return state;  
  }  
}  
  
function categoryFilter(state = actions.CategoryFilters.SHOW_ALL, action\  
on) {  
  switch (action.type) {  
    case actions.SET_CATEGORY_FILTER:  
      return action.filter;  
    default:  
      return state;  
  }  
}  
  
function searchText(state = '', action) {  
  switch (action.type) {  
    case actions.SEARCH_TEXT:  
      return action.text;  
    default:  
      return state;  
  }  
}  
  
export default function roadmapApp(state = {}, action) {  
  return {  
    features: features(state.features, action),  
    searchText: searchText(state.searchText, action),  
    categoryFilter: categoryFilter(state.categoryFilter, action)  
  };  
}
```


We separate these top-level nodes into their own reducer functions, further simplifying their respective code.

Our roadmapApp reducer function is now only three lines of code.

And, we are not done yet! Let us do another round of optimization using Redux combineReducers utility. Replace the roadmapApp function with the following code.

```
import { combineReducers } from 'redux';  
// ... some code  
  
const roadmapApp =  
  combineReducers({ features,  
    searchText, categoryFilter });  
  
export default roadmapApp;
```

One single line of code for our roadmapApp reducer!

ES6 import in reducers/roadmapApp.js

Again, we can further do some ES6 magic and optimize our file organization even further.

If we separate the features, searchText, categoryFilter reducers in their own file, and use a separate file for roadmapApp, we can use ES6 import to do this.

```
import { combineReducers } from 'redux';  
import * as reducers from './roadmap';  
  
const roadmapApp = combineReducers(reducers);  
export default roadmapApp;
```

This makes a lot of sense for larger projects where you may have many more reducers which you may want to keep organized in separate files.

We remove the roadmapApp function from our reducers/roadmap.js file. Prefix export keyword to all the reducer functions.

Of course we need to update the `store/roadmap.js` to import from the new `roadmapApp.js` file.

All along this journey refactoring our Redux app for optimizations, we were running the test suite defined earlier to ensure all tests are passing.

Component hierarchy spec in 03_roadmap.spec.js

Just like the spec we wrote earlier, let us write the component specification for our Roadmap app.

```
import { describe, it } from 'mocha';

describe('<Roadmap />', () => {
  it('should create one .roadmap component');

  describe('roadmap__navigation', () => {
    it('should create one .roadmap__navigation component');

    describe('roadmap__search', () => {
      it('should create one .roadmap__search component');
      it('should initialize default value for searchText');
      it('should execute enterSearch() when user presses Enter in search box');
      it('should update state tree after enterSearch() is called');
    });

    describe('roadmap__category', () => {
      it('should create N .roadmap__category components');
      it('should execute selectFilter() when user selects a filter');
      it('should update state tree after selectFilter() is called');
    });
  });

  describe('feature__list', () => {
    it('should create one .feature__list component');

    describe('feature', () => {
      it('should create N .feature components');

      describe('feature__likes', () => {
        it('should create one .feature__likes component per .feature');
      });

      describe('feature__detail', () => {
        it('should create one .feature__detail component per .feature');
      });

      describe('feature__category', () => {
        it('should create one .feature__category component per .feature');
      });
    });
  });
});
```

When we run this test using `npm test` we notice following test results.

...

```
<Roadmap />
- should create one .roadmap component
roadmap__navigation
- should create one .roadmap__navigation component
roadmap__search
- should create one .roadmap__search component
- should initialize default value for searchText
- should execute enterSearch() when user presses Enter in sear\
ch box
- should update state tree after enterSearch() is called
roadmap__category
- should create N .roadmap__category components
- should execute selectFilter() when user selects a filter
- should update state tree after selectFilter() is called
feature__list
- should create one .feature__list component
feature
- should create N .feature components
feature__likes
- should create one .feature__likes component per .feature
feature__detail
- should create one .feature__detail component per .feature
feature__category
- should create one .feature__category component per .feature

16 passing (252ms)
17 pending
```

Here are the strategies to consider when creating component specification for your app.

- **Component hierarchy.** Represent the component hierarchy starting at the top level owner component traversing through child or owned components.
- **JSX naming.** Identify component name using JSX closing tag statement `<Component />` even if the component has props.
- **Class name.** The `it` spec statements can refer to `className` associated with the component. This `className` can then be used by `Enzyme find()` method as well.
- **Cardinality.** Specify cardinality (zero, one, or many) of component(s) expected to be created during normal use case of the application. This can be checked in the test implementation.

- **Props.** Consider representing component props during specification stage as additional `it` spec statements.
- **Events.** Events and life-cycle methods can also be specified at this stage.

Rapid prototype hierarchy in roadmap.jsx

Like real-world apps, as our repository of components grows, we should be creating new functionality by mixing existing components, refactoring them for new use cases.

Let us rapidly prototype our front-end for Roadmap app, by reusing the Card component, and bringing together Button, and form components we created earlier. We also reuse the IconText component.

```
import React from 'react';

import Card from './Card.jsx';
import IconText from './IconText.jsx';
import Button from './Button';
import Input from './Input';
import InputLabel from './InputLabel';
import InputField from './InputField';

const Roadmap = () => (
  <div className="roadmap">
    <Card plain>
      <h1>Roadmap</h1>
    </Card>
    <div className="roadmap__navigation">
      <Card plain className="col--half">
        <Input className="roadmap__search">
          <InputLabel label="Search" />
          <InputField placeholder="Enter feature name" />
        </Input>
      </Card>
      <Card plain>
        <Button className="roadmap__category"
          label="Component" color="default" icon="cubes" />
        <Button className="roadmap__category"
          label="App" color="primary" icon="cloud" />
        <Button className="roadmap__category"
          label="Chapter" color="secondary" icon="book" />
      </Card>
    </div>
    <div className="feature__list">
      <div className="feature">
        <Card plain>
          <IconText
            className="success feature__likes"
            icon="heart"
            size="2x"
            text="21 likes"
            ribbon
          />
        </Card>
        <Card plain className="col--half">
          <div className="feature__detail">
```

```

        <b>Feature title here!</b><br />
        Details spilling to next line here.
    </div>
</Card>
<Card plain>
  <IconText
    icon="book"
    size="2x" className="secondary feature__category" />
  </Card>
</div>

<div className="feature">
  <Card plain>
    <IconText
      className="warning feature__likes"
      icon="heart"
      size="2x"
      text="1 like"
      ribbon
    />
  </Card>
  <Card plain className="col--half">
    <div className="feature__detail">
      <b>Feature two title here!</b><br />
      More details spilling to next line here.
    </div>
  </Card>
  <Card plain>
    <IconText
      icon="cubes"
      size="2x" className="default feature__category" />
    </Card>
  </div>
</div>
</div>
);
export default Roadmap;

```

We can also connect this rapid prototype with a route and develop the UI interactively using Hot Module Replacement.

Styles in roadmap.css

We define the styles for our prototype like so.

```

.feature__list,
.roadmap {
  display: flex;
  flex-flow: column;
  justify-content: flex-start;
}

.roadmap__navigation {
  display: flex;
  flex-flow: row wrap;
  justify-content: flex-start;
}

```

```
.feature {  
  display: flex;  
  flex-flow: row wrap;  
  justify-content: flex-start;  
  border-top: 1px solid lightgrey;  
}
```

We are using dummy data as the components are not yet wired to our Redux app.

Note that we are using same class identifiers as defined in our test suite spec. We can start implementing some of our test suite now.

```
it('should create one .roadmap component', () => {  
  const wrapper = shallow(<Roadmap />);  
  expect(wrapper.is('.roadmap')).toEqual(true);  
});
```

Once we are relatively satisfied with our UI prototype, we can further optimize our component hierarchy by extracting Roadmap app components into separate files before we start wiring our Redux actions, reducers, and store.

Data in fixtures/roadmap/features.js

As we extract components we may want to feed actual data into the components to continue prototyping our app before we connect the Redux store.

We will define our fixture data in such a manner that it conforms to our component shape (or schema) and at the same time it can be reused later on to hydrate (or initialize) our Redux store.

```
import * as actions from '../actions/roadmap';

const features = [
  { id: 1,
    title: 'Roadmap',
    about: `The app implements a features roadmap for ReactSpeed.
    The app is built using Redux and available live on ReactSpeed webs\
ite.`,
    category: actions.Categories.APP,
    likes: 3
  },
  { id: 2,
    title: 'Navigation',
    about: `This component renders main menu navigation items. It also
    renders React Router Links as child components.`,
    category: actions.Categories.COMPONENT,
    likes: 1
  },
  { id: 3,
    title: 'Test App Components',
    about: `The chapter discusses ESLint, StyleLint, Browsersync setup\
    using Webpack.
    It also introduces Behavior-Driven Development using Mocha, Chai, \
    and Enzyme.`,
    category: actions.Categories.CHAPTER,
    likes: 15
  }
];

export default features;
```

Extract Feature.jsx component

Now we extract the Feature component based on the shape we defined in our data fixture.

