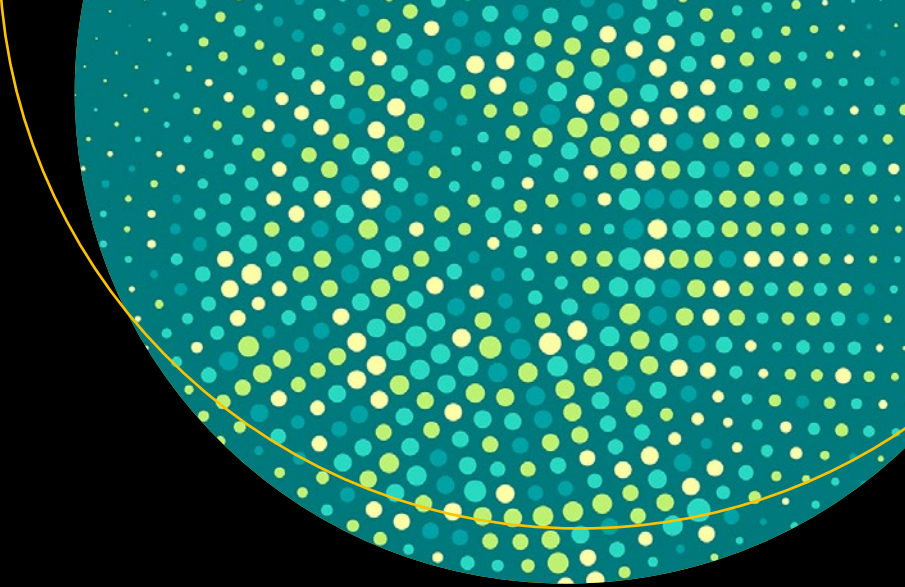




Pro MERN Stack

Full Stack Web App Development with
Mongo, Express, React, and Node

Vasan Subramanian



Apress®

Pro MERN Stack

Full Stack Web App Development
with Mongo, Express, React,
and Node



Vasan Subramanian

Apress®

Pro MERN Stack

Vasan Subramanian
Bangalore, Karnataka, India

ISBN-13 (pbk): 978-1-4842-2652-0
DOI 10.1007/978-1-4842-2653-7

ISBN-13 (electronic): 978-1-4842-2653-7

Library of Congress Control Number: 2017933833

Copyright © 2017 by Vasan Subramanian

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Pramila Balan
Development Editor: Poonam Jain
Technical Reviewer: Anshul Chanchlani
Coordinating Editor: Prachi Mehta
Copy Editor: Mary Behr
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover image designed by Freepik

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-2652-0. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

To Sandeep and Fazle.

Contents at a Glance

- About the Author xvii
- Chapter 1: Introduction 1
- Chapter 2: Hello World..... 17
- Chapter 3: React Components 37
- Chapter 4: React State..... 55
- Chapter 5: Express REST APIs 69
- Chapter 6: Using MongoDB..... 93
- Chapter 7: Modularization and Webpack 115
- Chapter 8: Routing with React Router 151
- Chapter 9: Forms 173
- Chapter 10: React-Bootstrap 207
- Chapter 11: Server Rendering 245
- Chapter 12: Advanced Features 275
- Chapter 13: Looking Ahead 319
- Index..... 325

Contents

About the Author xvii

■ Chapter 1: Introduction 1

 What Is MERN? 1

 Who Should Read This Book..... 2

 Structure of the Book 2

 Conventions..... 3

 What You Need 5

 MERN Components..... 5

 React 5

 Node.js..... 7

 Express 9

 MongoDB 10

 Tools and Libraries 12

 Why MERN? 13

 JavaScript Everywhere..... 13

 JSON Everywhere 14

 Node.js Performance 14

 The npm Ecosystem 14

 Isomorphic..... 14

 It's not a Framework! 15

 Summary..... 15

■ Chapter 2: Hello World	17
Server-Less Hello World	17
Server Setup	20
npm	21
Node.js	21
Project	22
npm	22
Express	24
Build-Time JSX Compilation	26
Separate Script File	27
Transform	28
Automate	29
React Library	30
ES2015	30
Summary	33
Answers to Exercises	34
Exercise: JSX	34
Exercise: npm	34
Exercise: Express	35
Exercise: Babel	35
Exercise: ES2015	36
■ Chapter 3: React Components	37
Issue Tracker	37
React Classes	38
Composing Components	40

Passing Data	42
Using Properties	42
Property Validation.....	44
Using Children	45
Dynamic Composition.....	47
Summary	51
Answers to Exercises	51
Exercise: React Classes	51
Exercise: Passing Data	52
Exercise: Dynamic Composition.....	52
■ Chapter 4: React State.....	55
Setting State.....	55
Async State Initialization	58
Event Handling	60
Communicating from Child to Parent	60
Stateless Components.....	63
Designing Components	65
State vs. props.....	65
Component Hierarchy	65
Communication.....	66
Stateless Components.....	66
Summary	66
Answers to Exercises	67
Exercise: Setting State.....	67
Exercise: Communicate Child to Parent.....	68

■ Chapter 5: Express REST APIs	69
REST	69
Resource Based.....	69
HTTP Methods as Actions	70
JSON.....	71
Express.....	72
Routing	72
Handler Function	73
Middleware.....	75
The List API.....	76
Automatic Server Restart	77
Testing.....	77
The Create API.....	80
Using the List API	82
Using the Create API.....	84
Error Handling	85
Summary.....	88
Answers to Exercises	89
Exercise: The List API.....	89
Exercise: Create API.....	90
Exercise: Using the List API	90
Exercise: Using the Create API.....	91
Exercise: Error Handling	91
■ Chapter 6: Using MongoDB.....	93
MongoDB Basics	93
Documents	93
Collections	94
Query Language	94

Installation	95
The mongo Shell.....	95
Shell Scripting	99
Schema Initialization.....	99
MongoDB Node.js Driver	101
Callbacks	103
Promises.....	104
Generator and co Module	104
The async Module.....	105
Reading from MongoDB	107
Writing to MongoDB	109
Summary.....	111
Answers to Exercises	112
Exercise: Mongo Shell	112
Exercise: Schema Initialization.....	112
Exercise: Reading from MongoDB	113
Exercise: Writing to MongoDB	113
■ Chapter 7: Modularization and Webpack	115
Server-Side Modules.....	115
Introduction to Webpack	117
Using Webpack Manually	118
Transform and Bundle	120
Libraries Bundle	125
Hot Module Replacement	129
HMR Using Middleware	132
Comparison of HMR Alternatives.....	133
Debugging	134
Server-Side ES2015	135

ESLint	140
Environment	142
Summary	147
Answers to Exercises	148
Exercise: Transform and Bundle	148
Exercise: Hot Module Replacement	148
Exercise: Server-Side ES2015	149
Exercise: ESLint	150
■ Chapter 8: Routing with React Router	151
Routing Techniques	152
Simple Routing	152
Route Parameters.....	154
Route Query String	157
Programmatic Navigation.....	161
Nested Routes	164
Browser History.....	167
Summary	169
Answers to Exercises	169
Exercise: Route Parameters.....	169
Exercise: Route Query String	169
Exercise: Programmatic Navigation.....	170
■ Chapter 9: Forms	173
More Filters in the List API	173
Filter Form.....	174
The Get API	180
Edit Page	182

UI Components	186
Number Input.....	186
Date Input	190
Update API	195
Using Update API	198
Delete API	200
Using the Delete API	201
Summary	203
Answers to Exercises	203
Exercise: More Filters in List API	203
Exercise: Filter Form	203
Exercise: Edit Page	204
Exercise: Date Input.....	204
Exercise: Update API	205
■ Chapter 10: React-Bootstrap	207
Bootstrap Installation	207
Navigation	210
Table and Panel	216
Forms	218
Grid-Based Forms.....	218
Inline Forms.....	222
Horizontal Forms	224
Alerts	229
Validations	229
Results.....	231
Modals.....	237
Summary.....	242

Answers to Exercises	243
Exercise: Navigation	243
Exercise: Grid-Based Forms	243
Exercise: Inline Forms	243
Exercise: Modals.....	244
■ Chapter 11: Server Rendering	245
Basic Server Rendering.....	245
Handling State.....	250
Initial State	252
Server-Side Bundle	254
Back-End HMR	256
Routed Server Rendering	260
Encapsulated Fetch.....	268
Summary	272
Answers to Exercises	273
Back-End HMR.....	273
Routed Server Rendering	273
■ Chapter 12: Advanced Features	275
MongoDB Aggregate.....	275
Pagination	284
Higher Order Components.....	288
Search Bar.....	297
Google Sign-In.....	303
Session Handling.....	310
Summary	317

■ Chapter 13: Looking Ahead	319
Mongoose.....	319
Flux.....	320
Deployment	322
mern.io	323
That's All, Folks!	324
Index.....	325

About the Author



Vasan Subramanian has experienced all kinds of programming, from 8-bit, hand-assembled code on an 8085 to AWS Lambda. He not only loves to solve problems using software, but he also looks for the right mix of technology and processes to make a software product team most efficient. He learned software development at companies such as Corel, Wipro, and Barracuda Networks, not just as a programmer but also as a leader of teams at those companies.

Vasan studied at IIT Madras and IIM Bangalore. In his current job as CTO at Accel, he mentors startups on all things tech. While not mentoring or coding (or writing books!), Vasan runs half marathons and plays 5-a-side soccer. He can be contacted at vasan.promern@gmail.com for boquets, brickbats, or anything in-between.

CHAPTER 1



Introduction

Web application development is not what it used to be even a couple of years back. Today, there are so many options, and the uninitiated are often confused about what's good for them. This applies not just to the broad *stack* (the various tiers or technologies used), but also to the tools that aid in development; there are so many choices. This book stakes a claim that the MERN stack is great for developing a complete web application, and it takes the reader through all that is necessary to get it done.

In this chapter, I'll give a broad overview of the technologies that make up the MERN stack. I won't go into details or examples in this chapter; I'll just introduce the high-level concepts. This chapter will focus on how these concepts affect an evaluation of whether MERN is a good choice for your next web application project.

What Is MERN?

Any web application is made by using multiple technologies. The combination of these technologies is called a “stack,” popularized by the LAMP stack, which is an acronym for Linux, Apache, MySQL, and PHP, which are all open-source components. As web development matured and interactivity came to the fore, single page applications (SPAs) became more popular. An SPA is a web application paradigm that avoids refreshing a web page to display new content; it instead uses lightweight calls to the server to get some data or snippets and updates the web page. The result looks quite nifty when compared to the old way of reloading the page entirely. This brought about a rise in front-end frameworks, since much of the work was done on the client side. At approximately the same time, although completely unrelated, NoSQL databases also started gaining popularity.

The MEAN (MongoDB, Express, AngularJS, Node.js) stack was one of the early open-source stacks that epitomized this shift towards SPAs and the adoption of NoSQL. AngularJS, a front-end framework based on the model-view-controller (MVC) design pattern, anchored this stack. MongoDB, a very popular NoSQL database, was used for persistent data storage. Node.js, a server-side JavaScript runtime environment, and Express, a web server built on Node.js, formed the middle tier, or the web server. This stack is arguably the most popular stack for any new web application these days.

Not exactly competing, but React, an alternate front-end technology from Facebook, has been gaining popularity and offers a replacement to AngularJS. It thus replaces the “A” with an “R” in MEAN, to give us the MERN Stack. I said “not exactly” since React is not a full-fledged MVC framework. It is a JavaScript library for building user interfaces, so in some sense it’s the View part of the MVC.

Although we pick a few defining technologies to define a stack, these are not enough to build a complete web application. Other tools are required to help the process of development, and other libraries are needed to complement React. This book is about all of them: how to build a complete web application based on the MERN stack, using other complementary tools that make it easy for us to do it.

Who Should Read This Book

Developers and architects who have prior experience in any web app stack other than the MERN stack will find this book useful for learning about this modern stack. Prior knowledge of how web applications work is required. Knowledge of JavaScript is also required. It is further assumed that the reader knows the basics of HTML and CSS. It will greatly help if you are also familiar with the version control tool git; you can try out the code just by cloning the git repository that holds all the source code described in this book, and running each step by just checking out a branch.

If you have decided that your new app will use the MERN stack, then this book will help you quickly get off the ground. Even if you have not made any decision, reading the book will get you excited about MERN and equip you with enough knowledge to make that decision for a future project. The most important thing you will learn is how to put together multiple technologies and build a complete, functional web application; by the book’s end, you’ll be a full-stack developer or architect on MERN.

Structure of the Book

Although the focus of the book is to teach you how to build a complete web application, most of the book revolves around React. That’s just because, as is true of most modern web applications, the front-end code forms the bulk. And in this case, React is used for the front end.

The tone of the book is tutorial-like. What this means is that unless you try out the code and solve the exercises yourself, you will not get the full benefit of reading the book. There are plenty of code listings in the book (this code is also available online in a GitHub repository, at <https://github.com/vasansr/pro-mern-stack>). I encourage you *not* to copy/paste; instead, please type out the code yourself. I find this very valuable in the learning process. There are very small nuances, such as the types of quotes, which can cause a big difference. When you actually type out the code, you are much more conscious of these things than when you are just reading it. Clone the repository only when you are stuck and want to compare it with my code, which has been tested and confirmed to work. And if you do copy/paste small sections, don’t do it from the electronic version of the book, as the typography of the book may not be a faithful reproduction of the actual code.

I have also added a checkpoint (a git branch, in fact) after every change that can be tested in isolation, so that you can look at the exact diffs between two checkpoints, online. The checkpoints and links to the diffs are listed in the home page (the README) of the repository. You may find this more useful than looking at the entire source, or even the listings in the text of this book, as GitHub diffs are far more expressive than what I can do in this book.

Rather than cover one topic or technology per section, I have adopted a more practical and problem-solving approach. You will have developed a full-fledged working application by the end of the book, but you'll start small with a Hello World example. Just as in a real project, you will add more features to the application as you progress. When you do this, you'll encounter tasks that need additional concepts or knowledge to proceed. For each of these tasks, I will introduce the concept or technology that can be used, and I'll discuss it in detail. Thus, you may not find one chapter or section devoted purely to one topic or technology; instead, each chapter will be a set of goals you want to achieve in the application. You will be switching between technologies and tools as you progress.

I have included exercises wherever possible to make you either think or look up various documentation pages on the Internet. This is so that you know where to get additional information for things that are not covered in the book, such as very advanced topics or APIs.

I have chosen an issue tracking application as the application that you'll build. It's something most developers can relate to, and it has many of the attributes and requirements that any enterprise application will have, commonly referred to as a "CRUD" application (CRUD stands for Create, Read, Update, Delete of a database record).

Conventions

Many of the conventions used in the book are quite obvious, so I won't explain all of them. However, I will cover some conventions with respect to how the code is shown if they're not obvious.

Each chapter has multiple sections, and each section is devoted to one set of code changes that results in a working application and can be tested. One section can have multiple listings, each of which may not be testable by itself. Every section will also have a corresponding entry in the GitHub repository, where you can see the complete source of the application at the end of that section, as well as the differences between the previous section and the current section. You will find the difference view very useful to identify the changes made in the section.

All code changes will appear in the listings within the section, but do not rely on their accuracy. The reliable and working code can be found in the GitHub repository, which may even have undergone last minute changes that couldn't make it to the book in time. All listings will have a listing caption, which will include the name of the file being changed or created.

A listing is a full listing if it contains a file, a class, a function, or an object in its entirety. A full listing may also contain two or more classes, functions, or objects, but not multiple files. In such a case, if the entities are not consecutive, I'll use ellipses to indicate chunks of unchanged code.

Listing 1-1 is an example of a full listing, the contents of an entire file.

Listing 1-1. server.js: Express server

```
const express = require('express');

const app = express();
app.use(express.static('static'));

app.listen(3000, function () {
  console.log('App started on port 3000');
});
```

A partial listing, on the other hand, will not list complete files, functions, or objects. It will start and end with an ellipsis, and will have ellipses in the middle to skip chunks of code that have not changed. Wherever possible, the actual changes will be highlighted. The changes will be highlighted in bold, and the unchanged code will be in the normal font. Listing 1-2 is an example of a partial listing that has small changes.

Listing 1-2. package.json: Adding Scripts for Transformation

```
...
"scripts": {
  "compile": "babel src --presets react,es2015 --out-dir static",
  "watch": "babel src --presets react,es2015 --out-dir static --watch",
  "test": "echo \"Error: no test specified\" && exit 1"
},
...
```

Deleted code will be shown using strikethrough, as in Listing 1-3.

Listing 1-3. index.html: Change in Script Name and Type

```
...
<script
  src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser-
min.js"></script>
...
```

Code blocks are used within regular text to cull out changes in code for discussion, and are often a repetition of code in listings. These are not listings, and are often just a line or two. The following is an example, where the line is extracted out of a listing, and one word is highlighted:

```
...
const contentNode = ...
...
```

All commands that need to be executed on the console will be in the form a code block starting with \$. Here is an example:

```
$ npm install express
```

What You Need

You will need a computer where you can run your server and do other tasks such as compilation. You also need a browser to test your application. I recommend a Linux-based computer running Ubuntu or a Mac as your development server, but with minor changes, you could also use a Windows PC.

If you have a Windows PC, an option is to run an Ubuntu server virtual machine using Vagrant (www.vagrantup.com/). This is helpful because you will eventually need to deploy your code on a Linux-based server, and it is best to get used to that environment from the beginning. But you may find it difficult to edit files using the console. In that case, an Ubuntu desktop variant may work better for you, but it requires more memory for the virtual machine.

Running Node.js directly on Windows will also work, but the code samples in this book assume a Linux-based PC or Mac. If you choose to run directly on a Windows PC, you may have to make the appropriate changes, especially when running commands in the shell, using a copy instead of using soft links, and in rare cases, to deal with `'\'` vs. `'/'` in path separators.

Further, to keep the book concise, I have not included installation instructions for packages, and they are different for different operating systems. You will need to follow the installation instructions from the package providers' websites. And in many cases I have not included direct links to websites even though I ask you to look them up. This is for a couple of reasons. The first is to let you learn by yourself how to search for them. The second is that the link I may provide may have moved to another location due to the fast-paced changes that the MERN stack was experiencing at the time of writing this book.

MERN Components

I'll give a quick introduction to the main components that form the MERN stack and a few other libraries and tools that you'll be using to build your web application. I'll just touch upon the salient features, and leave the details to other chapters where they are more appropriate.

React

React anchors the MERN stack. In some sense, it is the defining component of the MERN stack.

React is an open-source JavaScript library maintained by Facebook that can be used for creating views rendered in HTML. Unlike AngularJS, React is not a framework. It is a library. Thus, it does not, by itself, dictate a framework pattern such as the MVC pattern. You use React to render a view (the V in MVC), but how to tie the rest of the application together is completely up to you.

I'll discuss a few things about React that make it stand out.

Why Facebook Invented React

The Facebook folks built React for their own use, and later they open-sourced it. Why did they have to build a new library when there are tons of them out there?

React was born not in the Facebook application that we all see, but rather in Facebook's Ads organization. Originally, they used a typical client-side MVC model, which had all of the regular two-way data binding and templates. Views would listen to changes on models, and they would respond to those changes by updating themselves.

Soon, this got pretty hairy as the application became more and more complex. What would happen was that a change would cause an update, which would cause another update (because something changed due to that update), which would cause yet another, and so on. Such cascading updates became difficult to maintain because there were subtle difference in the code to update the view, depending on the root cause of the update.

Then they thought, why do we need to deal with all this, when all the code to depict the model in a view is already there? Aren't we replicating the code by adding smaller and smaller snippets to manage transitions? Why can't we use the *templates* (that is, the views) themselves to manage state changes?

That's when they started thinking of building something **declarative** rather than **imperative**.

Declarative

React views are declarative. What this really means is that you, as a programmer, don't have to worry about managing the effect of changes in the view's state or the data. In other words, you don't worry about transitions or mutations in the DOM caused by changes to the view's state. How does this work?

A React component *declares* how the view looks like, given the data. When the data changes, if you are used to the jQuery way of doing things, you'd typically do some DOM manipulation. Not in React. You just don't do anything! The React library figures out how the new view looks, and just applies the changes between the old view and the new view. This makes the views consistent, predictable, easier to maintain, and simpler to understand.

Won't this be too slow? Won't it cause the entire screen to be refreshed on every data change? Well, React takes care of this using its **virtual DOM** technology. You declare how the view looks, not in the form of HTML or a DOM, but in the form of a virtual representation, an in-memory data structure. React can compute the differences in the virtual DOM very efficiently, and can apply only these changes to the actual DOM. Compared to manual updates which do only the required DOM changes, this adds very little overhead because the algorithm to compute the differences in the virtual DOM has been optimized to the hilt.

Component-Based

The fundamental building block of React is a component, which maintains its own state and renders itself.

In React, all you do is build components. Then, you put components together to make another component that depicts a complete view or page. A component encapsulates the state of data and the view, or how it is rendered. This makes writing and reasoning about the entire application easier, by splitting it into components and focusing on one thing at a time.

Components talk to each other by sharing state information in the form of read-only properties to their child components and by callbacks to their parent components. I'll dig deeper into this concept in a later chapter, but the gist of it is that components in React are very cohesive, and the coupling with one another is minimal.

No Templates

Many web application frameworks rely on templates to automate the task of creating repetitive HTML or DOM elements. The templating language in these frameworks is something that the developer will have to learn and practice. Not in React.

React uses a full-featured programming language to construct repetitive or conditional DOM elements. That language is none other than JavaScript. For example, when you want to construct a table, you write a `for(...)` loop in JavaScript, or use the `map()` function of an `Array`.

There is an intermediate language to represent a virtual DOM, and that is JSX, which is very similar to HTML. It lets you create nested DOM elements in a familiar language rather than hand-construct them using JavaScript functions. Note that JSX is not a programming language; it is a representational markup like HTML. It's also very similar to HTML so you don't have to learn too much. More about this later.

You don't have to use JSX; you can write pure JavaScript to create your virtual DOM if you prefer. But if you are used to HTML, it's simpler to just use JSX. Don't worry about it; it's really not a new language that you'll need to learn.

Isomorphic

React can be run on the server too. That's what *isomorphic* means: the same code can run on both server and the browser.

This allows you to create pages on the server when required, for example, for SEO purposes. The same code can be shared on the server to achieve this. On the server, you'll need something that can run JavaScript, and this is where I introduce Node.js.

Node.js

Simply put, Node.js is JavaScript outside of a browser. The creators of Node.js just took Chrome's V8 JavaScript engine and made it run independently as a JavaScript runtime. If you are familiar with the Java runtime that runs Java programs, you can easily relate to the JavaScript runtime: the Node.js runtime runs JavaScript programs.

Node.js Modules

In a browser, you can load multiple JavaScript files, but you need an HTML page to do all that. You cannot refer to another JavaScript file from one JavaScript file. But for Node.js, there is no HTML page that starts it all. In the absence of the enclosing HTML page, Node.js uses its own module system based on CommonJS to put together multiple JavaScript files.

Modules are like libraries. You can include the functionality of another JavaScript file (provided it's written to follow a module's specifications) by using the keyword `require` (which you won't find in a browser's JavaScript). You can therefore split your code into files or modules for the sake of better organization, and load one or another using `require`. I'll talk about the exact syntax in a later chapter; at this point it's enough to note that compared to JavaScript on the browser, there is a cleaner way to modularize your code using Node.js.

Node.js ships with a bunch of core modules compiled into the binary. These modules provide access to the operating system elements such as the file system, networking, input/output, etc. They also provide some utility functions that are commonly required by most programs.

Apart from your own files and the core modules, you can also find a great amount of third-party open source libraries available for easy installation. This brings us to npm.

Node.js and npm

npm is the default package manager for Node.js. You can use npm to install third-party libraries (packages) and also manage dependencies between them. The npm registry (www.npmjs.com) is a public repository of all modules published by people for the purpose of sharing.

Although npm started off as a repository for Node.js modules, it quickly transformed into a package manager for delivering other JavaScript-based modules, notably those that can be used in the browser. jQuery, by far the most popular client-side JavaScript library, is available as an npm module. In fact, even though React is largely client-side code and can be included directly in your HTML as a script file, it is recommended instead that React is installed via npm. But, once installed as a package, we need something to put all the code together that can be included in the HTML so that the browser can get access to the code. For this, there are build tools such as browserify or webpack that can put together your own modules as well as third-party libraries in a bundle that can be included in the HTML.

As of the writing this book, npm tops the list of module or package repositories, having more than 250,000 packages (source: www.modulecounts.com). Maven, which used to be the biggest two years back, has just half the number now. This shows that npm is not just the largest, but also the fastest growing repository. It is often touted that the success of Node.js is largely owed to npm and the module ecosystem that has sprung around it.

npm is not just easy to use both for creating and using modules; it also has a unique conflict resolution technique that allows multiple conflicting versions of a module to exist side-by-side to satisfy dependencies. Thus, in most cases, npm just works.

Node.js Is Event Driven

Node.js has an asynchronous, event-driven, non-blocking input/output (I/O) model, as opposed to using threads to achieve multitasking.

Most languages depend on threads to do things simultaneously. But in fact, there is no such thing as simultaneous when it comes to a single processor running your code. Threads give the feeling of simultaneousness by letting other pieces of code run while one piece waits (blocks) for some event to complete. Typically, these are I/O events such as reading from a file or communicating over the network. For a programmer, this means that you write your code sequentially. For example, on one line, you make a call to open a file, and on the next line, you have your file handle ready. What really happens is that your code is *blocked* while the file is being opened. If you have another thread running, the operating system or the language can switch out this code and start running some other code during the blocked period.

Node.js, on the other hand, has no threads. It relies on *callbacks* to let you know that a pending task is completed. So, if you write a line of code to open a file, you supply it with a callback function to receive the results. On the next line, you continue to do other things that don't require the file handle. If you are accustomed to asynchronous Ajax calls, you will immediately know what I mean. Event-driven programming is natural to Node.js due to the underlying language constructs such as closures.

Node.js achieves multitasking using an *event loop*. This is nothing but a queue of events that need to be processed and callbacks to be run on those events. In the above example, the file that is ready to be read is an event that will trigger the callback you supplied while opening it. If you don't understand this completely, don't worry. The examples in the rest of this book should make you comfortable about how it really works.

On one hand, an event-based approach makes Node.js applications fast and lets the programmer be blissfully oblivious of the semaphores and locks that are utilized to synchronize multi-threaded events. On the other hand, getting used to this model takes some learning and practice.

Express

Node.js is just a runtime environment that can run JavaScript. To write a full-fledged web server by hand on Node.js directly is not that easy, nor is it necessary. Express is the framework that simplifies the task of writing your server code.

The Express framework lets you define *routes*, specifications of what to do when a HTTP request matching a certain pattern arrives. The matching specification is regular expression (regex) based and is very flexible, like most other web application frameworks. The what-to-do part is just a function that is given the parsed HTTP request.

Express parses request URL, headers, and parameters for you. On the response side, it has, as expected, all of the functionality required by web applications. This includes setting response codes, setting cookies, sending custom headers, etc. Further, you can write Express middleware, which are custom pieces of code that can be inserted in any request/response processing path to achieve common functionality such as logging, authentication, etc.

Express does not have a template engine built in, but it supports any template engine of your choice such as pug, mustache, etc. But, for an SPA, you will not need to use a

server-side template engine. This is because all dynamic content generation is done on the client, and the web server only serves static files and data via API calls. Especially with MERN stack, page generation is handled by React itself on the server side.

In summary, Express is a web server framework meant for Node.js, and it is not very different from many other server-side frameworks in terms of what you can achieve with it.

MongoDB

MongoDB is the database used in the MERN stack. It is a NoSQL document-oriented database, with a flexible schema and a JSON-based query language. I'll discuss a few things that MongoDB is (and is not) here.

NoSQL

NoSQL stands for “non-relational,” no matter what the acronym expands to. It's essentially *not* a conventional database where you have tables and columns (called a relational database). I find that there are two attributes of NoSQL that differentiate it from the conventional.

The first is the ability to horizontally scale by distributing the load over multiple servers. NoSQL databases do this by sacrificing an important (for some) aspect of the traditional databases: strong consistency. That is, the data is not necessarily consistent for very brief amounts of time across replicas. For more information, read up on the “CAP theorem” (https://en.wikipedia.org/wiki/CAP_theorem). But in reality, very few applications require web scale, and this aspect of NoSQL databases comes into play very rarely.

The second, and to me, more important, aspect is that NoSQL databases are not necessarily relational databases. You don't have to think of your data in terms of rows and columns of tables. The difference in the representation in the application and on disk is sometimes called impedance mismatch. This is a term borrowed from electrical engineering, and it means, roughly, that we're not talking the same language. In MongoDB, instead, you can think of the persisted data just as you see it in your application code; that is, as objects or documents. This helps a programmer avoid a translation layer, whereby one has to convert or map the objects that the code deals with to relational tables. Such translations are called object relational mapping (ORM) layers.

Document-Oriented

Compared to relational databases where data is stored in the form of relations, or tables, MongoDB is a document-oriented database. The unit of storage (comparable to a row) is a *document*, or an object, and multiple documents are stored in *collections* (comparable to a table). Every document in a collection has a unique identifier by which it can be accessed. The identifier is indexed automatically.

Imagine the storage structure of an invoice, with the customer name, address, etc. and a list of items (lines) in the invoice. If you had to store this in a relational database, you would use two tables, say, `invoice` and `invoice_lines`, with the lines or items referring to the invoice via a foreign-key *relation*. Not so in MongoDB. You would store the entire invoice as a single document, fetch it, and update it in an atomic operation. This applies not just to line items in an invoice. The document can be any kind of deeply nested object.

Modern relational databases have started supporting one level of nesting by allowing array fields and JSON fields, but it is not the same as a true document database. MongoDB has the ability to index on deeply nested fields, which relational databases cannot do.

The downside is that the data is stored denormalized. This means that data is sometimes duplicated, requiring more storage space. Also, things like renaming a master (catalog) entry name would mean sweeping through the database. But then, storage has become relatively cheap these days, and renaming master entries are rare operations.

Schema-Less

Storing an object in a MongoDB database does not have to follow a prescribed schema. All documents in a collection need not have the same set of fields.

This means that, especially during early stages of development, you don't need to add/rename columns in the schema. You can quickly add fields in your application code without having to worry about database migration scripts. At first, this may seem a boon, but in effect all it does is transfer the responsibility of data sanity from the database to your application code. I find that in larger teams and more stable products, it is better to have a strict or semi-strict schema. Using object document mapping libraries such as `mongoose` (not covered in this book) alleviates this problem.

JavaScript Based

MongoDB's language is JavaScript.

For relational databases, there is a query language called SQL. For MongoDB, the query language is based on JSON: you create, search for, make changes, and delete documents by specifying the operation in a JSON object. The query language is not English-like (you don't `SELECT` or say `WHERE`), and therefore much easier to construct programmatically.

Data is also interchanged in JSON format. In fact, the data is natively stored in a variation of JSON called BSON (where B stands for Binary) in order to efficiently utilize space. When you retrieve a document from a collection, it is returned as a JSON object.

MongoDB comes with a shell that is built on top of a JavaScript runtime like `Node.js`. This means that you have a powerful and familiar scripting language (JavaScript) to interact with the database via command line. You can also write code snippets in JavaScript that can be saved and run on the server (the equivalent of stored procedures).

Tools and Libraries

It's hard to build any web application without using tools to help you on your way. Here's a brief introduction to the other tools apart from the MERN stack components that you will be using to develop your sample application in this book.

React-Router

React supplies only the view rendering capability and helps manage interactions in a single component. When it comes to transitioning between different views of the component and keeping the browser URL in sync with the current state of the view, we need something more.

This capability of managing URLs and history is called routing. It is similar to the server-side routing that Express does: a URL is parsed, and based on its components, a piece of code is associated with the URL. React-Router not only does this, but also manages the browser's Back button functionality so that we can transition between what seem as pages without loading the entire page from the server. We could have built this ourselves, but React-Router is a very easy-to-use library that manages this for us.

React-Bootstrap

Bootstrap, the most popular CSS framework, has been adapted to React and the project is called React-Bootstrap. This library not only gives us most of the Bootstrap functionality, but the components and widgets provided by this library also give us a wealth of information on how to design our own widgets and components.

There are other component/CSS libraries built for React (such as Material-UI, MUI, Elemental UI, etc.) and also individual components (such as react-select, react-treeview, and react-date-picker). All these are good choices too, depending on what you are trying to achieve. But I have found that React-Bootstrap is the most comprehensive single library with the familiarity of Bootstrap (which I presume most of you know already).

Webpack

Webpack is indispensable when it comes to modularizing code. There are other competing tools such as Bower and Browserify which also serve the purpose of modularizing and bundling all the client code, but I found that webpack is easier to use and does not require another tool (like gulp or grunt) to manage the build process.

We will be using webpack not just to modularize and build the client-side code into a bundle to deliver to the browser, but also to “compile” some code. The compilation step is needed to generate pure JavaScript from React code written in JSX.

Other Libraries

Very often, there's a need for a library to address a common problem. In this book, we'll use body-parser (to parse POST data in the form of JSON, or form data), ESLint (for ensuring that the code follows conventions), and express-session, all on the server side, and some more like react-select on the client side.

Why MERN?

So now you have a fair idea of the MERN stack and what it is based on. But is it really far superior to any other stack, say, LAMP, MEAN, J2EE, etc.? By all means, all of these stacks are good enough for most modern web applications. All said and done, familiarity is the crux of productivity in software, so I wouldn't advise a MERN beginner to blindly start their new project on MERN, especially if they have an aggressive deadline. I'd advise them to choose the stack that they are already familiar with.

But MERN does have its special place. It is ideally suited for web applications that have a large amount of interactivity built into the front-end. Go back and reread the section on "Why Facebook built React." It will give you some insights. You could perhaps achieve the same with other stacks, but you'll find that it is most convenient to do so with MERN. So, if you do have a choice of stacks, and the luxury of a little time to get familiar, you may find that MERN is a good choice. I'll talk about a few things that I like about MERN, which may help you decide.

JavaScript Everywhere

The best part about MERN is that there is a single language used everywhere. It uses JavaScript for client-side code as well as server-side code. Even if you have database scripts (in MongoDB), you write them in JavaScript. So, the only language you need to know and be comfortable with is JavaScript.

This is kind of true of all other stacks based on MongoDB and Node.js, especially the MEAN stack. But what makes the MERN stack stand out is that you don't even need a template language to generate pages. In the React way, you programmatically generate HTML (actually DOM elements) using JavaScript. So, not only do you avoid learning a new language, you also get the full power of JavaScript. This is in contrast to a template language, which will have its own limitations. Of course, you will need to know HTML and CSS, but these are not programming languages, and there is no way you can avoid learning HTML and CSS (not just the markup, but the paradigm and the structure).

Apart from the obvious advantage of not having to switch contexts while writing client-side and server-side code, having a single language across tiers also lets you share code between them. I can think of functions that execute business logic, do validation, etc. that can be shared. They need to be run on the client side so that user experience is better by being more responsive to user inputs. They also need to be run on the server side to protect the data model.

JSON Everywhere

When using the MERN stack, object representation is JSON (JavaScript Object Notation) everywhere: in the database, in the application server, and on the client, and even on the wire.

I have found that this often saves me a lot of hassle in terms of transformations. No object relational mapping (ORM), no having to force fit an object model into rows and columns, no special serializing and de-serializing code. An object document mapper (ODM) such as mongoose may help enforce a schema and make things even simpler, but the bottom line is that you save a *lot* of data transformation code.

Further, it just lets me *think* in terms of native objects, and see them as their natural selves even when inspecting the database directly using a shell.

Node.js Performance

Due to its event-driven architecture and non-blocking I/O, the claim is that Node.js is very fast and a resilient web server.

Although it takes a little getting used to, I have no doubt that when your application starts scaling and receiving a lot of traffic, this will play an important role in cutting costs as well as savings in terms of time spent in trouble-shooting server CPU and I/O problems.

The npm Ecosystem

I've already discussed the huge number of npm packages available freely for everyone to use. Any problem that you face will have an npm package already. Even if it doesn't fit your needs exactly, you can fork it and make your own npm package.

npm has been developed on the shoulders of other great package managers and has therefore built into it a lot of best practices. I find that npm is by far the easiest to use and fastest package manager I have used to date. Part of the reason is that most npm packages are so small, due to the compact nature of JavaScript code.

Isomorphic

SPAs used to have the problem that they were not SEO friendly. We had to use workarounds like running PhantomJS on the server to pseudo-generate HTML pages, or use Prerender.io services that did the same for us. This introduced an additional complexity.

With the MERN stack, serving pages out of the server is natural and doesn't require tools that are after-thoughts. This is made possible due to the virtual DOM technique used by React. Once you have a virtual DOM, the layer that translates it to a renderable page can be abstracted. For the browser, it is the real DOM. For the server side, it is HTML. In fact, React Native has taken it to another extreme: it can even be a mobile app!

I don't cover React Native in this book, but this should give you a feel of what virtual DOM can do for you in future.

It's not a Framework!

Not many people like or appreciate this, but I really like the fact that React is a library, not a framework.

A framework is opinionated; it has a set way of doing things. The framework asks you to fill in variations of what it thinks you want to get done. A library, on the other hand, gives you tools to use to construct your application. In the short term, a framework helps a lot by getting most of the standard stuff out of the way. But over time, vagaries of the framework, its assumptions about what you want to get done, and the learning curve will make you wish you had some control over what's happening under the hood, especially when you have some special requirements.

With a library, an experienced architect can design his or her application with the complete freedom to pick and choose from the library's functions, and build their own framework that fits their application's unique needs and vagaries. So, for an experienced architect or very unique application needs, a library is better, even though a framework can get you started quickly.

Summary

This book lets you experience what it takes, and what it is like, to develop an application using the MERN stack. The work that we will do as part of this book encourages thinking and experimenting rather than reading. That's why I have a lot of examples; at the same time, there are exercises that make you think. Finally, it uses the least common denominator to get this done: the CRUD app.

If you are game, read on. Code ahoy!

CHAPTER 2



Hello World

As is customary, we will start with a Hello World example, something that is a bare minimum application that uses MERN. The main purpose of any Hello World exercise is to create the environment that has most of the technology of the stack.

In the Hello World exercise, we'll use React to render a simple page and use Node.js and Express to serve that page from a web server. This will also give you some insight into npm, and JSX transformation, some tools that you'll need to get used to as you go along.

Server-Less Hello World

To quickly get off the ground, let's write a simple piece of code in a single HTML file that uses React to display a simple page on the browser. No installations, no downloads, no server!

Open up your favorite editor and create an HTML file with a head and body, like Listing 2-1.

Listing 2-1. index.html: Server-less Hello World

```
<!DOCTYPE HTML>
<html>

<head>
  <meta charset="UTF-8" />
  <title>Pro MERN Stack</title>
  <script src=
    "https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js">
  </script>
  <script src=
    "https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js">
  </script>
  <script src=
    "https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js">
  </script>
</head>
```

```

<body>
  <div id="contents"></div><!-- this is where our component will appear -->
  <script type="text/babel">
    var contentNode = document.getElementById('contents');
    var component = <h1>Hello World!</h1>; // A simple JSX component
    ReactDOM.render(component, contentNode); // Render the component inside
                                              the content Node
  </script>
</body>

</html>

```

It's very likely that you will copy/paste this code rather than type it yourself, despite my advice that typing it in is better. OK just this once, but only because you're in a hurry to get it to work! But do compare the code line by line, and make sure it has been pasted correctly.

Save it as `index.html` anywhere on your file system. Open it in your browser and check it out. It may take a few seconds to load, but you should see the browser displaying the caption, as seen in Figure 2-1.



Figure 2-1. *Hello World written in React*

Now, let's analyze the code and see what happened here. The first thing to look at is the following line inside the inline script:

```

...
ReactDOM.render(component, contentNode);
...

```

ReactDOM, as you probably guessed, is defined in the included script, `react-dom.js`. The above line asks the ReactDOM library to render the component within the content node. In this case, the content node is the one with the ID `contents`.

But the component is a little more interesting. Look at this line:

```
...
var component = <h1>Hello World!</h1>;
...
```

This is probably familiar to you; it looks like HTML. But it should immediately strike you that the component is *not* a string that's being used as an innerHTML. That's because it's not enclosed in quotes. It's not even valid JavaScript! How did it even parse?

It is, instead, a special HTML-like language called JSX. It, in fact, gets transformed into JavaScript code that generates an element in React's virtual DOM. After transformation, this is what the code that is generated will look like:

```
...
var component = React.createElement('h1', null, 'Hello World!');
...
```

This essentially creates a React `<h1>` element (which is not the same as the HTML `<h1>` element, but very similar). How and when did the code get compiled into the `React.createElement` call? Well, there are two lines that are responsible for this transformation. The first is the inclusion of the following script:

```
...
<script src=
  "https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js">
</script>
...
```

The babel library is a JSX transformer. I will not go into the details of how this is done, because soon we will stop using the browser-based transformer, and instead have a build step during the development, which will create a pre-transformed JavaScript file that we can include in the HTML. The browser-based compiler is not intended for use in production; it is just a try-me-out tool.

Then, there is the type of the script that you specify in the wrapping `<script>` tags around the JSX code:

```
...
<script type="text/babel">
  var contentNode = document.getElementById('contents');
...
```

The browser-based JSX compiler looks for all inline scripts of type `"text/babel"` and compiles the contents into the corresponding JavaScript. The other two scripts, `react.js` and `react-dom.js`, are the core React libraries that handle react component creation and rendering.

At this point in time, you should play around with the code and try a few things to appreciate and understand what is happening under the hood.

EXERCISE: JSX

1. Remove `type="text/babel"` from the script. What happens, and can you explain why? Remove the babel JavaScript library instead. What happens now?
2. Add a class to the `h1` element (you will also need to define the class in a `<style>` section within the `<head>` section.) Hint: Search for “how to specify class in jsx” on stackoverflow.com.
3. We used the minified version of babel, but not for react and react-dom. Can you guess why? Switch to the minified version and introduce a runtime error in React. For example, introduce a typo in the id of the content node so there is no place to mount the component. What happens?

Answers are available at the end of the chapter.

Now, given all the differences and the complexity in comparison to HTML, why do you need to learn JSX? What value does it add? Why not write the JavaScript itself directly? One of the things I talked about in the Introduction chapter is that MERN has just one language throughout, so isn't this contrary to that statement?

As you explore React further, you will soon find that the differences between HTML and JSX are not earth-shaking and they are very logical. You will not need to remember a lot or need to look up things as long as you understand and internalize the logic. Further, though writing JavaScript directly to create the virtual DOM elements is indeed an option, I find that it is quite tedious and doesn't help me visualize the DOM. More so, when it comes to deeply nested elements.