```
import React, { PropTypes } from 'react';
import Card from './Card.jsx';
import IconText from './IconText.jsx';
import { Categories } from '../actions/roadmap';

const Feature = ({ onClickLikes, title, about,
  category, likes }) => {
  let renderCategory = '';
  switch (category) {
    case Categories.COMPONENT:
      renderCategory = (
        <IconText
          icon="cubes"
          size="2x" className="default feature__category" />
      );
      break;
    case Categories.APP:
      renderCategory = (
        <IconText
          icon="cloud"
          size="2x" className="primary feature__category" />
      );
      break;
    case Categories.CHAPTER:
      renderCategory = (
        <IconText
          icon="book"
          size="2x" className="secondary feature__category" />
      );
      break;
    default:
      renderCategory = '';
  }

  const renderLikesClass = (likes > 10)
    ? 'success feature__likes' : 'warning feature__likes';

  return (
    <div className="feature">
      <Card onClick={onClickLikes} className="col--twentieth" plain>
        <IconText
          className={renderLikesClass}
          icon="heart"
          size="2x"
          text={` ${likes} likes` }
          ribbon
        />
      </Card>
      <Card plain className="col--half">
        <div className="feature__detail">
          <b>{title}</b><br />
          <small>{about}</small>
        </div>
      </Card>
    </div>
  );
}
```

```

        </Card>
        <Card plain>
          {renderCategory}
        </Card>
      </div>
    );
  };

  Feature.propTypes = {
    onClickLikes: PropTypes.func.isRequired,
    title: PropTypes.string.isRequired,
    about: PropTypes.string.isRequired,
    category: PropTypes.string.isRequired,
    likes: PropTypes.number.isRequired
  };

  export default Feature;

```

We note that the component takes an event handler as property for handling click on feature likes.

We have also added some logic to the component for rendering icons based on category of the feature. We also render color for likes icon based on number of likes.

Extract FeatureList.jsx component

Next we need to create the FeatureList component to map given features fixture and render Feature components.

```
import React, { PropTypes } from 'react';
import Feature from './Feature';

const FeatureList = ({ features, onClickLikes }) => (
  <div className="feature__list">
    {features.map(feature =>
      <Feature
        key={feature.id}
        {...feature}
        onClickLikes={() => onClickLikes(feature.id)}
      />
    )}
  </div>
);

FeatureList.propTypes = {
  features: PropTypes.arrayOf(PropTypes.shape({
    id: PropTypes.number.isRequired,
    title: PropTypes.string.isRequired,
    about: PropTypes.string.isRequired,
    category: PropTypes.string.isRequired,
    likes: PropTypes.number.isRequired
  }).isRequired).isRequired,
  onClickLikes: PropTypes.func.isRequired
};

export default FeatureList;
```

This component passes the event handler with feature id parameter. We use the object spread operator `{...feature}` to pass on the remaining props. It also defines expected shape of the features prop.

Refactoring Roadmap.jsx app

Our app code now reduces to just rendering the FeaturesList component while passing it the fixtures data on features.

```
import React from 'react';

import Card from './Card.jsx';
import Button from './Button';
import Input from './Input';
import InputLabel from './InputLabel';
import InputField from './InputField';
import FeatureList from './FeatureList';
import features from '../fixtures/roadmap/features';

const likesClick = (id) => {
  // to be implemented
  console.log(`likesClick id = ${id}`);
};

const Roadmap = () => (
  <div className="roadmap">
    <Card plain>
      <h1>Roadmap</h1>
    </Card>
    <div className="roadmap__navigation">
      <Card plain className="col--half">
        <Input className="roadmap__search">
          <InputLabel label="Search" />
          <InputField placeholder="Enter feature name" />
        </Input>
      </Card>
      <Card plain>
        <Button className="roadmap__category"
          label="Component" color="default" icon="cubes" />
        <Button className="roadmap__category"
          label="App" color="primary" icon="cloud" />
        <Button className="roadmap__category"
          label="Chapter" color="secondary" icon="book" />
      </Card>
    </div>
    <FeatureList
      features={features}
      onClickLikes={likesClick}
    />
  </div>
);

export default Roadmap;
```

We also create placeholder event handler likesClick within our Roadmap component, passing this as a prop to the FeaturesList component.

Once we are done with this refactoring we can run our app and see the app render with the fixtures data.

Add tests to 03_roadmap.spec.js

We can continue implementing our tests to match the components we have created.

```
import React from 'react';
import { describe, it } from 'mocha';
import { expect } from 'chai';
import { shallow, render } from 'enzyme';
import Roadmap from '../../app/components/Roadmap';

describe('<Roadmap />', () => {
  it('should create one .roadmap component', () => {
    const wrapper = shallow(<Roadmap />);
    expect(wrapper.is('.roadmap')).to.equal(true);
  });

  describe('roadmap_navigation', () => {
    it('should create one .roadmap_navigation component');

    describe('roadmap_search', () => {
      it('should create one .roadmap_search component', () => {
        const wrapper = render(<Roadmap />);
        expect(wrapper.find('.roadmap_search')).to.have.length(1);
      });
      it('should initialize default value for searchText');
      it('should execute enterSearch() when user presses Enter in search box');
      it('should update state tree after enterSearch() is called');
    });

    describe('roadmap_category', () => {
      it('should create 3 .roadmap_category components', () => {
        const wrapper = render(<Roadmap />);
        expect(wrapper.find('.roadmap_category')).to.have.length(3);
      });
      it('should execute selectFilter() when user selects a filter');
      it('should update state tree after selectFilter() is called');
    });
  });

  describe('feature_list', () => {
    it('should create one .feature_list component', () => {
      const wrapper = render(<Roadmap />);
      expect(wrapper.find('.feature_list')).to.have.length(1);
    });
    describe('feature', () => {
      it('should create N .feature components', () => {
        const wrapper = render(<Roadmap />);
        expect(wrapper.find('.feature')).to.have.length.above(1);
      });
      describe('feature_likes', () => {
        it('should create N .feature_likes components', () => {
          const wrapper = render(<Roadmap />);
          expect(wrapper.find('.feature_likes')).to.have.length.above(1);
        });
      });
    });
    describe('feature_detail', () => {
```

```

    it('should create N .feature__detail components', () => {
      const wrapper = render(<Roadmap />);
      expect(wrapper.find('.feature__detail')).toHaveLength(1);
    });
  });

  describe('feature__category', () => {
    it('should create N .feature__category components', () => {
      const wrapper = render(<Roadmap />);
      expect(wrapper.find('.feature__category')).toHaveLength(1);
    });
  });
});

```

As we run the test we will see more tests passing.

In the next chapter we will wire Redux into our app.

Redux Wiring App

This chapter continues developing on the Redux State Container chapter and wiring Redux into our Roadmap app.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c13 origin/c13-redux-wiring-app
npm install
npm start
```


Extract CategoryButton.jsx component

Next we implement the CategoryButton presentational component.

```
import React, { PropTypes } from 'react';
import { CategoryFilters } from '../actions/roadmap';
import Button from './Button';

const CategoryButton = ({ selected, filter, onClick }) => {
  let categoryColor = '';
  let categoryIcon = '';

  switch (filter) {
    case CategoryFilters.SHOW_APPS:
      categoryColor = 'primary';
      categoryIcon = 'cloud';
      break;
    case CategoryFilters.SHOW_COMPONENTS:
      categoryColor = 'default';
      categoryIcon = 'cubes';
      break;
    case CategoryFilters.SHOW_CHAPTERS:
      categoryColor = 'secondary';
      categoryIcon = 'book';
      break;
    default:
      categoryColor = 'warning';
      categoryIcon = 'star';
  }
  if (selected) {
    return (
      <Button className="roadmap__category"
        color={categoryColor} icon={categoryIcon} />
    );
  }
  return (
    <Button className="roadmap__category"
      color={categoryColor} icon={categoryIcon}
      onClick={e => {
        e.preventDefault();
        onClick();
      }} />
  );
};

CategoryButton.propTypes = {
  selected: PropTypes.bool.isRequired,
  filter: PropTypes.string.isRequired,
  onClick: PropTypes.func.isRequired
};

export default CategoryButton;
```

Refactor data fixture features.js

We will refactor our data fixture to a schema that is closer to our production use case.

```
import * as actions from '../..actions/roadmap';

const features = [
  { id: 1,
    title: 'Roadmap',
    about: `The app implements a features roadmap for ReactSpeed.
    The app is built using Redux and available live on ReactSpeed webs\
ite.`,
    category: actions.Categories.APP,
    likes: 32,
    link: 'https://github.com/...'
  },
  { id: 2,
    title: 'Navigation',
    about: `This component renders main menu navigation items. It also
    renders React Router Links as child components.`,
    category: actions.Categories.COMPONENT,
    likes: 23,
    link: 'https://github.com/...'
  }
];

export default features;
```