Since most of us already know the basic HTML syntax, writing JSX will probably work best. It is easy to understand what the screen will look like when you read JSX because it is very similar to HTML.

Server Setup

The server-less setup allowed you to get familiarized with React without any installations or firing up a server. But as you may have noticed, it's good neither for development nor for production. During development, some additional time is introduced for loading the scripts from a content delivery network (CDN). As you may have noticed, that's how you're getting all of the React library into your web page. Take a look at the size of each of the scripts using the Network tab of the browser's developer console, and you'll see that the babel compiler (even the minified version) is really big. On production, especially in larger projects, runtime compilation of JSX to JavaScript will affect performance.

So, let's get a little organized and serve all files from an HTTP server. We will, of course, use other components of the MERN stack to achieve this.

nvm

To start, let's install nvm. This is the Node Version Manager that makes installation and switching between multiple versions of Node.js easy. This is not a mandatory step, but I've found that having nvm installed at the very beginning made my life easier when I had to start a new project and I wanted to use the latest and greatest version of Node.js at that point in time.

If you are using Mac OS or any Linux-based distribution, follow the instructions on nvm's GitHub page at <https://github.com/creationix/nvm>. Windows users can follow nvm for Windows (please search for it in your favorite search engine), or install Node.js directly without nvm. Generally, I advise Windows users to install a Linux virtual machine (VM), preferably using Vagrant, and do all the server-side coding within the VM. This usually works best, especially when you want to deploy your code on a Linux server.

One tricky thing about nvm is knowing how it initializes your PATH. This works differently on different operating systems, so make sure you read up on the nuances. Essentially, it adds a few lines to your shell's initialization scripts so that next time you open a shell, your PATH is initialized and executes nvm's initialization scripts. This lets nvm know about the different versions of Node.js that are installed, and the path to the default executable.

For this reason, it's always a good idea to start a new shell right after you install nvm rather than continue in the shell where you installed it. Once you get the right path for your nvm, things follow smoothly.

Node.js

Now that we have installed nvm, let's install Node.js using nvm. Visit the Node.js website to find out which version of Node.js fits the requirement: the Long Term Support (LTS) or the latest stable version. If you like to use the latest version, just type

```
$ nvm install node
```

An alternative is to choose the LTS version, which is assured to have support for a longer term than other versions. This means that although you cannot expect feature upgrades, you can expect security and performance fixes that are backward compatible, and you can continue to install incrementally newer versions without worrying about breaking your existing code. For this alternative, you have to type in the version number. As of the writing of this book, 4.5.0 was the LTS version, so you can use it:

```
$ nvm install 4.5
```

All the code in this book was tested with Node.js version 4.5.0, so I recommend that you do the same. Also, don't forget to make the just installed version of node the default. For example,

```
$ nvm alias default 4.5
```

Otherwise, the next time you enter the shell, node will not be in your PATH. Or you may get the wrong version, if you have installed another version. Confirm the version of the node that's been installed as your default by typing the following in a new shell / terminal:

```
$ node --version
```

Note that installing Node.js via nvm will also install npm, the package manager. If you are using Windows and installing Node.js directly, ensure that you have installed a compatible version of npm as well. Confirm this by noting down the version of npm that was installed along with Node.js:

```
$ npm --version
```

With Node.js 4.5, you are likely to see 2.15.9 as the npm version.

Project

Before we install any third-party packages with npm, it's a good idea to initialize the project. With npm, even an application is considered a package. A package defines various attributes of the application. One such important attribute is a list of other packages that it depends upon. This will change over time, as we find a need to use more libraries.

But to start with, we need at least a placeholder where these things will be saved. For that, we need to initialize the package. Create a directory, say `pro-mern-stack`, navigate to it in your shell, and type the following:

```
$ npm init
```

Most questions that this command asks of you should be easy to answer. The defaults will work fine too. From now on, make sure you are in the project directory for all commands, especially npm commands (which I'll describe below). This will ensure that all changes and installations are localized to the project directory.

npm

To install anything using npm, the command that you use is `npm install <package>`. To start off, and because we need an HTTP server, let's install Express using npm. Installation of Express is as simple as

```
$ npm install express
```

Go ahead and install it. Once done, you will notice that it prints a tree of other packages Express depends upon. Now, uninstall and install it again with the `--save` options:

```
$ npm uninstall express  
$ npm install express --save
```

■ **Note** While installing, you can give a specific version to install using the suffix `@<version>`, for example, `npm install express@4.14.10`. When not specified, it installs the latest version. If something is not working as described in this book, it may be due to a difference in version that you have installed vis-à-vis the version I used when writing the book. In such cases, you can try installing the same version as I have used for this book. For a list of the version of all packages, please refer to the final `package.json` file in the GitHub repository that contains the source code for this book.

EXERCISE: NPM

1. Find out what difference `--save` made. Try uninstalling and re-installing `express` with and without this option. Hint: The difference is in `package.json`.
2. When was `package.json` created? If you can't guess, inspect the contents for a hint. Still can't figure out? Go back and redo your steps. Start with creation of the project directory, and look at the directory contents at each step.
3. Uninstall `Express`. Now, just type `npm install`. What happens? Add another dependency, `mongodb`, manually to `package.json` this time. Use version as “latest”. Now, type `npm install`. What happens? Uninstall `mongodb`, and use `--save` this time to affect `package.json`.
4. Use `--save-dev` instead of `--save` while installing a package. What difference do you see in `package.json`? What difference do you think it will make?
5. Where do you think the packages files are installed? Type `npm ls --depth=0` to check all the currently installed packages. Clean up any packages you do not need.

Play around with `npm` installation and uninstallation a bit. This will be useful in general. Learn more about `npm` version syntax from the documentation: <https://docs.npmjs.com/files/package.json#dependencies>.

Answers are available at the end of the chapter.

npm is extremely powerful, and its options are vast. For the moment, we will concern ourselves only with the installation of packages and a few other useful things. The location of the installed files under the project directory is a conscious choice that the makers of npm made. This has the following effect:

1. All installations are *local* to the project directory. This means that another project can use a different version of any of the installed packages. This may at first seem unnecessary and feel like a lot of duplication. But you will really appreciate this feature of npm when you start multiple Node.js projects and don't want to deal with a package upgrade that you don't need. Further, you will notice that the entire Express package, including all dependencies, is just 1.8Mb. With such tiny sizes, you needn't be worried about excessive disk usage either.
2. A package's dependencies are also isolated within the package. Thus, you could, by all means, have two packages depending on different versions of a common package, and they would each have their own copy and therefore work flawlessly.
3. You don't need administrator (superuser) rights to install a package.

There is, of course, an option to install packages globally, and sometimes it is useful to do this. One use case is when a command-line utility is packaged as an npm package. You typically want the command line to be available regardless of the current directory, and in this case, you would choose a global install. In such cases, the `-g` option of npm install can be used, but you may need admin or root access to do this depending on your OS and type of installation.

Express

Express, if you remember the introduction in the previous chapter, is the best way to run an HTTP server in the Node.js environment. For starters, we'll use Express to serve only static files. This is so that you get used to what Express does, without getting into a lot of server-side coding.

To start using Express, you need to import the module, and use the top level function that the module exports, in order to instantiate an application. You can create multiple applications, which listen on different ports, but we won't do that. We'll instantiate just one application. The `listen()` method of the application takes in a port number and starts the server, which then waits eternally for requests. But before you start listening on a port, you need to set up the application to tell it how to respond to certain requests.

Express is a framework that does minimal work by itself; instead, it gets most of the job done by functions called *middleware*. A middleware is a function that takes in an HTTP request and response object, plus the next middleware function in the chain. The function can look at and modify the request and response objects, respond to requests, or decide to continue with middleware chain by calling the next function.

The `express.static` generator function generates one such middleware function. This middleware responds to a request by trying to match the request URL with a file under a directory specified by the parameter to the generator function. If a file exists, it returns the contents of the file as the response; if not, it chains to the next middleware function. The middleware is *mounted* on the application using the application's `use()` method.

Let's put all this together in a file called `server.js` in the root directory of the project. Listing 2-2 shows the code that is needed to achieve a simple static server.

Listing 2-2. `server.js`: Express Server

```
const express = require('express');

const app = express();
app.use(express.static('static'));

app.listen(3000, function () {
  console.log('App started on port 3000');
});
```

Let's create a directory named `static` under the project directory and move `index.html` (which we created in the previous section) into this directory. Now we can start the web server and serve `index.html`. To start the web-server, do this on the console:

```
$ npm start
```

You should see a message saying the application has started on port 3000. Open up your browser and type `http://localhost:3000` in the URL bar. You should see the same Hello World page.

Let's examine the server code and understand its contents in detail.

```
...
const express = require('express');
...
```

`require` is a JavaScript keyword specific to Node.js, and it is used to import other modules. In the above line, we loaded up the module called `express` and saved the top-level *thing* that the module exports, in the constant named `express`. Node.js allows the thing to be a function, an object, or whatever can fit into a variable. The type and form of what the module exports is really up to the module, and the documentation of the module will tell you how to use it. In the case of Express, the module exports a function that can be used to instantiate an application.

Note that we are using `const` and not `var`. This is because we are using the ECMAScript 2015 (ES2015) specification of JavaScript. You may also see references to ECMAScript 6th Edition (ES6), which is just an older name for ES2015. ECMAScript 6th edition was renamed to ECMAScript 2015 recently.

Node.js supports ES2015 to a large extent, so we will be using the ES2015 features and style of programming. We'll be using ES2015 for the client-side code, so we might as well use the same on the server, so as to have a consistent style throughout.

```
...  
const app = express();  
app.use(express.static('static'));  
...
```

This instantiates the application and then mounts a middleware. The middleware generator takes the parameter `static` to indicate that this is the directory where all the static files reside. Remember that we moved `index.html` into this directory.

The `express.static` generated middleware function is also smart enough to translate a request to `/` (the root of the website) and respond by looking for `index.html` in the directory. This is similar to what other static web servers such as Apache would have done. You could also have used `http://localhost:3000/index.html` to access the application and seen the familiar Hello World.

EXERCISE: EXPRESS

1. Change the name of the file `index.html` to something else, say, `hello.html`. How does this affect the application?
2. If you wanted all static files to be accessed by a prefixed URL, for example `/public`, what change would you make? Hint: Take a look at the Express documentation for static files and try this out.
3. Change the name of the web application file from `server.js` to something else, say, `app.js`. Now, what options do you have for starting up the web server? Hint: Look up the npm documentation, in specific, CLI commands and the command `run-script`.

Answers are available at the end of the chapter.

Build-Time JSX Compilation

As you saw in earlier sections, the transformation of JSX to JavaScript happens at runtime. This is inefficient and quite unnecessary. Let's instead move the transformation to the build stage in your development, so that we can deploy a ready-to-use distribution of the application.

Separate Script File

First, we need to separate out the JSX script from your all-in-one `index.html`, and refer to it as an external script. This way, we can keep the HTML as pure HTML and all the scripts that need compilation in a separate file. Let's call this external script `App.jsx`, and place it in the static directory, so that it can be referred to as `/App.jsx` from the browser. The new modified files are shown in Listings 2-3 and 2-4.

Listing 2-3. `index.html`: Separate HTML and JSX

```
<!DOCTYPE HTML>
<html>

<head>
  <meta charset="UTF-8" />
  <title>Pro MERN Stack</title>
  <script src=
    "https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js">
  </script>
  <script src=
    "https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js">
  </script>
  <script src=
    "https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js">
  </script>
</head>

<body>
  <div id="contents"></div><!-- this is where our component will appear -->
  <script type="text/babel" src="/App.jsx"></script>
</body>

</html>
```

Listing 2-4. `App.jsx`: JSX Part Separated Out from the HTML

```
var contentNode = document.getElementById('contents');
var component = <h1>Hello World!</h1>; // A simple JSX component
ReactDOM.render(component, contentNode); // Render the component inside ↵
                                         the content Node
```

At this point, the app should continue to work. Test it to make sure it does (point your browser to `http://localhost:3000`). Until now, we have only separated the files; we have not moved the transform to build-time. The JSX continues to get transformed by the babel library script. We'll move the transform to build-time in the next step.

Transform

Create a directory to keep all the JSX files, which will be transformed into JavaScript and placed in the static folder. Let's call this directory `src`. Let's move `App.jsx` into this directory.

To transform the JSX, we need to install some babel tools. In this case, we need a couple of tools: `babel-cli` (the command line tool that invokes the transformation) and `babel-preset-react` (the plugin that handles React JSX transformation). Note that the babel tool handles not just JSX but many other transformations, which you can read about on the website. For now, we'll only be installing the React transform, so execute the following command:

```
$ npm install --save-dev babel-cli babel-preset-react
```

Now we're ready to transform `App.jsx` into pure JavaScript. Do this:

```
$ node_modules/.bin/babel src --presets react --out-dir static
```

Since we did not install `babel-cli` globally, you must type the path to the command, relative to the current directory, which is `node_modules/.bin`. We could have installed `babel-cli` globally using the `--global` (or `-g`) option of `npm`. That way, we would have access to the command in any directory, without having to prefix the path. But as discussed earlier, it's a good practice to keep all installations local to a project. This is so that we don't have to deal with version differences of a package across projects. An alternative is to add the `.bin` directory to the `PATH` environment variable, but you'll see later that there is a better way to do all this, so let's keep it like this for now.

We also need to change `index.html` to replace the reference to `App.jsx` to `App.js` and indicate the new type of this script; it is now JavaScript and not JSX. So, let's just remove the `type="text/babel"` in the script specification. We no longer need the runtime transformer to be loaded in `index.html`, so we can get rid of the `babel-core` script library. These changes are shown in Listing 2-5.

Listing 2-5. `index.html`: Changes in Script Name and Type

```
...
<script
  src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser-
  min.js"></script>
...
<body>
  <div id="contents"></div><!-- this is where our component will appear -->
  <script src="/App.js" type="text/babel"></script>
</body>
...
```

It's time to test this set of changes and ensure that things work as before. For good measure, use the browser's developer console to ensure it is `App.js` that's being fetched, and not `App.jsx`. The developer console can be found on most browsers; you may need to look at your browser's documentation for instructions to get to it.

EXERCISE: BABEL

1. Inspect the contents of `App.js`, the output of the transform. You'll find it in the static directory. What do you see? Can you explain this?
2. Why did you use `--save-dev` instead of `--save` while installing `babel-cli`? Hint: Read the npm documentation, the CLI command for install.
3. Can you explain each of the command-line arguments of `babel`? Experiment with the arguments to find out.
4. Look at the babel website (<http://babeljs.io>). What other transforms do you think will be of use?

Answers are available at the end of the chapter.

Automate

npm provides an automatic way of running command-line binaries that belong to locally installed packages. This lets you pay less attention to the location of command-line binaries. (Didn't I say npm was powerful? This is one of the things that it does, even though this is not real package manager functionality.)

You can define your own npm commands by specifying them in the scripts section of `package.json`. These commands can then be called using `npm run <script>` from the console. Let's add a script called "compile" whose command line is the babel command line, but without the prefix to the binary location. This is because npm automatically figures out the location. The changes in `package.json` are shown in Listing 2-6. You can now run the transform like this:

```
$ npm run compile
```

Make a few temporary changes to `App.jsx`, recompile it, and ensure that the changes are visible in the browser. To recompile, just run `npm run compile` again.

Note Avoid using npm sub-command names which are also npm first-level commands such as `build` and `rebuild`, because this leads to silent errors if and when you miss out the 'run' in the npm command.

When we work on the client-side code and change the source files frequently, we have to manually recompile it. Wouldn't it be nice if someone detects these changes for us and recompiles the source into JavaScript? Well, babel supports this out of the box via the option `--watch`. To make use of it, let's add another script called `watch` with this additional option to the babel command line. The final set of scripts added is shown in Listing 2-6.

Listing 2-6. Package.json: Adding Scripts for Transformation

```
...
"scripts": {
  "compile": "babel src -presets react -out-dir static",
  "watch": "babel src -presets react -out-dir static -watch",
  "test": "echo \"Error: no test specified\" && exit 1"
},
...
```

Go ahead and run the new command using `npm watch`. You will notice that it does one transform, but doesn't return to the shell. It's actually waiting in a permanent loop, watching for changes to the source files. Test out its effect by making a small change to `App.jsx` and saving the file. You'll see that it prints out a fresh line like this for every change:

```
src/App.jsx -> static/App.js
```

Refresh your browser after the new line is printed when a change in `App.jsx` is made and ensure that your changes are reflected in the application.

React Library

Although it is possible to serve the React library scripts also from the server and gain a little more speed during development, we won't be doing this just yet. The reason is that in a later step, we'll be bundling not just the React library but also other libraries and the application's JavaScript files together using a tool called Webpack. The added benefit of moving the React scripts to the server right now is very minimal.

ES2015

Until now, we were using ES5 (old JavaScript) to write your client-side code. Let's switch to ES2015 instead. This not only gives us a few useful features, but it also makes the code more readable by allowing short-hand syntaxes such as arrow functions. We'll also be able to use extra conveniences that React gives you by using ES2015 classes.

For the server-side code, we didn't have to do anything special; we just wrote the code in ES2015. But we can't do this for the client-side code because not all browsers uniformly support ES2015 specifications. Luckily, babel provides the ability to deal with ES2015 in addition to JSX into ES5 JavaScript. Conversion from one specification of JavaScript to another is called *transpiling* (though you may also use *compiling* in the rest of the book to mean the same thing). Let's install another babel plugin for this:

```
$ npm install --save-dev babel-preset-es2015
```


Let's now modify the compile npm scripts in `package.json` to include an ES2015 compilation in addition to React transformation. The changed script commands are shown in Listing 2-7, with the changes highlighted in bold.

Listing 2-7. `package.json`: Including ES2015 Transformation

```
...
"scripts": {
  "compile": "babel src --presets react,es2015 --out-dir static",
  "watch": "babel src --presets react,es2015 --out-dir static --watch",
  "test": "echo \"Error: no test specified\" && exit 1"
},
...
```

All the new syntaxes and language features can now be used in your client-side code. But there's one more thing: ES2015 also comes with many built-in objects and extensions, which are not syntactic changes. For example, the Promise object is an ES2015 specification, as is the array `find()` method. These are not supported by older browsers such as Internet Explorer 11 and earlier. And, a transpile is not enough to add these new capabilities.

If you must support those browsers and want to use these built-in functions, you need to add support to the browser via a JavaScript library. These are called *polyfills*, things that supplement browser capabilities or global functions. If the browser natively supports a certain feature, a polyfill will choose to use the native objects or methods rather than use the JavaScript-based supplement. There are individual polyfills for various features; for example, you will find a polyfill for Promises. But rather than install individual polyfills, let's use the babel recommended polyfill that emulates a full ES2015 environment. This is called babel-polyfill. Let's use this, like the React libraries, from a CDN. So, change the `index.html` to include the babel-polyfill library as shown in Listing 2-8.

Listing 2-8. `index.html`: Changes for Including ES2015 Polyfill

```
...
<title>Pro MERN Stack</title>
<script src=
  "https://cdnjs.cloudflare.com/ajax/libs/babel-polyfill/6.13.0/polyfill.js">
</script>
<script src=
  "https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js">
</script>
...
```

Now we're ready to use ES2015 in `App.jsx`. Let's replace the contents with the code in Listing 2-9, which you'll examine soon.

Listing 2-9. App.jsx: Rewritten in ES2015

```
const contentNode = document.getElementById('contents');

const continents = ['Africa', 'America', 'Asia', 'Australia', 'Europe'];
const message = continents.map(c => `Hello ${c}!`).join(' ');

const component = <p>{message}</p>;    // A simple JSX component
ReactDOM.render(component, contentNode); // Render the component inside the
                                         content Node
```

Compile the new App.jsx by running `npm run watch` or `npm run compile`. If you already running `npm run watch`, you will need to break that process using Ctrl-C and restart it. That's because it needs to load the changes to the watch command which now includes the ES2015 preset.

Refresh your browser (you should be running `npm start` in a different console by now), and you should now see a greeting for each continent rather than the entire world. Let's discuss the new code, which includes a few ES2015 features and also one React feature.

```
...
const contentNode = ...
...
```

In ES2015, the `const` and `let` keywords replace the `var` keyword. You use `const` to declare constants (variables that don't change once assigned). You use `let` to declare and initialize a variable that may take other values during the course of the script execution. In the above piece of code, we used `const` to say that the variable `contentNode` is not going to be assigned later on to anything else.

```
...
continents.map(c => ...)
...
```

This is an *arrow function*. It's a shorthand for defining a function taking a single parameter and returning an expression. Note that there is no `return` keyword, it reads as "c maps to ``Hello ${c}!``".

```
...
`Hello ${c}!`
...
```

Pay close attention to the quotes: these are back-ticks and not single-quotes. This construct is called a *template string*. You can plug in variables using `${}` within the string, and the variables will be expanded inline, just like string interpolation in perl and python. In fact, you can use any JavaScript expression rather than just a variable.

```
...
const component = <p>{message}</p>;
...
```

Similar to ES2015 string interpolation, this is a JSX feature. The curly braces allow you to insert JavaScript expressions inside JSX, and these will be replaced by the value of the expression. This works not only for HTML text nodes but also within attributes. For example, the class name for an element can be a JavaScript variable.

EXERCISE: ES2015

1. Try to format the message as one per line by using `<br />` to join the individual messages instead of a space. Are you able to do it? Why not?
2. Inspect the generated `App.js` now. What is new because of ES2015 transformation?

Answers are available at the end of the chapter.

Please read all about ES2015 on the Babel website: <http://babeljs.io/docs/learn-es2015/>. We will be using many ES2015 features in this book, but I will not explain each of them henceforth. You don't need to be an expert, but you should know the basics and be able to find help when needed. I will assume that you are familiar with ES2015 from now on.

Summary

In this chapter, we saw the basics of how React applications can be built. We started with a simple piece of code written React JSX that we compiled at runtime; then we moved the compilation to the server. We also looked at ES2015, and how to *transpile*, that is, compile from one specification of the language to another.

We also got a whiff of what Node.js with Express can do, and how to use Babel to compile JSX into plain JavaScript. We didn't MongoDB, the M in the MERN stack, but I hope you got a good view of the others. In the next two chapters, we'll dive deeper into React, and then surface up to look at the big picture, dealing with REST APIs, MongoDB, and Express.

Answers to Exercises

Exercise: JSX

1. The attribute `"type=text/babel"` on the script element indicates that the contents are JSX, as opposed to regular JavaScript, the default if you don't specify a type. The babel compiler compiles all such script elements into JavaScript and injects it back. Removing either the babel compiler or the type attribute will cause the script not to be compiled into JavaScript and will cause errors on the JavaScript console.
2. To specify a class in JSX, you need to use `className=<name>` instead of `class=<name>` as you would have done in HTML.
3. A minified version of React hides or shortens runtime errors. A non-minified version gives full errors and helpful warnings as well.

Exercise: npm

1. The npm option `--save` causes the installation/uninstallation to be recorded under the dependencies section of `package.json`.
2. `package.json` got created when we created the project using `npm init`. In fact, all of the responses to the prompts when we ran `npm init` were recorded in `package.json`.
3. Running `npm install` without any further options or parameters causes all dependencies to be installed, by looking at `package.json`. Thus, you could add dependencies manually to `package.json` and just use `npm install`. This gives you better control over which version of the package you want.
4. The option `--save-dev` adds the package in `devDependencies` instead of `dependencies`. The list of dev dependencies will not be installed in production, which is indicated by the environment variable `NODE_ENV` being set to the string `production`.
5. Package files are installed under the directory `node_modules` under the project. `npm ls` lists all the packages installed, in a tree-like manner. `--depth=0` restricts the tree depth to the top-level packages. Deleting the entire `node_modules` directory is one way of ensuring you start clean.

Exercise: Express

1. The static file middleware does not specially treat `hello.html` as it did `index.html`, so you will have to access the application with the name of the file like this: `http://localhost:3000/hello.html`.
2. For accessing static files via a different mount point, specify that prefix in the middleware generated helper function as the first parameter. The directory name will now be the second parameter. For example, `express.static('/public', 'static')`.
3. By default, `npm start` runs the command `node server.js`. If you use some other file name, say `app.js`, you should add `"start": "node app.js"` as a script under the `scripts` section of `package.json` to tell `npm` that this is the actual start command. Alternatively, you could just run the server yourself without using `npm`, using the command `node app.js`.

Exercise: Babel

1. `App.js` now contains pure JavaScript, as discussed in the beginning of the chapter. JSX, the short-hand notation for `React.createElement()` function calls, has been transformed into just that.
2. When you deploy the code, you only deploy a pre-built version of the application. That is, you transform the JSX on a build server or your development environment, and push out the resulting JavaScript to your production server. Thus, on the production server, you will not need the tools that are required to build the application. Therefore, you use `--save-dev` so that on the production server, the package need not be installed.
3. The first argument to `babel` is the source: this can be a directory or an individual file. `--presets react` tells it to run the React (JSX) transformation. `--out-dir` tells it to place output files (derived from the input file name) at the specified location. You could have used `--out-file` to specify any custom file name.
4. We'll definitely use the `es2015` transform because we'll soon switch over to ES2015 for the client-side code as well. There are other useful transforms such as `object-assign` which also can be considered.

Exercise: ES2015

1. React does this on purpose, to avoid cross-site scripting vulnerabilities. It is not easy to insert HTML markup, although there is a way using the `dangerouslySetInnerHTML` attribute of an element. The correct way to do this is to compose an array of components. We will explore this in later chapters.
2. Without ES2015 transformation, only JSX snippets were transformed. Now, with ES2015 transformation, even JavaScript is modified.

CHAPTER 3



React Components

In the Hello World example, we created a very basic React native component, using pure JSX. In the real world, you will want to do much more than what a simple single line JSX can do. This is where React components come in. React components react to user input, change state, interact with other components, and much more.

However, before going into all that detail, let me first describe the requirements of the application that we will build. After that, at every step that we take forward, we'll take one feature or task that needs to be addressed and complete it. I like this approach because I learn things the best when I put them to immediate use. This approach not only lets you appreciate and internalize the concepts because you put them to use, but also brings the more useful and practical concepts to the forefront.

The application I've come up with is something that most developers can relate to.

Issue Tracker

I'm sure that most of you are familiar with GitHub or JIRA issues. It's essentially a CRUD application (Create, Read, Update, and Delete a record in a database). The CRUD pattern is so useful because pretty much all enterprise applications are built around the CRUD pattern on different entities or objects.

In our case, we'll only deal with a single object/record, because that's good enough to depict the pattern. Once you grasp the fundamentals of how to implement the CRUD pattern in MERN, you'll be able to replicate the pattern and create a real-life application.

Here's the requirement list:

- We should be able to view a list of issues, with the ability to filter the list by various parameters.
- We should be able to add new issues by supplying the initial values of the issue's fields.
- We should be able to update an issue by changing its field values.
- We should be able to delete issues.

An issue should be described by the following attributes:

- A title that summarizes the issue (free-form text)
- An owner to whom the issue is assigned to (free-form short text)

- A status indicator (a list of possible status values)
- Creation date (a date, automatically assigned)
- Effort required to address the issue (number of days, a number)
- Estimated completion date (a date, entered by the user)

Note that I've included different types of fields (lists, date, number, text) to make sure you learn how to deal with different data types. We'll start simple, build one feature at a time, and learn about the MERN stack as we go along.

In this chapter, we'll create React classes and instantiate components. We'll also create bigger components by putting together smaller components. Finally, we'll pass data among these components and create components dynamically from data. In terms of features, the objective in this chapter is to lay out the main UI page of the Issue Tracker: a list of issues. We'll hard-code the data that is used to display the page, and leave persistence and retrieval of the data to a later chapter.

React Classes

In this section, the objective is to convert the single line JSX into a simple React component instantiated from a React class, so that we can later use the full power of first class React components.

React classes let us create real components (as opposed to the templated HTML that we saw in the previous chapter), reuse them within other components, handle events, and so much more. To start with, we'll replace the Hello World example with a simple class, as the starting point for the Issue Tracker application.

React classes are created by extending `React.Component`. There is also a non-ES2015 class way of doing this by calling `React.createClass`, which is the only way if you are not using ES2015. Though there are a few differences in the style and the methods, you can achieve the same in both options. Some people prefer using `React.createClass`, but the React team recommends ES2015 classes. In this book, we'll follow that recommendation.

Within the class definition, we need, at the minimum, a `render()` function. To start, let's just return a `<div>` with some placeholder text from the `render()` method. The new contents of `App.jsx` are shown in Listing 3-1.

Listing 3-1. `App.jsx`: A Simple React Class

```
const contentNode = document.getElementById('contents');

class IssueList extends React.Component {
  render() {
    return (
      <div>This is a placeholder for the issue list.</div>
    );
  }
}

ReactDOM.render(<IssueList />, contentNode); // Render the component inside ↵
                                              the content Node
```


If you refresh your browser, you should see the placeholder text. Now, let's examine what happened here in a little more detail.

The `render()` method is something that the React framework calls when it needs to display the component. There are other methods with special meaning to React that can be implemented, called the Lifecycle functions, which provide hooks into various stages of the component formation and events. I'll discuss them in later chapters. But `render()` is one that must be present; otherwise you'll have a component that has no screen presence.

Within the `render()` method, we're supposed to return a native component instance, or an instance of a component defined by us. In this case, we returned a `<div>` element, just like the first Hello World example used an `<h1>` element. We don't really need the brackets around the `<div>` but it's convention, and it helps readability when rendering a more complex or nested set of elements.

Let's take a closer look at the last line:

```
...
ReactDOM.render(<IssueList />, contentNode);
...
```

Here, we rendered an instantiation of the `IssueList` component into the `contents` element defined in `index.html`. To instantiate a component, you write it in JSX as if it were an element, enclosed within angle brackets. It's worth noting here that `div` and `h1` are built-in internal React components that you can directly instantiate, whereas `IssueList` is something that *you* define and later instantiate. And within `IssueList`, you use React's built-in `div` component.

I've used component and instance of a component interchangeably, like sometimes we tend to do with objects. But it should be obvious by now that `IssueList` and `div` are actually React component classes, whereas `<IssueList />` and `<div />` are tangible components or instances of the component class.

EXERCISE: REACT CLASSES

1. In the `render` function, instead of returning one `<div>`, return two `<div>` elements placed one after the other. What happens? Why, and what's the solution? Hint: Look in the React documentation under tips: maximum number of JSX Root nodes.
2. Create a runtime error by changing the string `'contents'` to `'main'` or some other string that doesn't identify an element in the HTML. Where is the error caught? What about JavaScript runtime errors like undefined variable references?

Answers are available at the end of the chapter.

Composing Components

In the previous section, you saw how to build a component by putting together built-in React components that are HTML element equivalents. It's possible to build a component that uses other user-defined components as well.

This is called component composition, and it is one of the most powerful features of React. It lets you split the UI into smaller independent pieces so that you can reason about each piece in isolation. Using components rather than building the UI in a monolithic fashion also encourages reuse. We'll see in a later chapter how one of the components that you built can easily be reused, even though we hadn't thought of reuse at the time of building the component.

A component takes inputs (called properties, which I'll discuss later) and its output is the rendered UI of the component. In this section, we will not use inputs, but put together fine-grained components to build a larger UI.

Let's design the main page of the application as three parts: a filter to select which issues to display, the list of issues, and finally an entry form for adding an issue. We're focusing on composing the components at this point in time, so we'll only use placeholders for these three parts. So, just like the `IssueList` class, let's define three placeholder classes: `IssueFilter`, `IssueTable`, and `IssueAdd`—very similar to the `IssueList` class, the only difference being the placeholder text. To put them together, and change the `IssueList` class, let's remove the placeholder text and replace it with an instance of each of the new placeholder classes separated by a `<hr>` or horizontal line.

Ideally, components should be isolated into their own files so that they can be reused. But at the moment, we only have placeholders, so for the sake of brevity, we'll keep all the classes in the same file. At a later stage, when the classes are expanded to their actual content, we'll separate them out.

Listing 3-2 shows the new contents of the file `App.jsx`.

Listing 3-2. `App.jsx`: Composing Components

```
const contentNode = document.getElementById('contents');

class IssueFilter extends React.Component {
  render() {
    return (
      <div>This is a placeholder for the Issue Filter.</div>
    )
  }
}

class IssueTable extends React.Component {
  render() {
    return (
      <div>This is a placeholder for a table of Issues.</div>
    )
  }
}
```

```

class IssueAdd extends React.Component {
  render() {
    return (
      <div>This is a placeholder for an Issue Add entry form.</div>
    )
  }
}

class IssueList extends React.Component {
  render() {
    return (
      <div>
        <h1>Issue Tracker</h1>
        <IssueFilter />
        <hr />
        <IssueTable />
        <hr />
        <IssueAdd />
      </div>
    );
  }
}

ReactDOM.render(<IssueList />, contentNode); // Render the component inside
                                              the content Node

```

The effect of this code is an uninteresting page, as in Figure 3-1.

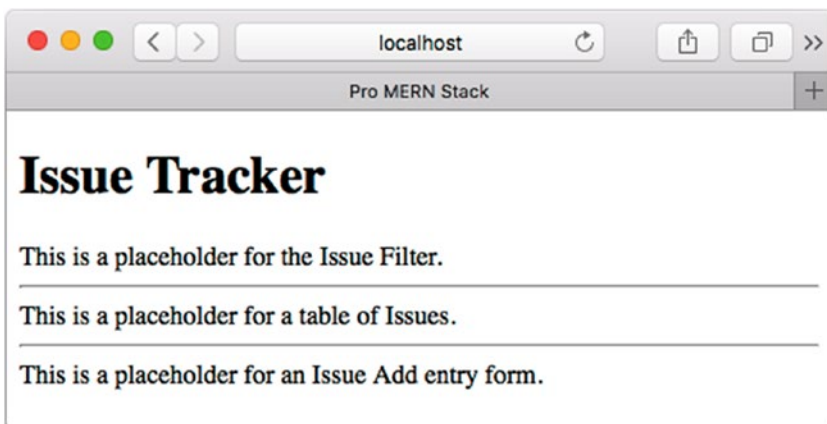


Figure 3-1. Issue tracker by composing components

Passing Data

Composing components without any variables is not so interesting. You should be able to pass data from a parent component to a child component and make it render differently on different instances. Displaying multiple rows of issues using a class for a single row is an ideal use case to demonstrate this.

Let's create a component called `IssueRow` to depict one row in a table, and then use this multiple times within `IssueTable`, passing in different data to show different issues.

Using Properties

You can pass data from a parent to a child component in different ways. One way to do this is using properties. Any data passed in from the parent can be accessed in the child component through a special variable, `this.props`. So, to access a property called `issue_title`, you use `this.props.issue_title`. For example, in the `IssueRow` child component, you access the property in a JSX snippet like this:

```
...
    <td>{this.props.issue_title}</td>
...

```

And, to pass this data, you use XML- or HTML-like attributes in the parent. For example, to pass the property `issue_title`, in the parent, you do this:

```
...
    <IssueRow title="Title of the first issue" />
...

```

You can pass not only strings but also JavaScript objects and other data types. In fact, any JavaScript expression can be passed along by using curly braces (`{}`) instead of quotes, as in the above example. Listing 3-3 shows the new `IssueRow` component and the modified `IssueTable` component.

Listing 3-3. `App.jsx`: Passing Data from `IssueTable` to `IssueRow` Component

```
class IssueRow extends React.Component {
  render() {
    const borderedStyle = {border: "1px solid silver", padding: 4};
    return (
      <tr>
        <td style={borderedStyle}>{this.props.issue_id}</td>
        <td style={borderedStyle}>{this.props.issue_title}</td>
      </tr>
    )
  }
}
```

```

class IssueTable extends React.Component {
  render() {
    const borderedStyle = {border: "1px solid silver", padding: 6};
    return (
      <table style={{borderCollapse: "collapse"}}>
        <thead>
          <tr>
            <th style={borderedStyle}>Id</th>
            <th style={borderedStyle}>Title</th>
          </tr>
        </thead>
        <tbody>
          <IssueRow issue_id={1}
            issue_title="Error in console when clicking Add" />
          <IssueRow issue_id={2}
            issue_title="Missing bottom border on panel" />
        </tbody>
      </table>
    )
  }
}

```

Let's examine what happened here. Let's first look at the table body, which contains multiple lines of the following form:

```

...
    <IssueRow issue_id={1}
      issue_title="Error in console when clicking Add" />
...

```

The issue title is passed in as a string using a quoted attribute, whereas the numbers 1 and 2 are enclosed in curly braces. We accessed the properties using `this.props` in the child component as seen in the `IssueRow` component:

```

...
    <td style={borderedStyle}>{this.props.issue_id}</td>
...

```

We passed in different values for the `id` and `title` properties to display two hard-coded rows of Issues. We added only two fields of an Issue for the sake of brevity; we'll add the other fields later. Further, we passed the attribute `style` to the native HTML element `<td>`:

```

...
    const borderedStyle = {border: "1px solid silver", padding: 4};
...
    <td style={borderedStyle}>{this.props.issue_id}</td>
...

```

The first line above creates an object describing a style. This style is then applied to all the table cells that are created.

But note that these are not really HTML attributes. Instead, they are also like properties being interpreted by the built-in native components. In most cases, like `style`, the name of the attribute is the same as the HTML attribute, but a few attributes cause conflict with JavaScript reserved words, so the naming requirements are different. Thus, the `class` HTML attribute is `className` in JSX. Also, hyphens in the HTML are replaced with camel cased names; for example, `max-length` becomes `maxLength`.

In most cases, the value of the attribute is a string, which looks the same as the HTML attribute value. But for the attribute `style`, you pass in an object describing the style rather than a string as in HTML. The style object contains a series of JavaScript key-value pairs. The keys are same as the CSS style name, except that instead of dashes (like `border-collapse`), they are camel cased (like `borderCollapse`). The values are CSS style values, just as in CSS. There is also a special shorthand for specifying pixel values: you can just use a number (like 4, above) instead of a string like `"4px"`.

Figure 3-2 shows the output of the above changes.

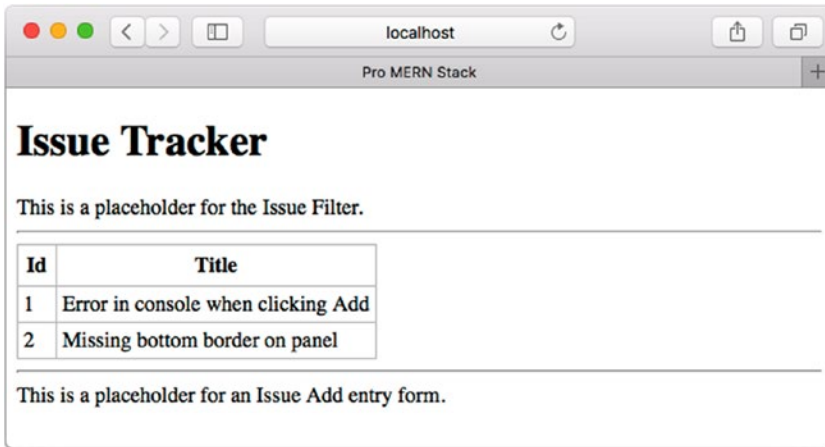


Figure 3-2. *Passing data to child components*

Property Validation

When you pass parameters in functions in strongly typed languages such as Java, you declare the types of the function parameters. This ensures that the caller knows the list and the types of parameters, and also ensures that passed-in parameters are validated against this specification.

Similarly, the properties being passed from one component to another can also be validated against a specification. This specification is supplied in the form of a static object called `propTypes` in the class, with the name of the property (e.g., `issue_title`) as the key and the validator as the value, which is one of the many constants exported by `React.PropTypes`, for example, `React.PropTypes.string`. To indicate that `issue_id`

and `issue_title` are the properties expected, the first being a mandatory value, you add the `propTypes` static variable to `IssueRow` like this:

```
IssueRow.propTypes = {
  issue_id: React.PropTypes.number.isRequired,
  issue_title: React.PropTypes.string
};
```

In ES2015, static members can only be functions; hence the class member must be declared outside the class declaration. If you prefer that the declaration be inside the class declaration, you can use a getter function instead, like this:

```
...
static get propTypes() {
  return {
    issue_id: React.PropTypes.number.isRequired,
    issue_title: React.PropTypes.string
  };
}
...
```

Property validation is checked only in development mode, and a warning is shown in the console when any validation fails. Since we are in an early stage in the development of the application, we expect more changes to the properties. We'll add property validations later, when the properties are reasonably stabilized.

Further, you can also default the property values when the parent does not supply the value. For example, if you want the title to be defaulted to something else, rather than show an empty string, you can do this:

```
IssueRow.defaultProps = {
  issue_title: '-- no title --',
};
```

Using Children

There is another way to pass data to other components, using the contents of the HTML-like node of the component. In the child component, this can be accessed using a special property of `this.props` called `this.props.children`.

As you have probably noticed, just like in regular HTML, you can nest components. We added the three sections within the `<div>` built-in component by nesting it. When the components are converted to HTML elements, the elements nest in the same order. Similarly, even for your own components, you can nest other components at the time the component is instantiated. In such cases, you'll find that the JSX expression includes both the opening and closing tags, and other components or JSX expressions within them. When the enclosing component programmatic needs access to the nested component, it can do that using `this.props.children`.

Say you wanted to wrap an arbitrary component with a bordered `<div>`. You would create such a wrapper component like this:

```
...
class BorderWrap extends React.Component {
  render() {
    const borderedStyle = {border: "1px solid silver", padding: 6};
    return (
      <div style={borderedStyle}>
        {this.props.children}
      </div>
    );
  }
}
...
```

Then, during the rendering, you could wrap *any* component with a border like this:

```
...
<BorderWrap>
  <ExampleComponent />
</BorderWrap>
...
```

Thus, instead of passing the issue title as a property to `IssueRow`, we now use this technique and embed it as contents of `<IssueRow>`. This change is shown in in Listing 3-4.

Listing 3-4. App.jsx: Using Children Instead of Props

```
...
    <td style={borderedStyle}>{this.props.issue_titlechildren}</td>
...
    <IssueRow issue_id={1} issue_title="Error in console when clicking Add" />
    <IssueRow issue_id={2} issue_title="Missing bottom border on panel" />
    <IssueRow issue_id={1}>Error in console when clicking Add</IssueRow>
    <IssueRow issue_id={2}>Missing bottom <b>border</b> on panel</IssueRow>
...
```

Now, as you can see, you are able to pass a formatted HTML content as the title directly to the `IssueRow` rather than a plain string.