Refactor actions in actions/roadmap.js

We can now refactor our actions to handle the new production schema.

```
import * as actions from '../actions/roadmap';

export function categoryFilter(state = actions.CategoryFilters.SHOW_ALL, action) {
  switch (action.type) {
    case actions.SET_CATEGORY_FILTER:
      return action.filter;
    default:
      return state;
  }
}

export function searchText(state = '', action) {
  switch (action.type) {
    case actions.SEARCH_TEXT:
      return action.text;
    default:
      return state;
  }
}

export function features(state = [], action) {
  switch (action.type) {
    case actions.LIKE_FEATURE:
      return state.map((feature, id) => {
        if (id === action.id - 1) {
          return { ...feature, likes: feature.likes + 1 };
        }
        return feature;
      });
    case actions.ADD_FEATURE:
      return [...state, {
        id: action.id,
        title: action.title,
        about: action.about,
        category: action.category,
        likes: action.likes,
        link: action.link
      }];
    default:
      return state;
  }
}
```

The Feature.jsx component also needs to be refactored to handle the new schema.

```
// ... add new props
const Feature = ({ onClickLikes, title, about,
  category, likes, link }) => {

// ... render new props
```

```
<Card plain className="col--half">
  <div className="feature_detail">
    <a href={link}>{title}</a><br /><br />
    <small>{about}</small>
  </div>
</Card>
```

```
// ... define new props
```

```
Feature.propTypes = {
  onClickLikes: PropTypes.func.isRequired,
  title: PropTypes.string.isRequired,
  about: PropTypes.string.isRequired,
  category: PropTypes.string.isRequired,
  likes: PropTypes.number.isRequired,
  link: PropTypes.string.isRequired
};
```

Extract SearchFeature.jsx component

Let us continue developing our Roadmap app further by extracting the SearchFeature component and connecting it with our Redux store.

Before we do this let us install the official React-Redux bindings.

```
npm install --save react-redux
```

The react-redux connect method helps us connect SearchFeature to the Redux store, by returning a new component class based on the component passed to the method. This is our first example of using higher order components or HOC. Higher order components help us add functionality to existing components. In this case add the Redux store without changing the existing component.

In case of SearchFeature we do not have any further mapping of state or event handlers required, so we are using the same function name for the new component.

```
import React from 'react';
import { connect } from 'react-redux';
import { setSearchText } from '../actions/roadmap';
import Input from './Input';
import InputLabel from './InputLabel';
import InputField from './InputField';

let SearchFeature = ({ dispatch }) => ( // eslint-disable-line import/\
no-mutable-exports
  <Input className="roadmap__search">
    <InputLabel label="Search" />
    <InputField
      onChange={e => {
        e.preventDefault();
        dispatch(setSearchText(e.target.value.trim()));
      }}
      placeholder="Enter feature name" />
    </Input>
  </Input>
);
SearchFeature.propTypes = {
  dispatch: React.PropTypes.func
};
SearchFeature = connect()(SearchFeature);

export default SearchFeature;
```

Connecting Store in VisibleFeatureList.jsx

It is time to connect our Redux store to our components. To do so we will create container components which will connect the store to presentational components we created so far.

We first create the `VisibleFeatureList` component which displays

visible features based on the current state. Right now we are only focusing on selected Category Filter, which determines the visible features.

Our `VisibleFeatureList` container component connects the store data to our presentational `FeatureList` component based on category filters selected by the user. `VisibleFeatureList` is a higher order component (HOC) which changes the functionality of `FeatureList` by filtering list of Features rendered. So, unlike `SearchFeature` component where we used the same name, we have a different name for this HOC as returned by the connect method.

This container component pattern starts with defining a `getVisibleFeatures` function which returns list of features based on category filter and search text. The `mapStateToProps` function sets features to this filtered return value. The `mapDispatchToProps` function maps event handlers.

```
import { connect } from 'react-redux';
import * as actions from '../actions/roadmap';
import FeatureList from './FeatureList';

const getVisibleFeatures = (features, filter, search) => {
  let searchedFeatures = features;
  if (search) {
    searchedFeatures = features
      .filter(
        f => `${f.title} ${f.about}`
          .toLowerCase().includes(search.toLowerCase())
      );
  }
}
```

```

switch (filter) {
case actions.CategoryFilters.SHOW_ALL:
  return searchedFeatures;
case actions.CategoryFilters.SHOW_APPS:
  return searchedFeatures
    .filter(f => f.category ===
      actions.Categories.APP);
case actions.CategoryFilters.SHOW_COMPONENTS:
  return searchedFeatures
    .filter(f => f.category ===
      actions.Categories.COMPONENT);
case actions.CategoryFilters.SHOW_CHAPTERS:
  return searchedFeatures
    .filter(f => f.category ===
      actions.Categories.CHAPTER);
default:
  return searchedFeatures;
}
};

const mapStateToProps = (state) => ({
  features:
    getVisibleFeatures(state.features,
      state.categoryFilter, state.searchText)
});

const mapDispatchToProps = (dispatch) => ({
  onClickLikes: (id) => {
    dispatch(actions.likeFeature(id));
  }
});

const VisibleFeatureList = connect(
  mapStateToProps,
  mapDispatchToProps
)(FeatureList);

export default VisibleFeatureList;

```

FilterCategoryButton.jsx container component

The FilterCategoryButton container component connects our CategoryButton presentational component properties and events with Redux actions, reducers, and store.

```
import { connect } from 'react-redux';
import { setCategoryFilter } from '../actions/roadmap';
import CategoryButton from './CategoryButton';

const mapStateToProps = (state, ownProps) => ({
  selected: ownProps.filter === state.categoryFilter,
  filter: ownProps.filter
});

const mapDispatchToProps = (dispatch, ownProps) => ({
  onClick: () => {
    dispatch(setCategoryFilter(ownProps.filter));
  }
});

const FilterCategoryButton = connect(
  mapStateToProps,
  mapDispatchToProps
)(CategoryButton);

export default FilterCategoryButton;
```

Refactor Roadmap.jsx

Our Roadmap app has fewer lines of code now.

```
import React from 'react';
import Card from './Card.jsx';
import SearchFeature from './SearchFeature';

import VisibleFeatureList from './VisibleFeatureList';
import FilterCategoryButton from './FilterCategoryButton';
import { CategoryFilters } from '../actions/roadmap';

const Roadmap = () => (
  <div className="roadmap">
    <Card plain>
      <h1>Roadmap</h1>
    </Card>
    <div className="roadmap__navigation">
      <Card plain className="col--half">
        <SearchFeature />
      </Card>
      <Card plain>
        <FilterCategoryButton filter={CategoryFilters.SHOW_ALL} />
        <FilterCategoryButton filter={CategoryFilters.SHOW_APPS} />
        <FilterCategoryButton filter={CategoryFilters.SHOW_CHAPTERS} />
      </Card>
    </div>
  </div>
);
```



```
        <FilterCategoryButton filter={CategoryFilters.SHOW_COMPONENTS}\
    />
    </Card>
  </div>
  <VisibleFeatureList />
</div>
);
```

```
export default Roadmap;
```

We remove the prototype event handlers as Redux actions will take over the UI interaction. We also render the `VisibleFeatureList` and `FilterCategoryButton` container components with relevant props.

Hydrate Redux app using reducers/hydrate.js

Now that we have connected our app components with Redux, we have one final step remaining. We need to hydrate our app with the fixture data for initial state.

```
import store from '../../store/roadmap';
import * as actions from '../../actions/roadmap';
import features from './features';

const roadmapHydrate = () => {
  for (let i = 0; i < features.length; i++) {
    store.dispatch(actions
      .addFeature(
        features[i].id,
        features[i].title,
        features[i].about,
        features[i].category,
        features[i].likes,
        features[i].link
      ));
  }
};

export default roadmapHydrate;
```

Pass store in index.js

To run our app we need to run hydrate and pass the store to our components.

```
//... some code

import { Provider } from 'react-redux';
import Roadmap from './components/Roadmap';
import store from './store/roadmap';
import roadmapHydrate from './fixtures/roadmap/hydrate';

roadmapHydrate();

//... some code

ReactDOM.render(
  <Provider store={store}>
    <Router history={browserHistory} routes={routeConfig} />
  </Provider>,
  document.getElementById('app')
);
```

That's it. Our Roadmap Redux app is functional. When you first run the app it seems like magic how UI events impact state of this relatively complex app and React automatically

handles re-rendering of the app based on new state. Try clicking on likes to increment beyond 10 likes, or selecting any of the category filters.