EXERCISE: PASSING DATA

1. Add an attribute `border=1` for the table, as you would in regular HTML. What happens? Why? Hint: Read up on supported tags and attributes in the reference section of the React documentation.
2. Why is there a double curly brace in the inline style for the table? Hint: Compare it with the other style, where you declared a variable and used that instead of specifying it inline.
3. The curly braces are a way to escape into JavaScript in the middle of JSX markup. Compare this to similar techniques in other templating languages such as PHP.
4. When is it appropriate to pass data as props vis-à-vis children? Hint: Think about what is it that you want to pass.

Answers are available at the end of the chapter.

Dynamic Composition

In this section, we'll replace the hard-coded set of `IssueRow` components with a programmatically generated set. For the moment, we'll use a simple JavaScript array in memory to store the list of issues to be displayed. In later chapters, we'll get more sophisticated by getting the data from the server, and then from a database. Using this in-memory array, we'll generate an array of `IssueRow` components. We'll also include as many fields of an issue as possible in this array. Listing 3-5 shows this in-memory array, declared globally just before the `IssueList` class declaration.

Listing 3-5. App.jsx: In-Memory Array of Issues

```
const issues = [
  {
    id: 1, status: 'Open', owner: 'Ravan',
    created: new Date('2016-08-15'), effort: 5, completionDate: undefined,
    title: 'Error in console when clicking Add',
  },
  {
    id: 2, status: 'Assigned', owner: 'Eddie',
    created: new Date('2016-08-16'), effort: 14,
    completionDate: new Date('2016-08-30'),
    title: 'Missing bottom border on panel',
  },
];
```

We added an array called `issues` that holds two issues objects. We left `completionDate` undefined in the first object, to indicate that this is an optional field. We can add more example issues, but two is sufficient to demonstrate dynamic composition. Now, let's modify the `IssueList` class to pass this array as a property to `IssueTable`. The changes are shown in Listing 3-6.

Listing 3-6. App.jsx: Pass Issues from `IssueList` to `IssueTable`

```
...
    <hr />
    <IssueTable issues={issues} />
    <hr />
...
```

We could have kept the array scoped within the `render()` function of `IssueList`, but keeping it global gives us some convenience later on. Also, since this is a simulation of a list of issues that needs to be fetched from the server, it is a more accurate replacement if we keep it in the global scope.

Within the `IssueTable` class' `render()` method, we need to iterate over the issues array and generate an array of `IssueRows` from it. The `map()` method of `Array` comes in handy to do this, as we can map an issue object to an `IssueRow` instance. This time, instead of passing each field as a property, we pass the issue object itself apart from the key property, which is required for arrays. This is how we create the array of `IssueRow` components:

```
...
issues.map(issue => <IssueRow key={issue.id} issue={issue} />)
...
```

If we use a `for` loop instead, we can't do it within the JSX, because JSX is not really a templating language. We must create a variable in the `render()` method and use that in the JSX. Let's do this anyway for readability. Replace the two hard-coded issue components with this variable instead.

In other frameworks and templating languages, creating multiple elements using a template requires a special `for-loop` construct (e.g., `ng-repeat` in `AngularJS`) within that templating language. But in `React`, you just use regular `JavaScript` for all programmatic constructs, not only giving you the full power of `JavaScript` to manipulate templates, but also a lesser number of constructs to learn and remember.

In the `IssueTable` class, we also need to expand the header row for the table to include all fields. Further, let's replace the inline styling with a `CSS` class. The new `IssueTable` class is shown in Listing 3-7.

Listing 3-7. App.jsx: `IssueTable` Class with an Array of `IssueRows`

```
class IssueTable extends React.Component {
  render() {
    const issueRows = this.props.issues.map(issue => <IssueRow
    key={issue.id} issue={issue} />)
```

```

return (
  <table className="bordered-table">
    <thead>
      <tr>
        <th>Id</th>
        <th>Status</th>
        <th>Owner</th>
        <th>Created</th>
        <th>Effort</th>
        <th>Completion Date</th>
        <th>Title</th>
      </tr>
    </thead>
    <tbody>{issueRows}</tbody>
  </table>
)
}
}

```

The changes in `IssueRow` are quite simple. We can remove the inline styles, and we need to add a few more columns for each of the added fields. The new `IssueRow` class is shown in Listing 3-8.

Listing 3-8. `App.jsx`: New `IssueRow` Class Using Issue Object Property

```

class IssueRow extends React.Component {
  render() {
    const issue = this.props.issue;
    return (
      <tr>
        <td>{issue.id}</td>
        <td>{issue.status}</td>
        <td>{issue.owner}</td>
        <td>{issue.created.toString()}</td>
        <td>{issue.effort}</td>
        <td>{issue.completionDate ? ←
          issue.completionDate.toString() : ''}</td>
        <td>{issue.title}</td>
      </tr>
    )
  }
}

```

We also used a local variable `issue` instead of referring to `this.props` all the time; this is just for readability. There are some things to note in this line:

```
...
    <td>{issue.completionDate ? ←
      issue.completionDate.toString() : ''}</td>
...
```

Firstly, we used the `?:` ternary operator to deal with a conditional display. Since anything within the curly braces is a JavaScript expression, this is a simple way of specifying an if-then-else condition. Secondly, we have to call `toString()` explicitly to format the date. React does not call a `toString()` automatically on objects, because it expects all objects as children of components to be React components if they are not strings.

To replace the inline styles, we need a style section in `index.html`, as seen in Listing 3-9.

Listing 3-9. `index.html`: Using CSS Styles

```
...
<script src=
  "https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js">
</script>
<style>
  table.bordered-table th, td {border: 1px solid silver; padding: 4px;}
  table.bordered-table {border-collapse: collapse};
</style>
</head>
...
```

After the above changes, the screen should look like Figure 3-3.

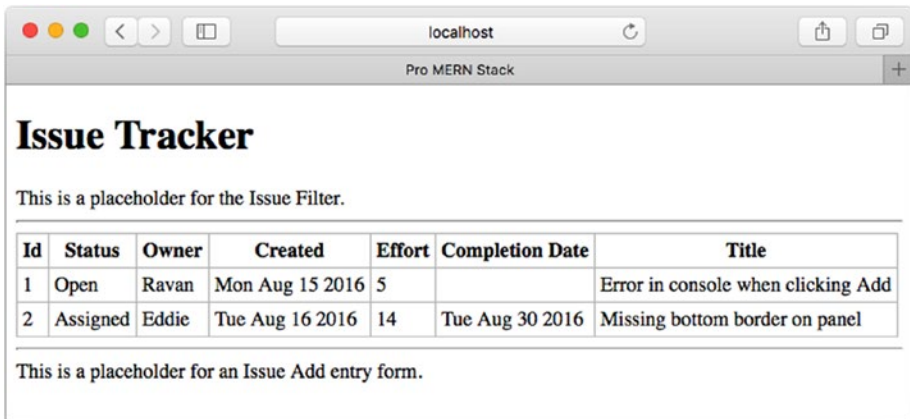


Figure 3-3. Rows constructed programmatically from an array

EXERCISE: DYNAMIC COMPOSITION

1. Why did we pass the property `issues` from `IssueTable`? Couldn't we have passed it from `IssueList`?
2. Remove the `key` property when constructing the array of `IssueRow` components. What happens? Explain this. Hint: Read about multiple components in the React documentation.
3. We used the issue's `id` field as the value of `key`. What other keys could have been used? Which would you choose?
4. In the previous section, we passed every field of an issue as a separate property to `IssueRow`. In this section, we passed the entire issue object. Why?
5. Instead of using a local variable `issueRows`, try using the `map` expression directly inside the `<tbody>`. Does it work? What does it tell us?

Answers are available at the end of the chapter.

Summary

In this chapter, we created a barebones version of the main page of the Issue Tracker. We used some placeholders to depict components that you have yet to develop. We did this by writing fine-grained individual components and putting them together (composing) in an enclosing component. We also saw how to pass parameters or data from an enclosing component to its children, to be able to reuse a single component with different data.

The components themselves didn't do much apart from rendering themselves based on the data. In the next chapter, we'll see how to add user interactivity that changes the appearance and manipulates data.

Answers to Exercises

Exercise: React Classes

1. Compilation will fail with an error, "Adjacent JSX elements must be wrapped in an enclosing tag". The `render()` method can only have a single return value, thus, it can return only one element. Enclosing the two `<div>`s in another `<div>` is one solution.
2. React prints an error in the browser's JavaScript console when it is a React error. Regular JavaScript errors are also shown in the console, but the code displayed is not the original code; it is the compiled code. You'll learn how to debug using the original source in later chapters.

Exercise: Passing Data

1. A border will not be displayed. How the JSX parser interprets each attribute is different from how an HTML parser does it. The border attribute is not one of the supported attributes. It completely ignores the border attribute, and it expects the style attribute to be an object, not a string.
2. The outer braces denote that the attribute value is a JavaScript expression. The inner braces specify an object, which is the attribute's value.
3. The curly braces of React are similar to `<?php ... ?>` of PHP, with a slight difference. The contents within a `<?php ... ?>` tag are full-fledged programs, whereas in JSX, you can only have JavaScript expressions. All programming constructs like loops are done outside the JSX in plain JavaScript.
4. props are flexible and useful for passing in any kind of data. On the other hand, children are components that can also be deeply nested. Thus, if you have simple data, pass it as props. If you have a component to pass, you could use a child if it is deeply nested and naturally appears within the child component. Components can also be passed as props, typically when you want to pass multiple components or when the component is not a natural child content of the parent.

Exercise: Dynamic Composition

1. It is best to keep the data at the topmost component that contains all the components that have a chance to deal with and manipulate the data. You'll learn more about this in the next chapter. But the gist is that `IssueAdd` and `IssueFilter` may also need access to the issue array, so it really belongs in `IssueList`.
2. The key property is essential for arrays of components. If you don't supply a key, React throws a warning that each child in an array or iterator should have a unique key property. React uses this key to uniquely identify every element in a row.
3. Another choice for the key property is the array index, because it is also unique. If the key is a large value like a UUID, you may think that it is more efficient to use the array index, but in reality it is not. React uses the key to *identify* the row. If it finds the same key, it assumes it is the same row. If the row has not changed, it does not rerender the row.

Thus, if you insert a row, React will be more efficient in shuffling existing rows rather than rerendering the entire table if the rows were the ID of the object. If you used the array index instead, it would think that every row after the inserted row has changed and rerender each of them.

4. Passing the entire object is obviously more concise. I would choose to pass individual properties only if the number of properties that are being passed is a small subset of the full set of properties of the object.
5. It works, despite the fact that you have JSX within the expression. Anything within the curly braces is parsed as a JavaScript expression. But since you are using a JSX transform on JavaScript expressions, these snippets will also go through the transform. It is possible to nest this deeper and use another set of curly braces within the nested piece of JSX and so on.

CHAPTER 4



React State

Until now, you've only seen static components. To make components that respond to user input and other events, React uses a data structure called *state* in the component. The state essentially holds the model, something that can change, as opposed to the immutable properties in the form of props that you saw earlier. It is only the change of state that can change the rendered view.

For this chapter, the goal is to add a button and append a row to the initial list of issues on the click of that button. We'll add this button below the Issues table. By doing that, you'll learn about a component's state, how to manipulate it, how to handle events, and how to communicate between components.

Setting State

We'll start by appending a row without user interaction. We'll do this using a timer rather than a button so that we can focus on the state and modifications without having to deal with UI complexity.

React treats the component as a simple state machine. Whenever the state changes, it triggers a re-render of the component and the view automatically changes. The way to inform React of a state change is by using the `setState()` method. This method takes in an object, and the top-level properties are *merged* into the existing state. Within the component, you can access the properties via the `this.state` variable. The initialization of the state is done in the constructor.

What we include in the state is up to us so we'll keep a list of issues in the state. Thus, the state will have a single property called `issues`, which we will initialize to the global `issues` array in the constructor. Also, in the constructor, we'll add a timer that appends a new issue to the list and sets it in the new state. We are not supposed to modify the state directly, so we need to make a copy or a clone of the existing array in the state, append to it, and set it as the state. Listing 4-1 shows the new `IssueList` class.

Listing 4-1. App.jsx: Initializing and Modifying State

```
class IssueList extends React.Component {
  constructor() {
    super();
    this.state = { issues: issues };
  }
}
```



```

    setTimeout(this.createTestIssue.bind(this), 2000);
  }

  createIssue(newIssue) {
    const newIssues = this.state.issues.slice();
    newIssue.id = this.state.issues.length + 1;
    newIssues.push(newIssue);
    this.setState({ issues: newIssues });
  }

  createTestIssue() {
    this.createIssue({
      status: 'New', owner: 'Pieta', created: new Date(),
      title: 'Completion date should be optional',
    });
  }

  render() {
    return (
      <div>
        <h1>Issue Tracker</h1>
        <IssueFilter />
        <hr />
        <IssueTable issues={this.state.issues} />
        <hr />
        <IssueAdd />
      </div>
    );
  }
}

```

On running this set of changes and refreshing the browser, you'll see that there are two rows of issues to start with. After two seconds, another row is added. Let's examine what happened here.

Firstly, let's look at the state initialization in the constructor:

```

...
  this.state = { issues: issues };
...

```

Initializing the state is as simple as setting the `this.state` variable to the state object.

■ **Note** If you are using `React.createClass` instead of extending from `React.Component`, the state has to be initialized within the method `getInitialState()`. The return value of that function is the initial state.

We then passed the data contained in the state to the `IssueTable` via properties, as discussed in the previous chapter, replacing the global array with the state data, as follows:

```
...
    <IssueTable issues={this.state.issues} />
...
```

The initial rendering of the `IssueTable` component will now use this array as its source data. Thus, same as in the previous chapter, you will see two rows of issues displayed in the table. But in the constructor, we also added a timer to do something:

```
...
    setTimeout(this.createTestIssue.bind(this), 2000);
...
```

This means that 2000 milliseconds after the constructor is called, `this.createIssue` will be called. Note that we had to include a `bind(this)` on the function instead of passing it as is. This is because we want the context, or the `this` variable when the function is called, to be this component's instance. If we don't do this, the `this` variable will be set to the event that called the function. When the timer fires, `createTestIssue` is called, which uses a test issue object and calls `createIssue` using that object.

Let's look at `createIssue` more closely. Here's the code:

```
...
    const newIssues = this.state.issues.slice();
    newIssue.id = this.state.issues.length + 1;
    newIssues.push(newIssue);
    this.setState({ issues: newIssues });
...
```

In the first line, we made a copy of the `issues` array in the state by calling `slice()` on it. We then pushed the new issue to be created into the array. Lastly, we called `this.setState` with the new array, thus modifying the state of the component. When React sees the state being modified this way, it triggers a rerendering process for the component, and *all* *descendent* components where properties get affected because of the state change.

Thus, `IssueTable` and all `IssueRows` will also be rerendered, and when they are, their properties will reflect the new state of the parent `IssueList` component automatically. And this will include the new issue. This is what the declarative programming paradigm is all about: you just mutate the model (state), and the view rerenders itself to reflect the changes. We did not have to write code for inserting a row into the DOM; it was automatic.

Note that we made a *copy* of the state value `issues`. This is important, because you are not supposed to modify the state directly. The following may seem to work:

```
...
    this.state.issues.push(newIssue);
    this.setState({ issues: this.state.issues });
...
```

But this will have unintended consequences in some of the Lifecycle methods within descendent components. Especially in those methods that compare the old and new properties, you'll find that the old and new properties are the same. There are React add-ons such as the update add-on to help you with creating copies when the change is deeply nested within the state. For now, we'll manage the copy ourselves.

EXERCISE: SETTING STATE

1. Remove `bind(this)` in the `setTimeout()` call. What happens?
2. We passed the function `this.createIssue` as a variable only once. Instead, if we refer to it multiple times, we have to do the `bind` in each of those references. Can you think of alternatives? Hint: Look up the guide called *Reusable Components* in the React documentation. There is a section titled "No Autobinding." Read it.
3. Add a `console.log` in the `IssueRow`'s `render()` method. How many times is `render()` called?

Answers are available at the end of the chapter.

Async State Initialization

In reality, you will not have an initial set of issues available to you. This list will be fetched from your web server. Let's simulate this condition: the initial set of issues will be empty, and it will be loaded via an asynchronous call as soon as the component is ready.

In the constructor, let's modify the state initialization to an empty array. Then, let's add a method for loading data called `loadData()`, which will use the global issues list to set the state. We'll simulate the asynchronous nature of an AJAX call by using a timer (with a small timeout, something that is reasonable for an AJAX call to a server) to wrap the `setState()` call.

Finally, we'll need to make a call to `loadData()` somewhere. A good place to do this is when you're sure the component is mounted and ready to receive `setState()` calls. React provides a Lifecycle method called `componentDidMount()` to indicate that the component is ready so let's use it.

Listing 4-2 shows the new code: the constructor is modified and two new methods, `componentDidMount()` and `loadData()`, have been introduced.

Listing 4-2. App.jsx, IssueList: Loading State Asynchronously

```

...
constructor() {
  super();
  this.state = { issues: [] };

  setTimeout(this.createTestIssue.bind(this), 2000);
}

componentDidMount() {
  this.loadData();
}

loadData() {
  setTimeout(() => {
    this.setState({ issues: issues });
    }, 500);
}
...

```

Note that we don't have to bind `loadData` to `this` because we used an arrow function, which uses the lexical `this`. Thus in the anonymous function that's passed to `setTimeout`, the `this` variable is initialized to the component instance.

We also used the first Lifecycle method hook, `componentDidMount()`. Apart from this, there are other hooks into the component lifecycle that React calls and lets us take action on those events. The method `componentDidMount()`, as the name indicates, is called after the component is *mounted*, that is, created and placed into the DOM. The other hooks related to mounting are `componentWillMount()` and `componentWillUnmount()`.

The hooks related to an update, that is, when the state or props of a component change are `componentWillReceiveProps()`, `componentWillUpdate()`, `componentDidUpdate()`, and `shouldComponentUpdate()`. We will be using some of them in later chapters to hook into events that indicate change of props. Of these hooks, `shouldComponentUpdate()` is an optimization hook that can be used to let React know exactly when there is a change in the display of the component; otherwise React plays it safe and rerenders the component on *any* state or props change, which may or may not change the display.

■ **Note** It is tempting to initiate `loadData()` within the constructor, but that should not be done. That's because there is a chance that the load finishes even before the component is ready (i.e., not yet rendered). Calling `setState()` can cause unexpected behavior if the component is not yet ready.

Event Handling

Let's now add an issue interactively on the click of a button. We'll first add a simple button component next to the `IssueTable`. To handle the button's click event, all you need to do is attach a handler function to the event. We do this by supplying the name of the handler to the `onClick` attribute. We can directly set the `createTestIssue` function as the handler to this event.

Listing 4-3 shows a modified constructor and a modified `render()` function of `IssueList`.

Listing 4-3. `App.jsx`, `IssueList`: Button and Event Handler

```
...
constructor() {
  super();
  this.state = { issues: [] };

  this.createTestIssue = this.createTestIssue.bind(this);
  setTimeout(this.createTestIssue, 2000);
}
...
    <IssueTable issues={this.state.issues} />
    <button onClick={this.createTestIssue}>Add</button>
    <hr />
...

```

The `createTestIssue` method takes no parameters and appends the sample issue to the list of issues in the state. We retained the timer way of adding an issue as well, just to ensure that works too. We also get rid of multiple binds by replacing `this.createTestIssue` with a permanently bound version in the constructor. Going forward, we'll use this strategy in all methods that need to be bound.

When you try out the changes, you'll find that a test row is added 2 seconds after the page loads, and then, on the click of the Add button, any new number of rows can be added interactively from the UI.

Communicating from Child to Parent

Ideally, the Add button should be within the `IssueAdd` component, as that's where its functionality belongs. We didn't do this earlier to avoid the complexity of communicating from a child component to its parent. Let's move the button now to where it belongs, and see how to communicate from child to parent. Since the button will move, its handler will also have to move to the `IssueAdd` component.

Instead of a hard-coded test issue, let's create a form in this component, with input fields that we'll use for the values of the new issue's fields. This handler will create a new issue object based on the form's input fields.

But how will this handler function get access to the `createIssue` method, which is in its parent, `IssueList`? Does the child component have a handle to its parent?

For good reason, no, the child does not have access to the parent's methods. The way to communicate from the child to a parent is by passing callbacks from the parent to the child, which it can call to achieve specific tasks. In this case, you pass `createIssue` as a callback property from `IssueTable` to `IssueAdd`. From the child, you just call the passed in function in your handler to create a new issue.

Listing 4-4 shows the new `IssueAdd` class. The click handler is called `handleSubmit`, and within this method, we read the form's input values and using them, we call the `createIssue` function, which is available to the component via `this.props`.

Listing 4-4. `App.jsx`, `IssueAdd`: Handling Add from This Component

```
class IssueAdd extends React.Component {
  constructor() {
    super();
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleSubmit(e) {
    e.preventDefault();
    var form = document.forms.issueAdd;
    this.props.createIssue({
      owner: form.owner.value,
      title: form.title.value,
      status: 'New',
      created: new Date(),
    });
    // clear the form for the next input
    form.owner.value = ""; form.title.value = "";
  }

  render() {
    return (
      <div>
        <form name="issueAdd" onSubmit={this.handleSubmit}>
          <input type="text" name="owner" placeholder="Owner" />
          <input type="text" name="title" placeholder="Title" />
          <button>Add</button>
        </form>
      </div>
    )
  }
}
```

Let's discuss a few things in the new components. The first thing we did was create a form with two text input fields for accepting Owner and Title of a new issue from the user. We also included an Add button in the form.

Note that unlike the previous step, we are handling the form's `onSubmit` event rather than the button's `onClick` event. Both methods are acceptable, but using `onSubmit` will allow the user to press Enter to add a new issue in addition to clicking on the Add button.

We also gave a name to the form so that we could access the form's input fields programmatically. In the submit handler, the first thing we did was prevent the default behavior of the form:

```
...
  handleSubmit(e) {
    e.preventDefault();
  }
...
```

The rest of the event handler is straightforward. We collected the form input values, constructed a new issue object with some default values for the other fields, and called the parent's `createIssue` method via the callback that we had in `this.props.createIssue`.

Now, let's make changes to `IssueList`. The main thing we need to do is pass the `createIssue` method as a property to `IssueAdd`. Note that we must bind this method in the constructor since it's now being called from another component (so that the `this` variable during the call will be the calling component). We can also delete all of the code that was used for creating a test issue using a timer as well as from the button within `IssueList`. The changes to `IssueList` are shown in Listing 4-5.

Listing 4-5. `App.jsx`, `IssueList`: Moved Add Functionality to Child

```
...
super();
  this.state = { issues: [] };

  this.createTestIssue = this.createTestIssue.bind(this);
  setTimeout(this.createTestIssue, 2000);
  this.createIssue = this.createIssue.bind(this);
}
...
createTestIssue(){
  this.createIssue({
    status: 'New', owner: 'Pieta', created: new Date(),
    title: 'Completion date should be optional',
  });
}
...
  <IssueTable issues={this.state.issues} />
  <button onClick={this.createTestIssue}>Add</button>
  <hr />
  <IssueAdd createIssue={this.createIssue} />
  </div>
);
...
```

The new screen (shown in in Figure 4-1) now has a form for entering values and adding a new issue. You can test it by entering some values in the input fields and clicking the Add button to add a new issue.

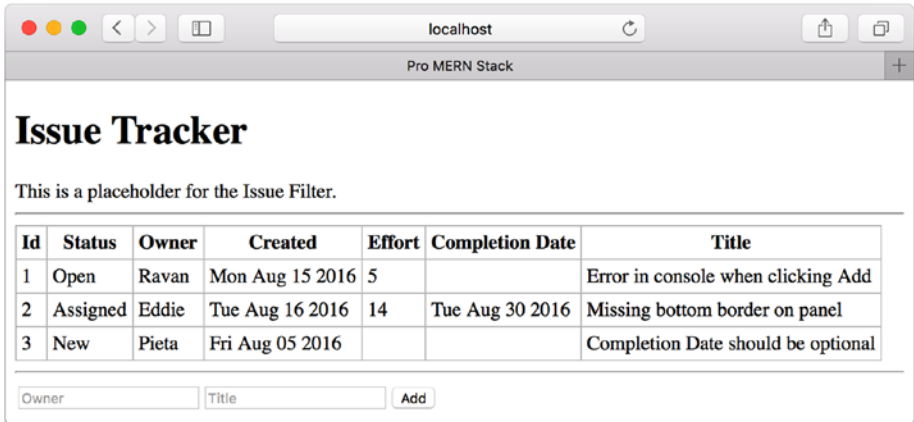


Figure 4-1. Issue Tracker with user interactivity

EXERCISE: COMMUNICATE CHILD TO PARENT

1. Remove `e.preventDefault()` in `handleSubmit()`. What happens? Can you explain why?
2. Would it have been possible to achieve all this if we had maintained the state in `IssueTable` instead of `IssueList`?
3. Refresh the browser; you'll see that the added issues are gone. How can you persist the changes?

Answers are available at the end of the chapter.

Stateless Components

We have added three React components by now (another one, the Issue Filter is still a placeholder). But there is a difference among them.

`IssueList` has lots of methods, a state, initialization, and functions that modify the state. In comparison, `IssueAdd` has some interactivity, but no state.¹ But, if you notice, `IssueRow` and `IssueTable` have nothing but a `render()` method. For performance reasons, it is recommended that such components are written as functions rather than classes: a function that takes in props and just renders based on it. It's as if the component's view is a pure function of its props.

¹This is not entirely true. There is, in fact, state in this component: the state of the input fields as the user is typing. But you have not captured them as React state, and have let the browser's native handlers maintain it.

Listing 4-6 shows these components changed to stateless functions.

Listing 4-6. App.jsx, IssueRow, and IssueTable: Stateless Components

```
const IssueRow = (props) => (
  <tr>
    <td>{props.issue.id}</td>
    <td>{props.issue.status}</td>
    <td>{props.issue.owner}</td>
    <td>{props.issue.created.toString()}</td>
    <td>{props.issue.effort}</td>
    <td>{props.issue.completionDate ? ←
      props.issue.completionDate.toString() : ''}</td>
    <td>{props.issue.title}</td>
  </tr>
)

function IssueTable(props) {
  const issueRows = props.issues.map(issue =><IssueRow ←
    key={issue.id} issue={issue} />);
  return (
    <table className="bordered-table">
      <thead>
        <tr>
          <th>Id</th>
          <th>Status</th>
          <th>Owner</th>
          <th>Created</th>
          <th>Effort</th>
          <th>Completion Date</th>
          <th>Title</th>
        </tr>
      </thead>
      <tbody>{issueRows}</tbody>
    </table>
  );
}
```

We used two different styles for the two components. The first is the ES2015 arrow function style with only the return value as an expression. There are no curly braces, and no statements, just a JSX expression:

```
...
const IssueRow = (props) => (
...

```

The second style, a little less concise, is needed when the function is not a single expression. We initialized a variable called `issueRows`, which means we need a full-fledged function with a return value. The main difference is the use of a curly brace to indicate that there's going to be a return value, rather than the expression within the round braces being an implicit return of that expression's result:

```
...
function IssueTable(props) {
  const issueRows = props.issues.map(issue =><IssueRow
    key={issue.id} issue={issue} />);
  ...
}
```

Technically, we could have avoided defining the `issueRows` variable and replaced its reference within `<tbody>` with the variable expression itself. But let's keep it like this for the sake of readability.

Designing Components

Most beginners will have a bit of confusion between state and props, when to use which, what granularity of components should one choose, and how to go about it all. This section is devoted to discussing some principles and best practices.

State vs. props

Both state and props hold model information, but they are different. The props are immutable, whereas state is not. Typically, state variables are passed down to child components as props because the children don't maintain or modify them. They take in read-only copy and use it only to render the view of the component. If any event in the child affects the parent's state, the child calls a method defined in the parent. Access to this method should have been explicitly given by passing it as a callback via props.

Anything that *can* change due to an event anywhere in the component hierarchy qualifies as being part of the state. Avoid keeping computed values in the state; instead, simply compute them when needed, typically inside the `render()` method.

Do not copy props into state, just because props are immutable. If you feel the need to do this, think of modifying the original state from which these props were derived. One exception is when props are used as *initial* values to the state, and the state is truly disjointed from the original state after the initialization.

Component Hierarchy

Split the application into components and subcomponents. Typically, this will reflect the data model itself. For example, in the Issue Tracker, the issues array was represented by the `IssueTable` component, and each issue was represented by the `IssueRow` component.

Decide on the granularity just as you would for splitting functions and objects. The component should be self-contained with minimal and logical interfaces to the parent. If it is doing too many things, just like for functions, it should probably be split into multiple components, so that it follows the Single Responsibility principle (that is, every component should be responsible for one and only one thing). If you are passing in too many props to a component, it is an indication that either the component needs to be split, or it need not exist; the parent itself could do the job.

Communication

Communication between components depends on the direction. Parents communicate to children via props; when state changes, the props automatically change. Children communicate to parents via callbacks.

Siblings and cousins can't communicate with each other, so if there is a need, the information has to go up the hierarchy and then back down. This is what we did when adding a new issue. The component `IssueAdd` had to insert a row in `IssueTable`. It was achieved by keeping the state in the least common ancestor, `IssueList`. The addition was initiated by `IssueAdd` and a new array element added in `IssueList`'s state via a callback. The result was seen in `IssueTable` by passing the `issues` array down as props from `IssueList`.

If there is a need to know the state of a child in a parent, you're probably doing it wrong. Although React does offer a way using `refs`, you shouldn't feel the need if you follow the one-way data flow strictly: state flows as props into children, and events cause state changes, which flows back as props.

Stateless Components

In a well-designed application, most components are stateless functions of their properties. All state is captured in a few components at the top of the hierarchy, from where the props of all the descendants are derived.

We did just that with the `IssueList`, where we kept the state. We converted all descendent components to stateless components, relying only on props passed down the hierarchy to render themselves. We kept the state in `IssueList` because that was the least common component above all the descendants that depended on that state. It's also OK to invent a new component just to hold the state.

Summary

In this chapter, you learned how to use state and make changes to it on user interactions or other events. The more interesting aspect was how state values are propagated down the component hierarchy as props. You also had a glimpse of user interaction, the click of a button to add a new issue, and how that causes the state to change, and in turn, the rendering via props in all the descendant components.

But we used simulated asynchronous calls and local data to achieve all this.

In the next chapter, instead of using local data, we'll get the data from the server, and also save to it.

Answers to Exercises

Exercise: Setting State

1. If you remove `bind(this)`, you'll get an error because `this.state` will be undefined (since `this` is now the window object, not the component).
2. One alternative that is popular is to replace the function with a permanently bound function in the constructor like this:

```
...
this.state = { issues: issues };
this.createTestIssue = this.createTestIssue.bind(this);
...
```

Now, you can just use `this.createTestIssue` all the time. One of the reasons some people prefer `React.createClass` as compared to extending from `React.Component` is that `React.createClass` auto-binds all the functions. This extra step is then not required. This saves debugging time spent whenever you forget to bind a function.

Another alternative is to use ES2015 arrow functions, which use a lexical `this`, that is, picks it up from the surroundings rather than the caller's `this`, like this:

```
...
setTimeout(() => {this.createTestIssue()}, 2000);
...
```

3. Each row is rendered once when initialized (two renders, one for each row). After the new row is inserted, each row is rendered once too (three renders, one for each row). Although a render is called, this does not mean that the DOM is updated. Only the virtual DOM is created on each render. A real DOM update happens only where there are differences.

Exercise: Communicate Child to Parent

1. On removing `e.preventDefault()`, the default behavior of the form is executed, which is to really submit the form. This does a GET (the default action, if not specified) to the form's action URL, which is the same as the current URL. Thus, the effect is to refresh the page even before the event is handled.
2. Since there is no way to communicate between siblings (only parent to child and vice versa), keeping the state at the root of the hierarchy is the best strategy. If we had kept the state in `IssueTable`, tying up the Add action would have meant calling a function that belongs in `IssueTable` from `IssueAdd`. This is neither simple nor a good practice.
3. To persist the changes, you could either save the issues in local storage on the browser, or save it in the server. We'll be saving it in the server in a later chapter.

CHAPTER 5



Express REST APIs

Now that you know how to deal with states and models in the UI, let's spend some time integrating with the backend for the *real* data.

In this chapter, we'll start storing in and fetching the data from the Node.js server instead of the hard-coded array of issues in the browser's memory. The objective is to get used to interacting with the server via REST APIs. We will be modifying both front-end and back-end code because we'll be implementing as well as using REST APIs. As for persistence, let's leave it to the next chapter. For the moment, we'll save the data in an in-memory data structure on the server.

REST

REST (short for representational state transfer) is an architectural pattern for application programming interfaces (APIs). There are other older patterns such as SOAP and XML-RPC, but of late, the REST pattern has gained popularity.

Since the APIs are only for internal consumption, you can use any API pattern, or even invent your own. But REST is a good pattern to adopt because it is simple and has very few constructs. Further, adopting an existing popular pattern makes you think about your APIs and schema, and organize them better, forcing some best practices.

There are many architectural constraints that true REST APIs must follow. But keeping in mind that the APIs are only for internal consumption, we'll just use a few core concepts of the REST API architectural pattern rather than follow it very strictly, which requires a lot more work. If you ever decide to expose the APIs to others, you can strengthen the RESTfulness of the API set. I'll briefly discuss the core concepts that we will make use of in your API design.

Resource Based

The APIs are resource based (as opposed to action based). Thus, you will not find API names like `getSomething` or `saveSomething`. In fact, there are no API names because APIs are formed by a combination of resources and actions.

Resources are accessed based on a Uniform Resource Identifier (URI), also known as an *endpoint*. Resources are nouns (not verbs). You typically use two URIs per resource: one for the collection (like `/customers`) and one for the object (like `/customers/1234`, where 1234 uniquely identifies a customer).

Resources can also form a hierarchy. For example, the collection of orders of a customer is identified by `/customers/1234/orders`, and an order of that customer is identified by `/customers/1234/orders/43`.

HTTP Methods as Actions

To access and manipulate the resources, you use HTTP methods. While resources were nouns, the HTTP methods are verbs that operate on them. They map to CRUD (Create, Read, Update, Delete) operations on the resource. Tables 5-1 shows commonly used mapping of CRUD operations to HTTP methods and resources.

Table 5-1. *CRUD Mapping for Collections*

Operation	HTTP Method	Resource	Example	Remarks
Read – List	GET	Collection	GET <code>/customers</code>	Lists objects (additional query string can be used to filter)
Read	GET	Object	GET <code>/customers/1234</code>	Returns a single object (query string may be used to filter fields)
Create	POST	Collection	POST <code>/customers</code>	Creates an object, and the object is supplied in the body.
Update	PUT	Object	PUT <code>/customers/1234</code>	Replaces the object with the object supplied in the body.
Update	PATCH	Object	PATCH <code>/customers/1234</code>	Modifies some attributes of the object, specification in the body.
Delete	DELETE	Object	DELETE <code>/customers/1234</code>	Deletes the object

Some other operations such as DELETE and PUT in the collection may also be used to delete and modify the entire collection in one shot, but this is not common usage. HEAD and OPTIONS are also valid verbs that give out information about the resources rather than actual data. They are used mainly for APIs that are externally exposed and consumed by many different clients.

Two important concepts about the HTTP methods are *safety* and *idempotency* of the methods. A safe method is one whose results can be cached. Thus, GET, HEAD, and OPTIONS are safe methods; you can call them any number of times and get the same results. An idempotent method is one that has the same effect when called multiple times. Note that it's not the same result; instead it's the effect on the resource. A safe method is always idempotent, but not the other way round.

As for the rest of the HTTP methods, only PUT and DELETE are idempotent, whereas POST and PATCH are not. If you use PUT multiple times, you continue to replace the same resource with the *same* new contents, thus the outcome is the same for each attempt. DELETE, if seen as “let the resource not exist,” also has the same outcome, even though in the second and later attempts the resource is not deleted because it doesn't exist any longer. But these are not safe methods, because their results can be different. For example, DELETE when called the first time can return a success, whereas the second time can return a warning that the resource doesn't exist.

The reason the HTTP methods are mapped to the CRUD operations as listed in Table 5-1 is due to the idempotency and safety of these methods. This is also why PATCH and PUT are subtly different, even though both can be used to update a resource. You can use PATCH to modify a resource by adding to an array in the resource. For example, you can add a new phone number to a customer resource. When called multiple times, the resource keeps changing because the same element is added again and again to the array. The same is not true for PUT since it completely replaces the resource.

It's important to keep these two concepts in mind when creating specialized variations of a REST API. For example, if you know the ID of the resource that you want to create if it doesn't already exist (also called an *upsert* operation, a portmanteau of update and insert), you use PUT to create that resource, even though POST is the preferred method for creating. That's because if the resource doesn't exist, it is created, but if it does, it is replaced. The net effect is the same: at the end of the operation, you have a resource as specified. And this is an idempotent operation.

It is a good practice to prefix the URI with `/api/` to separate the APIs from application-related HTTP resources such as HTML files. Further, adding a version to the API ensures that you respect backward compatibility for external consumers, like `/api/v1/`. If you don't think you'll ever expose the APIs to an external entity, it's OK to skip the version indicator. Note that a mobile application should be considered an external entity, because you could potentially have different versions of the application accessing your APIs.

JSON

JSON is used as the preferred encoding for the data, both in request and response bodies. Some REST specifications allow the caller to specify the format (JSON, XML, CSV, etc.), but we will be supporting only JSON, because we don't expect your API to be exposed to other consumers who may want this choice. So we will skip the need for the client to specify the format.

Express

I briefly touched upon Express and how to serve static files using Express in the Hello World chapter. But Express can do much more than just serve static files. Express is a *minimal*, yet, flexible web application framework. It's minimal in the sense that by itself, Express does very little. It relies on other modules (middleware) to provide the functionality that most applications will need. The concepts are simple and easy to understand.

Routing

The first concept is that of routing. At the heart of Express is a router, which essentially takes a client request, matches it against any *routes* that are present, and executes the *handler* function that is associated with that route. The handler function is expected to generate the appropriate response.

A route specification consists of an HTTP method (GET, POST, etc.), a path specification that matches the request URI, and the route handler. The handler is passed in a request object and a response object. The request object can be inspected to get the various details of the request, and the response object's methods can be used to send the response to the client. All this may seem a little overwhelming, so let's just start with a simple example and explore the details:

```
const app = express();

app.get('/hello', (req, res) => {
  res.send('Hello World');
});
```

To use Express, you first need to create an application using its root-level exported function. You do that using `const app = express();`. You can now set up routes using this app. To set up a route, you use a function to indicate which HTTP method; for example, to handle the GET HTTP method, you use the app's `get()` function. To this function, you pass the pattern to match and a function to deal with the request if it does match.

Request Matching

When a request is received, the first thing that happens is request matching. The request's method is matched with the route method (the `get` function was called on `app`, indicating it should match only GET HTTP methods), and also the request URL with the path spec (`'/hello'` in the above example). If a request matches this spec, then the handler is called. In other words, the above example reads “If you receive a GET request to the URL `/hello`, then execute *this* piece of code.”

The method as well as the path need not be very specific. Normally, you would use `app.get()`, `app.post()`, `app.put()`, etc., but if you want to say “any method,” you could use `app.all()`. The path specification can also take regular expression-like patterns (like `'/*'`, `do'`) or regular expressions themselves. But regular expressions in paths are rarely used. Route *parameters* in the path are used often, so I'll discuss them in a little more detail.

Route Parameters

Route parameters are named segments in the path specification that match a part of the URL. If a match occurs, the value in that part of the URL is supplied as a variable in the request object. This is used in the following form:

```
app.get('/customers/:customerId', ...
```

The URL `/customers/1234` will match the above route specification, and so will `/customers/4567`. In either case, the customer ID will be captured and supplied to the handler function as part of the request in `req.params`, with the name of the parameter as the key. Thus, `req.params.customerId` will have the value `1234` or `4567` for each of these URLs, respectively.

It's no coincidence that route parameters work really well for what we discussed as good REST API endpoints. This feature was designed just for that. In fact, you can have multiple parameters, for example `/customers/:customerId/orders/:orderId`, to match a customer's order.

■ **Note** The query string is not part of the path specification, so you cannot have different handlers for different parameters or values of the query string.

Route Lookup

Multiple routes can be set up to match different URLs and patterns. The router does not try to find a best match; instead, it tries to match all routes in the order in which they are installed. The first match is used. So, if two routes are possible matches to a request, it will use the first defined one. It is up to you to define routes in the order of priority. So, when you add patterns rather than very specific paths, you should be careful to add the more generic pattern *after* the specific paths. For example, if you want to match everything that goes under `/api/`, that is, a pattern like `/api/*`, you should add this route only *after* all the specific routes that handle paths such as `/api/issues`.

Handler Function

Once a route is matched, the handler function is called, which in the above example was an anonymous function supplied to the route setup function. The parameters passed to the handler are a request object and a response object. Let's briefly look at the important properties and methods of these objects.

Request Objects

Any aspect of the request can be inspected using the request object's properties and methods. You already saw how to access a parameter value using `req.params`. The other important properties follow.

- `req.query`: This holds a parsed query string. It's an object with keys as the query string parameters and values as the query string values. Multiple keys with the same name are converted to arrays, and keys with a square bracket notation result in nested objects (e.g., `order[status]=closed` can be accessed as `req.query.order.status`).
- `req.header`, `req.get(header)`: The `get` method gives access to any header in the request. The `header` property is an object with all headers stored as key-value pairs. Some headers are treated specially (like `Accept`), and have dedicated methods in the request object for them. That's because common tasks that depend on these headers can be easily handled.
- `req.path`: This contains the path part of the URL, that is, everything up to any `?` that starts the query string. Usually, the path is part of the route if the route is not a pattern, but if the path is a pattern that can match different URLs, you can use this property to get the actual path that was received in the request.
- `req.url`, `req.originalURL`: These properties contain the complete URL, including the query string. Note that if you have any middleware that modifies the request URL, `originalURL` will hold the URL as it was received, *before* the modification.
- `req.body`: This contains the body of the request, valid for POST, PUT, and PATCH requests. Note that the body is not available (`req.body` will be undefined) unless a middleware is installed to read and optionally interpret or parse the body.

There are many other methods and properties; for a complete list, please refer to the Request documentation of Express at <http://expressjs.com/en/api.html#req> as well as Node.js' request object at https://nodejs.org/api/http.html#http_class_http_incomingmessage, from which the Express Request is extended.

Response Objects

The response object is used to construct and send a response to a request. If no response is sent, the client is left waiting.

- `res.send(body)`: You already saw the `res.send()` method briefly, which responded with a string. This method can also accept a buffer (in which case the content type is set as `application/octet-stream` as opposed to `text/html` in case of a string). If the body is an object or an array, it is automatically converted to a JSON string with an appropriate content type.