Update test suite 03_roadmap.spec.js

We can now update our tests to hydrate data from fixtures and render components using Redux provider.

```
import React from 'react';
import { describe, it } from 'mocha';
import { expect } from 'chai';
import { shallow, render } from 'enzyme';
import { Provider } from 'react-redux';
import Roadmap from '../../app/components/Roadmap';
import store from '../../app/store/roadmap';
import roadmapHydrate from '../../app/fixtures/roadmap/hydrate';

roadmapHydrate();

describe('<Roadmap />', () => {
  it('should create one .roadmap component', () => {
    const wrapper = shallow(<Roadmap />);
    expect(wrapper.is('.roadmap')).to.equal(true);
  });

  describe('roadmap__navigation', () => {
    it('should create one .roadmap__navigation component');

    describe('roadmap__search', () => {
      it('should create one .roadmap__search component', () => {
        const wrapper = render(<Provider store={store}><Roadmap /></Provider>);
        expect(wrapper.find('.roadmap__search')).to.have.length(1);
      });
    });

    describe('roadmap__category', () => {
      it('should create 4 .roadmap__category components', () => {
        const wrapper = render(<Provider store={store}><Roadmap /></Provider>);
        expect(wrapper.find('.roadmap__category')).to.have.length(4);
      });
    });

    describe('feature__list', () => {
      it('should create one .feature__list component', () => {
        const wrapper = render(<Provider store={store}><Roadmap /></Provider>);
        expect(wrapper.find('.feature__list')).to.have.length(1);
      });
    });
  });
});
```

Update 02_roadmap_redux.spec.js

We update the Roadmap Redux spec to match the new data fixture.

```

import { describe, it } from 'mocha';
import { expect } from 'chai';
import store from '../../app/store/roadmap';
import * as actions from '../../app/actions/roadmap';

describe('Roadmap Redux', () => {
  it('should get initial state for store', () => {
    expect(store.getState().features.length).to.equal(11);
    expect(store.getState().categoryFilter)
      .to.equal(actions.CategoryFilters.SHOW_ALL);
    expect(store.getState().searchText)
      .to.equal('');
  });
  it('should increment likes count for first feature', () => {
    store.dispatch(actions.likeFeature(0)); // likes = 1
    store.dispatch(actions.likeFeature(0)); // likes = 2
    expect(store.getState().features[0].likes).to.equal(43);
  });
  it('should set a new categoryFilter', () => {
    expect(store.getState().categoryFilter)
      .to.equal(actions.CategoryFilters.SHOW_ALL);
    store.dispatch(actions
      .setCategoryFilter(actions.CategoryFilters.SHOW_COMPONENTS));
    expect(store.getState().categoryFilter)
      .to.equal(actions.CategoryFilters.SHOW_COMPONENTS);
  });
  it('should set new search text', () => {
    expect(store.getState().searchText)
      .to.equal('');
    store.dispatch(actions
      .setSearchText('new search text'));
    expect(store.getState().searchText)
      .to.equal('new search text');
  });
});

```

Firestore React Integration

Firestore is a Platform as a Service offering managed, real-time database, user authentication APIs, and static (front-end) website hosting. We like Firestore as our backend and hosting platform as it is stable, backed by Google, provides well documented APIs, has an easy to learn visual database management toolset, and performant.

For the same price-point Firestore delivers much more than its competitors. As an example, it is hard to find a normal web host offering SSL custom domain hosting for free. Firestore offers way more for this price point. It also comes with a generous free plan which is easy to upgrade when you are ready to go production.

Here is what you will learn in this chapter.

- Compare Firestore with another popular framework Meteor.
- Host your front-end app using Firestore hosting.
- How Firestore stores files and data.
- Designing a REST API using Firestore.
- For what kind of apps is Firestore not ideal.
- Refactor Workflow component for Firestore integration.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview [complete demo website](#) hosted on Firebase, as you code this app by the end of this book.

View [current chapter demo](#) of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c14 origin/c14-firebase-react-integration
npm install
npm start
```

Comparing Firebase with Meteor

Both Meteor and Firebase can be used to build robust mobile-web apps. Differences in platform, database, hosting, and pricing are explained here.

Platform. Firebase = Real-time Database | Meteor = Web Framework

Firebase is focused on providing a real-time database platform and API which works well with Angular, React, Ember, effectively any popular front-end web framework or directly within mobile platforms like iOS and Android.

Meteor is focused on providing a web application framework that integrates nicely with other popular web frameworks like Angular and React. Meteor also integrates Cordova to enable write-once-publish-many across web and mobile for your apps. Meteor monetizes its open source efforts by providing paid Galaxy hosting for apps written in Meteor.

Database. Firebase = Real-time API over Mongo | Meteor = MongoDB API

Both Meteor and Firebase use MongoDB for database.

Meteor provides more direct, “power” Mongo access to the database API, within its framework. So, if you know Mongo, you will feel at home with many Meteor capabilities. Meteor also provides an innovative mini-Mongo API to create client-side data caching.

Firebase on the other hand provides its own real-time database management API which is more “constrained” and at the same time powerful and easier to learn for certain category of apps.

Pricing. Firebase = Easier bundling | Meteor = Scale and support based

Meteor Galaxy [pricing](#) is usage based or utility based pricing. Based on number of containers used by your app and number of hours these containers are running per month. Base pricing also varies based on level of support you require. Overall pricing will vary based on how your app scales and how it is used.

Firebase [pricing](#) is relatively simpler and offers more “visible” bundling of features. For a decent range of usage, scale, and performance, you can be assured of a relatively lower fixed price.

Front-end. Firebase = Angular, React, Many | Meteor = Blaze, Angular, React

Firebase provides API, integration libraries, and samples for web frameworks like Angular, Ember, Vue.js, and React. You can also deploy statically generated sites from tools like Jekyll and Harp.

Meteor supports Blaze as its front-end framework. It also integrates Angular and React.

Server-side. Firebase = Users, Security only | Meteor = Anything

Firebase provides limited “programmability” on the server-side through user management and authentication API and configurable database access security rules.

Meteor supports Universal JavaScript and server side processing capabilities out-of-the-box. So, using Meteor, you can build almost anything that requires server-side processing.

Hosting. Firebase = Web + CDN + Users + DB | Meteor = App only

Firebase provides hosting for front-end web apps written in plain HTML, CSS, and JS. Firebase also provides a Content Delivery Network (CDN) to help distribute your published websites in a performant way. Firebase hosting includes SSL for their lowest paid account.

Meteor provides Galaxy hosting for apps written in Meteor. As of this writing, Galaxy does not support database hosting. We personally find this constraint as “counter” Meteor goals of easing development on their platform, considering how neatly the database is tied into the whole framework with mini Mongo and server side Mongo database. You can use other paid services like [Compose](#) or [mLab](#) for database hosting.

Portability. Firebase = App portable | Meteor = Data queries portable

Portability is important for future-proofing your coding investment. A popular platform may stop general availability, like [Parse from Facebook](#). You may also decide to reuse your code across multiple platforms and projects.

Firebase front-end is almost entirely portable as it can be written in any front-end web framework. Where you write Firebase specific code is within limited data change listener methods. In fact as Firebase uses JSON for storing data, your Create-Read-Update-Delete (CRUD) code can be made relatively portable. If you are writing user authentication code, this uses OAuth protocols and Firebase API. Firebase data can be exported as JSON. Hosted apps on Firebase can move almost as-is to other static hosting providers like GitHub.

Meteor API cuts across client and server-side and Meteor is relatively more opinionated to ensure your app follows the best practices the platform supports including DDP, Livequery, and Blaze. The database access queries are written in MongoDB API, so these are relatively portable. Meteor also enables you to swap its Blaze front-end layer with Angular or React.

Organization. Firebase = Google | Meteor = MDG

Both Meteor and Firebase have robust organizations.

Firebase is [acquired](#) by Google.

Meteor has +\$31M in [funding](#) from leading Silicon Valley investors.

So, both platforms are a good choice for developing your next app. Firebase pricing is relatively easier to grasp and enables more flexibility in your technology stack decisions.

Firestore Hosting

Getting started with Firestore hosting is easy. Install their Command Line Interface (CLI) tools. Use a Google account to authenticate.

```
npm install -g firebase-tools
```

Next do `firebase init` to setup your deploy directory. In our case this is the `build` folder. This creates `firebase.json` file in our root with selected configuration.

```
=== Deploying to 'reactspeed'...
```

```
i  deploying hosting
i  preparing build directory for upload...
- 6 files uploaded successfully