- `res.status(code)`: This sets the response status code. If not set, it is defaulted to 200 OK. One common way of sending an error is by combining the `status()` and `send()` methods in a single call like `res.status(403).send("Access Denied")`.
- `res.json(object)`: This is the same as `res.send()`, except that this method forces conversion of the parameter passed into a JSON, whereas `res.send()` may treat some parameters like null differently. It also makes the code readable and explicit, stating that you are indeed sending out a JSON.
- `res.sendFile(path)`: This responds with the contents of the file at path. The content type of the response is guessed using the extension of the file.

There are many other methods and properties in the response object; you can look at the complete list in the Express documentation for Response at <http://expressjs.com/en/api.html#res> and also Node.js' Response object in the HTTP module at https://nodejs.org/api/http.html#http_class_http_serverresponse. But for the Issue Tracker application, the methods I've discussed should suffice.

Middleware

Express is a web framework that has minimal functionality of its own. An Express application is essentially a series of middleware function calls. In fact, the Router itself is nothing but a middleware function.

Middleware functions are those that have access to the [request object](#) (`req`), the [response object](#) (`res`), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named `next`. I won't go into the details of how to write your own middleware functions, since we will not be writing new middleware in the application. But we will use some middleware for sure.

We already used one middleware called `express.static` in the Hello World example, to serve static files. This is the only built-in middleware (other than the router) available as part of Express. But there are other very useful middleware supported by the Express team, of which we will be using `body-parser` in this chapter. Third-party middleware is available via npm.

Middleware can be at the application level (that is, applies to all requests) or the router level (applies to specific request path patterns). The way to use a middleware at the application level is to simply supply the function to the application, like this:

```
app.use(middlewareFunction);
```

The actual middleware function is supplied by the module in a module-specific way. For example, the `static` middleware used a "factory" function to "manufacture" a middleware function. We supplied the factory function one parameter: the location of the static files. This created the real middleware function that knows the location of static files to serve.

In order to use the same middleware in a route-specific way, you could have done the following:

```
app.use('/public', express.static('static'));
```

This would have mounted the static files on the path `/public` and all static files would have to be accessed with the prefix `/public`, for example, `/public/index.html`.

The List API

Now that you have learned the basic concepts of Express, let's start by implementing the first API. This will be the List API, which lists all issues. We're not integrating this with the front-end code just yet; instead we'll test it by other means. We'll directly call this API from the browser or the shell using `curl`. We also won't use the version indicator in the APIs, since they are for internal consumption only.

Since the Issue Tracker application has a single resource for now, the endpoint or URI that we'll use is straightforward: `/api/issues`. All the List API has to do is return the complete list of issues.

We will store the list of issues in your server's memory, by moving the global array from the file `App.jsx` to the server. Then, we will add a `get` route to the application, which just sends out this global array of issues as a JSON using `res.json()`.

The new file, `server.js`, is shown in Listing 5-1.

Listing 5-1. `server.js`: New List API

```
const express = require('express');

const app = express();
app.use(express.static('static'));

const issues = [
  {
    id: 1, status: 'Open', owner: 'Ravan',
    created: new Date('2016-08-15'), effort: 5, completionDate: undefined,
    title: 'Error in console when clicking Add',
  },
  {
    id: 2, status: 'Assigned', owner: 'Eddie',
    created: new Date('2016-08-16'), effort: 14, ←
    completionDate: new Date('2016-08-30'),
    title: 'Missing bottom border on panel',
  },
];

app.get('/api/issues', (req, res) => {
  const metadata = { total_count: issues.length };
  res.json({ _metadata: metadata, records: issues });
});
```

```
app.listen(3000, () => {
  console.log('App started on port 3000');
});
```

Automatic Server Restart

In order to have the changes take effect, you need to restart the server. You can stop the server using Ctrl-C, and start it again using `npm start`. But, if you are going to make many changes and test them often, this soon becomes a hassle. It's not just the act of restarting, but you'll find yourself spending time debugging why something didn't work and realize it was all because the server was not restarted!

To automatically restart the server on changes, let's install and use the package *nodemon*. You may also find by searching the Internet that *forever* is another package that can be used to achieve the same goal. Typically, *forever* is used to restart the server on crashes rather than watch for changes to files. The best practice is to use *nodemon* during development (where you watch for changes) and *forever* on production (where you restart on crashes). So, let's install *nodemon* now:

```
$ npm install nodemon --save-dev
```

Since it is a local install, you need to use a script in `package.json` that tells `npm` to use *nodemon* instead of running the server directly. You can either modify the `start` command or add a new command. We will just modify the `start` command, assuming that on production, we won't be using `npm start`. Here are the changes in `package.json`:

```
...
"scripts": {
  "start": "nodemon -w server.js server.js",
...

```

The `-w` command line option is to tell *nodemon* which files to watch for changes. If we hadn't supplied that command line option, it would have watched for changes in *any* file in the current directory and subdirectories. Thus, it would have restarted even when front-end code changed, and that's not what we want.

After the installation and changes to `package.json`, you can run `npm start` and see that any change to `server.js` automatically restarts the server. You can also change `App.jsx` and see that it does not restart the server, even when `App.js` is generated and saved in the static directory.

Testing

Now that you've restarted the server, let's test the newly created API. To test it, you can just type `http://localhost:3000/api/issues` in the browser's URL. Or, you can use the command line utility `curl`, which will come in handy while testing HTTP methods other than GET (because you can only simulate a GET by typing in the URL). It may also be

useful to install browser extensions or apps such as *getpostman* (www.getpostman.com/), a full-fledged API tester. The extension *jsonview* is also useful to see JSON output formatted nicely in the browser.

You can choose your favorite tool, but I personally prefer using Chrome's Developer Console to inspect the network traffic. It also automatically parses JSON and shows it as a collapsible tree. For the purpose of showing results in this book, I will show just a curl execution snippet. The curl command-line and the output of testing the List API are as follows:

```
$ curl -s http://localhost:3000/api/issues | json_pp
{
  "_metadata" : {
    "total_count" : 2
  },
  "records" : [
    {
      "id" : 1,
      "title" : "Error in console when clicking Add",
      "effort" : 5,
      "created" : "2016-08-15T00:00:00.000Z",
      "owner" : "Ravan",
      "status" : "Open"
    },
    {
      "title" : "Missing bottom border on panel",
      "created" : "2016-08-16T00:00:00.000Z",
      "effort" : 14,
      "id" : 2,
      "owner" : "Eddie",
      "status" : "Assigned",
      "completionDate" : "2016-08-30T00:00:00.000Z"
    }
  ]
}
```

Let's examine the interesting parts of the code. The entry point for the API is the Express route for the resource URI:

```
...
app.get('/api/issues', (req, res) => {
...

```

This code sets up a route such that a request to `/api/issues` (only GET requests, though) will be handled by the function that is defined here. The part of this function that returns a response is in the following line:

```
...
res.json({ _metadata: metadata, records: issues });
...
```

The above is actually a shortcut for the following:

```
...
res.set('Content-Type', 'application/json');
res.send(JSON.stringify({ _metadata: metadata, records: issues }));
...
```

This essentially sends out a JSON string representation of the object that we passed to the function. In this case, the object consists of a metadata and the set of records. For the set of records, we just used the in-memory array of issues. The metadata at the moment consists only of the total count of records. This can be expanded to add other useful metadata such as pagination information in the future.

EXERCISE: THE LIST API

1. Using the Chrome browser, type the API endpoint URL to test the API. Open the Network section of the Developer Console and hit refresh. You will see that the status code for the request is 304, and not a 200. Why is this happening? Hint: Search the Internet for “express etag 304”. Will it be a problem if the list of issues is modified?
2. Print a log statement just before sending the response. Now restart the server. Do you see the response being sent on a new request? Did you expect a 304 this time too, or a 200? What does this tell you about what Express might be doing?
3. If you look closely in the List API testing results, you will see that the output of curl is piped to json_pp. How will the output look if this was not done? If you wanted to return pretty-printed JSON always, what would you do? Hint: Look up the documentation of `JSON.stringify()` and also `Express app.set()`.
4. Search the Internet for “rest api link header,” especially *The GitHub Developer Guide*. Link headers are an alternative way of including pagination metadata. What are the pros and cons of using that approach as compared to what you have done (a wrapper on the output)?
5. Is the JSON output identical to the `issues` array? What are the differences? How do you think this will affect the client?

Answers are available at the end of the chapter.

The Create API

The application supports adding a new issue as well, as of now. So, in order to complete the transition of the data to the server, we need a Create API that supports adding a new issue.

Looking at the HTTP methods mapping in Table 5-1, you can see that the resource URI for this API needs to be `/api/issues` and the method needs to be a POST. The request body will contain the new issue object to be created.

Express does not have an in-built parser that can parse request bodies to convert them into objects. So, we first need to install an npm package that helps us do that. The package `body-parser` can parse various types of request bodies including URL Encoded form data and JSON. Let's use it:

```
$ npm install body-parser --save
```

Then, we need to include it in `server.js` (using a `require` statement) and we also need to let Express know that this middleware should be used. This is so that it can intercept requests, look at the content type, and deal with the body appropriately. At the moment, we only need the JSON parser, so only that middleware is set up. The JSON parser middleware is created using `bodyParser.json()`, and the middleware is mounted at the application level using `app.use()`.

Listing 5-2 shows the changes to `server.js` to get this done.

Listing 5-2. `server.js`: Including the `body-parser` Module

```
...
const bodyParser = require('body-parser');
...
app.use(express.static('static'));
app.use(bodyParser.json());
...
```

Now, we are ready to add the Express route for handling a POST to the endpoint `/api/issues`. The `body-parser` middleware places the result of parsing in the request's `body` property. So, within the handler, we need to use `req.body` to gain access to the new issue object that was passed in. After a little bit of processing, which includes generating an ID for the new issue and defaulting a few values, we will append this new issue to the in-memory `issues` array.

It is also good practice to return the newly created issue as the result of the operation. The client may use this to access the field values that were set in the server. Listing 5-3 shows the new route handler.

Listing 5-3. `server.js`: Creating an API POST Handler

```
app.post('/api/issues', (req, res) => {
  const newIssue = req.body;
  newIssue.id = issues.length + 1;
  newIssue.created = new Date();
```

```

if (!newIssue.status)
  newIssue.status = 'New';

issues.push(newIssue);

res.json(newIssue);
});

```

Testing the Create API will result in a transcript like this:

```

$ curl -s http://localhost:3000/api/issues \
--data '{"title": "Test Test", "owner": "me"}' \
--header 'Content-Type: application/json' | json_pp
{
  "title" : "Test Test",
  "owner" : "me",
  "created" : "2016-08-07T15:20:36.579Z",
  "status" : "New",
  "id" : 3
}

```

Most of the code is self-explanatory. We grabbed the new issue to be added from the request body (which was already parsed by `body-parser`). We then added some new fields like `id` and `created`, and set some fields to their defaults if not supplied.

The last statement in the handler is responsible for returning the new issue as a response to the request. An alternative is to return nothing and let the client query the entire issue list again after the addition. But we need *some* response, so we may as well return the new object that was created. You can let the client decide if they want to query for the entire list or use the return value as per convenience.

Note that we did not specify the method as `POST` in the `curl` command line. This is because `curl` automatically uses `POST` when the `--data` option is supplied.

EXERCISE: THE CREATE API

1. Test the API with a malformed JSON as the request body. What happens? Is this acceptable?
2. Change the Content Type header in the POST to say, `text/x-json`. What happens? Why? What if you wanted to handle both? Hint: Look at the `body-parser` documentation.
3. The API allows creation of an issue without a title or owner. You can also create an issue with an invalid status (not one of the enumerations you want). Is this acceptable?

Answers are available at the end of the chapter.

Using the List API

In this section, we will use the List API in the application front end and replace the in-memory list of issues. To use the APIs, we need to make asynchronous API calls, or Ajax calls. The popular library jQuery is an easy way to use the `$.ajax()` function, but including the entire jQuery library just for this purpose seems like overkill.

Fortunately, there are many libraries that provide this functionality. Better still, modern browsers support asynchronous calls natively via the Fetch API. It's my expectation that in the near future, all browsers will support this API and its full specification. It would be good to use `fetch()` with a polyfill for the browsers that don't support it yet, so that in the future you can get rid of the polyfill and use `fetch()` natively. The polyfill is available in a package called `whatwg-fetch`. We'll use it directly from a CDN and include it in `index.html` like this:

```
...
<script src=
  "https://cdnjs.cloudflare.com/ajax/libs/fetch/1.0.0/fetch.min.js">
</script>
...
```

■ **Note** The polyfill is required only for Safari or Internet Explorer. For other browsers such as the latest Chrome, Firefox, or Edge, you don't need to include the polyfill, as these browsers support `fetch()` natively. You may skip this step if you are testing on these browsers, and don't expect your users to use Safari or IE.

Next, let's replace the method `loadData()` in the `IssueList` component with an Ajax call using `fetch()`. This API is similar to `$.ajax()`, if you have used that jQuery function. It takes in the path of the URL to be fetched, and returns a promise with the response as the value. If you are not familiar with promises, please read up on them. Searching the Internet for "javascript promise" should give you a lot of resources. The one at MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) is a very concise and great resource. We will be using promises in later chapters as well.

We need to parse the response, for which we can use the `json()` method of the response itself. This too returns a promise with the value as the parsed data. The parsed data will reflect what we sent from the server, which includes the properties `_metadata` and `records`. We can use `records` to directly set the state, but before that, we need to do some data transformations for converting date strings to date objects. Also, since we don't use the `metadata` yet, we just print it using a console log statement. Listing 5-4 shows the new `loadData()` method.

Listing 5-4. App.jsx, IssueLlist: Loading Data Using fetch()

```

loadData() {
  fetch('/api/issues').then(response =>
    response.json()
  ).then(data => {
    console.log("Total count of records:", data._metadata.total_count);
    data.records.forEach(issue => {
      issue.created = new Date(issue.created);
      if (issue.completionDate)
        issue.completionDate = new Date(issue.completionDate);
    });
    this.setState({ issues: data.records });
  }).catch(err => {
    console.log(err);
  });
}
}

```

To convert dates, we introduced a `forEach` loop that does the conversion. We need a check for `completionDate` being defined since that field is optional. It's now a good time to get rid of the in-memory array of issues in `App.jsx`. The modifications (only deletions) are shown in Listing 5-5.

Listing 5-5. App.jsx: Removing the In-Memory Array

```

...
const issues = [
  {
    id: 1, status: 'Open', owner: 'Ravan',
    created: new Date('2016-08-15'), effort: 5, completionDate: undefined,
    title: 'Error in console when clicking Add',
  },
  {
    id: 2, status: 'Assigned', owner: 'Eddie',
    created: new Date('2016-08-16'), effort: 14, completionDate: new
    Date('2016-08-30'),
    title: 'Missing bottom border on panel',
  },
];
...

```

On testing this set of changes, you will find that the application behaves the same as before. Adding a new issue via the UI also works. Importantly, refreshing the browser continues to revert to the original list. That's because we have not yet integrated the Create API to save any newly created issue.

EXERCISE: USING THE LIST API

1. Instead of converting the dates into Date objects just after fetching the data and before setting state, you could have done it in the `IssueRow` stateless function. What are the pros and cons of doing that?

Answers are available at the end of the chapter.

Using the Create API

The next step is to modify the `createIssue()` method, by including a call to the Create API. Thus, instead of directly appending to the list of issues, we'll first send it to the server, and use the updated issue returned by the server to append to the list of issues.

The `fetch()` API for POST methods needs a little more information than just the URL path. This information is supplied as an options object in the second parameter. You need to include the method, the Content Type header, and finally the body, which is a JSON representation of the new issue.

On the return path, you know that the server is going to give you a JSON representation of the new issue created. You can either reload the entire list or use this new object to append to the existing list of issues in the current state. Let's use the new object, which is faster than a round trip to the server, for the new list. Parsing of the response is similar to what we did in the List API, using the `json()` method of the response object. As for the setting the new state, we need to create a new list of issues. We can do this by making a copy of the existing list from the current state, that is `this.state`. Listing 5-6 shows the modified `createIssue()` method.

Listing 5-6. Using the Create API to Create an Issue

```
createIssue(newIssue) {
  fetch('/api/issues', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(newIssue),
  }).then(response => response.json())
  ).then(updatedIssue => {
    updatedIssue.created = new Date(updatedIssue.created);
    if (updatedIssue.completionDate)
      updatedIssue.completionDate = new Date(updatedIssue.completionDate);
    const newIssues = this.state.issues.concat(updatedIssue);
    this.setState({ issues: newIssues });
  }).catch(err => {
    alert("Error in sending data to server: " + err.message);
  });
}
```

Let's take a closer look at the setting of the new state:

```
...
    const newIssues = this.state.issues.concat(updatedIssue);
    this.setState({ issues: newIssues });
...
```

Remember that the state is immutable, so you cannot make modifications to it. So, we used the `concat()` function of Array, which creates a copy of the original array, and therefore, is safe. We used `splice()` earlier, but this is more readable.

On testing, you can see that a browser refresh keeps any newly added issues. Of course, restarting the server will cause the list of issues to be reset to the original. To permanently save the issues, we need to persist it in a database, which we will deal with in the next chapter.

EXERCISE: USING THE CREATE API

1. Can you think of a situation where using the returned issue to append to the list is not appropriate?

Answers are available at the end of the chapter.

Error Handling

I discussed error handling in the Create API section as part of the exercises, but we did not implement it then. Let's do so now. We will not handle malformed JSON kind of errors, because they don't cause great harm. Also, handling malformed JSON is a bit complicated, because we need to write a own middleware for this. As for application-level validations, we'll handle missing required fields and incorrect list values.

One decision that we need to make is: how do we return errors back to the client? The success or failure of any REST API call is typically reflected in the HTTP status code. Some prefer returning a 400 `Bad Request` status code, but in my opinion, a 400 status indicates a malformed request, or something that does not conform to HTTP standards, or for syntactically incorrect requests. Application errors are not problems with the syntax of the input, so we'll instead use 422 `Unprocessable Entity` as the error status code for them. If and when you implement errors for malformed JSON, you could use the 400 error code.

The error message (a description of what went wrong) itself can be returned in the response body, again as a JSON string. We will return an object with a single property called `message` that holds a readable as the description. For programmatic interpretation of the errors, it may also be useful to include an error code, but we won't do that in the application because the API is only for internal consumption.

At the server, sending an error is simple; all you need to do is set the status using `res.status()` and send the error message as the response. Apart from this, the rest of the changes (as seen in Listing 5-7) are straightforward logic, which can be implemented in many different ways. Listing 5-7 shows a few new global variables, a validation function, and the new modified POST handler for creating an issue.

Listing 5-7. server.js: Adding Server-Side Validation

```

const validIssueStatus = {
  New: true,
  Open: true,
  Assigned: true,
  Fixed: true,
  Verified: true,
  Closed: true,
};

const issueFieldType = {
  id: 'required',
  status: 'required',
  owner: 'required',
  effort: 'optional',
  created: 'required',
  completionDate: 'optional',
  title: 'required',
};

function validateIssue(issue) {
  for (const field in issueFieldType) {
    const type = issueFieldType[field];
    if (!type) {
      delete issue[field];
    } else if (type === 'required' && !issue[field]) {
      return `${field} is required.`;
    }
  }

  if (!validIssueStatus[issue.status])
    return `${issue.status} is not a valid status.`;

  return null;
}

app.post('/api/issues', (req, res) => {
  const newIssue = req.body;
  newIssue.id = issues.length + 1;
  newIssue.created = new Date();
  if (!newIssue.status)
    newIssue.status = 'New';

  const err = validateIssue(newIssue)
  if (err) {
    res.status(422).json({ message: `Invalid request: ${err}` });
    return;
  }
});

```

```

    }
    issues.push(newIssue);

    res.json(newIssue);
  });

```

The first two global objects are a kind of schema definition to indicate what is a valid Issue object. The function `validateIssue` checks against this specification and returns an error if the validation fails. Note that we also *deleted* any fields that do not belong, effectively ignoring such errors rather than reporting them back to the client. It's a matter of preference whether you want to report or ignore them; you could handle them either way.

In the handler, we called the validation function and in case of an error, we set the status to 422 and sent back an object with an appropriate message. The single line containing `res.status()` and `json()` is a shortcut for `res.status()` followed by a `res.json()` as two different statements.

At the client, we need to modify the code to detect a non-success HTTP status code. Note that the Fetch API does not throw an error for failure HTTP status codes, so relying on the catch section is not going to work. We must check the response's property `response.ok`, and if it is not OK, we need to show an error. Listing 5-8 shows the client-side code, the complete `createIssue()` method.

Listing 5-8. `App.jsx`, `IssueList`'s `createIssue` method: Client-Side Error Handling

```

createIssue(newIssue) {
  fetch('/api/issues', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(newIssue),
  }).then(response => {
    if (response.ok) {
      response.json().then(updatedIssue => {
        updatedIssue.created = new Date(updatedIssue.created);
        if (updatedIssue.completionDate)
          updatedIssue.completionDate = new Date(updatedIssue.completionDate);
        const newIssues = this.state.issues.concat(updatedIssue);
        this.setState({ issues: newIssues });
      });
    } else {
      response.json().then(error => {
        alert("Failed to add issue: " + error.message);
      });
    }
  }).catch(err => {
    alert("Error in sending data to server: " + err.message);
  });
}

```


We moved the entire success path processing to within an `if (response.ok)` check. The `else` part just shows an error message as an alert. In both cases, we had to parse the response body. In the success case, the response body represents the updated Issue object, and in the error case, the response body contains the error object with the message.

Another way to handle this could be to throw an error if `response.ok` indicates a failure, so that you can handle all this in a single catch block. But this complicates things, especially since you want to show the error message that's within the response body, as parsing the response is another asynchronous call. If you did not care about the response message, you could have just thrown an error in the first step, where you only pass on the status text message instead of the application error message. Like this:

```
...
}).then(response => {
  if (!response.ok) {
    throw new Error(response.statusText);
  } else {
    return response.json();
  }
})
...
```

To simulate the errors and test the new code, create an issue via the UI, skipping the mandatory field. An error message should show. For an invalid status, you need to test via a direct API call, or hardcode an invalid status to send to the server, because there's no UI for setting the status yet.

EXERCISE: ERROR HANDLING

1. Do you think the validations that you implemented are enough? If not, what other validations can you think of?
2. Looks like errors due to HTTP Status error codes like 503 or 422 are not thrown as errors in `fetch()`. So, which types of errors are thrown? Can you simulate one such error? Hint: Refer to the fetch API documentation.
3. Can you think of other ways of sending the error message?

Summary

APIs are all about intuitiveness and predictability. REST gives you a framework for how to think about and structure APIs. We implemented the C and R of CRUD in this chapter, to get a whiff of what REST looks like and how to implement and consume APIs. We'll explore U and D much later, when we start implementing the features that require them.

In the meanwhile, it's a good time to find out how to persist and read that persisted data. In the next chapter, you'll learn a bit about how to do so with MongoDB.

Answers to Exercises

Exercise: The List API

1. Express uses ETags to identify the version of a resource. This is a kind of hash, which will change if the resource changes. If you look at the request headers, you will see that Chrome is sending an ETag header in the request. This means that Chrome already has a version of the resource, and if it matches the version that the server has, you are OK to use the cached version.

No, it will not be a problem if the issues list changes; see the next answer.

2. Express does not cache the response, as you may have initially suspected. Instead, the response is always generated (as seen by the log statement). It calculates the hash again, and if it matches what the browser has, it sends a 304 status code instead of the response. This is a far more reliable way of avoiding unnecessary network traffic compared to conventional caching mechanisms such as Cache-Control, Max-Age, and If-Modified-Since headers.

But note that this does not avoid processing, since the response is generated anyway. This optimization only saves some network traffic.

3. The pipe to `json_pp` (stands for JSON pretty-print) formats and pretty-prints the JSON output, which otherwise would have been in a single line. To always return a formatted JSON, you could replace `res.json` with `res.send` and construct the JSON string yourself using `JSON.stringify()`. This would allow you to pass parameters that give you a formatted string. Or, you could set the application settings `json spaces` using `app.set()` so that *all* responses are pretty printed.
4. Link headers are useful if all you are doing is adding Next/Previous links in the UI. It is more standard than a wrapper, so it needs less documentation. On the other hand, a wrapper is easier to parse for the client, and allows for programmatically generating related actions. Also, the wrapper can include other useful information such as count, which can be used to show a total count of results in the UI.
5. The difference between the array and the JSON string is in the dates. Since JSON allows only primitive types, dates are encoded as an ISO 8601 string in the JSON. If the client needs to manipulate or do calculations on dates, it must convert the string to a Date object.

Exercise: Create API

1. On testing with a malformed JSON, an error and stack trace are printed in the server's console. Also, the output of curl is an HTML response of the same stack trace. This means that somewhere along the response chain, the error was caught and sent as an HTML response to the request.

Ideally, you should catch such errors and return a JSON indicating an error code, and hide the stack trace. This involves installing your own error handling middleware in Express, which is an advanced topic. Since our API is internal, such errors will be caused by our own programming errors on the client, so, for the moment, we'll live with this.

2. The middleware `bodyParser` looks at the `Content-Type` header and determines if and how the body can be parsed. For JSON, the default content type is `application/json`. In the absence of this header, `bodyParser` does not parse the request body, and the variable `req.body` ends up as an empty object. Thus, a new issue is created without a title or owner. If you want to handle both content types, you need to instantiate the JSON parser by supplying an option. This option is an array that has all the content types that you want to accept as JSON and trigger the JSON parser.
3. I said that it's OK to not handle malformed JSON errors because the API is for internal consumption, and they are programming errors. Getting invalid objects is also very likely a programming error, but there is a difference. No harm is done if malformed JSON errors are not handled; the request just fails. But without validation, you may end up with invalid objects in the database, and that can cause havoc later. The application *must* handle validation errors to protect data integrity. I'll address this validation later in the book.

Exercise: Using the List API

1. Doing the date conversions from strings to actual dates at the point of usage is also a valid strategy. In this case, you think of the *model* as always containing strings. This will work well when you need to do only formatting conversions.

On the other hand, if you expect to manipulate dates (e.g., to show the number of days from today left for completion, or the age of the issue from the date created), it's more convenient to convert them to dates right after the fetch. Otherwise, you'll be doing the same conversion multiple times.

Exercise: Using the Create API

1. We are just appending the newly created issue. If the order is important (for example, the list is sorted on some field), appending may ruin the order. In this case, it is better to refetch the entire list in the correct order from the server.

Exercise: Error Handling

1. The validations are not exhaustive. Ideally, you need to validate the type of every field, for example, effort has to be a number and dates have to be proper dates. Further, there could also be validations such as completion date that cannot be lesser than the created or current date. You could define a more elaborate schema, or use something like JSON Schema, or even use mongoose rather than invent your own.

We will not implement exhaustive validations as part of this book, because that exercise will not give new insights into the MERN stack. But remember that in a real-life application, having such validations is essential.

You should also introduce front-end validations for more instant user-friendly error messages. I'll leave them for later chapters, as you have explored how to return errors, and that was the objective of this section.

2. Only network-related errors are thrown by fetch, such as an unreachable server. Try shutting down the server before creating an issue to simulate a network error.
3. A popular alternative is to always wrap the result in an object that contains the success status, error message, and the result. This means that the status checking will have to be inside the second stage, post response parsing to JSON. This way, clients don't have to look at the HTTP Status code to find out about application-level error messages.

CHAPTER 6



Using MongoDB

In this chapter, you'll learn about MongoDB. The goal is to get rid of the in-memory list of issues and start using a MongoDB database to add and retrieve issues. To achieve this, we will need to install MongoDB, learn about how to add to and list records from the database directly, and then modify the server code to use MongoDB to persist the list of issues.

MongoDB Basics

This section is an introductory section, where we will not be modifying the application. We'll look at the following core concepts in this section: MongoDB, documents, and collections. Then, we'll install MongoDB and explore these concepts with examples using the `mongo` shell while reading and writing to the database.

Documents

MongoDB is a *document* database, which means that the equivalent of a record is a document, or an object. In a relational database, one has to use rows and columns, whereas in a document database, an entire object can be written as a document.

For simple objects, this may seem no different from a relational database. But let's say you have objects with nested objects (called embedded documents) and arrays. Now, when using a relational database, you typically need multiple tables. For example, an Invoice object may be stored in a combination of an `invoice` table and `invoice_lines` table in a relational database. In MongoDB, you store the entire Invoice object as one document.

A document is a data structure composed of field and value pairs. The values of fields may include other documents, arrays, and arrays of documents. MongoDB documents are similar to JSON objects, so it is easy to think of them as JavaScript objects. Compared to a JSON object, a MongoDB document has support not only for the primitive data types boolean, numbers, and strings, but also other common data types such as dates, timestamps, regular expressions, and binary data. You can even store JavaScript functions as document fields.

You will see examples of documents when you explore the `mongo` shell later in this chapter.

Collections

A collection is like a table in a relational database. It is a set of documents, and you access each document via the collection. Just like in a relational database, you can have a primary key and indexes on the collection. But there are a few differences.

A primary key is mandated in MongoDB, and it has the reserved field name `_id`. Even if you don't supply an `_id` field when creating a document, MongoDB creates this field and auto-generates a unique key for every document. More often than not, the auto-generation is used as is, since it is convenient and guaranteed to produce unique keys even when multiple clients are writing to the database simultaneously. MongoDB uses a special data type called the `ObjectId` for the primary key.

The `_id` field is automatically indexed. Apart from this, indexes can be created on other fields, and this includes fields within embedded documents and array fields. Indexes are used to efficiently access a subset of documents in a collection.

Unlike a relational database, MongoDB does not require you to define a schema for a collection. The only requirement is that all documents in a collection must have a unique `_id`, but the actual documents may have completely different fields. In practice, though, all documents in a collection do have the same fields. Although a flexible schema may seem very convenient for schema changes during the initial stages of an application, this can cause problems if you don't have some kind of schema checking in the application code.

Query Language

Unlike the universal English-like SQL in a relational database, the MongoDB query language is made up of *methods* to achieve various operations. The main methods for read and write operations are the CRUD methods. Other methods include aggregation, text search, and geospatial queries.

All methods operate on a collection and take parameters as JavaScript objects that specify the details of the operation. Each method has its own specification. For example, to insert a document, the only parameter you need is the document itself. For querying, the parameters are a match specification and a list of fields to return.

Unlike relational databases, there is no method that can operate on multiple collections at once. All methods operate on only one collection at a time. If there is a need to combine the result of multiple collections, each collection has to be queried separately and manipulated by the client. In a relational database, you can use *joins* to combine tables using fields that are common to the tables, so that the result includes the contents of both tables. You can't do this in MongoDB and many other NoSQL databases. This lets NoSQL databases scale by using *shards*, or multiple servers to distribute documents part of the same collection.

Also, unlike relational databases, MongoDB encourages *denormalization*, that is, storing related parts of a document as embedded subdocuments rather than as separate collections (tables) in a relational database. Take an example of people (name, gender, etc.) and their contact information (primary address, secondary address etc.). In a relational database, you would have separate tables for People and Contacts, then join the two tables when you need all of the information together. In MongoDB, on the other hand, you store the list of contacts *within* the same People document, thus avoiding a join.

Installation

MongoDB can be installed easily on OS X, Windows, and most distributions based on Linux. The installation instructions are different for each operating system and have a few variations depending on the OS flavor as well. Please install MongoDB by following the instructions at the MongoDB website (<https://docs.mongodb.com/manual/installation/>) or search for “mongodb installation” in your search engine). Choose version 3.2 or higher, preferably the latest, as some of the examples use features introduced only in version 3.2. Most installation options let you install the server, the shell, and tools all in one. Check that this is the case; if not, you may have to install them separately.

After installation, ensure that you have started MongoDB server (the name of the daemon or service is `mongod`), if it is not already started by the installation process. Test the installation by running the mongo shell like this:

```
$ mongo
```

On a Windows system, you may need to add `.exe` to the command. The command may require a path depending on your installation method. If the shell starts successfully, it will also connect to the local MongoDB server instance. You should see the version of MongoDB printed on the console, the database it is connecting to (the default is `test`), and a command prompt, like this:

```
MongoDB shell version: 3.2.4
connecting to: test
>
```

If, instead, you see an error message, revisit the installation and server starting procedure.

The mongo Shell

The mongo shell is an interactive JavaScript shell, very much like the Node.js shell. In this interactive shell, there are a few non-JavaScript conveniences apart from the full power of JavaScript. In this section, you’ll look at the basic operations that are possible via the shell, those that are most commonly used. For a full reference of all you can do with the shell, take a look at the MongoDB documentation.

By default, the shell connects to a database called `test`. At any time, to know the current database, use the special command `db` like this:

```
> db
```

It should print the current database, which is by default, `test`. To connect to another database, say a `playground` database, do this:

```
> use playground
```

Note that a database does not have to exist to connect to it. The first document creation will initiate the database creation if it doesn't exist. The same applies to collections: the first creation of a document in a collection creates the collection. You can see the proof of this by listing the databases and collections in the current database:

```
> show databases
> show collections
```

You will see that `playground` is not listed in the databases, and the collections list is empty. Let's create an `employees` collection by inserting a document. To insert a document, you use the `insert()` method on the collection, which is referred to by a property with the name of the database, of the special variable `db`:

```
> db.employees.insert({name: {first: 'John', last: 'Doe'}, age: 44});
```

Now, if you list the databases and collections, you will find both `playground` and `employees` listed. Let's also make sure that the first employee record has been created. To list the contents of a collection, you need to use the `find()` method of the collection:

```
> db.employees.find();
{ "_id" : ObjectId("57b1caea3475bb1784747ccb"), "name" : { "first" : "John",
"last" : "Doe" }, "age" : 44 }
```

You can see that `_id` was automatically generated and assigned. If you wanted a prettier, indented listing of employees, you should use the `pretty()` method on the results of `find()` like this:

```
> db.employees.find().pretty()
{
  "_id" : ObjectId("57b1caea3475bb1784747ccb"),
  "name" : {
    "first" : "John",
    "last" : "Doe"
  },
  "age" : 44
}
```

Now, insert a few more documents with different names and ages. Add a middle name for someone, like this:

```
> db.employees.insert({name: {first: 'John', middle: 'H', last: 'Doe'}, ←
age: 22});
```


This is what a flexible schema lets you do: you can enhance the schema whenever you discover a new data point that you need to capture, without having to explicitly modify the schema. In this case, it is implicit that any `employee` document where the middle field under `name` is missing indicates an employee without a middle name. If, on the other hand, you added a field that didn't have an implicit meaning when absent, you'd either have to handle the absence in the code, or run a migration to modify all documents and add the field with a default value.

Note that MongoDB automatically generated the primary key for each of the documents, which is displayed as `ObjectId("57b1caea3475bb1784747ccb")` in the `find()` output. Just to reiterate, the `ObjectId` is a special data type, which is why it is displayed like that. You can convert an `ObjectId` to and from strings, which you'll see a little later.

The `insert()` method can take in an array when inserting multiple documents together. The variations `insertOne()` and `insertMany()` were introduced in version 3.2 to make it more explicit whether the parameter is a single document or an array.

To retrieve only some documents that you're interested in, you need to supply a filter to `find()`. The filter specification is an object where the property name is the field that you want to filter on, and the value is its value that you want to match. Say you want to find all employees aged 44; this is what you would do:

```
> db.employees.find({age: 44});
```

The output would be similar to the output of `find()` without filters as described previously. The filter is actually a shortcut for `age: { $eq: 44 }`, where `$eq` is the operator. Other operators for comparison are available, such as `$gt` for greater than. If you need to compare and match fields within embedded documents, you can refer to field using the dot notation (which will require you to specify quotes around the field name). If there are multiple field specifications, all of them have to match.

A second parameter can be passed to restrict the fields that are returned. This is called the *projection*. The format of this specification is an object with one or more field names as the key and the value as 0 or 1, to indicate exclusion or inclusion. Unfortunately, you cannot combine 0s and 1s: you can only start with nothing and include all the fields using 1s, or start with everything and exclude fields using 0s. The `_id` field is an exception; it is always included unless you specify a 0. The following will find employees whose first name is John, aged 44 or more, and print only their first names and ages:

```
> db.employees.find({'name.first': 'John', age: {$gte: 44}}, {
{'name.first': 1, age: 1})
```

The method `findOne()` is a variation that returns a single document rather than an cursor that can be iterated over.

In order to update a document, you first need to specify the filter that matches the document to update, and then specify the modifications. The filter specification is the same as for a read operation. Typically, the filter is the ID of the document so that you are sure you update one and only one document. You can replace the entire document by supplying the document as the second parameter. If you want to only change a few fields, you do it by using the `$set` operator, like this:

```
> db.employees.update({_id: ObjectId("57b1caea3475bb1784747ccb")}, {
  $set: {age: 44}})
```

You identify the document to modify using its primary key, `_id`. In order to generate the special data type `ObjectId`, you need to use the mongo shell's built-in function `ObjectId()` and pass it a string representation of the ID. The update results in a message like this (the output may vary depending on the version of MongoDB you have installed):

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

The `update()` method can take more options, one of which is the `upsert` option. When this is set to true, MongoDB will find the document based on the filter, but if it doesn't find one, it will *create* one based on the document supplied. Of course, this isn't useful if you are using the object ID to identify the object. It's useful if the key to search for is a unique key (like the employee number). Also, the second parameter in this case must be the entire document; it can't be a patch specification using `$set`.

The variations `updateMany()` and `updateOne()` were introduced in version 3.2 to make it explicit as to the intention of the update. Use of these variations is recommended over the plain `update()` since even if the filter matches multiple or single documents, the update will affect one or many documents depending on which method was called.

To delete a document, use the `remove` method with a filter, just as in the `find` method:

```
> db.employees.remove({"_id" : ObjectId("57b1caea3475bb1784747ccb")})
```

```
WriteResult({ "nRemoved" : 1 })
```

If you think a field is often used to filter the list, you should create an index on the field to make the search more efficient. Failing this, MongoDB searches the entire database for a match. To create an index on the `age` field, do this:

```
> db.employees.createIndex({age: 1})
```

Shell Scripting

A mongo shell script is a regular JavaScript program, with all the collection methods available as built-ins. One difference from the interactive shell is that you don't have the convenience commands such as `use` and the default global variable `db`. You must initialize them within the shell script programmatically, like this:

```
var db = new Mongo().getDB("playground");
```

Add a few more statements in a script file, the same that you typed in the mongo shell for inserting and reading documents. To execute the script, supply it as a parameter to the mongo shell like this (if you have saved the file as `test.mongo.js`):

```
$ mongo test.mongo.js
```

EXERCISE: MONGO SHELL

1. Does the mongo shell support ES2015 features? Hint: Try out a few ES2015 features in the mongo shell.
2. Write a simple statement to retrieve all employees who have middle names. Hint: Look up the MongoDB documentation for query operators.
3. Is the filter specification a JSON? Hint: Think about date objects and quotes around field names.
4. Say an employee's middle name was set mistakenly, and you need to remove it. Write a statement to do this. Hint: Look up the MongoDB documentation for update operators.
5. During index creation, what did the 1 indicate? What other valid values are allowed? Hint: Look up the MongoDB indexes documentation.

Answers are available at the end of the chapter.

Schema Initialization

Since MongoDB does not enforce a schema, there is really no such thing as a schema initialization as you may do in other databases. The only thing you really want to do is create indexes that will prove useful for often used filters in the application. While we're at it, let's also initialize the database with some sample records to ease your testing.

Let's create a mongo shell script called `init.mongo.js` and place it in a new scripts directory in the project directory. We need to write the initialization statements, which should include setting up the `db` variable; removing all existing issues, if any; inserting a few sample records; and creating indexes. Listing 6-1 shows the contents of the script.

Listing 6-1. `init.mongo.js`: DB Initialization Script

```
db = new Mongo().getDB('issuetracker');

db.issues.remove({});

db.issues.insert([
  {
    status: 'Open', owner: 'Ravan',
    created: new Date('2016-08-15'), effort: 5, ←
    completionDate: undefined,
    title: 'Error in console when clicking Add',
  },
  {
    status: 'Assigned', owner: 'Eddie',
    created: new Date('2016-08-16'), effort: 14, ←
    completionDate: new Date('2016-08-30'),
    title: 'Missing bottom border on panel',
  },
]);

db.issues.createIndex({ status: 1 });
db.issues.createIndex({ owner: 1 });
db.issues.createIndex({ created: 1 });
```

The only notable thing we did differently from what I discussed in the previous section is use an array to insert multiple records at once in the `insert` method. The array of documents is a copy of what we had in `server.js`, except that we removed the field called `id`. Run this script from the command line like this:

```
$ mongo scripts/init.mongo.js
```

It should run without any errors. To check the effect of the script, open up the mongo shell, list all documents using `find()`, and list all indexes using `getIndexes()`. The `find()` should return the two documents that you inserted, with auto-generated ObjectIDs in the `_id` field. And, `getIndexes()` should list four indexes: the three that we created on `status`, `owner`, and `created`, and the auto-created index on `_id`.

EXERCISE: SCHEMA INITIALIZATION

1. We let MongoDB generate the `_id` primary key. What are the pros and cons of generating the primary key ourselves instead?
2. Are there any other indexes that may be useful? Hint: What if you needed a search input field in the application?

Answers are available at the end of the chapter.

MongoDB Node.js Driver

This is the Node.js driver that lets you connect and interact with the MongoDB server. It is, as you probably guessed, an npm module. It provides methods very similar to what you saw in the mongo shell, but not exactly the same.

Another option is to use the mongoose client, which is an object-document-mapper as discussed in Chapter 1. But since I want you to explore the nuts and bolts of how the native driver works, which works similar to the shell commands, let's use the native driver itself.

To start, let's install the driver:

```
$ npm install mongodb --save
```

To connect to the database from a Node.js program, you call the `connect` method on the `MongoClient` object provided by the module. The `mongodb` module exports many functions and objects, of which `MongoClient` gives you the ability to act as a client, mainly, the `connect` method. The parameter to the `connect` function is a URL-like string starting with `mongodb://` followed by the server name, and then the database name separated by a `/`. You can optionally include a port after the server name, separated by a `:`, but if you're using the default port, this can be skipped.

Once you acquire a connection, to get a handle to any collection, you need to call its `collection()` method. To this method, you supply the name of the collection as the parameter to indicate which collection. Then, the CRUD operation methods can be called on the handle to the collection. For example, to connect to a database called `playground` and fetch some documents from a collection called `employees`, you do the following:

```
...
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost/playground', function(err, db) {
  db.collection('employees').find().toArray(function(err, docs) {
    console.log('Result of find:', docs);
    db.close();
  });
});
...
```

In the above, a `find()` returns a cursor which you could iterate over. Calling `toArray()` on the cursor runs through all the documents and makes an array out of them. It calls the callback when the array is ready to be processed, passing the array as a parameter to the callback.