- Deploy complete!
```

```
URL: https://reactspeed.firebaseio.com
Dashboard: https://reactspeed.firebaseio.com
```

Visit the URL above or run `firebase open`

Run `npm run build` and then `firebase deploy`. You are done. You can then `firebase open` from the terminal to open your new or updated website in your favorite browser. It is that easy.

The screenshot shows the Firebase Hosting dashboard. At the top, there's a dark header with the 'Dashboard' logo, a dropdown menu set to 'VIEWING REACTSPEED', and a user profile icon with 'Account Settings'. On the left is a dark sidebar with icons for Data, Security & Rules (with a red alert icon), Simulator, Analytics, Login & Auth, Hosting (highlighted with a blue bar), and Secrets. The main content area has a light blue header with 'Firebase Hosting' and a link to 'Hosting Documentation'. Below this, the default domain 'https://reactspeed.firebaseio.com' is shown with a 'Use a Custom Domain' button. A deployment summary shows '#1 - Deployed Apr 17th 16, 1:14am'. A section titled 'HISTORY FOR REACTSPEED' contains a table with one entry:

HISTORY FOR REACTSPEED			
1	deployed	Deployed by new@reactspeed.com	04/17/16 1:14am

Firebase Hosting Panel

Firebase hosting also works well for Single Page Apps written to be served entirely from the client side. This includes features like handling search engine friendly URLs and clean URLs.

You can also replace the development HTTP server with Firebase's own development staging server. So far we are using `serve` using the `npm serve` file server. Now we can use `firebase serve` using the Firebase development server packaged as part of Firebase CLI. Done! Now you can test your apps under same configuration as on the production server before hitting the `firebase deploy` command.

Behind the scenes Firebase is using Superstatic on its hosting infrastructure. Superstatic came with Divshot's acquisition by Firebase. It is an open source project and a "static file server for fancy apps". Read more on the [Firebase blog about their hosting features](#).

How Firebase stores files and data

JSON documents. Firebase is built on top of one of the most mature and powerful NoSQL databases around, MongoDB. So Firebase data store is tree-like, JavaScript Object Notation (JSON) documents.

Real-time API. Firebase value adds on top of MongoDB with its real-time publish/subscribe API. What this means is you could have microservice A write to your Firebase data store and microservice B picks up the changes as these happen in near real-time. A and B microservices do not need to be on the same server or developed by the same entity. This makes Firebase really powerful for applications like chatting servers, streaming services, location intelligence, among others.

Static Hosting and CDN. Firebase also stores and serves front-end files (HTML, CSS, JS) using a Content Delivery Network (CDN) as a world class hosting platform. You could dynamically generate these files from a Node.js backed development server, use modern web application frameworks like React and Angular to deliver very capable mobile-web apps wired to a real-time database.

Geo-spatial data. Firebase has basic capabilities to store and retrieve Latitude/Longitude information along with JSON documents. This makes it ideal for many location aware apps, combined with its real-time data API, and performant front-end hosting for mapping embeds.

User Profiles. Firebase also stores basic user profiles and registration data. That is one less worry and an important aspect of any multi-tenant SaaS app.

Complex UI state management. This is experimental, however we are keen to explore if Firebase can be used to persist in near real-time, complex UI state management. Use cases include deep personalization of app UI state, so you come back to same settings you left your app last time you used it.

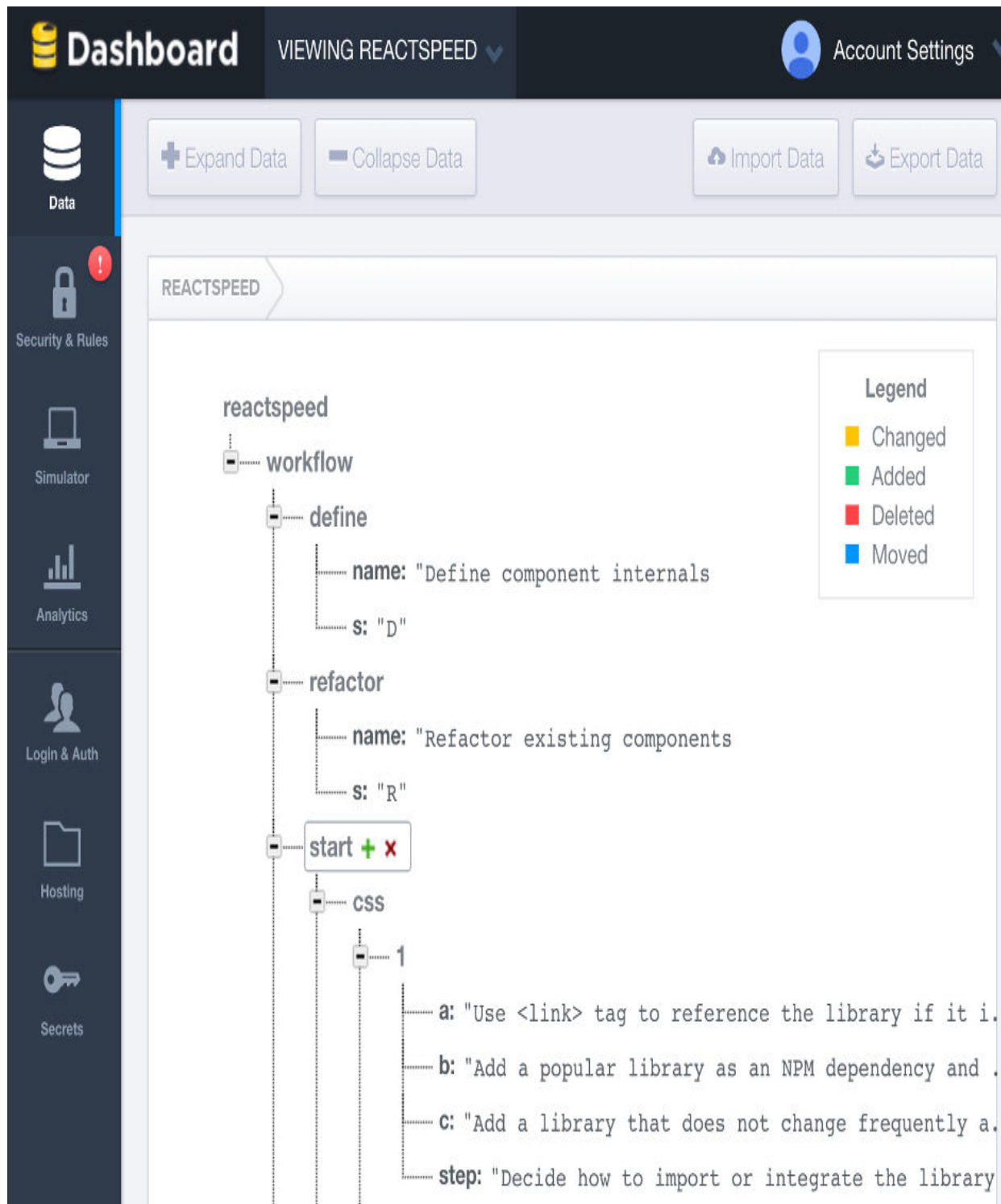
Decision Trees. JSON hierarchical data schema, real-time data change listening, REST API interface to your data schema, all combined can be a good use case for elaborate decision trees. These can support knowledge management and machine learning apps.

Gaming state. Another variation on complex UI state management. Firebase could be used to create, read, update, and delete user gaming state in real-time.

Designing a REST API using Firebase

Providing a REST API is essential for any app that works with data that can be consumed by other apps. Normally writing an API for an app is an involved process.

Firebase provides visual data management tools that enable you to rapidly design an API simply using a well designed JSON data schema. The structure of your JSON schema becomes the API endpoints right out-of-the-box. You can then secure your API with several authentication schemes, or just keep it available as an open data set.



Designing Firebase API using visual tools

For ReactSpeed API we want to expose component design workflow to other apps which can consume this and add more features. One of the use cases may be a project

management app which brings up required workflow steps as a pre-populated Todo List for its users.

We are effectively codifying or API enabling the entire React Speed Coding book outline of contents for consuming within third-party apps!

For what kind of apps is Firebase not ideal

Server side continuous processes. Firebase is not ideal, if your app requires continuous server side processing, like receiving streaming data from a source like Twitter, analyzing this in some way, before consuming the analytics results in your app in real-time.

Full-stack or universal app frameworks. You cannot run Meteor or Universal Angular on a platform like Firebase as these frameworks require server-side scripting.

Large binary file processing. If you are creating an image processing app, or a an online video post-processing tool, Firebase does not have much to offer in terms of shifting large binary files or processing these on the server side.

Refactor Workflow component for Firebase integration

It is time for us to truly create our Serverless Architecture by integrating front-end React components with Firebase real-time database as a backend service.

We will start by refactoring our Workflow component to read data from JSON fixture instead of an array data source.

This has several benefits as we refactor Workflow component to read from Firebase data source.

- Our data fixture in JSON format can be directly imported into Firebase database with a couple of clicks.
- Firebase retrieves objects with sequential keys as arrays, so we can continue processing the data in our Workflow component without much change.
- Having same data structure for local store and Firebase remote store means we can compensate for latency, as app connects to Firebase, we could display local stored data. This will make our user experience more reactive.

Data in fixtures/workflow/steps.json

We create our workflow data fixture in JSON format that can be readily imported into Firebase.

```
{
  "0": { "workflow": "Start", "symbol": "Se", "scenario": "Embed to Re\
act", "sequence": 1,
    "img": "img/embed-to-react-w300.jpg" },
  "1": { "workflow": "Start", "symbol": "Se", "scenario": "Embed to Re\
act", "sequence": 2,
    "text": "Customize the embed code within target platform to suit y\
our site or app styles and placement." },
  "2": { "workflow": "Start", "symbol": "Se", "scenario": "Embed to Re\
act", "sequence": 3,
    "text": "Optionally, parametrize the embed code attributes using R\
eact props." },
  "3": { "workflow": "Start", "symbol": "Se", "scenario": "Embed to Re\
act", "sequence": 4,
```

```

    "text": "Use stateless component as you will most likely not maintain embed UI state locally in your component." },
    "4": { "workflow": "Define", "symbol": "Df", "scenario": "Naming conventions", "sequence": 1,
      "img": "img/webpack-workflow-w300.jpg" },
    "5": { "workflow": "Define", "symbol": "Df", "scenario": "Naming conventions", "sequence": 2,
      "text": "Use .jsx extension for React components." },
    "6": { "workflow": "Define", "symbol": "Df", "scenario": "Naming conventions", "sequence": 3,
      "text": "Entry point for app is /app/index.jsx file." },
    "7": { "workflow": "Wire", "symbol": "We", "scenario": "Events", "sequence": 1,
      "text": "As event handlers often manipulate state, they are best defined where state is defined." },
    "8": { "workflow": "Wire", "symbol": "We", "scenario": "Events", "sequence": 2,
      "text": "Define the event handler in outermost owner component." },
    "9": { "workflow": "Wire", "symbol": "We", "scenario": "Events", "sequence": 3,
      "text": "Consume on<Event> property within owned components down the multi-component hierarchy." }
  }
}

```

You will notice we are making some more changes to our data structure.

- We are adding a numeric sequence as id to each step object.
- We are also supporting image display instead of text description for certain workflow steps.

Next we will install the required dependencies for Firebase integration and JSON processing.