To insert a document, you need to use the `insertOne()` method on the collection, and pass it one parameter: the object to be inserted. The result of an insert contains various things, one of which is the new `_id` that has been generated, in a property called `insertedId`.

Note that all calls to the driver are asynchronous calls, which means that you don't get the result of the call as a return value to the function call. In the above example, you supplied a callback to each of the MongoDB driver methods. When the result is ready, these callbacks will be called. The MongoDB driver documentation gives you three different paradigms for dealing with this asynchronous nature: the callbacks paradigm, one using promises, and another using the `co` module and generator functions. Let's explore these three options, and one more option using the `async` module, which is not mentioned in the driver documentation.

Let's do all this in a JavaScript program called `trymongo.js` so that you have a ready test if and when required. Let's initialize this file with a test wrapper that we will use to exercise each of the paradigms. We'll use a command line argument to the program to call a different function for each of the paradigms. Command line arguments to Node.js programs are available in the array `process.argv`. The first two elements in the array are `node` and the name of the program. All user arguments follow after these two. The initial contents of the file are shown in Listing 6-2.

Listing 6-2. `trymongo.js`: Initial Contents, Command Line Arguments Handling

```
'use strict';
const MongoClient = require('mongodb');

function usage() {
  console.log('Usage:');
  console.log('node', __filename, '<option>');
  console.log('Where option is one of:');
  console.log('  callbacks    Use the callbacks paradigm');
  console.log('  promises     Use the Promises paradigm');
  console.log('  generator    Use the Generator paradigm');
  console.log('  async       Use the async module');
}

if (process.argv.length < 3) {
  console.log("Incorrect number of arguments");
  usage();
} else {
  if (process.argv[2] === 'callbacks') {
    testWithCallbacks();
  } else if (process.argv[2] === 'promises') {
    testWithPromises();
  }
}
```

```

    } else if (process.argv[2] === 'generator') {
      testWithGenerator();
    } else if (process.argv[2] === 'async') {
      testWithAsync();
    } else {
      console.log("Invalid option:", process.argv[2]);
      usage();
    }
  }
}

```

We gave a name and associated a function with each of the paradigms: Callbacks, Promises, Generator, and Async. We will fill in the functions in the following subsections.

Callbacks

The conventional and oldest way to deal with asynchronous calls is to provide a callback to handle the result of the operation. As seen in the above example, it is minimal JavaScript. You pass a callback to the method, with the first parameter of the callback expecting any errors, and the second (or more) parameters expecting the result of the operation.

Callbacks are easy to understand and reason about. They work in conventional ES5 JavaScript as well. They have very little constructs that you need to learn. The callback paradigm is not particular to the MongoDB driver. There are many APIs, including the core Node.js library, that follow this paradigm.

Listing 6-3 shows how to get a database connection, use the connection to insert into a collection, and then use the result of the insert operation to retrieve the object. As you can see, one problem with this paradigm is that it can get deeply nested and complicated, depending on the depth of the chain: the result of one operation being passed to the next.

Listing 6-3. trymongo.js, testWithCallbacks: Callbacks Paradigm

```

function testWithCallbacks() {
  MongoClient.connect('mongodb://localhost/playground', function(err, db) {
    db.collection('employees').insertOne({id: 1, name: 'A. Callback'},
    function(err, result) {
      console.log("Result of insert:", result.insertedId);
      db.collection('employees').find({id: 1}).toArray(function(err, docs) {
        console.log('Result of find:', docs);
        db.close();
      });
    });
  });
}

```

It can get even more deeply nested when you also have to handle errors (or other conditions: you'll soon find yourself writing the same set of statements multiple times (imagine `db.close()` in every error condition). This is often referred to as *callback hell*. The only remedy if you want to stick to the callback paradigm is to split each small piece of code into its own function and pass that function as a parameter to a call, chaining the callback along.

Promises

Using ES2015 promises, the nesting can be avoided, and the chaining can become seemingly sequential. The above can be written as shown in Listing 6-4, using the promises paradigm.

Listing 6-4. `trymongo.js`, `testWithPromises`: Promises Paradigm

```
function testWithPromises() {
  let db;
  MongoClient.connect('mongodb://localhost/playground').then(connection => {
    db = connection;
    return db.collection('employees')...insertOne({id: 1, name: 'B. Promises'});

  }).then(result => {
    console.log("Result of insert:", result.insertedId);
    return db.collection('employees').find({id: 1}).toArray();

  }).then(docs => {
    console.log('Result of find:', docs);
    db.close();

  }).catch(err => {
    console.log('ERROR', err);
  });
}
```

The result of every call is a promise, on to which you attach a `then`, which returns another promise, and so on. Finally, there is a `catch` block that consumes all errors. Assuming all calls throw errors, you'll find that error handling isn't needed in each individual block: just one final `catch()` for errors at any stage is enough.

Generator and `co` Module

ES2015 introduces generator functions, which can be exited temporarily and called again. The temporary exits are done using the `yield` statement. Between multiple calls, the function retains the execution state. These functions are declared using an asterisk after the function keyword, like `function*()`.

A module called `co` takes advantage of ES2015 generators and promises to make asynchronous calls look sequential. It achieves this by asking you to sequence the asynchronous calls within one function. Then, the `co` module makes multiple calls to this function, where each asynchronous step temporarily exits the function.

To try this paradigm, let's first install the `co` module:

```
$ npm install co
```

We did not use `--save` because we will not be using this module other than for trying out this paradigm. The previous example can be rewritten using the `co` module as shown in Listing 6-5.

Listing 6-5. `trymongo.js`, `testWithGenerator`: Generator Paradigm with `co` Module

```
function testWithGenerator() {
  const co = require('co');
  co(function*() {
    const db = yield MongoClient.connect('mongodb://localhost/playground');

    const result = yield db.collection('employees')...insertOne({id: 1, ←
      name: 'C. Generator'});
    console.log('Result of insert:', result.insertedId);

    const docs = yield db.collection('employees').find({id: 1}).toArray();
    console.log('Result of find:', docs);

    db.close();
  }).catch(err => {
    console.log('ERROR', err);
  });
}
```

As you can see, every asynchronous call was preceded by the keyword `yield`. This causes a *temporary* return from the function, after which the function can be resumed where it left off, if called again. The `co` module does the repeated calling, which is why we needed to wrap the function around `co()`.

The async Module

Yet another way to manage callbacks is by using the `async` module, though this method is not mentioned in the MongoDB driver documentation. To start, let's install the `async` module (again without `--save` because we will not be using this paradigm in the application, only trying it out in this section.)

```
$ npm install async
```

Apart from many other useful utilities for managing asynchronous calls, this module provides a method called *waterfall*, which lets you *pipe* the result of one asynchronous call to another. This method takes an array of functions to run. Each function is passed the results of the previous function, and a callback (which takes an error and results as its parameters). Each function in the array must call this callback when it is done. The results are passed through as a *waterfall* from one function to the next, that is, the outputs of one are passed to the next.

Since all the driver methods follow the same callback convention of error and results, it's easy to pass the callbacks through the waterfall. Listing 6-6 demonstrates this.

Listing 6-6. trymongo.js, testWithAsync: Async Paradigm

```
function testWithAsync() {
  const async = require('async');
  let db;
  async.waterfall([
    next => {
      MongoClient.connect('mongodb://localhost/playground', next);
    },
    (connection, next) => {
      db = connection;
      db.collection('employees').insertOne({id: 1, name: 'D. Async'}, next);
    },
    (insertResult, next) => {
      console.log('Insert result:', insertResult.insertedId);
      db.collection('employees').find({id: 1}).toArray(next);
    },
    (docs, next) => {
      console.log('Result of find:', docs);
      db.close();
      next(null, 'All done');
    }
  ], (err, result) => {
    if (err)
      console.log('ERROR', err);
    else
      console.log(result);
  });
}
```

Only in the last function did we explicitly call the callback with a null error and a string as the final result. In all of the other function calls, we just passed the callback through to the MongoDB driver method. The driver function calls the callback when the results are available. We also named the callback *next*, just to be clear that it will be the next function in the array that will be called when done.

Choosing any one of the paradigms is a matter of taste and familiarity. All of them are valid choices; it is really up to you as to which one you pick. For the purpose of the Issue Tracker application, I am going to pick the promises paradigm, mainly because it does not depend on any other module. Also, it lets you perform parallel tasks if you choose to do so for certain operations.

Reading from MongoDB

Let's first modify the List API to read from the database instead of the in-memory list of issues on your server. Since we've initialized the database with a set of initial issues, you should be able to test this easily.

To start, we have to include the MongoDB driver that we have already installed, in `server.js`. Next, we need to connect to the MongoDB server and keep the connection open for future calls to the database. We do this rather than acquire a connection in every request because creating a connection takes a bit of time. Also, the MongoDB driver keeps a pool of connections automatically and reuses them if you reuse the connection handle. So, we'll just save the connection in a global variable called `db` for later use.

We'll also start the Express server only once we get the connection. Let's call the database `issuetracker`. Listing 6-7 shows the connection acquisition and modified server startup section in `server.js`.

Listing 6-7. `server.js`: Modified Initialization Sequence with MongoDB connection

```
...
const MongoClient = require('mongodb').MongoClient;
...
let db;
MongoClient.connect('mongodb://localhost/issuetracker').then(connection => {
  db = connection;
  app.listen(3000, () => {
    console.log('App started on port 3000');
  });
}).catch(error => {
  console.log('ERROR:', error);
});
...
```

Now, we can modify the endpoint handler for `/api/issues` to read from the database. All we need to do is call a `find()` on the `issues` collection, convert it to an array, and return the documents returned by this call. Listing 6-8 shows the modified endpoint handler.

Listing 6-8. server.js: List API Modified to Use MongoDB

```
app.get('/api/issues', (req, res) => {
  db.collection('issues').find().toArray().then(issues => {
    const metadata = { total_count: issues.length };
    res.json({ _metadata: metadata, records: issues })
  }).catch(error => {
    console.log(error);
    res.status(500).json({ message: `Internal Server Error: ${error}` });
  });
});
```

■ **Caution** Never skip the catch block when using promises. If you do so, any runtime error within any of the blocks will not be caught and will be silently ignored. If you don't have a catch block, you may fail to “catch” errors in your code, even those such as a typo in one of the variable names.

Since we have changed the field name `id` to `_id`, the front-end code referring to `id` also needs to change. There are two places in `App.jsx` where this has to be done. Listing 6-9 shows the changed lines, with the change highlighted in bold.

Listing 6-9. App.jsx: Front-End Changes, Replacing `id` with `_id`

```
...
  <td>{props.issue._id}_id} issue={issue} />)
...

```

Finally, now that even the List API can return a non-successful HTTP Status code, let's make sure that we handle that in the front end, just like we did for the Create API, by looking at `response.ok`. The modified method `loadData()` is shown in Listing 6-10.

Listing 6-10. App.jsx, IssueList `loadData()`: Error Handling

```
loadData() {
  fetch('/api/issues').then(response => {
    if (response.ok) {
      response.json().then(data => {
        console.log("Total count of records:", data._metadata.total_count);
        data.records.forEach(issue => {
          issue.created = new Date(issue.created);
          if (issue.completionDate)
            issue.completionDate = new Date(issue.completionDate);
        });
      });
    }
  });
}
```

```

        this.setState({ issues: data.records });
    });
    } else {
        response.json().then(error => {
            alert("Failed to fetch issues:" + error.message)
        });
    }
}).catch(err => {
    alert("Error in fetching data from server:", err);
});
}

```

Now, the changes can be tested. Refresh the browser, and the two issues that we initialized using the mongo shell script should be displayed. The only change is that the ID is now a long string instead of what used to be 1 and 2. Of course, adding a new issue won't work, which we'll deal with in the next section.

EXERCISE: READING FROM MONGODB

1. We are saving the connection in a global variable. What happens when the connection is lost? Restart MongoDB server to see what happens. Does the connection still work?
2. Shut down the MongoDB server, refresh the browser, and then start the server and see what happens. Try the sequence again; this time wait for a much longer time before starting the server. What happens? Hint: Look up the documentation of MongoDB driver, look for the tutorial on connection failures.
3. We used `toArray()` to convert the entire list of issues into an array. What if the list is too big, say, a million documents? How would you deal with this?

Answers are available at the end of the chapter.

Writing to MongoDB

You saw how to write to MongoDB using the `insert` method on a collection. We'll use that method to create a new record in the Create API. We will need to, as in the MongoDB driver trial, read back the object that was just created and return it as the result of the API call.

The `find()` call to read back the object is guaranteed to return a single document since you are using the `_id` as the filter criterion. In such a case, it's recommended that you do not use `toArray()`; instead, you use the `next()` method on the cursor returned by `find()`. Another alternative is to use the `findOne()` method. This is just to prevent a large array being created in memory inadvertently and also to make it clear that you only want *one* document and not an array of one document.

The modified Create API is shown in Listing 6-11.

Listing 6-11. server.js: Modified Create API That Writes to MongoDB

```
app.post('/api/issues', (req, res) => {
  const newIssue = req.body;
  newIssue.id = issues.length + 1;
  newIssue.created = new Date();
  if (!newIssue.status)
    newIssue.status = 'New';

  const err = validateIssue(newIssue)
  if (err) {
    res.status(422).json({ message: `Invalid request: ${err}` });
    return;
  }

  db.collection('issues').insertOne(newIssue).then(result =>
    db.collection('issues').find({ _id: result.insertedId }).limit(1).next()
  ).then(newIssue => {
    res.json(newIssue);
  }).catch(error => {
    console.log(error);
    res.status(500).json({ message: `Internal Server Error: ${error}` });
  });
});
```

Since MongoDB will now generate the ID for each issue, we removed the line where we assigned the field `id` based on the array length. We also need to remove the validation for the field `id`. All we have to do is remove the corresponding property in the `issueFieldType` array. Finally, we can also get rid of the in-memory array, now that we are saving the list in a database. These two changes are shown in Listing 6-12.

Listing 6-12. server.js: Clean up Validations and Remove In-Memory Array

```
...
const issueFieldType = {
  id: 'required',
  status: 'required',
...
const issues = [
  {
    id: 1, status: 'Open', owner: 'Ravan',
    created: new Date('2016-08-15'), effort: 5, completionDate: undefined,
    title: 'Error in console when clicking Add',
  },
];
```

```

    {
      id: 2, status: 'Assigned', owner: 'Eddie',
      created: new Date('2016-08-16'), effort: 14, completionDate: new
      Date('2016-08-30'),
      title: 'Missing bottom border on panel',
    },
  ],
  ...

```

Testing this set of changes will show that new issues can be added, and even on a restart of the Node.js server, or the database server, the newly added issues are still there. As a cross-check, use the mongo shell to look at the collection after every change from the UI.

EXERCISE: WRITING TO MONGODB

1. Could we have just added the `_id` to the passed-in object, and returned that instead of doing a `find()` for the inserted object?
2. We could have used `insert()` instead of `insertOne()`. What would be the difference? Why did we choose to use the latter?
Hint: Look up the MongoDB driver documentation.

Answers are available at the end of the chapter.

Summary

In this chapter, we looked at the installation and use of MongoDB, both via the mongo shell as well as from Node.js code via the native driver. We explored a few methods, the pillars of MongoDB via the shell as well as a test program in Node.js. We then used these learnings to insert and read a document from the database, thus making the issue list persistent.

You should now have a good understanding of the fundamentals of MongoDB and the paradigms that it uses. We did not go deeper into other operations such as update and delete, or getting summary reports via aggregates. We'll do all of that later, when we implement features in the Issue Tracker that require them to be used.

In the next chapter, we'll take a break from implementing features. Instead, we'll start getting organized. We'll modularize the code and use tools to improve our productivity since the project is starting to get bigger.

Answers to Exercises

Exercise: Mongo Shell

1. Yes, the mongo shell does support some ES2015 features. Notably, you can use arrow functions, string interpolation, and `const` variables.
2. This can be done using the `$exists` operator like this:

```
> db.employees.find({"name.middle": {$exists: true}})
```

3. The filter specification is not a JSON document, because it is not a string. It is a regular JavaScript object, which is why you are able to skip the quotes around the property names. You will also be able to have real Date objects as field values, unlike a JSON string.

4. The `$unset` operator in an update can be used to unset a field (which is actually different from setting it to null). Here is an example:

```
> db.employees.update({_id: ObjectId("57b1caea3475bb1784747ccb")}, {"name.middle": {$unset: null}})
```

Although we supplied `null` as the value for `$unset`, this value is ignored. It can be anything.

5. The 1 indicates an *ascending* sort order for traversing the index. -1 is used to indicate a descending sort order. This is useful only for compound indexes, because a simple index on one field can be used to traverse the collection in either direction.

Exercise: Schema Initialization

1. Creating our own `_id` would make the display and identification of the issue more user friendly, because it can be a continuous number like 1, 2, 3, and so on. The `ObjectId` is harder to refer to. On the other hand, generating a continuous number is more work, and may not scale well when there are a large number of documents and simultaneous inserts from multiple clients. Read the MongoDB tutorial called *Create an Auto-Incrementing Sequence Field* for a more detailed discussion.
2. A search bar is quite helpful when searching for issues. A text index (an index based on the words) on the title field would be useful in this case. Find out more about text indexes in the MongoDB tutorial. We'll implement a text index towards the end of the book.

Exercise: Reading from MongoDB

1. The connection object is in fact a connection pool. It automatically figures out the best thing to do: reuse an existing TCP connection, reestablish a new connection when the connection is broken, etc. Using a global variable (at least, reusing the connection object) is *the* recommended usage. If you don't like global variables, you can create a module and encapsulate the connection within its namespace.
2. If the database is unavailable for a short period (30 seconds), the driver waits and reconnects when the database is available again. If the database is unavailable for a longer period, the read throws an error. The driver is also unable to reestablish a connection when the database is restored. The application server needs to be restarted in this case.

If you'd rather throw an error if the database is unreachable, set `bufferMaxEntries` to 0 when creating the connection. This is perhaps the preferred approach for a single server. When you have a replica set, the default behavior works better.

3. One option is to use `limit()` on the result to limit the return value to a maximum number of records. For example, `find().limit(100)` returns the first 100 documents. If you were to paginate the output in the UI, you could also use the `skip()` method to specify where to start the list.

Another option is to use `forEach()` or `stream()` to stream the result back to the client, without occupying too much memory on the server to hold the entire array. This assumes that the client can handle large results.

Exercise: Writing to MongoDB

1. Adding the `_id` and returning the object passed in would have worked, so long as you know for a fact that the write was a success and the object was written to the database as is. In most cases, this would be true, but it's a good practice to get the results from the database, as that is the ultimate truth. Since we are querying on the `_id` (which is automatically indexed), the extra call is very minimal in terms of latency added to the query.
2. The result from an `insert()` is slightly different from `insertOne()`. Instead of `insertedId`, it returns an array of `insertedIds`. You use `insertOne()` since `insert()` is deprecated in favor of `insertOne()` and `insertMany()`, and the usage is unambiguous.

CHAPTER 7



Modularization and Webpack

In this chapter, we'll take a break from regular coding and adding features. Instead, we'll get a bit organized so that the application can grow bigger, yet be manageable. The focus will be on development tools.

The goal of this chapter is to be able to split the code into multiple files, both on the server and the client side. Despite this, we should be able to continue the development process as before: automatically restart or rebuild whenever files change, except that this would be *any* file. Further, we'll even get the browser to refresh automatically when files change during development. Finally, we'll add checks to verify that the code we write follows some standards and good practices, and catches possible bugs earlier than the testing cycle.

Server-Side Modules

You saw in the Hello World chapter how to include modules in a Node.js file after installing the module using `npm install`. But what about your own modules? How do you write a file that can be included in another file?

It's surprisingly simple to do this. There is a special variable that Node.js looks for called `module.exports` within any file. If that file is used in a `require` statement, Node.js just uses the value of this `module.exports` variable, and returns the same in the `require` call. Thus, anything that you want to be available to be *exported*, you can just add to the `module.exports` object.

Let's aim to separate out code related to the Issue object (or *model*) into its own file. To start with, let's create a directory called `server` where we will keep all server-side files, since there is going to be more than one file going forward. Let's move `server.js` into this directory. We'll also create a new file called `issue.js` and cut/paste all of the issue validation-related code into this file. We need to add `'use strict';` at the beginning of the file to let Node.js accept `const` declarations. Finally, we define `module.exports` as an object with the property `validateIssue` set to the function `validateIssue`.

The contents of the new `issue.js` file are shown in Listing 7-1.

Listing 7-1. `issue.js`: New File for the Issue Model

```
'use strict';

const validIssueStatus = {
  New: true,
  Open: true,
  Assigned: true,
  Fixed: true,
  Verified: true,
  Closed: true,
};

const issueFieldType = {
  status: 'required',
  owner: 'required',
  effort: 'optional',
  created: 'required',
  completionDate: 'optional',
  title: 'required',
};

function validateIssue(issue) {
  for (const field in issueFieldType) {
    const type = issueFieldType[field];
    if (!type) {
      delete issue[field];
    } else if (type === 'required' && !issue[field]) {
      return `${field} is required.`;
    }
  }

  if (!validIssueStatus[issue.status])
    return `${issue.status} is not a valid status.`;

  return null;
}

module.exports = {
  validateIssue: validateIssue
};
```

■ **Note** The first line, `'use strict';`, is important. Without it, the behavior of `const` and `let` are different in Node.js version 4.5. Ensure that you have not missed this line.

To use this new module in `server.js`, we must include the module that we just created, using the (by now familiar) `require` statement. When you refer to your own modules rather than modules installed via npm, you need to tell Node.js the *path* of the module's file rather than just the name. So, in this case, we must use `'./issue.js'` rather than a plain `'issue'`. The changes are shown in Listing 7-2.

Listing 7-2. `server.js`: Changes to Using `issue.js` Module

```
...
const MongoClient = require('mongodb').MongoClient;
const Issue = require('./issue.js');
...
app.post('/api/issues', (req, res) => {
...
  const err = Issue.validateIssue(newIssue);
...

```

We could have chosen to assign `module.exports` directly to `validateIssue` (like `module.exports = validateIssue`), in which case, the return value of the `require` statement in `server.js` would have been the function itself, and we could use it as is (like `const validateIssue = require('./issue.js');`). But we chose to use an object to enclose the exported function. The expectation is that we may export more things related to the `Issue` object later, and they can all be inside this object.

Finally, to let nodemon watch all the files in the new server directory, we need to modify the `start` command in `package.json`. Now that we have a directory for all of the server-side code, the `-w` argument can take the directory `server` as the value instead of a single file. Further, the starting point is `server/server.js` rather than simply `server.js`. This change is listed in Listing 7-3.

Listing 7-3. `package.json`: Changes to Start Script

```
...
  "start": "nodemon -w server.js server.js",
  "start": "nodemon -w server server/server.js",
...

```

If you restart the server (by pressing Ctrl-C in the console where it was started, and running `npm start` again), you should be able to test your changes. You need to do this because `package.json` itself changed, and it needed to be reloaded. The application should function as before, because we have only refactored the code. We have not introduced any functional changes.

Introduction to Webpack

Traditionally, one would split client-side JavaScript code into multiple files, and include them all (or whichever are required) using `<script>` tags in the main HTML file. This is less than ideal because the dependency management is done by the developer, by maintaining a certain order of files in the HTML file. Further, when the number of files becomes large, this becomes unmanageable.

Tools such as webpack and browserify provide alternatives that let you define dependencies as you would in a Node.js application using `require` or equivalent statements. They automatically figure out not just your own dependent modules, but also third-party libraries. Then, they put together these individual files into one or more bundles of pure JavaScript that has *all* the required code that can be included in the HTML file.

The only downside is that this requires a build step. But then, we already have a build step to transform JSX and ES2015 into plain JavaScript. It's not much of a change in habit to let the build step also create a bundle based on multiple files. Both webpack and browserify are good tools and can be used to achieve the goals. But I chose webpack, because it is simpler to get all that we want done, which includes separate bundles for third-party libraries and our own modules. It has a single pipeline to transform, bundle, and watch for changes and generate new bundles as fast as possible.

If you choose Browserify instead, you will need other task runners such as gulp or grunt to automate watching and include multiple transforms. This is because Browserify does only one thing: bundle. In order to combine bundle and transform (using babel) and watch for file changes, you need something that puts all of them together, and gulp is one such utility. In comparison, webpack (with a little help from loaders, which we'll explore soon) can not only bundle, but can also do many more things such as transforms and watching for changes to files. You don't need additional task runners to use webpack.

Note that webpack can also handle other static assets such as CSS files. It can even split the bundles such that they can be loaded asynchronously. We will not be exercising those aspects of webpack; instead, we'll focus on the goal of being able to modularize the client-side code, which is mainly JavaScript at this point in time.

Using Webpack Manually

To get used to what webpack really does, we'll use webpack from the command-line just like we did for the JSX transform using babel command line. Let's first install webpack:

```
$ npm install --save-dev webpack
```

We installed webpack locally because, just like with babel, we'll eventually move all commands into commands defined in `package.json`, so that they can be run using `npm run`. We don't need webpack globally. Let's see what webpack does. You can run it on the client-side JavaScript file `App.js` and produce a bundle called `app.bundle.js` like this:

```
$ node_modules/.bin/webpack static/App.js static/app.bundle.js
```

You can see that webpack creates `app.bundle.js`, which is not very different from `App.js` itself. Note also that we didn't run it against the React file `App.jsx`, because webpack cannot handle JSX natively. What webpack did in this bundling is hardly interesting. We did it just to make sure we've installed it correctly and are able to run it.

To start the modularization exercise, let's split `App.jsx` into two files by separating out one component, `IssueAdd`. Let's create a new file called `IssueAdd.jsx` under the `src` directory, and move the entire `IssueAdd` class to this file. To include the new file, we could use the `require` style imports and exports as in the server-side modules. But ES2015 supports a new style with `import` statements, which are easier to read than `require` statements. We could not take advantage of this style for the server side-code because Node.js does not support this style natively as of the latest version at the time of writing this book.

Using the new ES2015 style to export a class is as simple as prefixing `export` before the class definition. Further, you can add the keyword `default` if you are exporting a single class, and you want it to be the result of an `import` statement directly (or a *top-level* export).

Listing 7-4 shows the changes to the class definition. The rest of the new file is the original contents of the class unchanged.

Listing 7-4. `IssueAdd.jsx`: New File, Class Contents Moved Here from `App.jsx`

```
...
export default class IssueAdd extends React.Component {
  ...
}
```

To import this class in `App.jsx`, we need to use an `import` statement right at the beginning of the file. This, as well as the removal of the `IssueAdd` class, is shown in Listing 7-5

Listing 7-5. `App.jsx`: Move `IssueAdd` Class and Import It

```
...
import IssueAdd from './IssueAdd.js';
...
class IssueAdd extends React.Component {
    constructor() {
        super();
        this.handleSubmit = this.handleSubmit.bind(this);
    }
  ...
}
...
```

Let's run the JSX transformation using `npm run compile` (or let the babel watch automatically recompile upon detecting changes). This will transform both files, since we'd specified the entire `src` directory as input source to babel. You'll see that two files, `App.js` and `IssueAdd.js`, are generated. Now, when you run webpack with the same command as before, you will notice that it automatically includes `IssueAdd.js` in the bundle, without you ever telling it to do so. Here is a sample result of a webpack run:

```
Time: 88ms
      Asset      Size  Chunks             Chunk Names
app.bundle.js  11.7 kB          0 [emitted]  main
    [0] ./static/App.js  7.06 kB {0} [built]
    [1] ./static/IssueAdd.js  2.9 kB {0} [built]
```

You could create many more such files, and webpack does not need to be told about any of them. That's because it looks at the `import` statements within each of the JavaScript files and figures out the dependency tree. Finally, we need to change the script file referenced in `index.html` to the new `app.bundle.js` instead of `App.js`. This change is shown in Listing 7-6.

Listing 7-6. `index.html`: Replace `App.js` Script with `app.bundle.js`

```
...
<script src="/app.bundle.js"></script>
...
```

At this point, you should test the application to see if it works as before. There will be some residual temporary files, `App.js` and `IssueAdd.js`, in the static directory, which you will not require any longer, so they can be removed.

Transform and Bundle

In the previous step, we had to manually transform JSX files, and then use webpack to bundle them together. The good news is that webpack is capable of combining these two steps, obviating the need for intermediate files. But it can't do that on its own; it needs some helpers called *loaders*. All transforms and file types other than pure JavaScript require loaders in webpack. These are separate modules.

In this case, we need the babel loader to handle JSX transforms, so let's install it:

```
$ npm install --save-dev babel-loader
```

Using loaders in the command line is cumbersome, especially if you want to pass parameters to the loaders (babel, in this case, needs presets `react` and `es2015`). You can instead use a configuration file to give these parameters to webpack. The default configuration file that webpack looks for is called `webpack.config.js`, so let's create that file.

Webpack loads this configuration file as a module and gets its parameters from this module. Everything has to be under one exported object. Since we won't be transforming this file, let's use the `module.exports` syntax rather than the ES2015 export syntax. Inside the exported object, webpack looks for various properties. We'll use the properties `entry` and `output` to replace what we did with the command line parameters until now. The entry point is now `App.jsx` rather than `App.js`. The output property takes `path` and `filename` as two subproperties.

To add the babel loader, we need to define the configuration property `module.loaders` and supply it with an array of loaders. We will have only one element, the babel loader in this array. A loader specification includes a test (regular expression) to match files, the loader to apply (in this case, `babel-loader`), and finally a set of options to the loader specified by the property `query`.

The final contents of the configuration file are shown in Listing 7-7.

Listing 7-7. `webpack.config.js`: Configuration File for webpack

```
module.exports = {
  entry: './src/App.jsx',
  output: {
    path: './static',
    filename: 'app.bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.jsx$/,
        loader: 'babel-loader',
        query: {
          presets: ['react', 'es2015']
        }
      }
    ]
  }
};
```

The option for babel loader is an array of presets, very similar to the babel command line:

```
...
    query: {
      presets: ['react', 'es2015']
    }
...

```

The import statement in `App.jsx` referred to `IssueAdd.js`, but this file will no longer be created because webpack will transform `IssueAdd.jsx` to `IssueAdd.js` *on the fly*. So, for the dependency tree to be built correctly, we will have to import the pretransformed file with the `jsx` extension, as shown in Listing 7-8.

Listing 7-8. App.jsx: Import the jsx File Instead of js

```
...
import IssueAdd from './IssueAdd.jsx';
...
```

Now, webpack can be run on the command line without any command line parameters, like this:

```
$ node_modules/.bin/webpack
```

You will notice that it takes quite a while to complete. You don't want to wait this long every time you modify a source file, do you? Nor do you want to run this command over and over again. Fortunately, just like babel, webpack has a watch mode, which looks for changes and processes *only* the changed files. So, run this command instead:

```
$ node_modules/.bin/webpack --watch
```

Modify some text (create a syntax error, for instance) and ensure that it rebuilds the bundle on every change, and note how long it takes to do this. Watch how errors are reported, too. Now is a good time to modify the npm script commands for building and watching for changed files. Let's replace the `compile` and `watch` script specifications in `package.json`, as shown in Listing 7-9.

Listing 7-9. package.json: New Compile and Watch Scripts

```
...
  "compile": "webpack",
  "watch": "webpack --watch",
...
```

Now that you know what webpack is capable of, let's organize your files into one file per class: `IssueList`, `IssueAdd`, and `IssueFilter`. Let the stateless components `IssueTable` and `IssueRow` remain with `IssueList` in the same file. The entry file, `App.jsx`, will only import the other classes and mount the main component. This will create a two-level hierarchy of imports: `App.jsx` will import `IssueList.jsx`, which in turn will import `IssueAdd.jsx` and `IssueFilter.jsx`.

To start with, let's move the placeholder class `IssueFilter` to its own file. This is shown in Listing 7-10.

Listing 7-10. IssueFilter.jsx: Move Class IssueFilter Here and Add Export

```
export default class IssueFilter extends React.Component {
  render() {
    return (
      <div>This is a placeholder for the Issue Filter.</div>
    )
  }
}
```

Similarly, let's take the class `IssueAdd` from the file `App.jsx` and create a new file called `IssueAdd.jsx`. Listing 7-11 shows the partial contents of the new file: the class as such is unchanged, except that in the class declaration you have the keywords `export default`.

Listing 7-11. `IssueAdd.jsx`: New File, Move Class `IssueAdd` Here, and Add Export

```
export default class IssueAdd extends React.Component {
  constructor() {
    super();
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  ...
}
```

Listing 7-12 shows another change, but here we extract three classes (`IssueRow`, `IssueTable`, and `IssueList`) from `App.jsx`. The new file created is called `IssueList.jsx`. Since this file needs the other extracted classes, `IssueAdd` and `IssueFilter`, we also need import statements for them in addition to the classes that are moved into this file. Only the class `IssueList` is exported, so we need to add the keywords `export default` only to that class declaration. The other two classes are for internal use only, so they are not exported.

Listing 7-12. `IssueList.jsx`: Move Classes `IssueList`, `IssueTable`, and `Issue Row` Here, and Add Import and Export

```
import IssueAdd from './IssueAdd.jsx';
import IssueFilter from './IssueFilter.jsx';

const IssueRow = (props) => (
  <tr>
    ...
  </tr>
)

function IssueTable(props) {
  const issueRows = props.issues.map(issue => <IssueRow ↵
    key={issue._id} issue={issue} />)
  return (
    ...
  );
}
```

```

export default class IssueList extends React.Component {
  constructor() {
    super();
    this.state = { issues: [] };

    this.createIssue = this.createIssue.bind(this);
  }
  ...
}

```

Now that most of the contents of `App.jsx` have been moved out to their individual files, we're left with just the rendering of the component. The contents of this file are shown in Listing 7-13.

Listing 7-13. `App.jsx`: Complete Contents After Moving All Classes Out

```

import IssueList from './IssueList.jsx';

const contentNode = document.getElementById('contents');
ReactDOM.render(<IssueList />, contentNode); // Render the component inside ←
                                              the content Node

```

If you run `npm run watch`, you will find that all the compilations are done and the bundle is ready for you to test the application. If not, just start the `npm start` and `npm run watch` commands in different consoles, and then test the application to ensure that it behaves the same as before.

EXERCISE: TRANSFORM AND BUNDLE

1. Save any JSX file with only a spacing change, while running `npm run watch`. Does webpack rebuild the bundle? Why not?
2. Why did we separate the mounting of the component and the component into different files? Hint: Think about what other pages you'll need in the future.
3. What would happen if you did not use the `default` keyword while exporting a class? Hint: Revisit how you had choices for `validateIssue` to be exported. Also look up the Babel documentation and/or Mozilla Developer Network (MDN) documentation on JavaScript export statement.

Answers are available at the end of the chapter.

Libraries Bundle

For the server side, we started by importing third-party libraries, and then we created modules from our own code. For the client-side code, we have only used modules of our own code until now.

But we do have library dependencies, notably React itself. We included them as scripts from a CDN. In this section, we'll use webpack to create a bundle that includes these libraries. If you remember, I discussed that npm is used not only for server-side libraries, but also client-side ones. What's more, webpack understands this and can deal with client-side libraries installed via npm.

So, let's first install, using npm, the client-side libraries that we have used until now. This is the same list as the list of `<script>`s in `index.html`.

```
$ npm install --save-dev react react-dom whatwg-fetch babel-polyfill
```

Next, to use these installed libraries, let's them in all the client-side files where they are needed, just like we imported our own files. Listings 7-14 to 7-17 show these changes.

Listing 7-14. `App.jsx`: Imports for Using React and ReactDOM

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

Listing 7-15. `IssueAdd.jsx`: Imports for Using React

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

Listing 7-16. `IssueFilter.jsx`: Imports for Using React

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

Listing 7-17. `IssueList.jsx`: Imports for Using React and Fetch

```
...
import React from 'react';
import 'whatwg-fetch';
...
```

You can now remove all the script references from `index.html`, as shown in Listing 7-18.

Listing 7-18. `index.html`: Remove Scripts Loaded from CDN

```
...
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-
js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom-
js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/fetch/1.0.0/fetch.min-
js"></script>
...
```

Note that unlike React and React-DOM, `fetch` is imported into the global namespace, because it is expected to be a substitute for `window.fetch()`, which should be available in the browser in any case. If you have `npm run watch` already running, you will notice that the number of hidden modules has shot up from four to more than a hundred, and also that the size of `app.bundle.js` has increased from few kB to more than 1Mb. The console output of webpack before this step would have looked like this:

```
Hash: 8f5de02c8f9e672b4ccb
Version: webpack 1.13.2
Time: 10593ms

   Asset      Size  Chunks             Chunk Names
app.bundle.js  13.8 kB      0 [emitted]      main
   + 4 hidden modules
```

And after the change, it will look like this:

```
Hash: 983fd62bdc8309eb9fd5
Version: webpack 1.13.2
Time: 11886ms

   Asset      Size  Chunks             Chunk Names
app.bundle.js  1.02 MB      0 [emitted]      main
   + 472 hidden modules
```

The fact that the bundle includes all of the libraries is a minor problem. The libraries will not change often, but the application code will, especially during the development and testing. Even when the application code undergoes a small change, the *entire* bundle is rebuilt, and therefore, a client will have to fetch (the now big) bundle from the server. We're not taking advantage of the fact that the browser can cache scripts when they are not changed. This not only affects the development process, but even in production, users will not have the optimum experience.

A better alternative is to have *two* bundles, one for the application code and another for all the libraries. It turns out that a plugin called `CommonsChunkPlugin`, a webpack built-in plugin, can do this quite easily. The modified `webpack.config.js` is shown in Listing 7-19.

Listing 7-19. `webpack.config.js`: Separate Vendor Bundle Configuration

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

module.exports = {
  entry: './src/App.jsx',
  entry: {
    app: './src/App.jsx',
    vendor: ['react', 'react-dom', 'whatwg-fetch'],
  },
  output: {
    path: './static',
    filename: 'app.bundle.js'
  },
  plugins: [
    new webpack.optimize.CommonsChunkPlugin('vendor', 'vendor.bundle.js')
  ],
  module: {
    loaders: [
      {
        test: /\.jsx$/,
        loader: 'babel-loader',
        query: {
          presets: ['react', 'es2015']
        }
      },
    ]
  }
};
```

First, we specified two entry points to webpack in the configuration file, one for the app and the other for the libraries. Thus, the configuration variable `entry` is now an object with two keys: one for the app and the other for the libraries, which we called `vendor`.

Next, we added the plugin in a new configuration variable called `plugins`. The first line, where we *require* webpack, is needed because the plugin is available as part of the webpack module. We passed in two parameters to the plugin, the first being the name of the entry identifier, and the second its output file name.

You must reload the configuration file by restarting `npm run watch`. You'll now see that there are *two* bundles being created. You may see something similar to the following console output:

```
Hash: 82e5bb4d86c63757adb8
Version: webpack 1.13.2
Time: 13826ms

      Asset      Size  Chunks             Chunk Names
  app.bundle.js  13.8 kB      0  [emitted]  app
  vendor.bundle.js    1 MB      1  [emitted]  vendor
    [0] multi vendor 52 bytes {1} [built]
      + 472 hidden modules
```

Further, if you modify any source file, you'll see that only the application bundle is being rebuilt, that too in much lesser time.

```
Hash: d8ef5154f4ffe1780984
Version: webpack 1.13.2
Time: 231ms

      Asset      Size  Chunks             Chunk Names
  app.bundle.js  13.8 kB      0  [emitted]  app
      + 473 hidden modules
```

Now, let's include the vendor bundle in `index.html`, so that the contents are available to the rest of the code. This change is shown in Listing 7-20.

Listing 7-20. `index.html`: Including the Vendor Bundle Script

```
...
<script src="/vendor.bundle.js"></script>
<script src="/app.bundle.js"></script>
...
```

Refresh the browser and check whether `vendor.bundle.js` is getting reloaded in the browser. Using the developer console, you should see a 304 response for the vendor bundle, which means that it is not fetching the bundle if it is already there in the browser's cache. This should happen even when you change any source code and regenerate the application bundle, and that's the advantage of splitting the bundle in two.

Hot Module Replacement

The watch mode of webpack works well for client-side code, but there is potential pitfall with this approach. You must keep an eye on the console that is running the command `npm run watch` to ensure that bundling is complete, before you refresh your browser to see the effect of your changes. If you hit the refresh a little too soon, you will end up with the previous version of your client-side code, and scratch your head wondering why your changes didn't work, and then spend time debugging.

The webpack-dev-server comes in handy here. Not only can it watch for changes, but it can speed up delivery of the bundle by keeping the bundle completely in memory, not writing it to disk. But the best part is that it makes the browser *wait* until the bundle is ready, thus ensuring that if you have changed any client-side code, you are guaranteed to **not** be served the previous version, saving some scratches on your head.

Let's start by installing the webpack-dev-server:

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

Since it is part of webpack, the webpack-dev-server understands `webpack.config.js`, and produces `app.bundle.js` and `vendor.bundle.js` according to the specifications in there. You just need to give it a couple parameters, the first of which is the port on which it will listen as a server. The second parameter is to tell it where static files *other* than the bundles reside. In our case, we need it to serve `index.html` from the static directory. Thus, we can run webpack-dev-server as follows:

```
$ node_modules/.bin/webpack-dev-server --port=8000 --content-base=static
```

Open up a *new* window or tab in your browser and go to `http://localhost:8000/`. Note that now you are using the port of the webpack dev server (8000), not the Express web server (3000). If you make a change in any of the client-side code, you will find that while the bundling is in process, the browser is made to wait as opposed to being given a previous version of the bundles. But you will also find that although the assets are loaded, the REST API calls fail, because webpack-dev-server has no clue how to handle them.

The original URL, `http://localhost:3000/`, continues to work in the other browser window or tab. This is because the APIs and the static files are both still available via this application, which is hosted by the Express server. What we want is the APIs to be served from the Express server and the static content and the bundles to be served from the webpack-dev-server. But we want it all from a single server.

The solution is to tell webpack-dev-server to act as a proxy for API requests, and *forward* them to the Express server. The webpack-dev-server is capable of doing this, but setting this up is a little involved, so it's better done in a configuration file rather than command line parameters. The webpack-dev-server looks into a special section called `devServer` in `webpack.config.js` for its parameters. Let's configure the proxy here, and also move the command line parameters into this configuration. The partial changes are shown in Listing 7-21.

Listing 7-21. webpack.config.js: Webpack-dev-server Configuration Added

```
...
},
devServer: {
  port: 8000,
  contentBase: 'static',
  proxy: {
    '/api/*': {
      target: 'http://localhost:3000'
    }
  }
}
}
```

The first two parameters, `port` and `contentBase`, were replacements for the command line. The proxy configuration tells webpack-dev-server to look for requests matching `'/api/*'` and forwards them to your Express server, which runs at the original URL at port 3000. Let's restart webpack-dev-server, this time without any command line parameters because the configuration is now available as part of `webpack.config.js`.