```

npm i --save-dev babel-plugin-transform-runtime
npm i --save-dev babel-plugin-transform-object-rest-spread
npm i --save-dev babel-preset-es2015
npm i --save-dev babel-preset-es2017
npm i --save-dev json-loader
npm i --save firebase

```

We are adding dependencies to achieve specific goals.

- **babel-plugin-transform-runtime**, **babel-preset-es2017** are required so we can use ES7 feature `Object.values` and process JSON directly within our component.

- **json-loader** is required in Webpack to process JSON files.
- **firebase** capabilities are loaded at run-time to connect, authenticate, and query the database.

Next we need to update `.babelrc` to use ES7 `Object.values` feature.

```
{
  "presets": ["react", "es2015", "es2017", "airbnb"],
  "env": {
    "development": {
      "presets": ["react-hmre"]
    }
  },
  "plugins": [
    "transform-class-properties",
    "transform-object-rest-spread",
    "transform-runtime"
  ]
}
```

Refactor Workflow.jsx

Now we can refactor Workflow component to read the new JSON fixture locally.

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

const steps = require('../fixtures/workflow/steps.json');

export default class Workflow extends React.Component {
  static propTypes = {
    steps: PropTypes.array.isRequired
  }
  static defaultProps = { steps: Object.values(steps) }
  constructor(props) {
    super(props);
    this.state = { stepsIndex: 0 };
    this.cycleSequence = this.cycleSequence.bind(this);
    this.cycleScenario = this.cycleScenario.bind(this);
  }
  cycleSequence() {
    const nextIndex =
      this.state.stepsIndex === (this.props.steps.length - 1)
      ? 0
      : this.state.stepsIndex + 1;

    this.setState({ stepsIndex: nextIndex });
  }
  cycleScenario() {
    const stepsList = this.props.steps;
    const currentStep = stepsList[this.state.stepsIndex];
    let stepsCount = 0;
```

```

    for (let i = 0; i < stepsList.length; ++i) {
      if (stepsList[i].symbol === currentStep.symbol) stepsCount++;
    }
    const currentScenario = currentStep.strategy;
    const loopStart =
      (this.state.stepsIndex + stepsCount) >= stepsList.length
      ? 0
      : this.state.stepsIndex + 1;
    for (let i = loopStart; i < stepsList.length; ++i) {
      if (stepsList[i].strategy !== currentScenario) {
        this.setState({ stepsIndex: i });
        break;
      }
    }
  }
  render() {
    const stepsList = this.props.steps;
    const currentStep = stepsList[this.state.stepsIndex];

    return (
      <div className="workflow">
        <div className="workflow__scenario">
          {currentStep.strategy}
        </div>
        <div className="workflow__text">
          {currentStep.text}
          ? currentStep.text
          : <img src={currentStep.img}
            alt={currentStep.strategy} />
        </div>
        <div className="workflow__nav">
          <div onClick={this.cycleScenario} className="workflow__symbo\
l">
            {currentStep.symbol}
          </div>
          <div className="workflow__steps">
            {currentStep.workflow}
          </div>
          <div onClick={this.cycleSequence} className="workflow__seque\
nce">
            {currentStep.sequence}
          </div>
        </div>
      </div>
    );
  }
}

```

Changes to our Workflow component are as follows:

- We use `Object.values(steps)` to convert JSON object to array of objects containing the steps data.
- We add capability to process images and text as workflow step content.

We are now ready to create Workflow component with Firebase integration. First we need to create a utility for connecting with Firebase.

Firestore config data in fixtures/rsdb.js

The rsdb.js utility is using Firebase connection configuration from ReactSpeed Firebase console. Note that you will need to replace this with your own configuration as you may want to use your own Firebase instance for your app.

Note that your configuration will be different from this one here. You can easily create it when you setup a new database Firebase will prompt with your specific configuration.

```
const firebase = require('firebase/app');
require('firebase/database');

const config = {
  apiKey: 'AIzaSyDtclK4CEcD9TgoU1Wa3zrJwSSTZxaAwj8',
  authDomain: 'reactspeed.firebaseio.com',
  databaseURL: 'https://reactspeed.firebaseio.com',
  storageBucket: 'project-2919339734930458575.appspot.com'
};
firebase.initializeApp(config);
const rsdb = firebase.database();

export default rsdb;
```

Pass route param using index.js

Now we need to pass the rsdb database as router param in index.js to AboutWorkflow component.

```
import rsdb from './fixtures/rsdb.js';
// ... some code
```

```
<Route path="/workflow"
  rsdb={rsdb}
  component={AboutWorkflow} />
```

Refactor AboutWorkflow.jsx

We refactor AboutWorkflow to add WorkflowFire component which we will create in the next section.

```

import React from 'react';
import Card from './Card';
import Workflow from './Workflow';
import WorkflowFire from './WorkflowFire';

const steps = require('../fixtures/workflow/steps.json');

const AboutWorkflow = ({ route }) => (
  <section>
    <section className="stripe">
      <Card plain className="col--half text--center">
        <h1>Custom Workflow Component</h1>
        <p className="subtext">
          Navigate React development strategies.
        </p>
      </Card>
      <Card className="col--one-third">
        <Workflow />
      </Card>
    </section>
    <section className="stripe back--default">
      <Card className="col--one-third back--white">
        <WorkflowFire steps={steps} rsdb={route.rsdb} />
      </Card>
      <Card plain className="col--half text--center">
        <h1>Firebase Wired Workflow Component</h1>
        <p className="subtext">
          Navigate React development strategies straight from
          a realtime database.
        </p>
      </Card>
    </section>
  </section>
);

AboutWorkflow.propTypes = {
  route: React.PropTypes.object
};

export default AboutWorkflow;

```

Create WorkflowFire.jsx

Now we duplicate Workflow component to WorkflowFire and add the Firebase integration.

```

import React, { PropTypes } from 'react';

export default class WorkflowFire extends React.Component {
  static propTypes = {
    steps: PropTypes.object.isRequired,
    rsdb: PropTypes.object.isRequired,
    realtime: PropTypes.bool,
    stepChange: PropTypes.func
  }
  static defaultProps = { realtime: false }
  constructor(props) {
    super(props);
    this.state = {

```

```

        stepsIndex: 0,
        steps: Object.values(this.props.steps),
        firebase: false,
        stepsCount: Object.values(this.props.steps).length };
    this.cycleSequence = this.cycleSequence.bind(this);
    this.cycleScenario = this.cycleScenario.bind(this);
  }
  componentDidMount() {
    const getSteps = (snap) => {
      this.setState({
        steps: snap.val(),
        firebase: true,
        stepsCount: snap.numChildren()
      });
    };
    if (this.props.realtime) {
      this.props.rsdB.ref('steps').on('value', getSteps);
    } else {
      this.props.rsdB.ref('steps').once('value').then(getSteps);
    }
  }
  cycleSequence() {
    const nextIndex =
      this.state.stepsIndex === (this.state.steps.length - 1)
        ? 0
        : this.state.stepsIndex + 1;

    if (this.props.stepChange) {
      const stepsList = this.state.steps;
      const nextStep = stepsList[nextIndex];
      this.props.stepChange(nextStep.workflow, nextStep.strategy, next\
Step.sequence);
    }

    this.setState({ stepsIndex: nextIndex });
  }
  cycleScenario() {
    const stepsList = this.state.steps;
    const currentStep = stepsList[this.state.stepsIndex];
    let stepsCount = 0;
    for (let i = 0; i < stepsList.length; ++i) {
      if (stepsList[i].symbol === currentStep.symbol) stepsCount++;
    }
    const currentScenario = currentStep.strategy;
    const loopStart =
      (this.state.stepsIndex + stepsCount) >= stepsList.length
        ? 0
        : this.state.stepsIndex + 1;
    for (let i = loopStart; i < stepsList.length; ++i) {
      if (stepsList[i].strategy !== currentScenario) {
        if (this.props.stepChange) {
          const nextStep = stepsList[i];
          this.props.stepChange(nextStep.workflow, nextStep.strategy, \
nextStep.sequence);
        }

        this.setState({ stepsIndex: i });
        break;
      }
    }
  }
  render() {

```

```

const stepsList = this.state.steps;
const currentStep = stepsList[this.state.stepsIndex];
return (
  <div className="workflow">
    <div className="workflow__scenario">
      {currentStep.strategy}
    </div>
    <div className="workflow__text">
      {currentStep.text
        ? currentStep.text
        : <img src={currentStep.img}
          alt={currentStep.strategy} />}
    </div>
    <div className="workflow__nav">
      <div onClick={this.cycleScenario} className="workflow__symbol\
l">
        {currentStep.symbol}
      </div>
      <div className="workflow__steps">
        {currentStep.workflow}
      </div>
      <div onClick={this.cycleSequence} className="workflow__seque\
nce">
        {currentStep.sequence}
      </div>
    </div>
    <br />
    <div className="text--center">
      <small>
        Data Source: {this.state.firebase ? 'Firebase' : 'Local'}
        &nbsp;| Steps count: {this.state.stepsCount}
      </small>
    </div>
  </div>
  );
}
}

```

First interesting thing you will note is how similar this component is to our local Workflow component. Firebase integration is that easy and straight forward.