```
$ node_modules/.bin/webpack-dev-server
```

Now, when you refresh the browser page pointing to the 8000 port page, you will find that the API calls are also working. If you make a change in any of the client-side code, you will also see the webpack-dev-server rebuilding the bundles. This is great, but the webpack-dev-server is capable of some more magic: hot module replacement (HMR).

HMR automatically causes a browser refresh whenever there is a change in the client-side bundle. In fact, it *incrementally* replaces only the modules that have changed, making it very efficient in terms of network data transfer. This is really useful for impatient programmers. To enable HMR, all we have to do is add a couple of command line parameters to the webpack-dev-server invocation:

```
$ node_modules/.bin/webpack-dev-server --hot --inline
```

Now, when you refresh the browser (for the last time, I may say), you will see the following in the browser's console log:

```
[HMR] Waiting for update signal from WDS...
[WDS] Hot Module Replacement enabled.
```

If you change any client-side code, you will see more log messages in the console, and you will find that the bundle is rebuilt, and the browser automatically refreshes and gets the new changes!

There's still one small caveat, though: you will find that the browser is refreshing rather than hot replacing the module. If you look at the console log (you may need to preserve log across requests), you will also see a message saying it's unable to replace.

That is because your client-side code that is responsible for dealing with modules is yet to *accept* HMR.

To enable this, we need a change in the topmost module, that is, `App.jsx`, that accepts HMR. The new `App.jsx` is shown in Listing 7-22.

Listing 7-22. `App.jsx`: Modified to Accept HMR

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

import IssueList from './IssueList.jsx';

const contentNode = document.getElementById('contents');
ReactDOM.render(<IssueList />, contentNode); // Render the component inside the content Node

if (module.hot) {
  module.hot.accept();
}
```

Now, you should be able to see that the browser doesn't do a full refresh. There is a good chance that you'll mistakenly type the familiar `http://localhost:3000` instead of your new 8000 port that webpack dev server is running on. To prevent this, it's best if you *delete* the files `app.bundle.js` and `vendor.bundle.js` so that they don't get served by the 3000 server, and an error will be immediately apparent.

Finally, let's replace your `watch` command in `package.json` to use `webpack-dev-server` rather than just `bundle` using `webpack`. This change is listed in Listing 7-23.

Listing 7-23. `package.json`: Change Watch Script to use `webpack-dev-server`

```
...
  "watch": "webpack-dev-server --hot --inline",
  ...
```

EXERCISE: HOT MODULE REPLACEMENT

1. How can you tell the difference between a browser refresh and a hot module replacement? Can you compare the performance of the two? Hint: Use the Network section of your browser's developer tools and watch what goes on.
2. Do you think `webpack-dev-server` is useful even without HMR? Hint: What problems does the dev server HMR solve? Which of them is damaging?

Answers are available at the end of the chapter.

HMR Using Middleware

Wouldn't it be great if you didn't have to run two servers and could thus avoid the confusion with the ports? It turns out that there is an alternative to using `webpack-dev-server`, by wiring in the HMR pieces within the Express code itself. This will make it a *single* server that serves your APIs, and at the same time watches for changes to client-side code, rebundles it, and sends incremental updates to the client.

The way to do this is to install HMR as middleware in the Express server. As discussed earlier, middleware is something that can intercept requests and do some special processing. The `webpack-dev-middleware` does the same work as it did as an independent `webpack-dev-server`: it looks for changes to files, and blocks requests to the bundle until the changes are incorporated into the bundle. The `webpack-hot-middleware` does a similar thing for HMR. Let's make these two middleware functions available by installing the respective modules:

```
$ npm install --save-dev webpack-dev-middleware webpack-hot-middleware
```

We'll use these modules within the Express server, but within an `if` condition that does it only in a development environment. We'll have to import these modules and initialize them with a webpack configuration. Most of the configuration is already there in `webpack.config.js`, but we need a few changes to the configuration for it to be used in an HMR environment. They are the following:

1. We need additional entry points (other than `App.jsx`) so that webpack can build the client-side code necessary for this extra functionality into the bundle.
2. We need to add a plugin (which got enabled using the `--hot` command line for the `webpack-dev-server`) that generates incremental updates rather than entire bundles that can be sent to the client.

We'll make these changes by patching the configuration on the fly within the Express server. But we need a few changes in `webpack.config.js` that will not affect the production configuration, yet will allow us to patch in the changes. The changes required are shown in Listing 7-24.

Listing 7-24. `webpack.config.js`: Compatible Changes to Allow Patching

```
...
app: './src/App.jsx',
app: ['./src/App.jsx'],
...
path: './static',
path: __dirname + './static',
...
```

The first change is to use an array instead of a single entry point. This will let us add more entry points on the fly. The next change is to use an absolute path name for the assets output directory name. This is a requirement for the middleware, but doesn't affect the webpack command line. Now we can make the changes in the Express server that initializes the HMR middleware. These changes are shown in Listing 7-25, which can be inserted before the first `app.get()` call.

Listing 7-25. `server.js`: Use HMR via Middleware for Development Convenience

```
...
if (process.env.NODE_ENV !== 'production') {
  const webpack = require('webpack');
  const webpackDevMiddleware = require('webpack-dev-middleware');
  const webpackHotMiddleware = require('webpack-hot-middleware');

  const config = require('../webpack.config');
  config.entry.app.push('webpack-hot-middleware/client',
    'webpack/hot/only-dev-server');
  config.plugins.push(new webpack.HotModuleReplacementPlugin());

  const bundler = webpack(config);
  app.use(webpackDevMiddleware(bundler, { noInfo: true }));
  app.use(webpackHotMiddleware(bundler, { log: console.log }));
}
...
```

The first three lines in the block load the newly installed modules. Then, we loaded up the default webpack configuration using another `require` statement, just as webpack itself would have. Note that this can be done thanks to the fact that webpack configurations are modules, not JSON files. Then we appended additional entry points and plugins that are required for HMR. Finally, we created a bundler using the new options, and passed it to the two middleware instantiations. To install the middleware, we use `app.use()`.

For HMR to work, just as in the previous section, you have to *accept* HMR in `App.jsx`, as shown in Listing 7-22 in the previous section. With these changes, the server should have automatically restarted if you were running `npm start` because nodemon was watching for server file changes. Now, go back to the 3000 port server, without the `webpack-dev-server` running. You need a manual browser refresh for the first time to kick off HMR, and then, for every new client side change, you can see that modules are being hot replaced automatically in the browser.

Comparison of HMR Alternatives

At first, I liked the middleware approach because it tied in nicely with the Express server, and I did not have to run two consoles and monitor the restarts on each. Further, it did not require another server and a listening port, and there was no question of making a mistake by connecting to the wrong server and scratching heads.

But a problem with the middleware approach is that a change in the server-side code causes the entire front-end bundle to be rebuilt, which takes approximately 13 seconds on my environment. This is quite a long wait, and quite unnecessary too, since I did not touch any client-side code. If there were a way to cache the bundles and not rebuild them on every restart, the middleware option would be ideal.

Since there is no way to do that (at least as of writing this book), I have chosen to revert back to the webpack-dev-server approach during my development. It does the job: you don't need to monitor the bundling and refresh the browser (because you're guaranteed to be not given stale bundles), so the extra console occupied for this can remain hidden. As for mistakenly connecting to the wrong server, I have ensured that I don't ever run `npm run compile`, and I frequently check that the bundles don't exist in the file system.

I encourage you to make your own choice. Depending on how often you plan to change the server-side code, you may find the middleware alternative more convenient, unlike me.

Debugging

Until the previous chapter, if you needed to do any debugging on the client side, you probably used `console.log` statements. Adding breakpoints using the developer tool of your browser could have worked too, but you would have noticed that the generated JavaScript code is quite different from the JSX we wrote. Creating a single bundle from four different source files makes it even worse.

Fortunately, webpack solves this problem by its ability to give you *source maps*, things that contain your original source code as you typed it in. The source maps also connect the line numbers in the transformed code to your original code. Browsers' development tools automatically understand source maps and correlate the two, letting you put breakpoints in *your* code and converting them to breakpoints in the transformed or transpiled code.

It's quite simple to enable source maps. The small change to `webpack.config.js` is listed in Listing 7-26.

Listing 7-26. `webpack.config.js`: Add Source Map Configuration

```
...
  devtool: 'source-map'
...
```

You need to restart `npm run watch` because of the configuration change. Note now that two maps are generated by webpack apart from the two bundles. This can be seen in the initial output of `webpack-dev-server` (you may need to scroll up in your console), or when making changes to client-side code. Now, if you open up your Resources section in your browser's development tool, you will find the JSX files also listed, within which you can set breakpoints. A sample development tool screen is shown in in Figure 7-1.

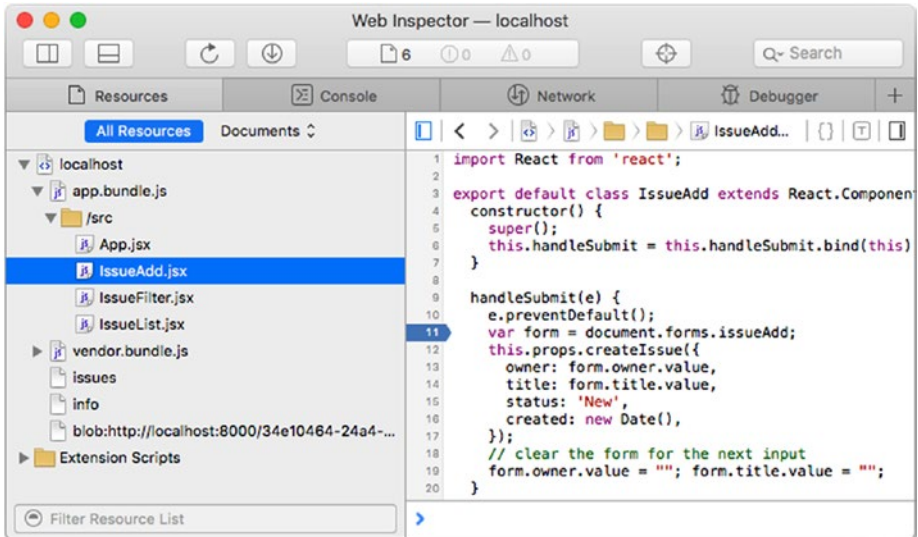


Figure 7-1. Developer console showing source files

Server-Side ES2015

Even though Node.js supports many ES2015 features, it is not complete. For example, you still need to use `require` instead of `import` to include other modules. Further, Node 4.5 does not support some useful things like destructuring assignments and spread operators. On the client side, we assumed full ES2015 support and used these features. It would be good to have the server side also support the same set, so that we don't have to be conscious of this fact when switching between client-side and server-side code. If we do that, our coding style and features used can be uniform across the board.

Let's start using the `import` and `export` style of modularization. Listing 7-27 shows the changes to `server.js`, which includes removal of `'use strict'` as well.

Listing 7-27. `server.js`: Replace `Require` with Imports

```
...
'use strict'

const express = require('express');
const bodyParser = require('body-parser');
const MongoClient = require('mongodb').MongoClient;
const Issue = require('./issue.js');
...
import express from 'express';
import bodyParser from 'body-parser';
import { MongoClient } from 'mongodb';
import Issue from './issue.js';
...
```

Listing 7-28 shows changes to `issue.js`, where we replace the `module.exports` with a simple `export default`, just like we did for the client-side code, as well as the removal of `use strict`, which is no longer required due to ES2015.

Listing 7-28. `issue.js`: Changes for Using ES2015 Export

```
...
'use strict'
...
export default {
  validateIssue: validateIssue
};
...
```

Let's also create a directory where the compiled files will be saved. Let's call this directory `dist` (short for *distribution*), and try out a transform using `babel`, as we did in the Hello World chapter, to transform all the JSX:

```
$ node_modules/.bin/babel server --presets es2015 --out-dir dist
```

Now, instead of running `server.js` from the original directory, you can run it from the `dist` directory. Try it out manually to ensure that the server starts up just fine. If you inspect the compiled files, you will notice that the files are fully ES5 compliant, that is, you will find `vars` instead of `consts`, and you will see that arrow functions have been converted to regular functions. That's great, but this means that you are not taking advantage of the fact that Node.js natively supports constructs such as `consts` and arrow functions.

Fortunately, there are other `babel` presets available that let you use what is available in Node.js and change only unsupported ES2015 features. Depending on the version of Node.js, the presets are named differently, corresponding to the version. For Node.js 4.5, it's called `babel-preset-es2015-node4`. Let's now install this and use the preset instead of the default `es2015` preset:

```
$ npm install --save-dev babel-preset-es2015-node4
```

For Node.js version 6, it is `babel-preset-es2015-node6`. Further, the compilation only takes care of the new syntax of ES2015, but does not provide new objects and methods that are part of the ES2015 standard library. For example, you will find that the `method values()` is missing in any array. To enable these objects and methods as you did in the client-side code, we need to include `babel-polyfill` once in the app. Let's include that in the server. The changes are listed in Listing 7-29.

Listing 7-29. `server.js`: Include ES2015 Polyfill

```
...
import 'babel-polyfill';
...
```

Because of the compilation, all errors will now be reported as if they occurred at the line number in the *compiled* file, not the source file. For a better reporting, we need to generate source maps and tell Node.js to use the source maps. Generating source maps is as easy as including a command line switch in the compilation:

```
$ node_modules/.bin/babel server --presets es2015 --out-dir dist --source-maps
```

To let Node.js report line numbers by using source maps, we need to install the `source-map-support` module, and also call the library in the application once. Let's first install the package.

```
$ npm install --save source-map-support
```

To use this, we need to change `server.js` as shown in Listing 7-30.

Listing 7-30. `server.js`: Add Source Map Support

```
...
import SourceMapSupport from 'source-map-support';
SourceMapSupport.install();
...
```

Now, if you introduce an error anywhere in the server, the error message printed on the console will have the line number from the source file instead of the compiled file. You can try this out by temporarily introducing the following line anywhere in `server.js`:

```
...
throw new Error('Test!');
...
```

Now that the server depends on `babel-polyfill` during runtime, we need to move the dependency from `devDependencies` to `dependencies` in `package.json`. While we're at it, let's also change `package.json` to add new commands for compiling the server and watching for changed files, and also change the `start` command to use the compiled file from the `dist` directory. The changes are shown in Listing 7-31.

Listing 7-31. `package.json`: Changes to Scripts and Dependencies

```
...
"scripts": {
  "start": "nodemon -w dist dist/server.js",
  "compile-server": "babel server --presets es2015-node4 ↵
  --out-dir dist --source-maps",
  "watch-server": "babel server --presets es2015-node4 ↵
  --out-dir dist --source-maps --watch",
  ...
}
```



```

"dependencies": {
  "babel-polyfill": "^6.13.0",
  ...
"devDependencies": {
  "babel-polyfill": "^6.13.0",
  ...

```

This also means that we need one more console to run `npm run watch-server`, taking the total number of consoles occupied to three. There is an alternative to static compilation that can do the transformation on the fly. This method is called the *require hook*. That's because it binds itself to Node.js' `require` and dynamically compiles any new module that is loaded. Let's install it to try it out:

```
$ npm install --save-dev babel-register
```

To use it, let's create a separate entry point, load the `babel-register` module, and then load your `server.js` to start the server. Let's call this new file `start_hook.js`; the contents are shown in Listing 7-32.

Listing 7-32. File `start_hook.js`

```

require('babel-register')({
  presets: ['es2015-node4']
});

require('./server.js');

```

We initialized the compiler with a preset (the same as for the static compilation using `babel-cli`) because this module does not load any presets by default. Also, this module automatically ignores libraries loaded from `node_modules`, so we did not have to explicitly mention them to be ignored. To start the server using the `require hook`, let's add a new command in `package.json` for use by `npm run`. The command is as follows:

```
"start-hook": "nodemon -w server server/start_hook.js",
```

Now, use the command line `npm run start-hook` to start the server. Ensure that functionally is all well, as before.

If you are not comfortable running different start and watch scripts on different consoles, it is best to combine the different scripts into one single script that runs all three in the background. You can do this by creating a new script that calls other `npm run <script>` commands, separated by an `&`. The `&` has the effect of placing the script preceding the `&` in background. Let's add two combination scripts for each of the methods that we used. The entire set of changes made to `package.json` in this section is listed in Listing 7-33.

Listing 7-33. package.json: Changes for New Commands and Moving Polyfill

```

...
"scripts": {
  "start": "nodemon -w server server/server.js",
  "start": "nodemon -w dist dist/server.js",
  "compile-server": "babel server --presets es2015-node4 ↵
  --out-dir dist --source-maps",
  "watch-server": "babel server --presets es2015-node4 ↵
  --out-dir dist --source-maps --watch",
  "start-hook": "nodemon -w server server/start_hook.js",
  "compile": "webpack",
  "watch": "webpack-dev-server --hot --inline",
  "dev-all": "npm run watch & npm run watch-server & npm start",
  "dev-all-hook": "npm run watch & npm run start-hook",
  "test": "echo \"Error: no test specified\" && exit 1"
},
...
"dependencies": {
  "babel-polyfill": "^6.13.0",
  "body-parser": "^1.15.2",
  ...
},
...
"devDependencies": {
  "babel-polyfill": "^6.13.0",
  ...
}
...

```

Now, to run all development tasks using the statically compiled method, just use `npm run dev-all`; for the start-hook method, use `npm run dev-all-hook`.

EXERCISE: SERVER-SIDE ES2015

1. We used the es2015-node4 preset rather than the es2015 preset, which would have worked for *all* versions of node, like on the client side. Why did we do this? Hint: Think about the advantages of native code.
2. Compare the pros and cons of the two ways of compiling: statically using babel-cli or dynamically using the require hook.

Answers are available at the end of the chapter.

ESLint

A linter (something that *lints*) checks for suspicious code that may be a bug. It can also check whether your code adheres to conventions and standards that you want to follow across your team to make the code predictably readable.

While there are multiple opinions and debates on what is a *good* standard (tabs vs. spaces, for example), there has been no debate on whether there needs to be a standard in the first place. For one team or one project, adopting *one* standard is far more important than adopting the *right* standard.

ESLint is a very flexible linter that lets you define the rules that you want to follow. Yet, we need something to start off with and the rule set that has appealed to me the most has been that of Airbnb. Part of the reason for its appeal has been its popularity: if more people adopt it, the more standardized it gets, so more people end up following it, and the cycle continues.

To get started, let's install ESLint. Along with it, we need a plugin that understands JSX, and then the Airbnb rule set. Installing the configuration automatically installs all dependencies, which you will be able to see in the console output as well as `package.json`.

```
$ npm install --save-dev eslint-config-airbnb
```

ESLint requires that the configuration and the rules to apply be specified in a starter file, `.eslintrc`. Let's create this file in the project directory, and initialize it by extending the Airbnb rule set. The initial file contents are shown in Listing 7-34.

Listing 7-34. `.eslintrc`: Default `.eslintrc` File Contents

```
{
  "extends": "airbnb"
}
```

To lint your code, you need to run the `eslint` command line against a file like this:

```
$ node_modules/.bin/eslint src/IssueAdd.jsx
```

Or, you can run it against an entire directory, say the server-side code, like this:

```
$ node_modules/.bin/eslint server
```

Since the default extension that ESLint looks for is only `js`, we need to tell it to look for `jsx` files also. Further, we need to tell it to handle the client source directory as well. To run it against *all* the code including the webpack configuration file, we need to do this:

```
$ node_modules/.bin/eslint --ext jsx,js src server webpack.config.js
```

If you do this, you'll see a *lot* of error reports. This is because we never gave full attention to writing good code or sticking to a standard until now. Let's set out to correct the errors and clean them up. At this stage, it's easier to deal with it one file at a time. Alternatively, if your editor supports it, you can install a linter plugin that automatically displays lint errors for you in the editor, so that you can correct them as you type. The popular code editors Atom and Sublime do have plugins to handle this; please follow the instructions on their respective websites to install the plugins.

As for the changes we need to make to fix the lint errors, there are too many to discuss or even show in the listings individually. Other than the major rewrites, I'll only discuss the *types* of changes needed in order to get a clean ESLint report. For individual changes, please refer to the GitHub repository accompanying this book and the differences between this step and the previous. This is one section where I've skipped the listings to show code changes.

For most of the errors, we are just going to change the code to adhere to the suggested standard. But in a few cases, we will make exceptions to the Airbnb rule. The first kind of exception is to override the rule itself, by setting different options for the rule in the `.eslintrc` configuration file. A rule is identified by its name (as printed in the console, for example, `no-use-before-define`). The value is the settings, which is an array. The first element indicates whether the rule is disabled and causes a warning or an error. Further elements are more options to the rule. Here is an example rule set:

```
...
"rules" {
  "no-console": ["off"]
}
...
```

This rule set has only one rule specification, which switches off the rule, thus allowing `console.log` statements throughout the project. It needs no further parameters. But some rules can also take parameters, specifying conditions on which to apply them. For example, to specify that parameter reassignment should be considered an error, except for the parameters properties, this is what you write:

```
"no-param-reassign": ["error", { "props": false }]
```

The second kind of exception that you can make is for temporary code and for things that are really localized, that is, not applicable for the project but for one particular instance of that error. ESLint allows you to make such exceptions on a per-line basis by adding an in-line comment starting with `eslint-disable-line` and optionally specifying which rule to disable a check for (otherwise, it disables checking completely), like this:

```
console.log('Must see message'); // eslint-disable-line no-console-log
```

■ **Note** The errors can be quite different from what I have discussed, based on the version of the Airbnb ESLint configuration (the module `eslint-config-airbnb`) because the configuration is under continuous change and improvement. I have used the configuration version 9.0.1.

Environment

ESLint needs to be told that a certain file is being used in a certain environment. This lets ESLint ignore otherwise invalid global variables such as `document`, which is available in the browser environment. Since you need to specify different environments for the server-side and client-side code, you'll have to create *override* configuration files (again called `.eslintrc`) in each of the directories.

The environment is specified not as a rule, but as a configuration variable at the same level as rules. The configuration takes in an environment name and whether the source is expected to run in that environment (`true/false`). For example, to set the browser environment, you add this configuration:

```
...
  "env": {
    "browser": true
  },
...
```

Now, let's discuss the types of errors and changes we need to do to fix them.

Syntax Consistency

JavaScript is quite flexible in syntax, so there are many ways to write the same code. The linter rules report some errors so that you use a consistent syntax throughout the project.

1. *Missing semicolon:* There's a lot of debate on whether semicolons everywhere or semicolons nowhere is better. Both work, except for a few cases where the absence of semicolons causes a behavior change. If you follow the no-semicolons standard, it requires you to remember those special cases. Let's go with the Airbnb default, which is to require a semicolon everywhere.
2. *Strings must use single quotes:* JavaScript allows both single and double-quotes. In order to standardize, it's better to consistently use one and only one style. Let's use the Airbnb default, single quotes.

3. *Chained calls require a newline per new function:* When chaining calls, especially promises, it improves readability to have each call on a new line. Thus, every `.then` and the final `.catch` need to start on their own new lines.
4. *Prefer shorthand:* When the key and value in an object have the same name, using the shorthand makes the code more concise, and thus easier to read.
5. *Prefer string templates:* String templates improve readability because you don't have a `+` character in the middle.

Editing Convenience

The following rules reduce the chance of an error when you edit code.

1. *Missing trailing comma:* Requiring a comma for the last item in a multi-line array or object is really handy when inserting new items. In addition, when looking at the difference between two versions in say, GitHub, the fact that a comma is added to the last line highlights that the line has been changed, whereas in reality, it hasn't.
2. *Require a curly after an if:* This is useful when you add more lines in the block. It's possible that you may forget to add the curly and you could end up with a dangling line that looks inside the block but is really outside. Forcing a curly avoids this problem.

If you need a very short `if` block, you could write the block in the same line without a curly. The curly is required only if the block is in a new line of its own.

Structural Issues

When the linter points out structural issues, there are very likely bugs lurking underneath.

1. *Variables should be defined before using:* This is a good practice so that all global variables are at the top, and the reader is not confused as to what the variable is all about when reading code top to bottom.
2. *Use `const/let` instead of `var`:* Using `var` to declare variables causes it to be at the function scope rather than block scope. This can lead to undesirable behavior, especially if the name is used again. Using `let` and `const` forces you to think and declare variables in the scope that they belong.

3. *For-in statements are disallowed:* The linter suggests that you guard this with a check for `hasOwnProperty`, since you may unexpectedly get properties if you don't. Note that `for-in` statements are therefore best avoided; you can replace them with a `forEach` iteration on the keys using `Object.keys()`.
4. *Already declared (shadow) variables:* It's best to avoid a variable in an inner scope having the same name as one in the outer scope. It's confusing, and also it hides access to the outer scope variable in case it is required to be accessed.
5. *Function parameter reassignment is disallowed:* This really exposed a bug in the code, and also points out that we should in fact have had two separate functions to validate and clean up the object. In the latter function, we should return a copy of the issue passed in as a function parameter. The two new functions are shown in Listing 7-35 and the changes to the caller (`server.js`) are shown in Listing 7-36.

Listing 7-35. Rewrite of Function `validateIssue` as Two Functions

```
function cleanupIssue(issue) {
  const cleanedUpIssue = {};
  Object.keys(issue).forEach(field => {
    if (issueFieldType[field]) cleanedUpIssue[field] = issue[field];
  });
  return cleanedUpIssue;
}

function validateIssue(issue) {
  const errors = [];
  Object.keys(issueFieldType).forEach(field => {
    if (issueFieldType[field] === 'required' && !issue[field]) {
      errors.push(`Missing mandatory field: ${field}`);
    }
  });

  if (!validIssueStatus[issue.status]) {
    errors.push(`${issue.status} is not a valid status.`);
  }

  return (errors.length ? errors.join('; ') : null);
}

export default {
  validateIssue,
  cleanupIssue,
};
```

Listing 7-36. Changes to server.js for cleanupIssue

```
...
db.collection('issues').insertOne(Issue.cleanupIssue(newIssue)). ←
then(result =>
...

```

But in `IssueList.jsx`, we do need to modify the properties of objects that are passed in during replacement of date strings with `Date` objects in `Issue` records. So, we chose to override the default `no-param-reassign` rule by allowing modification of *properties* alone. The rule for this in the root `.eslintrc` file is as follows:

```
"no-param-reassign": ["error", { "props": false }]
```

React Specifics

These are React-specific issues that are good practices to adopt.

1. *Enforcing propTypes*: Just like declaring function parameter types, it's a good practice to specify the properties and their types that you pass to components from parent to child. I discussed how this can be done in the chapter on React components in the "Passing Data Using Properties" section; now We'll do it for all components.
2. *Stateless functions*: The linter detects the fact that some components don't have any state, and they are better written as stateless functions. We have such a component in `IssueFilter`, but this is only a temporary placeholder. We'll add an in-line exception for this at the end of the line:

```
// eslint-disable-line
```

Rule Overrides

Rather than correct the code, we've chosen to override the default Airbnb rule to not report certain issues as errors.

1. *Dangling _*: These are meant to be internal variables that you're not supposed to use. But MongoDB's `_id` is so convenient for various things that you'll allow the dangling underscore only for this variable. The rule for this is

```
"no-underscore-dangle": ["error", { "allow": ["_id"] }],
```


2. *No console.log*: Typically, this is considered bad practice. In order to insert debugging statements, it is recommended that you use a module such as `debug`. We'll adhere to this for the client side but override this rule for the server-side code, because we do want to see console messages (and maybe save them in a log file) on the server. The rule for this, only in the server configuration, is

```
"no-console": ["off"]
```

3. *No alerts*: The original intention of this rule was to weed out debugging messages that were left unpruned. At least for the foreseeable future of the application, we'll be showing error messages to the user as alerts. Until we find a better way to do this, we'll allow alerts. The rule is part of the client-side `.eslintrc`, and is as follows:

```
"no-alert": ["off"]
```

The final set of `.eslintrc` files after including all the environment specifications and the override rules is shown in the following listings. Listings 7-37 is for the configuration at the root level, that is, common to all code. Listing 7-38 is for the server-side code and Listing 7-39 is for the client-side code.

Listing 7-37. `.eslintrc`: Final root `.eslintrc`

```
{
  "extends": "airbnb",
  "rules": {
    "no-underscore-dangle": ["error", { "allow": ["_id"] }],
    "no-param-reassign": ["error", { "props": false }]
  }
}
```

Listing 7-38. `server/.eslintrc`: Final server `.eslintrc`

```
{
  "env": {
    "node": true
  },
  "rules": {
    "no-console": ["off"]
  }
}
```

Listing 7-39. `src/.eslintrc`: Final client `.eslintrc`

```
{
  "env": {
    "browser": true
  },
  "rules": {
    "no-alert": ["off"]
  }
}
```

Finally, for convenience of running ESLint using `npm` without having to remember all the options that it takes, let's create a command under `scripts` in `package.json` that can be used to run the linter. This change is shown in Listing 7-40.

Listing 7-40. `package.json`: New Command for Running Lint

```
...
"scripts": {
  ...
  "lint": "eslint --ext js,jsx src server webpack.config.js",
  "test": "echo \"Error: no test specified\" && exit 1"
},
...
```

EXERCISE: ESLINT

1. We disabled ESLint checking completely, instead of disabling only the check for stateless functions. Ideally, we should have written `// eslint-disable-line react/prefer-stateless-function`. We didn't. Why not? What would happen if we did?

Answers are available at the end of the chapter.

Summary

Although we did not add any features to the application in this chapter, we made lots of changes to get organized and improve our productivity. Going forward, we'll closely watch lint errors, which help keep the code clean and uniform, but also detect errors that otherwise would only be seen after you run the application and test it. Setting up modularization on both the client and server also lets us keep code for different classes separated.

In the next chapter, we'll go back to adding more features. We'll explore an important concept of client-side routing that will allow us to show different components or *pages* and navigate between them.

Answers to Exercises

Exercise: Transform and Bundle

1. No, webpack does not rebuild if you save a file with just an extra space. This is because the preprocessing or loader stage produces a normalized JavaScript, which is no different from the original. A rebundling is triggered only if the normalized script is different.
2. As of now, we have only one page to display, the Issue List. Going forward, we'll have other pages to render, for example, a page to edit the issue, maybe another to list all users, yet another to show one's profile information, and so on. The file `App.jsx` was created keeping in mind different *page* components that could come up in the future.
3. Not using the `default` keyword has the effect of exporting the class as a property (rather than itself) of the object that's exported. In the import statement, you would have to do this:

```
import { IssueList } from './IssueList.jsx';
```

Note the curly braces around the LHS. This allows multiple objects to be exported from a single file, each object you want from the import being separated by a comma. If there's only one object being exported, you just *default* it, so that the curly braces are not required.

Exercise: Hot Module Replacement

1. There are many logs in the browser's console that tell you that HMR is being invoked. Further, even when the console log window setting "preserve logs across browser requests" is off, you will find that the logs still persist, which means there was no new browser refresh.

If you look at the network requests, you will find that for a browser refresh, requests are made to all of the assets. Take a look at the sizes of these assets. Typically, `vendor.bundle.js` is not fetched again when client-side code changes (it would return a 304 response), but `app.bundle.js` will be reloaded. But when HMR succeeds, you will see that the entire assets are not fetched; instead, incremental files, much smaller than `app.bundle.js`, are being transferred.

2. The main objective of webpack-dev-server is to prevent inadvertent errors, such as those caused by refreshing the browser even before a bundle operation has completed. This can be damaging because you may think you have changed the code, but the new code is not what is running on the browser. HMR is just icing on the cake. I would use webpack-dev-server even if it did not support HMR.

Exercise: Server-Side ES2015

1. Native support for any JavaScript feature is expected to be faster than the transpiled code, which uses ES5 constructs. On the client side, you cannot predict which browsers will be used by the end user, so you must use the least common denominator. Whereas, on the server side, the node version that you are going to use in production is predictable and under your control, and you can indeed take advantage of this.
2. The static compilation alternative requires you to start up two consoles just for the server and ensure that they are functioning OK, whereas the dynamic alternative needs only one console. Although we combined them into one npm script, the output can be intermingled and it may not be easy to spot any errors.

When there are syntax errors in your server code, in the static compilation alternative, the compile step fails, but the server continues to run the old code. This can lead to confusion during development, if you are not monitoring the server compilation console. In the require hook method, the server crashes, and you will be forced to see what happened, because the application will stop functioning.

The static compilation method is faster than the require hook, especially when there are incremental changes to the server code.

Using the static compilation, you are confident that the production code is identical to the development code (require hook is not intended for production, so you have to run the statically compiled code). This removes at least that one variable while chasing bugs in production.

My recommendation is to use static compilation regularly and monitor the console. To me, the fact that the development and production environments are identical is worth this trade-off.

Exercise: ESLint

1. If you disabled that specific rule, the length of the line would have exceeded 100 characters, triggering an error on another rule that restricts the maximum line length. We took a shortcut and disabled all linting for the line, knowing that we'll revisit this class soon.

An alternative is to disable both rules like this:

```
// eslint-disable-line max-len,react/prefer-stateless-function
```



Routing with React Router

Now that we've organized the project and added development tools, let's get back to adding more features to the Issue Tracker. In this chapter, we'll explore the concept of routing, or handling multiple pages that we may need to display. Even when you build a single-page application (SPA), there are in fact multiple *logical* pages (or views) within the application. It's just that the page load happens only the first time; after that, each of the other views is loaded by manipulating or changing the DOM.

To navigate between different views of the application, you need *routing*. Routing links the state of the page to the URL in the browser. It's not only an easy way to reason about what is displayed in the page based on the URL, it has the following very useful properties:

- The user can use the Forward/Back buttons of the browser to navigate between visited pages (actually, *views*) of the application.
- Individual views can be bookmarked and visited later.
- Links to views can be shared with others. Say you want to ask someone to help you with an issue, and you want to send them the link that displayed the issue. Emailing them the link is far easier than asking them to navigate through the user interface.

Before SPAs really matured, this was rather difficult, or SPAs just didn't let you do this. They had just a single page, which meant a single URL. All navigation had to be interactive; the user had to go through the application via predefined steps. You couldn't, for example, send someone a link to a specific issue; you had to tell the recipient to follow a sequence of steps on the SPA to reach that issue. But modern SPAs handle this gracefully.

In this chapter, we'll explore how to use React Router to ease the task of setting up navigations between views. We'll add another view to the application, one where the user can see and edit a single issue. Then, we'll create links between the two views so that the user can navigate between them. On the hyperlinks that we create, we'll add parameters that can be passed to the different views, for example, the ID of the issue that needs to be shown, to a view that shows a single issue. Finally, we'll see how to nest components and routes.

Routing Techniques

In order to effect routing, we first need to connect a page to something that the browser recognizes and indicates that “this is the page that the user is viewing.” In general, for SPAs, there are two ways to make this connection:

- Hash-based: This uses the anchor portion of the URL (everything following the #). This method is natural; you can think of the # portion as a location *within* the page, which is an SPA. And this location determines what section of the page is displayed. The portion before the # never changes; it is the one and only *page* (`index.html`) that is returned by the back end. This is simple to understand, and works well for most applications. In fact, implementing hash-based routing without a routing library is quite simple (but we won’t do it).
- Push state, also known as browser history: This uses a new HTML5 API that lets JavaScript handle the page transitions, at the same time preventing the browser from reloading the page when the URL changes. This is a little more complex to implement even with help from React Router (because it forces you to think about what happens when the server gets a request to the different URLs). But it’s quite handy when we want to render a complete page from the server itself, especially to let search engine bots get the content of the pages and index them.

We’ll start with the hash-based technique because it’s easy to understand, and then switch over to the browser history technique because eventually we’ll do server-side rendering.

Simple Routing

In this section, we’ll create two routes, one for the issue list that we’ve been working on all along, and another (a placeholder) for viewing and editing a single issue. To start, let’s install the package, React Router:

```
$ npm install --save-dev react-router
```

We need to also remember to add any new front-end packages that we install, to the vendor section of the webpack configuration. This is so that they don’t get included in the application’s bundle; instead, they go into the vendor bundle. This change is shown in Listing 8-1.

Listing 8-1. webpack.config.js: Add react-router as a Vendor Library

```
...
module.exports = {
  entry: {
    app: './src/App.jsx',
    vendor: ['react', 'react-dom', 'whatwg-fetch', 'react-router'],
  },
  ...
}
```

Let's also create a placeholder component for editing an issue. Listing 8-2 shows the complete code for the placeholder component, saved as `IssueEdit.jsx` in the `src` directory.

Listing 8-2. IssueEdit.jsx: Placeholder for a Component to Edit Issues

```
import React from 'react';

export default class IssueEdit extends React.Component {
  // eslint-disable-line
  render() {
    return (
      <div>This is a placeholder for the Issue Edit page.</div>
    );
  }
}
```

React Router works by taking control of the main component that is rendered in the DOM. So, instead of rendering the `App` component in the content node, we need to render a `Router` component. To the `Router` component, we supply the configuration that includes the paths and the component associated with the path. The configuration is a hierarchy of React components called `Route`, with properties specifying the path and the component associated. The `Router` component then renders these different components when the URI changes. Listing 8-3 shows the changed `App.jsx` code for this.

Listing 8-3. App.jsx Rewritten with Router

```
import 'babel-polyfill';
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, Route, hashHistory } from 'react-router';

import IssueList from './IssueList.jsx';
import IssueEdit from './IssueEdit.jsx';

const contentNode = document.getElementById('contents');
const NoMatch = () =><p>Page Not Found</p>;
```



```

const RoutedApp = () => (
  <Router history={hashHistory}>
    <Route path="/" component={IssueList} />
    <Route path="/issueEdit" component={IssueEdit} />
    <Route path="*" component={NoMatch} />
  </Router>
);

ReactDOM.render(<RoutedApp />, contentNode);

if (module.hot) {
  module.hot.accept();
}

```

Let's start by looking at the `render()` call. Instead of rendering the `IssueList` component, we are now rendering a `RoutedApp`. The only property for this component is the kind of history to use, and we used a hash history, which we imported from `react-router`. Nested within the `Router` are multiple `Route` components, each with a path and a component to display for that URL path. We also introduced a new component for showing unmatched routes, and this is associated with the fallback path, `"*`, indicating any path. The other changes are to import appropriate classes and components.

To see the Issue List page, the usual URL will work, since the path `"/"` is associated with the `IssueList` component in the first route. But to see the Issue Edit page at this point in time, you need to manually type in `http://localhost:8000/#/issueEdit` in the URL bar of the browser. Later, when we implement a hyperlink from the Issues List, this will become easier. Further, you can see that the Back and Forward buttons can be used to navigate between the two pages. Finally, if you change `issueEdit` to any other word in the URL, it shows a Page Not Found error, which is the fallback component matching the path `"*`.

Route Parameters

We saw that the route path can take wildcards like `"*`". In fact, the path can be a complex string pattern that can match optional segments and even specify parameters like REST API paths. Let's reorganize the paths so that

- `/issues` shows the list of issues
- `/issues/<id>` takes the user to the edit page of the issue with the corresponding `id`
- `/` redirects to `/issues`

A route parameter is specified using a `:` in front of the parameter. Thus, `/issues/:id` will match any path that starts with `/issues/` followed by any string. The value of that string will be available to the component of the route in a property object called `params`, with the key `id`. To implement a redirect, React Router provides a `Redirect` component. Listing 8-4 shows the partial listing of the modified Router configuration which uses a redirect and a route parameter.

Listing 8-4. Modified Router Configuration with Parameters and Redirect

```

...
import { Router, Route, Redirect, hashHistory } from 'react-router';
...
const RoutedApp = () => (
  <Router history={hashHistory} >
    <Redirect from="/" to="/issues" />
    <Route path="/issues" component={IssueList} />
    <Route path="/issues/:id" component={IssueEdit} />
    <Route path="*" component={NoMatch} />
  </Router>
);
...

```

Further, let's also add links between the two pages. In the Issue List, let's change the ID column to be a hyperlink to the Issue Edit page, with the destination URL including the ID. Then, another link from the Issue Edit page to take the user back to the Issues List page. To achieve this, we need to use the Link component from React Router, which is very similar to the HTML <a> tag. Listing 8-5 shows the new IssueEdit.jsx contents, and Listing 8-6 shows the changes to IssueList.jsx for these links. Let's also include the ID of the issue in the placeholder page to ensure that we can access the parameters in the IssueEdit component.

Listing 8-5. IssueEdit.jsx: Modifications for Adding Link and Accessing Parameters

```

import React from 'react';
import { Link } from 'react-router';

export default class IssueEdit extends React.Component {
  { // eslint-disable-line
    render() {
      return (
        <div>
          <p>This is a placeholder for editing issue {this.props.params.id}</p>
          <Link to="/issues">Back to issue list</Link>
        </div>
      );
    }
  }

  IssueEdit.propTypes = {
    params: React.PropTypes.object.isRequired,
};

```

The important changes in `IssueEdit.jsx` are the display of the id using `this.props.params.id` and the use of the `Link` component. The other changes are an import statement to make `Link` available, and a property definition to avoid link errors.

Listing 8-6. `IssueList.jsx`: Changes for Adding a Link on Each Issue's ID

```
...
import 'whatwg-fetch';
import { Link } from 'react-router';
...
<tr>
  <td><Link to={`/issues/${props.issue._id}`} ~
    {props.issue._id.substr(-4)}
  </Link></td>
  <td>{props.issue.status}</td>
</tr>
...
```

We used the `Link` component of React Router to create a hyperlink. This can be used in place of the `<a>` tag that you use in normal HTML, with the target of the hyperlink specified in the `to` attribute. Apart from the hyperlink, we also shortened the display to only the last four characters using the `substr` function on the ID. If we need to see the full ID, we can hover over the link and see it in the status bar of the browser.

When you test this set of changes, you'll see that there is a hyperlink on each of the ID fields in the issue list. On clicking these, you are taken to the placeholder page, which displays the ID of the issue that you clicked, as shown in Figure 8-1. You can also check if the index page (i.e., `/`) redirects to the issue list.

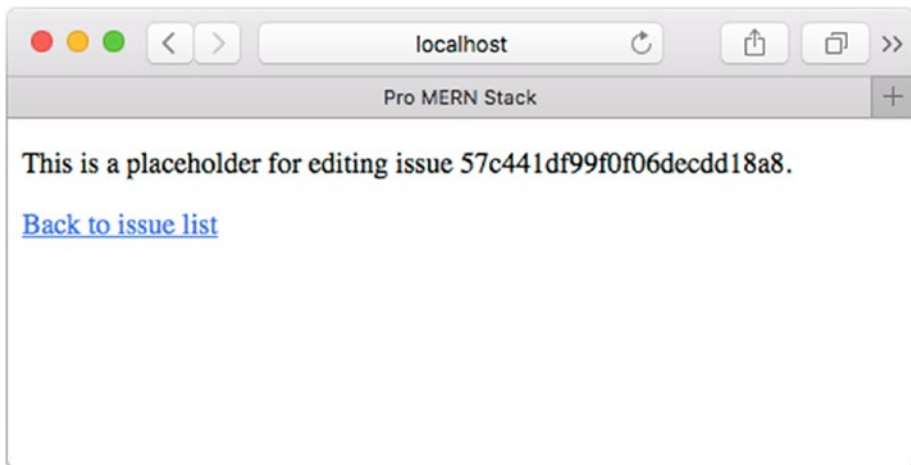


Figure 8-1. Placeholder page

We can fetch the issue object from the server using an AJAX call and set the state in `componentDidMount()` method as we did for the issue list. Editing involves a form, which I'll cover in the next chapter, so we'll leave it as a placeholder for now.

Apart from the component that is displayed depending on the route or the URL in the browser, there are other properties that you can set on the route. The `onLeave` property is particularly useful to alert the user if any changes have been made in a particular page, and the user is navigating away from the page without saving the changes. The `onLeave` property in a route lets you specify a function that can be called on this event. Other events that you can get notified for are `onEnter` (called when the user enters a route) and `onError` (when errors are encountered when matching a route).

EXERCISE: ROUTE PARAMETERS

1. Instead of `<Link>`, we could have used a simple `<a href ...>`. Why is the latter not recommended? Hint: Read up the API documentation of React Router for the `<Link>` component.

Answers are available at the end of the chapter.

Route Query String

Apart from the parameters that are part of the URL path, React Router also parses the query string, if any, and makes it available to the component in a property object called `location`. This object contains various keys, including the path, the unparsed query string, and a parsed query object.

Query strings are ideal for specifying a filter to the issue list. We'll use the same convention for both the Route URL's query string as well as the REST API query string: a set of key-value pairs where the key is the name of the field to filter on, and the value of the field to match. For example, `?status=New` will match all issues with the New status, and `?status=Open&owner=Vasan` will fetch the issues that match both conditions. For now, we'll implement a single filter field in the UI.

First, the back-end API needs to be changed to handle this. You already learned about filters in MongoDB's `find()` method in the MongoDB chapter. You also learned in the Express REST APIs chapter how to extract the query parameters from the Express Request object. Let's use these techniques to change the Issue List API. The modified API is shown in Listing 8-7.

Listing 8-7. `server.js`: Modified Issue List API, with One Filter

```
app.get('/api/issues', (req, res) => {
  const filter = {};
  if (req.query.status) filter.status = req.query.status;
  ...
});
```

The modified API can be tested using curl. You should be able to verify that only a subset of the issues is fetched, those that match the filter. The curl command is like this:

```
$ curl -s http://localhost:3000/api/issues?status=Open | json_pp
```

The next step is to integrate this into the client-side code. We'll first create links to a few hard-coded filters. The filter should ideally be a form where the user can choose different combinations, but since you haven't learned about forms yet, we'll leave that to the next chapter. We'll rewrite the `IssueFilter` placeholder component as shown in Listing 8-8.

Listing 8-8. `IssueFilter.jsx`: Rewritten with Hardcoded Filters

```
import React from 'react';
import { Link } from 'react-router';

export default class IssueFilter extends React.Component {
  // eslint-disable-line
  render() {
    const Separator = () => <span> | </span>;
    return (
      <div>
        <Link to="/issues">All Issues</Link>
        <Separator />
        <Link to={{ pathname: '/issues', query: { status: 'Open' } }}>
          Open Issues
        </Link>
        <Separator />
        <Link to="/issues?status=Assigned">Assigned Issues</Link>
      </div>
    );
  }
}
```

We created multiple links using variations in the query string. There are two ways to supply a query string to a `Link`, and I've used both, just to demonstrate the variation. The first method is more verbose but convenient when you need to generate multiple links programmatically. The second one is good for hard-coded query strings.

Finally, we need to pass the filter to the REST API call while loading the list of issues in `IssueList` component, in the call to `fetch()`. Since we kept the query string format same for the REST API as well as the URL, we just have to pass along the query string as is to the REST API, like in Listing 8-9.

Listing 8-9. `IssueList.jsx`: Changes to Pass a Filter to the REST API in `loadData()`

```
...
  fetch(`api/issues${this.props.location.search}`).then(response => {
...

```

The change from single quotes to back-ticks is subtle, to make it a template string. Make sure you don't miss this change. We don't use a `?` character because `location.search` already includes it. Now, when you navigate using the hyperlinks for each hard-coded filter, you will find that the list of issues doesn't change, even though the browser URL reflects the new query string. But it does work when you refresh the browser.

The reason is that we are calling `loadData()` only when the component is mounted. When there is a change in the query string, the component is not mounted again; instead, React reuses the component that's already mounted. On a browser refresh, the entire page is recreated, and therefore the component is mounted. It is also mounted when navigating between routes, for example, when you click on the ID of an issue to edit it and then press the back button, you will see that the correct list is loaded. Compared to that, the change in the query string is only a change in one of the properties, which doesn't warrant a remounting of the component.

I talked briefly about component lifecycle methods earlier. These are hooks into various changes that React does on the component. We used the lifecycle method `componentDidMount()` to hook into the initially ready state of the component. Similarly, we need to hook into a method that tells us that the route query string has changed, so that we can reload the list. The Component Lifecycle guide for React Router recommends we do it in the lifecycle method `componentDidUpdate()`, which gives us a hook whenever any property of the component changes. Let's do so; see Listing 8-10.

Listing 8-10. `IssueList.jsx`: Handle the Lifecycle Method `componentDidUpdate`

```
...
  componentDidUpdate(prevProps) {
    const oldQuery = prevProps.location.query;
    const newQuery = this.props.location.query;
    if (oldQuery.status === newQuery.status) {
      return;
    }
    this.loadData();
  }
...
IssueList.propTypes = {
  location: React.PropTypes.object.isRequired,
};
...
```

Now, testing will show you that navigating between the filter links indeed reloads the data. A screenshot of the main Issue List page is shown in Figure 8-2.

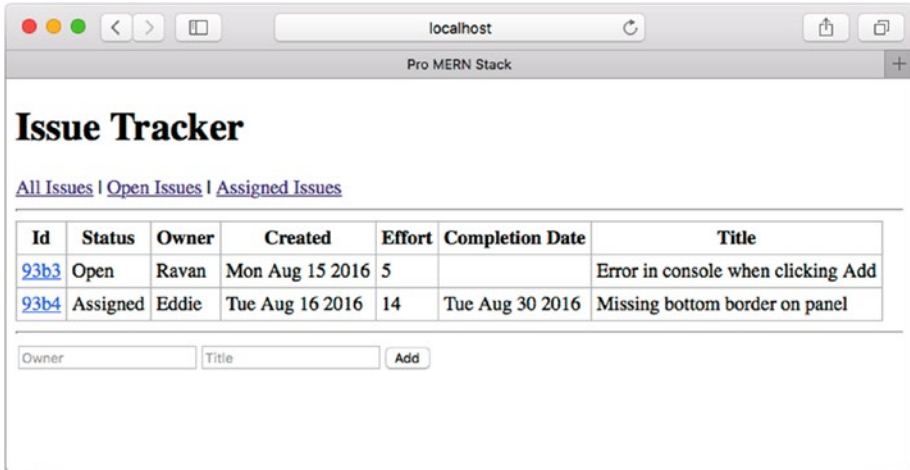


Figure 8-2. Issue list with hardcoded filters as hyperlinks

EXERCISE: ROUTE QUERY STRING

1. Create a bookmark for one of the issues (or copy/paste an issue's ID in the URL bar). Navigate between two different issues' edit pages. It works. Why didn't we need the lifecycle method hook in this case? Hint: Think about how a change in properties affected the Issue List vis-à-vis the Issue Edit page.
2. Which other lifecycle method could have been used in place of `componentDidUpdate()`? Hint: Read the "Component Lifecycle" section in the official documentation of React.
3. What happens if we remove the check for old and new queries being the same? That is, we always reload the data whenever the component updates?
4. Add a log message in the `render()` call of `IssueAdd`. You will find that this component is being rerendered when the filter changes. What are the performance implications? How can this be optimized? Hint: Read the "Component Lifecycle" section in the documentation on React.

Answers are available at the end of the chapter.

Programmatic Navigation

We are using the React Router's `Link` component to navigate using a hyperlink. But eventually we'll convert this hard-coded filter into a form that takes user inputs with an `Apply` button that constructs the query string programmatically. In this case, we can't use a `Link`; instead, we must navigate by changing the browser's URL using code.

React Router has a `router` object that can be passed around to components. This object has many methods, including methods for setting hooks on entering and leaving a route, to programmatically generate query strings given a route, but most importantly, to programmatically push into the history or replace the current route. Let's use the `router.push()` method to set the filter programmatically.

But the `router` object is not available to all the components automatically. There are two ways to get access to this. The first is using React's `Context` feature. This feature allows a property to be available in deeply nested components without having to explicitly pass it through each and every component in the hierarchy. You can read up more on this at the React's official documentation, but as described there, `Context` is an experimental feature that you should avoid when there are alternatives.

The second method is by injecting the `router` property into the components which need it, using React Router's `withRouter` method. This method wraps a given component and makes the `router` property available. This is described in detail in React Router's API documentation under `withRouter` and `<RouterContext>`. Let's make these changes in `App.jsx`, as shown in Listing 8-11.

Listing 8-11. `App.jsx`: Changes to Inject `router` to `IssueFilter`

```
...
import { Router, Route, Redirect, useHistory, withRouter } ←
from 'react-router';
...
<Route path="/issues" component={withRouter(IssueList)} />
...
```

Now, `IssueList` can use `this.props.router` to access the `router` object. Let's add a method in `IssueList` to set a new filter, given a set of field-value pairs, and use the `router` to change the URL based on the filter. We'll pass this method along to `IssueFilter` as a property for it to call when required. The changes to `IssueList` are shown in Listing 8-12.

Listing 8-12. `IssueList.jsx`: Changes to Use `Router` and Push a Filter

```
...
constructor() {
  ...
  this.setFilter = this.setFilter.bind(this);
}
```



```

...
  setFilter(query) {
    this.props.router.push({ pathname: this.props.location.pathname, query });
  }
...
    <h1>Issue Tracker</h1>
    <IssueFilter setFilter={this.setFilter} />
    <hr />
...
IssueList.propTypes = {
  location: React.PropTypes.object.isRequired,
  router: React.PropTypes.object,
};
...

```

The new `setFilter` method takes in a query object like `{ status: 'Open' }` and uses the `push` method of `router` to change only the query string part, keeping the `pathname` the same as before. We could have also passed in a string like `'/issues/?status=Open'` by constructing the query string, but that would mean writing some code to escape unsafe URL characters in the value of the filter field.

We had to bind the method to `this` in the constructor, because the method needs access to `this.props.router`. Also, for property validation, we added the `router` property as an object in the `propTypes` declaration. Finally, we passed the `setFilter` method as a property to `IssueFilter`.

To use the new programmatic way of setting the route, we'll rewrite `IssueFilter`. Instead of `<Link>`s, let's use regular `<a>` tags and in the `onClick` event of these anchor tags, we can programmatically set the filter or clear the filter. To do this, we'll have to use the `setFilter` method passed into this component via `props`. The entire new file is listed in Listing 8-13.

Listing 8-13. `IssueFilter.jsx`: Rewrite to Handle Navigation Programmatically

```

import React from 'react';

export default class IssueFilter extends React.Component {
  constructor() {
    super();
    this.clearFilter = this.clearFilter.bind(this);
    this.setFilterOpen = this.setFilterOpen.bind(this);
    this.setFilterAssigned = this.setFilterAssigned.bind(this);
  }

  setFilterOpen(e) {
    e.preventDefault();
    this.props.setFilter({ status: 'Open' });
  }
}

```

```

setFilterAssigned(e) {
  e.preventDefault();
  this.props.setFilter({ status: 'Assigned' });
}

clearFilter(e) {
  e.preventDefault();
  this.props.setFilter({});
}

render() {
  const Separator = () =><span> | </span>;
  return (
    <div>
      <a href="#" onClick={this.clearFilter}>All Issues</a>
      <Separator />
      <a href="#" onClick={this.setFilterOpen}>Open Issues</a>
      <Separator />
      <a href="#" onClick={this.setFilterAssigned}>Assigned Issues</a>
    </div>
  );
}
}

IssueFilter.propTypes = {
  setFilter: React.PropTypes.func.isRequired,
};

```

One thing to note in Listing 8-13 is the call to `e.preventDefault()`, which prevents the default action on clicking the hyperlink. This is required, just like in the `IssueAdd` component; otherwise the default action will be executed, which would be to set the URL to `#`. On testing this set of changes, you should see that the behavior is no different from the previous section.

EXERCISE: PROGRAMMATIC NAVIGATION

1. There is yet another way to navigate programmatically. What are the pros and cons of using that method? Hint: Read React Router's 2.0.0 upgrade guide to find the other method.
2. Change `router.push()` to `router.replace()`. What difference do you see? When would you use one versus the other? Hint: Play around with the Back/Forward browser buttons to discover the difference.

Answers are available at the end of the chapter.

Nested Routes

Most applications have a common header and footer across all views or pages. The header typically has the brand icon, the application, or website name, sometimes a menu, and also a search box for quick access to different parts of the application. The footer usually has links to useful information regarding the application.

In the Issue Tracker application, we just had a header in the issue list itself, which was not shared with the edit page. Let's create a common decoration for all pages across the application. To do this, we can use nesting of routes. This way, a component can have different children depending on the route. We'll use only one level of nesting, where a main component has a header, the contents section, and a footer. The exact component that is rendered in the contents section will depend on the route. Thus, the *root* route will be `/`, the contents section will render a list of issues when the route is `/issues`, or a single issue when the route is `/issues/<id>`.

To configure this, all we need to do is nest the routes in the router configuration, with paths relative to the parent route. Note that although we use only one level of nesting, React Router allows infinite levels. A great example is described in the "Introduction" section of the React Router documentation.

We need to create a new component corresponding to the decorator that holds the header and footer. Let's call this component `App`. React Router will pass via `props.children` the child component resolved as a result of route matching. Let's place it within the rendering function at the place where we want the contents to be displayed: a `<div>` with class `contents`. As for the header and footer, we'll just create `<div>`s and show a simple text. This can later be expanded to hold maybe a navigation menu and/or a search box.

As for the Router configuration, we need to change it so that `/` is mapped to the new `App` component, and the other routes are nested within this. Listing 8-14 shows the new contents of the file `App.jsx`.

Listing 8-14. `App.jsx`: Rewritten to Decorate and Nest Routing

```
import 'babel-polyfill';
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, Route, Redirect, hashHistory, withRouter } from 'react-router';

import IssueList from './IssueList.jsx';
import IssueEdit from './IssueEdit.jsx';

const contentNode = document.getElementById('contents');
const NoMatch = () =><p>Page Not Found</p>;

const App = (props) => (
  <div>
    <div className="header">
      <h1>Issue Tracker</h1>
    </div>
```

```

    <div className="contents">
      {props.children}
    </div>
    <div className="footer">
      Full source code available at this <a href= ↵
      "https://github.com/vasansr/pro-mern-stack">
      GitHub repository</a>.
    </div>
  </div>
);}

App.propTypes = {
  children: React.PropTypes.object.isRequired,
};

const RoutedApp = () => (
  <Router history={hashHistory} >
    <Redirect from="/" to="/issues" />
    <Route path="/" component={App} >
      <Route path="issues" component={withRouter(IssueList)} />
      <Route path="issues/:id" component={IssueEdit} />
      <Route path="*" component={NoMatch} />
    </Route>
  </Router>
);

ReactDOM.render(<RoutedApp />, contentNode);

if (module.hot) {
  module.hot.accept();
}

```

Now, we can remove the heading from `IssueList`, because this is now part of the decorator. This change is shown in Listing 8-15.

Listing 8-15. `IssueList.jsx`: Removal of Heading

```

...
    <div>
      <h1>Issue Tracker</h1>
      <IssueFilter setFilter={this.setFilter} />
    </div>
...

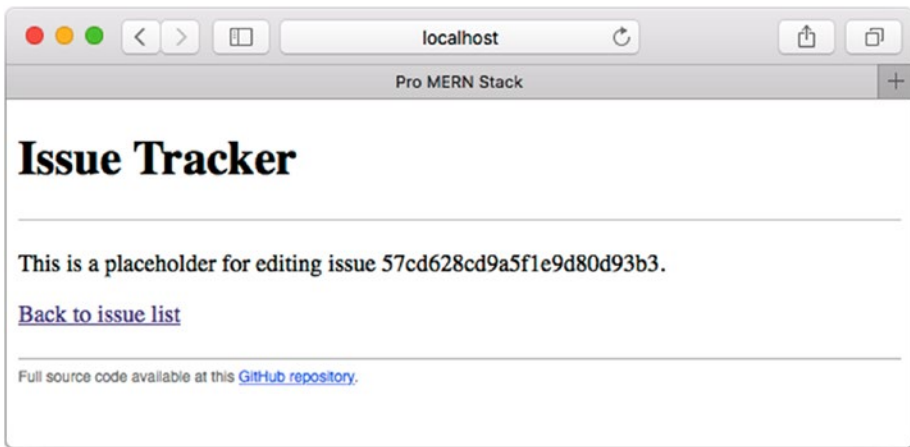
```

Finally, there are some trivial styles added to space the headers and footers for better readability. This is done in `index.html`; the changes are listed in Listing 8-16.

Listing 8-16. index.html: Styles Added for Header and Footer

```
...
.header {border-bottom: 1px solid silver; margin-bottom: 20px;}
.footer {
  border-top: 1px solid silver; padding-top: 5px; margin-top: 20px;
  font-family: Helvetica; font-size: 10px; color: grey;
}
...
```

When you test this, you can see that the plain-looking edit page now has a header and footer. What's more, even a Page Not Found error triggered by an invalid route has these decorations. The new edit page is shown in Figure 8-3.

**Figure 8-3.** Edit page, decorated

We redirected “/” to “/issues”, which means that we did not have an independent view or route for “/” itself. There can be use cases where we don’t want to do this and instead display, say, a Home page or a Help page. Let’s imagine a dashboard of sorts when a user lands at the Home page that gives a summary of all the issues at hand. To indicate that a particular component needs to be displayed if only the parent route is matched, we can use the `IndexRoute` component like this:

```
...
<Redirect from="/" to="/issues" />
<Route path="/" component={App} >
  <IndexRoute component={Dashboard} />
  <Route path="issues" component={withRouter(IssueList)} />
...
```

Without the redirect, the `/`, or the *index* of the parent route would have been an App without any children. But with an `IndexRoute` added, the URL with just a `/` will display the Dashboard component within the App. In the Issue Tracker, we don't have a need for an `IndexRoute`, so we won't implement it.

There is also an `IndexRedirect` component that is handy if you want to specify a redirect to the index. What we achieved using `Redirect` *outside* the root route could have been achieved using an `IndexRedirect` within the route too. Both methods are acceptable, but as you'll see later, `IndexRedirect` has some advantages due to the fact that it makes the entire route self-contained.

Browser History

Using the browser history method is recommended for two reasons. For one, the URL looks cleaner and easier to read. But most applications can live with the hash-based routing, because users rarely look at the URL or try to make sense out of it, so this reason is not compelling enough to switch to the browser history method. The other, more important, reason is that if you need to do server-side rendering (something that will help search engine bots index your pages correctly), the hash-based method will just not work.

That's because the hash-anchor is just a location *within* the page, not something that a bot will crawl. Take, for example, the links to each issue. Say these are "pages" that you want search engines to index. The only way the bots can get to them is via the Issues List page. But since the links are in-page anchors, the bots will not make a request to the server to fetch them. In fact, they cannot, since anything after the `#` is stripped out when identifying a URI.

Let's switch to the browser history method in anticipation that you want the application to be indexed by search engines. At first glance, it looks as if replacing `hashHistory` with `browserHistory` is all we need to do to start using the browser history method, but it's not as simple as that.

Whenever you hit the refresh button until now, the first request that went to the server was always to `/`. That's because all URLs ended at `/` before the `#`. Now, imagine a URL such as `/issues` rather than `/#/issues`. When navigating between routes, React Router ensures that a new component is mounted. But when you hit refresh on the browser, if the URL were `/issues`, the browser will try to fetch `/issues` (note: *not* `/api/issues`) from the server.

How do we deal with this? What are we supposed to return for the request to `/issues`? It turns out that the server needs to return the one and only real *page* in an SPA, `index.html`, regardless of the request (except, of course, the REST API calls). And then, React Router will take care of mounting the right components based on the URL.

One obvious place where we have to do this is the Express server. But we also have a webpack dev-server that we use during development. That too needs to be configured to handle this correctly. It turns out that webpack developers are already familiar with the use case, so they let you do it with a simple flag called `historyApiFallback`.

Let's carry out these server-side changes and then finally flip the switch to browser history in the client side. Listings 8-17 through 8-19 show the changes in each file that must be made to achieve this.

Listing 8-17. server.js: Return index.html for Any Request

```
...
import path from 'path';
...
app.get('*', (req, res) => {
  res.sendFile(path.resolve('static/index.html'));
});
...
```

The new express route has to be placed *after* all the other routes, so that it gets resolved only if none of the previous routes match. Also, we need to use the `resolve` method from the `path` module because `sendFile` accepts only absolute paths. We can't give it a path relative to the current directory.

Listing 8-18. webpack.config.js, devServer section: Fallback to Server

```
...
devServer: {
  ...
  proxy: {
    ...
  },
  historyApiFallback: true,
},
...
```

Listing 8-19. App.jsx: Flip the Switch to `browserHistory`

```
...
import { Router, Route, Redirect, browserHistory, withRouter } from 'react-router';
...
<Router history={ browserHistory } >
...
```

When you restart the webpack dev-server, you will notice a small change in webpack dev-server's output. It prints out the following in the console:

```
...
404s will fallback to /index.html
...
```

That's comforting, because we now know that for any request that doesn't match the proxy path or having static content, the dev-server will return `index.html`. And that's what we want. Let's test it in development mode, that is, by running the webpack dev-server. Not only should navigations work, but browser refreshes also should work.

Further, to test the production mode, you need to compile the client-side code, compile the server-side code, and then start the server. This too should work when navigating between pages and when refreshing. Finally, do check that the browser URL has no hashes, and it is a clean URL in each case.

Summary

In this chapter, you learned about an important aspect of single page applications: routing. We used the popular React Router library to achieve this, and we routed between two pages: Issue List and a placeholder for Issue Edit. More importantly, you learned how to add more routes if required, and also to deal with parameters and query strings, which act as input parameters to the pages.

We used a form as part of Issue Add, but we used it in the conventional HTML way. React has another way of dealing with forms and form inputs, called *controlled* forms, which can connect to state variables in components to achieve *two-way binding* between form fields and state variables. We'll look at this technique in the next chapter when we complete the Issue Edit page.

Answers to Exercises

Exercise: Route Parameters

1. One useful feature that `Link` gives us is the ability to set an active class. We can style a link differently based on whether the currently active route matches that link. This is useful in displaying menus or tabs that have multiple links, with the current link automatically highlighted.

Another effect of using a `Link` is that we don't need to be aware of the underlying mechanism of routing: it could be hash-based or it could be browser history-based. The path is the same.

Exercise: Route Query String

1. When properties are changed in a component, a `render()` is automatically called by React. The call to `render` now has access to the new properties. When a change in properties only affects rendering, we don't need to do anything more.

The difference in the `IssueList` was that the change in properties caused a change in the state. Further, this change was asynchronous. This change had to be triggered somewhere, and we chose the Lifecycle method `componentDidUpdate()` to do that. Eventually, even in `IssueEdit`, when we load the issue details in an asynchronous call to the server, we will have to hook into the Lifecycle method.

2. We could have used `componentWillReceiveProps()` instead of `componentDidUpdate()`. Both are OK to use, and React documentation recommends `componentDidUpdate()`. I like `componentWillReceiveProps()` because it is more indicative of what's happening; it makes the code more readable.
3. If we don't have the check for old and new properties being the same, we end up in an infinite loop. That's because setting a new state is also considered an *update* to the component, and this will trigger a new `loadData()`, which will do a `setState()` again, and the cycle will continue endlessly.
4. Any change in a parent component will trigger a render in the child, because it is assumed that the state of the parent can affect the child as well. Normally, this is not a problem, because the DOM itself is not updated; it is only the *virtual* DOM that is updated. React will not update the DOM, seeing no differences in the old and new virtual DOMs.

Updates to the virtual DOM are not that expensive, because they are no more than data structures in memory. But, in rare cases, especially when the component hierarchy is very deep and the number of components affected is very large, just the act of updating the virtual DOM may take a wee bit of time. If you want to optimize for this, React provides the Lifecycle method `shouldComponentUpdate()`, which lets *you* determine if an update is warranted.

Exercise: Programmatic Navigation

1. The other method is by directly importing the singleton `hashHistory` and using its `push` method to effect the navigation. We could have done this in `IssueList` instead of having to inject the router property into the component.

This is convenient, especially if we were to do the navigation in `IssueFilter` itself, which is one level deeper in the hierarchy: we would have had to pass the router property further down. But the advantage is that we don't have to hardcode the fact that you are using `hashHistory` throughout the application. Using the `withRouter` method, we can keep `IssueList` and other components agnostic to the kind of history we're using.

Note that `withRouter` and importing of the singleton `hashHistory` is available only from version 2.4.0 of React Router. Older versions, especially 1.x, had different techniques to do programmatic navigation.

2. The method `router.replace()` *replaces* the current URL such that the history does not have the old location. `router.push()` ensures that the user can use the Back button to go back to the previous view. Replacing can be used when the two routes are not really different. It is analogous to a HTTP redirect, where the contents of the request are the same, but available at a different location. In this case, you really don't want to remember the history of the first location.

CHAPTER 9



Forms

User input is an important part of any web application, and the Issue Tracker is no different. We had some user input when we used a form to create a new issue. But it was very rudimentary, and it did not demonstrate how forms are supposed to be dealt with in React.

In this chapter, we'll start taking in a lot more user input. We will convert the hard-coded filter to something more flexible with user input, and then we'll fill in the Edit page with a form, and finally, we'll add the ability to delete issues from the Issue List page. To be able to do this, we'll also have to modify the back-end APIs to support these functions. We'll modify the List API to take more filter options, and we'll create new Update and Delete APIs. We'll thus complete implementing all of the CRUD actions.

More Filters in the List API

The List API already has one filter field for the status of the issue. Let's add another, this time the integer field, effort, so that we can explore how different data types can be dealt with. We'll implement a from-to criterion for this field rather than an equality.

The difference between an equality operator and other comparison operators is that the other operators need to be specified as an object with the key being the operator, such as `{effort: {$gte: 0}}`. This way, multiple operators can be added for the same field. For the effort field, we need a greater than or equal to value, and also a less than or equal to value, both being optional. If both were specified, it would look like `{effort: {$gte: 0, $lte: 10}}`. If neither were specified, the filter wouldn't have an effort key at all. Further, we need two parameters in the query string to specify each of the values: the minimum and the maximum. Let's call them `effort_gte` and `effort_lte` to match the names of the MongoDB operators.

Another thing to note is that MongoDB is strict about data types. If we supply a string instead of a number, the match will return no documents. So, we have to convert the query parameters to an integer before setting the value in the filter.

Listing 9-1 shows the above changes in the List API.

Listing 9-1. server.js: Add Effort Filter to List API

```
...
const filter = {};
if (req.query.status) filter.status = req.query.status;
if (req.query.effort_lte || req.query.effort_gte) filter.effort = {};
if (req.query.effort_lte) filter.effort.$lte = ~
parseInt(req.query.effort_lte, 10);
if (req.query.effort_gte) filter.effort.$gte = ~
parseInt(req.query.effort_gte, 10);

db.collection('issues').find(filter).toArray()
...
```

These changes can be tested using curl as follows:

```
$ curl 'http://localhost:3000/api/issues?effort_lte=16&effort_gte=4'
```

Test with each comparison as a single parameter as well as both together to make sure the API returns the expected documents.

EXERCISE: MORE FILTERS IN THE LIST API

1. If you haven't added any issues other than the schema-initialized default ones, add them using the Add form. If you run the curl test, you will find that the added documents are never returned. Why? Hint: Read up on the \$gte operator reference in the MongoDB documentation.
2. If we do need documents without an effort to be returned always, what would the filter look like? Hint: Read up on the \$or operator reference in the MongoDB documentation.

Answers are available at the end of the chapter.

Filter Form

One of the challenges that a declarative style of programming faces is user interaction in form inputs. If, like, conventional HTML code, you set the value of a text `<input>` to a string, it means that the value is *always* that string because you *declared* it so. If you let the user edit the value, it means that any rerendering destroys the user edits and restores it to that original string.

To avoid that from happening, you have two options. The first is to supply no value to the input, which is what we did with the `IssueAdd` component. Whenever you need the user input, you get it from the HTML component, the conventional way. This may be

OK as long as you don't have to initialize the inputs with some value, as in the case of the `IssueAdd` component. In this approach, there is still an internal state of the component (the current state of the user-typed value), but it is not tracked by any state variable in React components. Whenever you need the component's state (value), for example, when the user hits the Add button, you peek into it using conventional HTML and DOM functions.

The other option is to supply a value that is a *variable*, something that is part of React's state. This connects a state variable to the value that is displayed. If you change the state variable, the value displayed in the component automatically changes, even on a rerender. Such form components, those that have their value bound to the state, are called *controlled* components. If not bound, like the components we used in `IssueAdd`, they are called *uncontrolled*.

Controlled components are great because their value can be dictated by state (or in many cases, props, which in turn depends on the state of a parent or ascendant component). But what about user interaction? What happens when the user starts typing in the component? There seems to be a deadlock because even if the value is not a hard-coded string, there doesn't seem to be an obvious way to change its value based on a user's keystrokes.

The solution many other frameworks (for example, Angular) offer is *two-way binding* out of the box. The components are not just bound to the value in the state, but also vice versa. Any user input automatically changes the state variable too. But in React, unidirectional data flow is important, so it does not support two-way binding as part of the library. In order to let the user's keystrokes affect the state, you need to handle the `onChange` event in the component yourself. On this event, you need to set the state variable based on the user's input. This will come back all the way, and set the value of the component, and thus its display.

To experience the difference, we'll leave the `IssueAdd` as is, using uncontrolled components, but we'll implement the `Filter` form using controlled components. Let's start with the `render()` method and replace the hyperlinks with a form. We need three input components: a `<select>` for the status filter and two `<input>`s for the effort from and to fields. We'll need to link the three components' values to state variables: `status`, `effort_gte`, and `effort_lte`, respectively. We'll also need to assign a change handler for each of the components.

Let's also add three buttons. The first, an `Apply` button, is for applying the filter. The second, a `Reset` button, to reset the filter to the values that were there *before* the user made any changes. We'll also make it so that the reset button will only be enabled if the user has made any changes and has yet to apply them. This will act as an indicator that something has been changed but has yet to be applied. We'll use a state variable, `changed` for this purpose. Finally, we'll add a `Clear` button, which will work like the `Clear` link in the filter's previous avatar. The new `render()` method implementing all of this is shown in Listing 9-2.

Listing 9-2. IssueFilter.jsx: render() Method with the Filter Form

```

render() {
  return (
    <div>
      Status:
      <select value={this.state.status} onChange={this.onChangeStatus}>
        <option value="">(Any)</option>
        <option value="New">New</option>
        <option value="Open">Open</option>
        <option value="Assigned">Assigned</option>
        <option value="Fixed">Fixed</option>
        <option value="Verified">Verified</option>
        <option value="Closed">Closed</option>
      </select>
      &nbsp;&nbsp;&nbsp;Effort between:
      <input size={5} value={this.state.effort_gte} ⚡
        onChange={this.onChangeEffortGte} />
      &nbsp;&nbsp;&nbsp;-&nbsp;&nbsp;&nbsp;
      <input size={5} value={this.state.effort_lte} ⚡
        onChange={this.onChangeEffortLte} />
      <button onClick={this.applyFilter}>Apply</button>
      <button onClick={this.resetFilter} disabled={!this.state.
        changed}>Reset</button>
      <button onClick={this.clearFilter}>Clear</button>
    </div>
  );
}

```

Next, we'll handle the `onChange` events from each of the inputs. All we need to do is set the state variable to the event target's value. But we can also take advantage of the fact that we know the value that is being typed, and so do some validations. For the number fields, we'll apply a mask and prevent any characters other than numeric characters from being input. If the user types any non-numeric character, we'll just ignore the new value and not set the state value. The `onChange` event handlers that sync user input in all the inputs to state variables are shown in Listing 9-3.

Listing 9-3. IssueFilter.jsx: `onChange` Event Handlers for the Inputs

```

onChangeStatus(e) {
  this.setState({ status: e.target.value, changed: true });
}

onChangeEffortGte(e) {
  const effortString = e.target.value;
  if (effortString.match(/^\d*$/)) {
    this.setState({ effort_gte: e.target.value, changed: true });
  }
}

```

```

onChangeEffortLte(e) {
  const effortString = e.target.value;
  if (effortString.match(/^\d*$/)) {
    this.setState({ effort_lte: e.target.value, changed: true });
  }
}

```

Note that the change of status did not require any mask or condition to propagate the user's input, but the other two handlers applied a mask to ignore any change where the resulting value had non-numeric characters.

Handling the apply and clear buttons is simple; we just call the `setFilter` handler with values from the state variables, which reflect the filter values. Let's also make the contract for `setFilter` such that any condition not specified will be undefined, and not an empty string. These handlers are shown in Listing 9-4.

Listing 9-4. `IssueFilter.jsx`: `applyFilter` and `clearFilter` Handlers

```

applyFilter() {
  const newFilter = {};
  if (this.state.status) newFilter.status = this.state.status;
  if (this.state.effort_gte) newFilter.effort_gte = this.state.effort_gte;
  if (this.state.effort_lte) newFilter.effort_lte = this.state.effort_lte;
  this.props.setFilter(newFilter);
}

clearFilter() {
  this.props.setFilter({});
}

```

Now comes the tricky part. We need to reflect the *current* filter in the inputs' values. At the same time, we need to let the user change them. The user may type the query string in the browser's URL bar, bookmark a list of issues, or use the UI to apply a new filter. We need to not just apply that filter, but also show the filter parameters that got applied. Then, when the user changes any value, we stop showing the current filter and instead show the *intended* new filter. The reset button will get enabled to show the distinction and also let the user undo this action.

Essentially, we need to have two things. One, the currently applied filter (let's call it the `initFilter`), which cannot change unless the filter is applied *and* is in effect. Second, the current state of the form, which reflects the changes the user has made before hitting Apply. We won't use a special variable for this; we'll use the state itself to store the field values. This is *initialized* to `initFilter` but maintains its own values from there on. Since `initFilter` cannot be changed by the filter form, we'll hold it as a property, passed in from the parent component based on the URL (`location`). We can copy its values to the state initially (in the constructor), as well as when the filter changes (when the component receives new properties because the location changes). Also, when the user hits Reset, we can use the values in `initFilter` to set the form's values back to their original values, to reflect the current filter that is applied.

This set of changes is shown in Listing 9-5. It also includes the changes to the constructor for binding handler methods to `this`.

Listing 9-5. IssueFilter.jsx: Using `initFilter` to Initialize and Reset the Form's State

```

constructor(props) {
  super(props);
  this.state = {
    status: props.initFilter.status || '',
    effort_gte: props.initFilter.effort_gte || '',
    effort_lte: props.initFilter.effort_lte || '',
    changed: false,
  };
  this.onChangeStatus = this.onChangeStatus.bind(this);
  this.onChangeEffortGte = this.onChangeEffortGte.bind(this);
  this.onChangeEffortLte = this.onChangeEffortLte.bind(this);
  this.applyFilter = this.applyFilter.bind(this);
  this.resetFilter = this.resetFilter.bind(this);
  this.clearFilter = this.clearFilter.bind(this);
}

componentWillReceiveProps(newProps) {
  this.setState({
    status: newProps.initFilter.status || '',
    effort_gte: newProps.initFilter.effort_gte || '',
    effort_lte: newProps.initFilter.effort_lte || '',
    changed: false,
  });
}

resetFilter() {
  this.setState({
    status: this.props.initFilter.status || '',
    effort_gte: this.props.initFilter.effort_gte || '',
    effort_lte: this.props.initFilter.effort_lte || '',
    changed: false,
  });
}

```

We also need a small change to `propTypes` to let React know that a new property `initFilter` has been introduced. This is shown in Listing 9-6.

Listing 9-6. IssueFilter.jsx: `propTypes` Now Includes `initFilter`

```

IssueFilter.propTypes = {
  setFilter: React.PropTypes.func.isRequired,
  initFilter: React.PropTypes.object.isRequired,
};

```

Finally, in the parent component `IssueList`, we need a couple of changes. First, we had a check in `componentDidUpdate()` before `loadData()` to see if the filter has changed. This should now include the new filter fields related to effort. Next, we need to pass the

initial filter from the location to the `IssueFilter` component. Since we already have the parsed query parameters in `location.query`, we can directly pass this to the child. These changes are shown in Listing 9-7.

Listing 9-7. `IssueList.jsx`: Pass the Initial Filter to `IssueFilter`, Include the Effort Field in Filter Changes

```
...
componentDidUpdate(prevProps) {
  ...
  if (oldQuery.status === newQuery.status
      && oldQuery.effort_gte === newQuery.effort_gte
      && oldQuery.effort_lte === newQuery.effort_lte) {
    return;
  }
  this.loadData();
}
...
<div>
  <IssueFilter setFilter={this.setFilter}
    initFilter={this.props.location.query} />
  <hr />
...

```

When you test these changes, you should find the screen looks like Figure 9-1. Instead of hyperlinks for a hard-coded filter, you now have a form that can generate various filter criteria. Do tests with different values of effort as well as a combination with the status. An interesting thing to try is how the numeric input's validation works, especially when typing invalid characters when the cursor is at various positions in the input.

Issue Tracker

Status: (Any) Effort between: - Apply Reset Clear

Id	Status	Owner	Created	Effort	Completion Date	Title
d05c	Open	Ravan	Mon Aug 15 2016	5		Error in console when clicking Add
d05d	Assigned	Eddie	Tue Aug 16 2016	14	Tue Aug 30 2016	Missing bottom border on panel

Owner Title Add

Full source code available at this [GitHub repository](#).

Figure 9-1. A hard-coded filter replaced with a form

As you'd have realized by now, masking the input has its downside. To demonstrate, place the cursor before 3 when the input's value is 1234. Then, hit any non-numeric key, say, a space. You'll see that the cursor jumps to the end of the input. This happens when the input's value is ignored in the `onChange` event handler.

Why this happens is roughly like this: when a keystroke is received in the input element, the changes first get applied to the value of the input. The actual DOM now has the value, which includes the user's keystroke. Then, an event is generated, which causes React's rerendering to kick in, regardless of whether you do a `setState`. At this point, React compares the input's value in the virtual DOM with the state variable. If the two are the same, it leaves the input alone. If they are different, it sets the value of the input *back* to the state variable. Since the input is being set with a new value (the original state value), the cursor moves to the end. Note that the cursor movement is a behavior that browsers implement, with or without React.

There are different approaches to address this issue. One option is to handle the cursor movement on every event. Packages like `react-input-mask` do this in addition to letting you define masks on the input characters. Another approach is to *always* let the state variable and the input's value be synchronized, and do any validation or masking when the input loses focus. We'll use the latter approach when designing the Date input.

EXERCISE: FILTER FORM

1. In `IssueList`, we had to compare the old props with the new props, whereas in the `IssueFilter`, we didn't, and got away with it. Why?
2. Try using `<input type="number">` rather than the default text. Test it on different browsers, say, Chrome, Safari, and Firefox. Type possibly valid number characters such as `.` (dot) and `E`. What do you observe? Why? Hint: Add some `console.log` statements within the change handler and observe the logs.

Answers are available at the end of the chapter.

The Get API

The List API gave us a list of issues that matched a criterion. To display a single issue in the Edit page, we need an API that can be used to retrieve a single issue, as opposed to a list. We could do this by implementing a filter on the ID of the issue. But that is not the REST convention. To get a single resource, REST dictates that we need an API of the form `/api/issues/<issue_id>`.

If you recollect, I discussed route parameters in the Express REST APIs chapter. We can specify an Express route of the form `/api/issues/:id`. In the handler, we get the ID as a parameter in the request object, as `req.params.id`. The ID will be a string, which we need to convert to a MongoDB ObjectID. Using that object ID, we can then retrieve a single record matching the ID and return the document as a JSON in the response.

The code for all this, including error handling, is shown in Listing 9-8.

Listing 9-8. server.js: New Express Route for Get API

```
app.get('/api/issues/:id', (req, res) => {
  let issueId;
  try {
    issueId = new ObjectId(req.params.id);
  } catch (error) {
    res.status(422).json({ message: `Invalid issue ID format: ${error}` });
    return;
  }

  db.collection('issues').find({ _id: issueId }).limit(1)
    .next()
    .then(issue => {
      if (!issue) res.status(404).json({ message: `No such issue: ${issueId}` });
      else res.json(issue);
    })
    .catch(error => {
      console.log(error);
      res.status(500).json({ message: `Internal Server Error: ${error}` });
    });
});
```

The code is quite similar to the Create API, except for the parsing and validation of the input parameter for the issue ID. You should be able to test this using curl and get a response like this console output:

```
$ curl -s http://localhost:8000/api/issues/57e14da9ca2d380662d9d05c |
  json_pp
{
  "effort" : 5,
  "owner" : "Ravan",
  "created" : "2016-08-15T00:00:00.000Z",
  "status" : "Open",
  "_id" : "57e14da9ca2d380662d9d05c",
  "title" : "Error in console when clicking Add"
}
```

You should also test for invalid format of the issue ID (for example with more than 24 characters), and with an issue ID that is valid in format, but does not exist in the database.

Edit Page

Now that we have an API to fetch just one issue, let's use this to render the contents of the Edit page. Let's create a form to display all the values of the issue, with the editable fields as form inputs. We will not handle the submission of the form yet; we'll leave that for the following sections.

The render method is simple. We'll first have two labels for the non-editable fields: ID and Created Date. As for the rest of the fields, we'll use a select for the status, and input fields for the rest. Let's make these controlled components by connecting them to an object in the state called `issue`. But, instead of handling each input's `onChange` individually like we did in `IssueFilter`, let's make a common method called `onChange`.