Let us understand how this component is working, navigating the source from the top.

One change you will notice is we are now treating steps as a state instead of a prop. The reason for doing so is to handle asynchronous data feed from our real-time data store. As our state gets hydrated from default local data store, our component immediately renders these values. When Firebase asynchronous callback within componentDidMount

returns with data, our state is refreshed, and render method called once more with Firebase stored values.

We are also adding another boolean state called `firebase` to indicate which data source is currently active. When you refresh the page that renders our new component, you will notice for a few micro-seconds how data source updates from Local to Firebase.

Firebase data fetch is a single line of code!

```
rsdb.ref('steps').once('value').then((snap) => {  
  this.setState({ steps: snap.val() });  
  this.setState({ firebase: true });  
});
```

What this code is doing is referencing a path `steps` within our Firebase database `rsdb` and returning any values found as `snap` snapshot of the datastore at that path. As explained earlier, because of how we sequentially created our JSON for import into Firebase, the `snap.val()` method actually returns an array of steps objects. By using `once` and `then` we are ensuring the values are only read once every app load and the database does not require active listening. Change this to `on('value', (snap) => {})` to turn on real-time database updates.

To summarize the steps required for Firebase integration with React components.

- Convert local data fixture to JSON structure.
- Read JSON using ES7 `Object.values` feature.
- For representing array data structure use sequential keys starting from 0..N in JSON object.
- Import JSON to Firebase database at a specific path.
- Use state to reference local JSON store as default data.
- Update state to Firebase snapshot data within `componentDidMount` using `async` callback.

- Firebase `snapshot.val()` method returns an array for JSON stored with sequential, contiguous keys starting with 0.

This demonstrates Firebase integration with React components for achieving a truly Serverless Architecture for our apps.

React Developer Experience

This chapter adds more tools and techniques to your already powerful development workflow, making it really awesome and fun to develop in React.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c15 origin/c15-react-developer-experience
npm install
npm start
```

Redux DevTools

Your Redux store is central to how your app manages state. To view how store changes state, calls actions as you run your app, you can install [Chrome extension for Redux DevTools](#).

Update store/roadmap.js

Once installed you will also require to update your store to recognize the DevTools in development and test environments.

```
import { createStore } from 'redux';
import roadmapApp from '../reducers/roadmapApp';

let storeByEnvironment = null;
if (process.env.NODE_ENV === 'production') {
  storeByEnvironment = createStore(roadmapApp);
} else {
```

```

storeByEnvironment =
  createStore(roadmapApp,
    window.devToolsExtension &&
    window.devToolsExtension());
}
const store = storeByEnvironment;
export default store;

```

Now run your app using `npm start` and you will notice the Redux DevTools icon on your Chrome browser bar light up.

The screenshot shows a web browser at `localhost:8080` displaying a 'Roadmap' application. The application has a search bar with the text 'test' and a list of articles. The Redux DevTools interface is open, showing a state tree with a 'features' array. A tooltip is visible over one of the features, showing its structure:

```

{
  "name": "searchText",
  "value": "test"
}

```

The state tree structure is as follows:

- state
 - categoryFilter
 - searchText
 - features
 - features[0]
 - features[1]
 - features[2]
 - features[3]
 - features[4]
 - features[5]
 - features[6]
 - features[7]
 - features[8]
 - features[9]
 - features[10]

The first article in the list is 'Test App Components' with 15 likes. The second article is 'Browsersync and Webpack For Testing Web Apps Across Multiple Devices' with 7 likes.

Redux DevTools Chart

When you click this icon, you can navigate the store for your app as you take UI actions within your app.

Kadira Storybook

So far we have tested our components using Enzyme along with Mocha and Chai. Sometimes you may want to test UI of a complex custom component in isolation, within minimal development or production like environment. You may want to do this test visually, interacting with the component, as you update its design and code.

This is where [Kadira Storybook](#) steps in. You can find several use cases for the Storybook. You can use it to rapidly prototype custom components in isolation of your overall app. You can also use the Storybook to create a component library, demo, documentation, generating a static version for publishing to your customers and stakeholders. We use Storybook for rapid visual testing of custom components as we design these for ReactSpeed website and book.

What we really like about this tool is how well documented it is. Thank you Kadira!

You can install this neat tool using NPM.

```
npm i --save-dev @kadira/storybook
```

Create /.storybook/config.js

Once installed you can configure your storybook like so. Notice that we are importing the CSS entry file for our app.

```
import { configure } from '@kadira/storybook';
import '../app/style.css';

function loadStories() {
  require('../app/stories/button');
  // require as many stories as you need.
}

configure(loadStories, module);
```

Create /app/stories/button.js

We are also requiring certain stories for our components, which we will define next. Here is the story for testing Button custom component.

```
import React from 'react';
import { storiesOf, action } from '@kadira/storybook';
import Button from '../components/Button.jsx';

storiesOf('Button', module)
  .add('with text, default color', () => (
    <Button color="default"
      label="My First Button"
      onClick={action('clicked')} />
  ))
  .add('with icon, primary color', () => (
    <Button color="primary" icon="cog" />
  ));
```

Create `./storybook/webpack.config.js`

Now all that remains is to give Storybook its own `webpack.config.js` for running the CSS loader. Notice that this is mostly copy of our app webpack config.

We have modified our paths to work from `.storybook` folder. We also remove `webpack-dev-server`, `hot` reloading config, and `JSX` loader as Storybook brings its own.

```
// Initialization
const webpack = require('webpack');
const postcssImport = require('postcss-easy-import');
const path = require('path');

const APP = path.join(__dirname, '../app');

// PostCSS support
const precss = require('precss');
const autoprefixer = require('autoprefixer');

module.exports = {
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css', 'postcss'],
        include: APP
      }
    ]
  },
  postcss: function processPostcss(webpack) { // eslint-disable-line \
no-shadow
    return [
      postcssImport({
```

```

        addDependencyTo: webpack
      }),
      precss,
      autoprefixer({ browsers: ['last 2 versions'] })
    ];
  }
};

```

You can add any CSS or scripts loaded in our HTML template by creating `.storybook/head.html` file like so.

```

<!-- Font Awesome CDN integration -->
<link rel="stylesheet"
      href="//maxcdn.bootstrapcdn.com/font-awesome/4.6.3/css/font-awesome.\
min.css">

<!-- jQuery for AJAX GitHub component -->
<script src="//ajax.googleapis.com/ajax/libs/jquery/2.2.2/jquery.min.j\
s"></script>

```

Now you can load Font Awesome icons and perform AJAX and other jQuery integration within the custom components prototyped using Storybook.

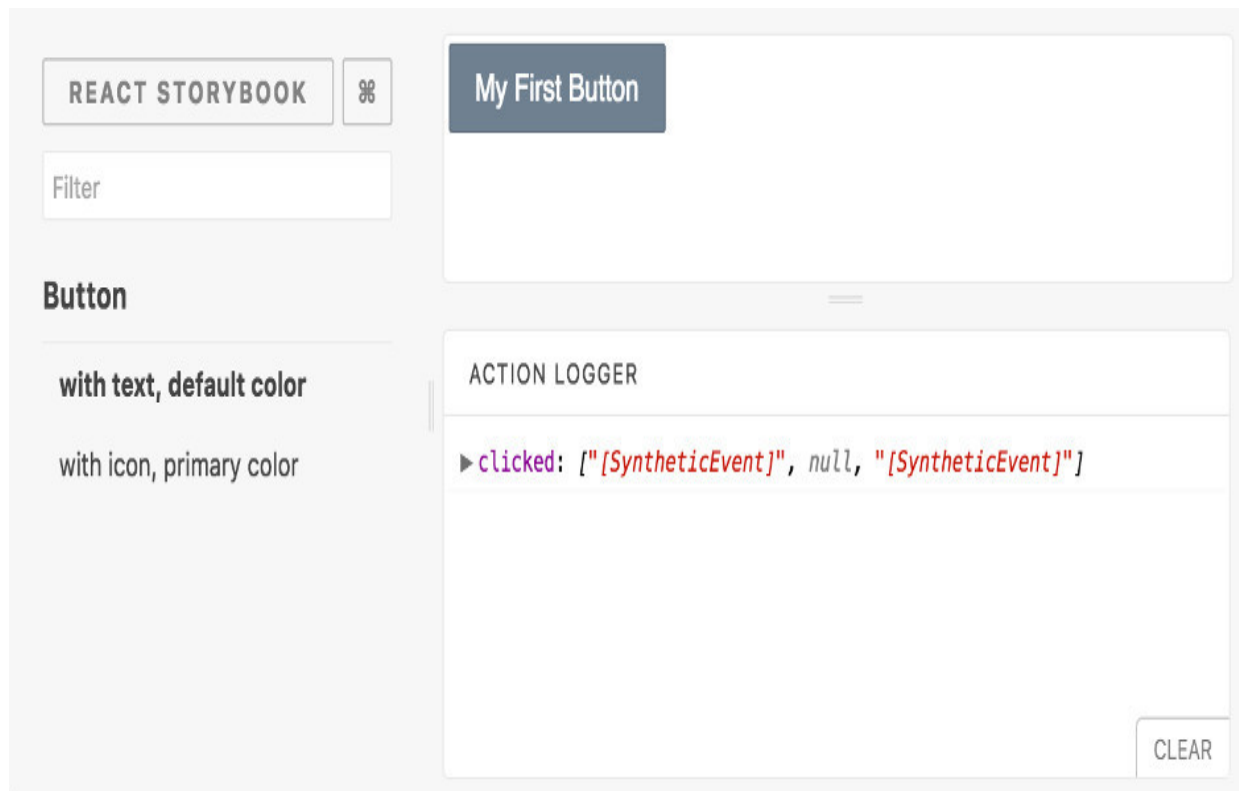
Now we can configure `package.json` to add shortcut script to run Storybook on port 9001. This also configures the static assets path that our components may use.

```

"storybook": "start-storybook -p 9001 -s ./app/public"

```

Run the storybook with `npm run storybook` and view results at `http://localhost:9001` address.



Kadira Storybook

You can also build a static version of your Storybook using following script in package.json.

```
"storybuild": "build-storybook -s ./app/public -o ../.storybook/build" \
```

The static version can be served from any HTTP/file server including Firebase static hosting or GitHub Pages.

Table of Contents

[Awesome React Ecosystem](#)

[Code Along Edition](#)

[Easy start React](#)

[Who this book is for](#)

[Development environment](#)

[Why React is awesome](#)

[Why read React Speed Coding](#)

[Prior art](#)

[Stakeholder perspectives on speed](#)

[Who is using React](#)

[Technology stack](#)

[Measuring speed](#)

[Why learn React comparing with Angular](#)

[Shared learning path between Angular 2 and React](#)

[Setup React Webpack](#)

[Installing starter dependencies](#)

[Configure Webpack in webpack.config.js](#)

[HTML webpack template for index.html](#)

[Configuring startup scripts in package.json](#)

[Run webpack setup](#)

[ES6 React Guide](#)

[Hello World React](#)

[Component composition and naming](#)

[Files and folder structure](#)

[Root component index.jsx](#)

[Module import](#)

[World.jsx component](#)

[Class definition](#)

[Constructor](#)

[Event Handlers and setState](#)

[JSX and the render method](#)

[Template literals and ternary conditionals](#)

[Properties](#)

- [Event UI binding](#)
- [Controlled components](#)
- [PropTypes and defaultProps](#)
- [ES7 Property Initializers](#)
- [Complete World.jsx listing](#)
- [Hello.jsx stateless component](#)
- [Component styles in world.css](#)
- [Base styles in element.css](#)
- [Entry CSS using style.css](#)
- [Run development server](#)
- [React Chrome Extension](#)

[Production Optimize Webpack](#)

- [Add production plugins and supporting dependencies](#)
- [HTML minifying](#)
- [CSS minifying](#)
- [Webpack production webpack.prod.config.js](#)
- [Initialization](#)
- [Entry points](#)
- [Loaders](#)
- [PostCSS and plugins](#)
- [Plugins](#)
- [Profiling Webpack build](#)

[ReactSpeed UI](#)

- [ReactSpeed UI objectives](#)
- [PostCSS processing](#)
- [BEM CSS naming method](#)
- [Configurable theme using variables.css](#)
- [Theme definition in theme.css](#)
- [Updating elements.css](#)
- [Typography in type.css](#)
- [Utilities in util.css](#)
- [Image styles in image.css](#)
- [Flexbox layout in layout.css](#)
- [Import styles in style.css](#)
- [Define card component styles in card.css](#)
- [Card.jsx component](#)

[Home.jsx landing_page component](#)

[Start Component Design](#)

[Embed to React](#)

[CSS libraries to React](#)

[API to React](#)

[Define Component Internals](#)

[Naming files, folders, and modules](#)

[Imports and exports](#)

[Stateless components and pure functions](#)

[Classes and inheritance](#)

[Constructor and binding](#)

[Properties and property types](#)

[State management](#)

[Lifecycle methods](#)

[Event handlers](#)

[Render and ReactDOM.render methods](#)

[JSX features and syntax](#)

[Workflow.jsx component](#)

[Wire Multiple Components](#)

[Render multiple components programmatically](#)

[Composition using parent child node tree](#)

[Presentational and container components](#)

[Reconciliation algorithm and keys for dynamic children](#)

[Integrating vendor components](#)

[Routing to wire component layouts](#)

[Route Component Layouts](#)

[Layout strategy](#)

[AboutBook.jsx component](#)

[AboutEmbeds.jsx component](#)

[Configure routes in index.js](#)

[Nav.jsx component](#)

[Navigation styles in nav.css](#)

[Footer.jsx component](#)

[Landing.jsx layout component](#)

[NavLink component](#)

[Programmatic routing in Ribbon.jsx](#)
[Refactoring IconText.jsx for ribbon menu](#)
[Search engine friendly URLs](#)
[Handling router exceptions](#)

[Lint React Apps](#)

[Browsersync multi-device testing](#)
[Lint config in webpack.lint.config.js](#)
[JavaScript lint using ESLint](#)
[Configuring eslint in eslintrc.js](#)
[Eslint command line interface](#)
[Fixing ESLint reported problems](#)
[ESLint webpack.lint.config.js integration](#)
[StyleLint for CSS](#)
[StyleLint CLI](#)
[Fixing StyleLint reported problems](#)
[Webpack integration for StyleLint](#)
[Complete webpack.lint.config.js listing](#)

[Test App Components](#)

[Mocha Chai Behavior-Driven Development](#)
[Test 01_mocha_timeout.spec.js](#)
[Test 02_mocha_chai.spec.js](#)
[Enzyme React component testing](#)
[JSDOM browser.js helper](#)
[Test 01_workflow.spec.js](#)
[Sinon spy methods and events](#)
[Istanbul code coverage](#)

[Refactor Existing Components](#)

[ES5 to ES6 conversion in TodoList.jsx](#)
[Adding TodoApp to AboutCustom.jsx](#)
[Testing and refactoring](#)
[Refactoring for converting standard React apps to Redux](#)
[Refactoring for optimizing React apps](#)
[Refactoring Font Awesome to SVG icons](#)
[IconSvg.jsx custom component](#)
[Icon data fixture in icons.js](#)

Redux State Container

The Roadmap app

Redux basics

State tree definition

Redux spec in 02_roadmap_redux.spec.js

Actions for Roadmap in /actions/roadmap.js

Reducers in /reducers/roadmap.js

Store in /store/roadmap.js

Test store, actions, and reducers

Optimize Redux app

Component hierarchy spec in 03_roadmap.spec.js

Rapid prototype hierarchy in roadmap.jsx

Data in fixtures/roadmap/features.js

Extract Feature.jsx component

Extract FeatureList.jsx component

Refactoring Roadmap.jsx app

Redux Wiring App

Extract CategoryButton.jsx component

Refactor data fixture features.js

Refactor actions in actions/roadmap.js

Extract SearchFeature.jsx component

Connecting Store in VisibleFeatureList.jsx

FilterCategoryButton.jsx container component

Refactor Roadmap.jsx

Hydrate Redux app using reducers/hydrate.js

Pass store in index.js

Update test suite 03_roadmap.spec.js

Update 02_roadmap_redux.spec.js

Firebase React Integration

Comparing Firebase with Meteor

Firebase Hosting

How Firebase stores files and data

Designing a REST API using Firebase

For what kind of apps is Firebase not ideal

Refactor Workflow component for Firebase integration

[Data in fixtures/workflow/steps.json](#)

[Refactor Workflow.jsx](#)

[Firebase config data in fixtures/rsdb.js](#)

[Pass route param using index.js](#)

[Refactor AboutWorkflow.jsx](#)

[Create WorkflowFire.jsx](#)

[React Developer Experience](#)

[Redux DevTools](#)

[Kadira Storybook](#)

[Create /.storybook/config.js](#)

[Create /app/stories/button.js](#)

[Create /.storybook/webpack.config.js](#)