To differentiate between the inputs that generate the `onChange` event, we'll have to supply the `name` property to each input, which will be set in the event's `target` field in the event handler. We'll use the field's name as the input name as well. This makes it easy to set the appropriate field of the state object in the event handler: we can just use the `target`'s name as the key to the state object, and modify its value.

As we did for the `IssueList`, we'll add a `load()` method to fetch the issue properties using the `Get` API we created in the previous section. We'll call this method from the `componentDidMount()` and `componentDidUpdate()` lifecycle methods.

There is one difference from the `IssueList` approach. Instead of keeping the field data types as their natural data types, we'll have to convert them to strings. That's because the input fields' value cannot be an object like `Date`; they can only handle strings.

Finally, we'll have to create an initial state in the constructor with empty strings. If not, React assumes that the original values were absent, and therefore, that the input fields were uncontrolled components. When the return of the API call sets the issue fields to non-null values, React assumes that we've converted an uncontrolled component to a controlled one and issues a warning.

The entire new file is shown in Listing 9-9.

Listing 9-9. `IssueEdit.jsx`

```
import React from 'react';
import { Link } from 'react-router';

export default class IssueEdit extends React.Component {
  constructor() {
    super();
    this.state = {
      issue: {
        _id: '', title: '', status: '', owner: '', effort: '',
        completionDate: '', created: '',
      },
    };
    this.onChange = this.onChange.bind(this);
  }

  componentDidMount() {
    this.loadData();
  }
}
```

```

componentDidUpdate(prevProps) {
  if (prevProps.params.id !== this.props.params.id) {
    this.loadData();
  }
}

onChange(event) {
  const issue = Object.assign({}, this.state.issue);
  issue[event.target.name] = event.target.value;
  this.setState({ issue });
}

loadData() {
  fetch(`/api/issues/${this.props.params.id}`).then(response => {
    if (response.ok) {
      response.json().then(issue => {
        issue.created = new Date(issue.created).toDateString();
        issue.completionDate = issue.completionDate != null ?
          new Date(issue.completionDate).toDateString() : '';
        issue.effort = issue.effort != null ? issue.effort.toString() : '';
        this.setState({ issue });
      });
    } else {
      response.json().then(error => {
        alert(`Failed to fetch issue: ${error.message}`);
      });
    }
  }).catch(err => {
    alert(`Error in fetching data from server: ${err.message}`);
  });
}

render() {
  const issue = this.state.issue;
  return (
    <div>
      <form>
        ID: {issue._id}
        <br />
        Created: {issue.created}
        <br />
        Status: <select name="status" value={issue.status} ↵
          onChange={this.onChange}>
          <option value="New">New</option>
          <option value="Open">Open</option>
          <option value="Assigned">Assigned</option>
          <option value="Fixed">Fixed</option>
          <option value="Verified">Verified</option>
        </select>
      </form>
    </div>
  );
}

```

```

        <option value="Closed">Closed</option>
      </select>
    <br />
    Owner: <input name="owner" value={issue.owner} ↵
    onChange={this.onChange} />
    <br />
    Effort: <input size={5} name="effort" value={issue.effort} ↵
    onChange={this.onChange} />
    <br />
    Completion Date: <input
      name="completionDate" value={issue.completionDate}
      onChange={this.onChange}
    />
    <br />
    Title: <input name="title" size={50} value={issue.title} ↵
    onChange={this.onChange} />
    <br />
    <button type="submit">Submit</button>
    <Link to="/issues">Back to issue list</Link>
  </form>
</div>
);
}
}

IssueEdit.propTypes = {
  params: React.PropTypes.object.isRequired,
};

```

Let's examine some parts of the code in greater detail. The following statement in `onChange` is interesting:

```

...
  issue[event.target.name] = event.target.value;
...

```

We used the target's name as the key in the state object to set the value in the state object. This technique helps us combine all of the inputs' `onChange` into one. This works only if we set the `name` property in the inputs, as shown in the last set of highlights within the `render()` method. For example, the name of the select component was set to `status`:

```

...
    Status: <select name="status" value={issue.status} ↵
    onChange={this.onChange}>
...

```

Also, note that the state variables are not exact copies of the object fields. We had to do some data type conversions after fetching the data, for example:

```
...
    issue.created = new Date(issue.created).toDateString();
...
```

The above set of changes will result in an Edit page that looks as in Figure 9-2. It's a good idea now to test it to ensure that all off the inputs are editable and faithfully replicate the user input.

Figure 9-2. Edit page placeholder replaced with a form

EXERCISE: EDIT PAGE

1. *Must* we hook into `componentDidUpdate()`? What will not work correctly if we don't hook into this method and call `loadData()` from the method? Hint: Think about what will cause only a property change, that is, without the component being mounted again. What are the ways the property can change?

Answers available at the end of the chapter.

UI Components

Although we saved some repetitive code by combining all of the `onChange` handlers into one handler, it should be immediately obvious that the approach has some scope for improvement.

1. When dealing with non-string data types, when you want to validate (for example, check if the completion date is not before today), you need to convert the string representations to the natural data type. The same conversion is needed before sending the modified issue to the server.
2. If there is more than one input of the same type (number or date), you must repeat the conversions for each input.
3. The inputs let you type anything and don't reject invalid numbers or dates. We already found that the HTML5 input types aren't much help, and since the `onChange` handler is a common one, you can't add masks there for different input types.

Ideally, we want the form's state to store the fields in their natural data types (number, date, etc.). We also want all of the data type conversion routines to be shared. A good solution to all this is to make reusable UI components for the non-string inputs, which emit natural data types in their `onChange` handlers. We could very well use some of the great packages like `react-number-input` and `react-datepicker` that provide these UI components. But for the sake of understanding how these UI components can be built, let's create our own, minimalistic components.

We'll first create a simple UI component for the number with simple validation and conversion. Then, we'll create a more sophisticated UI component for the date, which does more, like letting the user know if and when the value is invalid.

Number Input

Since form inputs work well with strings, but we need a number in the state, the first thing we must do is separate the two states: one that represents the value of the *persisted* issue's field and another that is the *transient* value, that is, the state while it's being edited by the user. The persisted state can be the natural data type, and the transient one must be a string.

We'll encapsulate the transient state along with all the other things needed to manage the editing and conversion within a UI component called `NumInput`. The transient state will be initialized to the persisted state (which will be owned by the parent component) when the component is mounted, via props. The two states will be synchronized as long as the input is *not* in the "being edited" state. Once the user starts editing (by placing the cursor and focusing on the input), the two states get disconnected. This also lets the user type in a temporarily invalid value. It may not seem such a big deal for plain numbers, but as you'll see in the next section, it is important for other data types like dates.

We can use the event of losing input focus from the input as a good indicator that the user is done editing. We can hook into this event by using an `onBlur()` handler in the input. It is at this point that we'll emit the `onChange()` event to the parent. The actual `onChange()` handler within the `NumInput` component will only affect the local, or the transient, state. When the parent's `onChange()` handler is called when the input loses focus, it will set its state (the persistent one) to the new value. At this point, the two states will be back in sync.

When calling the parent's `onChange`, we will need to pass the validated, natural data type of the input. Instead of modifying the event's target's value (which doesn't work), we'll use a second argument to the `onChange()` handler to pass the value converted to the natural data type. An additional benefit is that the parent can choose to use the event's target's value or the converted value as it deems fit.

It's also a good idea to let the parent specify some properties of the input field that can be passed through to the real HTML `<input>`. One useful property in this case is the `size`, which we set at 5 for the original input field. We'll do this using the *spread attribute* construct of JSX. The new `NumInput` component is shown in Listing 9-10.

Listing 9-10. `NumInput.jsx`

```
import React from 'react';

export default class NumInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: this.format(props.value) };
    this.onBlur = this.onBlur.bind(this);
    this.onChange = this.onChange.bind(this);
  }

  componentWillReceiveProps(newProps) {
    this.setState({ value: this.format(newProps.value) });
  }

  onBlur(e) {
    this.props.onChange(e, this.unformat(this.state.value));
  }

  onChange(e) {
    if (e.target.value.match(/^\d*$/)) {
      this.setState({ value: e.target.value });
    }
  }

  format(num) {
    return num != null ? num.toString() : '';
  }
}
```

```

unformat(str) {
  const val = parseInt(str, 10);
  return isNaN(val) ? null : val;
}

render() {
  return (
    <input
      type="text" {...this.props} value={this.state.value}
      onBlur={this.onBlur} onChange={this.onChange}
    />
  );
}
}

NumInput.propTypes = {
  value: React.PropTypes.number,
  onChange: React.PropTypes.func.isRequired,
};

```

Let's examine a few statements in the listing in a bit more detail. First, let's look at the `format` method of `NumInput`:

```

...
format(num) {
  return num != null ? num.toString() : '';
}
...

```

Here, while converting the natural data type to a string, we are also dealing with the fact that the field is optional. A null value needs to be shown as an empty string. The corresponding `unformat` method does the same by checking if the parsed value is a number:

```

...
unformat(str) {
  const val = parseInt(str, 10);
  return isNaN(val) ? null : val;
}
...

```

An empty string would have resulted in `isNaN` being true. This also has the effect of rejecting possibly invalid values and setting them to `null` instead. But the regex check within `onChange()` on the input characters makes it very difficult to actually type in invalid values.

The two methods `format()` and `unformat()` are used to convert the natural data type to and from a string. We used `format()` when initializing the local state (in the constructor as well as the `componentWillReceiveProps()` lifecycle method), and we used `unformat()` in the `onBlur()` handler to convert to a natural data type before sending it to the parent.

Let's also take a closer look at the spread attribute usage, which allowed us to pass an object that already has the properties directly to a component:

```
...
<input
  type="text" {...this.props} value={this.state.value}
  onBlur={this.onBlur} onChange={this.onChange}
/>
...
```

A construct of the form `{...<object>}` is the spread attribute. It places the key-value pairs of the supplied object as property-value pairs at that point. In the above code, we just passed through to the `<input>`, all the properties we received from the parent.

Note that we used `{...this.props}` *after* the type property but *before* the other properties like `value`. This is a way of indicating which properties can be overridden by the parent and which cannot. In JSX, if the same property is specified more than once, the last specification is the one that takes effect. Thus, if `this.props` contained a `type` property, it would take preference since it appears after the type specification in the input. Whereas if `this.props` contained `value`, `onBlur`, or `onChange` properties, they would not be passed through to the `<input>`. In fact, `this.props` does contain `onChange` and `value` properties, and they get overridden.

In order to use the new component, we need some changes to `IssueEdit`. We need to use `NumInput` instead of `input` for the effort field. Further, we need to change the `onChange` handler to look for an additional argument and use that if available. Finally, we can now use the natural data type in the issue object stored in the state, and the initial value can be `null` rather than an empty string. The modifications to the `IssueEdit` component are shown in Listing 9-11, with the changes highlighted.

Listing 9-11. `IssueEdit.jsx`: Changes for Using `NumInput`

```
...
import NumInput from './NumInput.jsx';
...
  this.state = {
    issue: {
      _id: '', title: '', status: '', owner: '', effort: null,
    },
    ...
  }
  onChange(event, convertedValue) {
    const issue = Object.assign({}, this.state.issue);
    const value = (convertedValue !== undefined) ? ↵
    convertedValue : event.target.value;
    issue[event.target.name] = event.target.value;
    issue[event.target.name] = value;
    this.setState({ issue });
  }
  ...
```

```

    response.json().then(issue => {
      ...
      issue.effort = issue.effort !== null ? issue.effort.toString() : '';
      this.setState({ issue });
    });
  ...
  Effort: <NumInput size={5} name="effort" value={issue.effort} ↵
    onChange={this.onChange} />
  ...

```

Testing with the above changes will show that the effort field is editable, but doesn't allow non-numeric characters. It also supports an empty value. More importantly, with a few `console.log()` statements, you'll be able to see that the form's state is updated only on losing focus from the input, and also that the value that we're storing in the state object is of the numeric data type.

Date Input

Now that we have a basic UI component working, let's take it to the next level with the Date field. A date is more challenging because we can't prevent invalid values by masking the input. The input has to allow partially typed dates (which are invalid at that point in time), and also handle the possibility that the focus is lost while in an invalid state.

We could clear the input whenever it's invalid and it loses focus. But that's not good user experience. What we really need is the ability to let the input be in an invalid state and lose focus, so that the user can come back and correct it. But we need to show that it needs correction, and also prevent submission of the form in this state.

The disconnected state technique helps us handle this. The transient state, which is a string, can hold invalid dates. But the persisted state (even when the focus is lost) cannot. Thus, we not only have to prevent calling `onChange()` in the parent with an invalid date, we also have to indicate to the parent that the input is invalid. This is so that the parent can take appropriate action, such as not allowing the submission to happen and give helpful messages to the user.

We'll also take advantage of the disconnected state to make the display of the value friendlier. We'll display the date in a long format, including the day of the week while not being edited. But when being edited, we need a more precise format, so let's use a YYYY-MM-DD format.

To achieve all this, apart from the transient state in the UI component, we will have to introduce two more state variables:

1. A state variable that indicates whether the focus is in the input field or not
2. Another state variable that indicates whether the currently typed in value is valid or not.

A combination of the two variables will affect how we render the text in the input. If the input is focused, we'll just render the transient state string (which could be invalid). If the input has lost focus, if the value is not valid, we'll continue to render the transient state string. We can also indicate to the user that the value is invalid using a CSS style. If the input is not focused and the value is a valid date, then we'll convert it to the long format and display it.

The validity state will also need to be communicated to the parent whenever it changes. For this, we'll need a new callback passed in from the parent via props. In the parent, we can keep a list of all invalid inputs and disable the submit button if the list is not empty. The list will be added to or removed from when the parent receives any change-of-validity event from the components.

The new component for the Date input is shown in Listing 9-12.

Listing 9-12. DateInput.jsx: UI Component to Handle Dates

```
import React from 'react';

export default class DateInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: this.editFormat(props.value), focused: false, ←
    valid: true };
    this.onFocus = this.onFocus.bind(this);
    this.onBlur = this.onBlur.bind(this);
    this.onChange = this.onChange.bind(this);
  }

  componentWillReceiveProps(newProps) {
    if (newProps.value !== this.props.value) {
      this.setState({ value: this.editFormat(newProps.value) });
    }
  }

  onFocus() {
    this.setState({ focused: true });
  }

  onBlur(e) {
    const value = this.unformat(this.state.value);
    const valid = this.state.value === '' || value != null;
    if (valid !== this.state.valid && this.props.onValidityChange) {
      this.props.onValidityChange(e, valid);
    }
    this.setState({ focused: false, valid });
    if (valid) this.props.onChange(e, value);
  }
}
```

```

onChange(e) {
  if (e.target.value.match(/^\d-*$/)) {
    this.setState({ value: e.target.value });
  }
}

displayFormat(date) {
  return (date != null) ? date.toDateString() : '';
}

editFormat(date) {
  return (date != null) ? date.toISOString().substr(0, 10) : '';
}

unformat(str) {
  const val = new Date(str);
  return isNaN(val.getTime()) ? null : val;
}

render() {
  const className = (!this.state.valid && !this.state.focused) ?
    'invalid' : null;
  const value = (this.state.focused || !this.state.valid) ? this.state.
    value
    : this.displayFormat(this.props.value);
  return (
    <input
      type="text" size={20} name={this.props.name} className={className}
      value={value}
      placeholder={this.state.focused ? 'yyyy-mm-dd' : null}
      onFocus={this.onFocus} onBlur={this.onBlur} onChange={this.onChange}
    />
  );
}
}

DateInput.propTypes = {
  value: React.PropTypes.object,
  onChange: React.PropTypes.func.isRequired,
  onValidityChange: React.PropTypes.func,
  name: React.PropTypes.string.isRequired,
};

```

Let's get into some of the details of the new component's code, mostly in comparison to the simpler `NumInput` component. For `NumInput`, we had only one format method; here we have two, one for a friendly displayable format and the other for the editable one. The editable format is used to display and handle the transient state within the component. This is the format that is also updated in the `onChange()` handler.

We had to introduce an `onFocus` handler so that we could set the state variable `focus`. The same is unset during `onBlur()`. When the input loses focus, we also checked the validity and it's here that we called the parent's validity handler.

Within the `componentWillReceiveProps()` lifecycle method, we introduced a check to see if the props have really changed. This was not required in the `NumInput` component because it did not have to deal with invalid values. Whereas in the `DateInput` component, since we have to preserve the invalid string, we couldn't receive new props without checking that it is indeed a change. Otherwise, it could overwrite user typed values.

Another difference from the `Number` input is the lack of `{...this.props}` in the input's attributes. This is because we have an extra property `onValidityChange` handler that will get passed on to the input causing a warning because it is an unrecognized property for the input.

Let's take a closer look how we showed that the input is invalid. We used a class called `invalid` to show this. The value of this class name is determined by the following:

```
...
    const className = (!this.state.valid && !this.state.focused) ? '
    'invalid' : null;
...

```

This behaves such that we highlight an invalid input with a red border (the actual style definition is shown in Listing 9-14), but only if the field is not focused. This is so that we don't distract the user when editing. Also, if we did show the red border even when the input was focused, the expectation would be that the input becomes normal as the user types and makes the input valid, which will mean that we have to validate the input on every keystroke rather than when it loses focus. That's also a valid approach, used by many applications, especially for password fields, where the validation is typically complex and it's good to have a confirmation on every keystroke.

To make use of the new `DateInput` component, we need a few changes in the parent, `IssueEdit`. This includes a new handler that tracks invalid field names, a state variable to store the invalid fields, making use of a `Date` type for the date input, and a validation message. The changes are shown in Listing 9-13.

Listing 9-13. `IssueEdit.jsx`: Changes to Use `DateInput`

```
...
    this.state = {
      issue: {
        _id: '', title: '', status: '', owner: '', effort: null,
        completionDate: null, created: '',
      },
      invalidFields: {},
    };
...
    this.onValidityChange = this.onValidityChange.bind(this);
...
    onValidityChange(event, valid) {
      const invalidFields = Object.assign({}, this.state.invalidFields);

```

```

    if (!valid) {
      invalidFields[event.target.name] = true;
    } else {
      delete invalidFields[event.target.name];
    }
    this.setState({ invalidFields });
  }
...
    response.json().then(issue => {
      ...
      issue.completionDate = issue.completionDate != null ?
        new Date(issue.completionDate) : null;
      ...
    });
...
  render() {
    const issue = this.state.issue;
    const validationMessage = Object.keys(this.state.invalidFields).
length === 0 ? null
    : (<div className="error">Please correct invalid fields before
submitting.</div>);
...
    Completion Date: <DateInput
      name="completionDate" value={issue.completionDate}
      onChange={this.onChange}
      onValidityChange={this.onValidityChange}
    />
...
    <br />
    {validationMessage}
    <button type="submit">Submit</button>
...

```

Finally, a couple of changes are required in the style section of `index.html` to give a distinct style to the error message and the invalid input. This is shown in Listing 9-14.

Listing 9-14. `index.html`: Changes to Add New Styles for Errors

```

...
}
input.invalid {border-color: red;}
div.error {color: red;}
</style>
...

```

Ensure that you are able to edit and test with valid and invalid values, and also that the validation message is shown whenever the user leaves the input in an invalid state. We could have added a way to dismiss the error message, but that requires special widgets, which I'll leave for the next chapter.

EXERCISE: DATE INPUT

1. Since we couldn't use `{...this.props}` we had to hard-code the size. If we want to let the parent optionally specify a different size, like we did for `NumInput`, what options do we have? Hint: Look up `defaultProps` in the React documentation.

Answers are available at the end of the chapter.

Update API

If you recall from Chapter 5 (Express Rest APIs), there are two ways to update an object using REST APIs. You can use the PATCH method and specify the changes, or you can use the PUT method to replace the object altogether. Let's briefly discuss the pros and cons of the two options.

The PATCH method allows the caller to alter individual fields without having to send across the entire object even for small changes. The savings in network traffic may not amount to much, but it eases the front-end code if it is able to modify individual fields. Imagine the Issue List page having a drop-down menu next to each issue that allows the user to quickly change the status. In this case, the front end needs to send only the change in status rather than the entire object.

The PATCH method is almost unavoidable if the front end does not have access to the complete object. Say you had some extra fields in the back end not exposed to the front end (for example, some bookkeeping fields or embedded documents such as a list of comments associated with each issue). Replacing the entire object would either mean destroying those fields not visible to the front end or an elaborate merge process in the back end. Finally, if you needed an audit log of who changed what, using the PATCH lets you just save the patch specification from the front end rather than trying to find the difference between the old and new documents.

The PUT way of doing things, on the other hand, is very simple to implement. You don't have to learn about and implement a format such as JSON Patch (<http://jsonpatch.com>) that would need to be used in the PATCH method. In the front end, you can just take the entire object, serialize it, and submit it. At the back end, all you need to do is a bit of validation to check if the document is valid, and then use the `updateOne` function in the MongoDB driver to save the document. In the Issue Tracker, the client is a trusted one (our own code), and we know that there are no bookkeeping fields, embedded documents, or a requirement for an audit log as of now. If we find a need for individually updating fields in the future, we can implement the PATCH method at that point in time.

To use the PUT method, the Update API will need a `put` route that matches `/api/issues/:id`, where `id` is the Object ID of the document, as in the case of Get API. The body of the request is the modified issue object, which will be automatically parsed if the client passes "application/json" as the content type, as in the Create API. Let's also decide to return the modified object in the response, like we did for the Create API.

We'll use the `updateOne` MongoDB function to save the document. Also, we need to convert the date strings in the input to the `Date` data type. We'll introduce a new function in `issue.js` that does this conversion. Apart from these, the rest of the code is very similar to the other APIs. The changes for the conversion function are shown in Listing 9-15 and the complete API is shown in Listing 9-16.

Listing 9-15. `issue.js`: New Conversion Function

```
...
function convertIssue(issue) {
  if (issue.created) issue.created = new Date(issue.created);
  if (issue.completionDate) issue.completionDate = new Date(issue.
    completionDate);
  return cleanupIssue(issue);
}
...
export default {
  validateIssue,
  cleanupIssue,
  convertIssue,
};
...
```

Listing 9-16. `server.js`: New Route for Update API

```
app.put('/api/issues/:id', (req, res) => {
  let issueId;
  try {
    issueId = new ObjectId(req.params.id);
  } catch (error) {
    res.status(422).json({ message: `Invalid issue ID format: ${error}` });
    return;
  }

  const issue = req.body;
  delete issue._id;

  const err = Issue.validateIssue(issue);
  if (err) {
    res.status(422).json({ message: `Invalid request: ${err}` });
    return;
  }

  db.collection('issues').update({ _id: issueId }, ←
  Issue.convertIssue(issue)).then(() =>
    db.collection('issues').find({ _id: issueId }).limit(1)
    .next()
  )
})
```

```

.then(savedIssue => {
  res.json(savedIssue);
})
.catch(error => {
  console.log(error);
  res.status(500).json({ message: `Internal Server Error: ${error}` });
});
});

```

Note that we converted all date strings to the Date data types using the `Issue.convertIssue` function:

```

...
db.collection('issues').update({ _id: issueId }, {
  Issue.convertIssue(issue)).then(() =>
...

```

But that function did not convert the ID from strings to `ObjectId`. Instead, we just deleted the ID, if present, from the object:

```

...
delete issue._id;
...

```

This is because the MongoDB update operation treats the ID specially, and leaves it intact even if not present in the document that replaces the existing document.

To test the new API, you can use curl, as shown in the console output below. Remember to change the ID string to a valid `ObjectId` that was generated in *your* database, by doing a `find()`. The ID shown below is unlikely to work for you.

```

$ curl -s -X PUT http://localhost:8000/api/issues/57e14da9ca2d380662d9d05c \
  --header 'Content-Type: application/json' \
  --data '{"title" : "Error in console when clicking Add",
    "status" : "Open", \
      "effort" : 6, "owner" : "Ravan",
      "created" : "2016-08-15T00:00:00.000Z" }' \
  | json_pp
{
  "effort" : 6,
  "owner" : "Ravan",
  "_id" : "57e14da9ca2d380662d9d05c",
  "title" : "Error in console when clicking Add",
  "created" : "2016-08-15T00:00:00.000Z",
  "status" : "Open"
}

```

After successful execution, you should also be able to confirm using the application in the browser that any changes you sent across have in fact taken effect. You should also test with invalid objects supplied in the request body. Finally, confirm that the dates have been converted to the Date data type by using the MongoDB shell and displaying the issues in the database. They should show up as `ISODate(...)` rather than strings.

EXERCISE: UPDATE API

1. In the Update API call, we had to pass a value for the `created` field. Ideally, we'd need to prevent changes to this field, even if the caller supplies it. You can't use the same approach as we did for the ID, because that would mean *removing* the field in the document. What options do we have? Hint: Look up MongoDB's update documentation, use something that doesn't replace the entire document.

Answers are available at the end of the chapter.

Using Update API

To use the Update API, we'll need to handle the form submission within the Edit page. In this handler method, we'll need to call the API using the PUT method. If successful, we'll have to do the usual data type conversions and set the returned document in the state. We'll also show an alert message saying the save was successful; otherwise, the user gets no indication that something changed.

While implementing the NumInput and DateInput components we had converted the effort and completion fields to their natural data types. We'll do the same for the created field now, so that all data types are their natural ones now. The changes to IssueEdit for using the Update API are shown in Listing 9-17.

Listing 9-17. IssueEdit.jsx: Changes to Handle Form Submission

```
...
  this.state = {
    issue: {
      _id: '', title: '', status: '', owner: '', effort: null,
      completionDate: null, created: null,
    },
  },
...
  this.onSubmit = this.onSubmit.bind(this);
...
  onSubmit(event) {
    event.preventDefault();
```

```

    if (Object.keys(this.state.invalidFields).length !== 0) {
      return;
    }

    fetch(`/api/issues/${this.props.params.id}`, {
      method: 'PUT',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(this.state.issue),
    }).then(response => {
      if (response.ok) {
        response.json().then(updatedIssue => {
          updatedIssue.created = new Date(updatedIssue.created);
          if (updatedIssue.completionDate) {
            updatedIssue.completionDate = new Date(updatedIssue.completionDate);
          }
          this.setState({ issue: updatedIssue });
          alert('Updated issue successfully.');
```

```

        });
      } else {
        response.json().then(error => {
          alert('Failed to update issue: ${error.message}');
        });
      }
    }).catch(err => {
      alert('Error in sending data to server: ${err.message}');
    });
  }
  ...
  loadData() {
    fetch(`/api/issues/${this.props.params.id}`).then(response => {
      if (response.ok) {
        response.json().then(issue => {
          issue.created = new Date(issue.created);
          issue.completionDate = issue.completionDate != null ?
            new Date(issue.completionDate) : null;
        });
      }
    });
    ...
    <form onSubmit={this.onSubmit}>
      ID: {issue._id}
    <br />
    Created: {issue.created ? issue.created.toDateString() : ''}
    <br />
    ...
  }
}

```

We used `event.preventDefault()` to disable the default action of the submit button, which would have been available to submit the form. We also silently ignored the submission if there were validation errors, since we know that the error message will be visible. We are now ready to test the complete Edit Page functionality: view, change, and save an issue. The screenshot for the Edit page is already shown in Figure 9-2.

Do test with invalid inputs as well ensure that the save is prevented. Also check that the object has been saved correctly, both in the Issue List page as well as in the database using MongoDB shell.

Delete API

It's unlikely that an application will actually want to delete database records. Usually, records are marked deleted and removed from listings. But for the sake of CRUD completion, let's also implement a Delete API.

We will use the MongoDB driver's `deleteOne` API to delete an object from the database. The REST API will use the `DELETE` method, which will translate to a route function call to `app.delete`. There is no object to return on a delete, so we'll only return a status whether the delete was successful. The result tells us how many objects were deleted; depending on this, we'll return an OK or a warning if the object was not found. We don't want to treat this as an error, since the absence of the object is what the caller is requesting, which is technically the case after the call, even if the object was not found.

The Delete API is shown in Listing 9-18.

Listing 9-18. server.js: Delete API Implementation

```
app.delete('/api/issues/:id', (req, res) => {
  let issueId;
  try {
    issueId = new ObjectId(req.params.id);
  } catch (error) {
    res.status(422).json({ message: `Invalid issue ID format: ${error}` });
    return;
  }

  db.collection('issues').deleteOne({ _id: issueId }).then((deleteResult) => {
    if (deleteResult.result.n === 1) res.json({ status: 'OK' });
    else res.json({ status: 'Warning: object not found' });
  })
  .catch(error => {
    console.log(error);
    res.status(500).json({ message: `Internal Server Error: ${error}` });
  });
});
```

The new API can be tested using curl like this:

```
$ curl -s -X DELETE http://localhost:8000/api/issues/581de551c47b3814fd20c23f -
| json_pp
{
  "status" : "OK"
}
```

If you run the same test using curl again, it should return a status message with a warning that the object was not found.

Using the Delete API

To implement delete functionality in the UI, we'll need to first introduce a button on each row in the Issue List. Upon the click of this button, we expect the issue to be deleted. This requires changes in the `IssueRow` component.

Rather than call the Delete API directly from the `IssueRow` component, we'll pass in a delete handler from `IssueList` via `props`. This keeps the `IssueRow` simple and isolates the actual mechanism of deletion. If we decide that we need a confirmation, or even let the delete be implemented as a checkbox in the UI and require a “submit” to confirm it after multiple deletes, these options would be easier done in the `IssueList`.

To handle the delete button's click, we must add an `onClick` handler, which needs to be a function that calls the delete handler with the ID of the current row as a parameter. We could do this by binding the delete handler with the ID of the issue in the component instance, but this is not recommended, as ESLint will tell us. Instead, we'll add an inner function for `onClick` that uses a closure to call the delete handler with the ID.

The modified `IssueRow` component is shown in Listing 9-19.

Listing 9-19. `IssueList.jsx`: `IssueRow` Class Rewritten

```
const IssueRow = (props) => {
  function onDeleteClick() {
    props.deleteIssue(props.issue._id);
  }

  return (
    <tr>
      <td><Link to={`/issues/${props.issue._id}`}>{props.issue._
        id.substr(-4)}</Link></td>
      <td>{props.issue.status}</td>
      <td>{props.issue.owner}</td>
      <td>{props.issue.created.toString()}</td>
      <td>{props.issue.effort}</td>
      <td>{props.issue.completionDate ?
        props.issue.completionDate.toString() : ''}</td>
      <td>{props.issue.title}</td>
      <td><button onClick={onDeleteClick}>Delete</button></td>
    </tr>
  );
};

IssueRow.propTypes = {
  issue: React.PropTypes.object.isRequired,
  deleteIssue: React.PropTypes.func.isRequired,
};
```

Since the function is no longer a simple expression, we must enclose the contents with curly braces. Also, we must add a return around the rendering. The substantial changes are the addition of a new `<td>` with a button, the `onClick` handler, and the addition of the handler to `propTypes`.

Next, we need to implement the actual delete handler in `IssueList` and pass it on as props (via `IssueTable`) to `IssueRow`. The delete handler will call the Delete API using `fetch()` and reload the data on completion to show the new list. Also, we need to add a column in `IssueTable` for the extra column header. These changes are also in the same file, `IssueList.jsx`, and shown in Listing 9-20.

Listing 9-20. `IssueList.jsx`: Other Changes for Using Delete API

```
...
function IssueTable(props) {
  const issueRows = props.issues.map(issue =>
    <IssueRow key={issue._id} issue={issue} deleteIssue={props.deleteIssue} />
  );
  ...
  return (
    <table className="bordered-table">
      ...
      <th>Title</th>
      <th></th>
    ...
  </table>
  );
  IssueTable.propTypes = {
    issues: React.PropTypes.array.isRequired,
    deleteIssue: React.PropTypes.func.isRequired,
  };
  ...
  export default class IssueList extends React.Component {
    constructor() {
      ...
      this.setFilter = this.setFilter.bind(this);
      this.deleteIssue = this.deleteIssue.bind(this);
    }
    ...
    deleteIssue(id) {
      fetch(`/api/issues/${id}`, { method: 'DELETE' }).then(response => {
        if (!response.ok) alert('Failed to delete issue');
        else this.loadData();
      });
    }
    ...
    <IssueTable issues={this.state.issues} deleteIssue={this.deleteIssue} />
    ...
  }
}
```


Now we can really call the CRUD implementation complete. We can add and delete objects from the List page, and view and modify a single object in the Edit page. At this stage, you should test the functionality of the complete application as a user: add, modify, and delete a few issues as a user would do.

Summary

We used the Edit page to explore forms and the difference between controlled and uncontrolled form components. We also added new APIs to cater to the needs of the new form, and completed the CRUD paradigm by adding a Delete operation as well.

While we did all this, a thought must have crossed your mind: can you make all this, especially the Edit page, look better in the browser? That's exactly what you'll set out to do in the next chapter. We'll use a popular CSS library adapted to React to add some polish to the UI.

Answers to Exercises

Exercise: More Filters in List API

1. MongoDB is strict with respect to data types. This also means that for a field that has no value, it cannot determine the type, thus the match cannot occur if the field has a filter criterion.
2. If we do need to return documents with the effort field missing, we'll have to create an `or` condition that includes the original filter, as well as a condition that says the effort is undefined. The `$or` operator takes an array of filters and matches the document against any of the filter conditions. To match a document where the effort field is not defined, we must use `{ $exists: false }` as the criterion for the field. Here's an example in the mongo shell:

```
> db.issues.find({$or: [{effort: {$lte: 10}}, ←
{effort: {$exists: false}}]});
```

Exercise: Filter Form

1. When loading data, we *asynchronously* updated the state, and React has no way of knowing that it was done as part of the lifecycle method. Even if we had used the method `ComponentWillReceiveProps`, we would have had to compare the old and new. In the case of the `IssueFilter`, although we are modifying the state, we are not doing it asynchronously. This allows React to handle the change gracefully, and not trigger another `ComponentWillReceiveProps`.

2. If you set the input's type as number, you find that (a) it behaves differently on different browsers, (b) masking does not work on some browsers, and (c) when it does allow invalid characters, you don't see them in `onChange`. This is because as per the HTML specification, when the type is specified, and the input does not conform to the specification, the `.value` of the input is supposed to return an empty string. It is also up to the browser how to deal with invalid values; for example, some browsers may display the fact that the input is invalid, whereas others may prevent an invalid entry.

When using React, it is best not to use the `type` attribute of input fields; instead, deal with the validation or masking yourself (or use packages that do it for you). This lets the behaviour be predictable across browsers, as well as make informed decisions on what to do with invalid input, especially input that is invalid *temporarily* in order to get to a valid value.

Exercise: Edit Page

1. The only property for this component is the ID of the issue being shown. The component will not be remounted if only the ID changed. This can happen if you make a switch between one issue's edit page and another.

One way to make this happen is by using the browser's History menu (not just the Back button). For example, in Safari, there is a History menu, and in Chrome, you can right-click the Back button to jump to any previous page. Thus, when viewing issue 1, you can click on "Back to Issue List," and then go to issue 2. Now, using the History menu, you can jump directly to issue 1. When this happens, the component is not mounted again, and without the hook into `componentDidUpdate()`, the page would be showing the wrong issue details.

Exercise: Date Input

1. One option is to copy the `this.props` into another object using `Object.assign()` and then remove the offending properties such as `onValiditychange`, and then use the new object as a spread attribute. This will enable the passing of more properties without explicitly specifying each of them.

Another option is to set the input's `size` property to `{this.props.size}`. In this case, you need to add `size` as a numeric property type in `propTypes`, and also define `defaultProps` with the `size` set to the default value like this:

```
...
DateInput.defaultProps = {
  size: 20,
};
...
```

This will allow the parent to skip the size specification, in which case the default will be used. If the parent did specify the size, then that size would be used.

Exercise: Update API

1. To prevent the API's caller from changing the created date, one option is to read the object from the database and overwrite the created date using the persisted value. Although this is a valid approach, and if we are reading the object for other reasons (for example, to record which fields have changed), it's OK to do so. If not, we could use the `$set` update operator after deleting the created field like this:

```
...
  delete issue.created;
...
db.collection('issues').updateOne({ _id: issueId }, ←
{ $set: issue }).then(() =>
...

```

This has the effect of saying, “Modify the fields specified in the issue to their corresponding values.” Since the created field has been deleted, its value will be left alone.

CHAPTER 10



React-Bootstrap

CSS frameworks like [Bootstrap](#) and [Foundation](#) have changed the way people build their websites. These tools make it a lot easier to make a web application look professional and responsive, that is, have it adapt even to mobile screens. Of course, the downside is that these existing frameworks may not give you fine-grained flexibility, and your application will look like many others. But, even if you do have the luxury or the capability to create your own end-to-end custom styles, I suggest that you start off with these frameworks. That's because there's a lot to learn from the patterns these frameworks use.

Since we are using React as the UI library, we need to choose something that fits into and plays well with React. I evaluated React + Foundation, Material UI, and React-Bootstrap because they appeared the most popular, based on Google searches. React + Foundation didn't appear to be very different from React-Bootstrap in terms of capability, but Bootstrap itself is far more popular. Material UI has an interesting CSS-in-JS and inline-style approach of styling that fits well into React's philosophy of isolating everything needed by a component, with the component itself. But this framework is much less popular and also seems to be a work in progress. And, perhaps the inline-style approach is too drastic a deviation from convention.

React-Bootstrap is a safe alternative that is built on top of the very popular Bootstrap and fits our needs (except for the lack of a date picker). I thus chose React-Bootstrap for this book. In this chapter, we'll look at how to make the application look professionally styled using React-Bootstrap. I won't be covering how to make custom themes and other advanced topics, but we'll learn just enough to appreciate what React Bootstrap is all about, so you can go further when required.

Bootstrap Installation

In this section, we'll install React-Bootstrap and confirm that it works by making a small change that is visible in the UI. Let's first install React-Bootstrap:

```
$ npm install --save-dev react-bootstrap
```

React-Bootstrap contains a library of React components, and has no CSS styles or themes itself. But it does require a Bootstrap stylesheet to be included in the application to use these components. The version or mechanism of including the stylesheet is left to us. The easiest way is to include the stylesheet from a CDN directly in `index.html`.

But since we have other dependencies locally installed, let's do the same for Bootstrap as well. So, let's install bootstrap also using npm so that its distribution files are available to you:

```
$ npm install --save bootstrap
```

The next step is to include the Bootstrap stylesheet in the application. One way to do this is by using webpack's style and CSS loaders. This allows us to use an `import` (or `require`) statement to include CSS files just like we import other React or JavaScript modules. Note that webpack then builds the dependency tree and includes all the styles that have been imported in the bundle that is created. Also note that webpack, by default, includes all required styles in a string within the JavaScript bundle that is built. When the application is loaded, the string is placed into the DOM as a `<style>` node.

To get this to work, we need to install the CSS and style loaders using npm. Then, we need to add pattern matches in the webpack configuration that triggers these loaders based on the file extension. We will also need loaders for the icons and fonts that the stylesheet may include. Finally, we need a single `import` statement that imports `bootstrap.css`, maybe just in `App.jsx`.

I find that for our needs, all of this is overkill. The purpose of webpack's CSS and Style loaders is to be able to *modularize* stylesheets just as we modularized the React code. If every component had its own set of styles separated out into their own CSS files, this method would work great. Unfortunately, Bootstrap is shipped as a monolithic stylesheet. We have to include all of it, even if we are using only a single component. Also, we don't envisage our own components being associated with their individual styles in a modular fashion.

Thus, all we'll do is keep a link to the Bootstrap distribution under the static directory, and include the CSS just as you would use other static files such as `index.html` itself. The command to run to achieve this is on a Mac or a Linux based computer is

```
$ ln -s ../node_modules/bootstrap/dist static/bootstrap
```

Alternatively, you can copy the entire `dist` directory under the Bootstrap library into the static directory. Since soft links are not supported under Windows, that's the only option for Windows users. If you now explore the new directory, you'll find three subdirectories: `css`, `fonts`, and `js`. We won't need the `js` directory because that's what React-Bootstrap replaces. In `index.html`, we'll add a link to this stylesheet, which is under the `css` directory of the directory that we linked or copied. We won't include the optional theme file, just the main minified Bootstrap style. The changes to `index.html` are shown in Listing 10-1.

Listing 10-1. `index.html`: Changes to Include Bootstrap Styles

```
...
<title>Pro MERN Stack</title>
<link rel="stylesheet" href="/bootstrap/css/bootstrap.min.css" >
...
```

■ **Note** When we installed Bootstrap, we used `--save` instead of `--save-dev`. This is because we are serving the Bootstrap files directly from the server, and they are needed at runtime, unlike all other front-end assets that are compiled and bundled from their sources.

To test that Bootstrap is properly installed and usable, and also to test that the fonts are available, let's make a few minor changes: let's replace the Delete button with a trash icon button. Let's also add some style to the Issues table with a Bootstrap `container-fluid` class.

For the trash icon, we need two components from React-Bootstrap: `Button` and `Glyphicon`. We can import them as we did other components such as `Link` from `react-router` in the `IssueList.jsx` file. As for the style, we can directly apply the Bootstrap class `container-fluid` to the content `<div>` that is the parent of all the routed components. The `container-fluid` class of Bootstrap is needed for most containers and layouts, letting the library know that the elements contained can be styled by Bootstrap. These changes are shown in Listing 10-2 and Listing 10-3, respectively.

Listing 10-2. `IssueList.jsx`: Changes for Trash Icon

```
...
import { Button, Glyphicon } from 'react-bootstrap';
...
    <td>{props.issue.title}</td>
    <td><button onClick={onDeleteClick}>Delete</button></td>
    <td>
      <Button bsSize="xsmall" onClick={onDeleteClick}><Glyphicon ~
        glyph="trash" /></Button>
    </td>
  </tr>
...

```

Listing 10-3. `App.jsx`: Adding `container-fluid` Class at Topmost `div`

```
...
<div className="container-fluid">
  {props.children}
</div>
...

```

The components available as part of React-Bootstrap can be found in the React-Bootstrap documentation site, and the glyphs and styles are described in the Bootstrap documentation site. For the `Button`, we added an `onClick` handler just as we would in a regular button. We also use the `xsmall` size attribute to use the smallest possible button size.

When specifying the `glyph` attribute, we need to only specify the name (as in `trash`), unlike Bootstrap styles which are prefixed with `glyphicon-` (like `glyphicon-trash`). You will find that this is a consistent theme in React-Bootstrap: you don't have to use the prefixes because, due to modularization, you don't have the possibility of a namespace conflict as in CSS.

On testing, you'll find that the new button and styles are indeed applied, as shown in Figure 10-1.

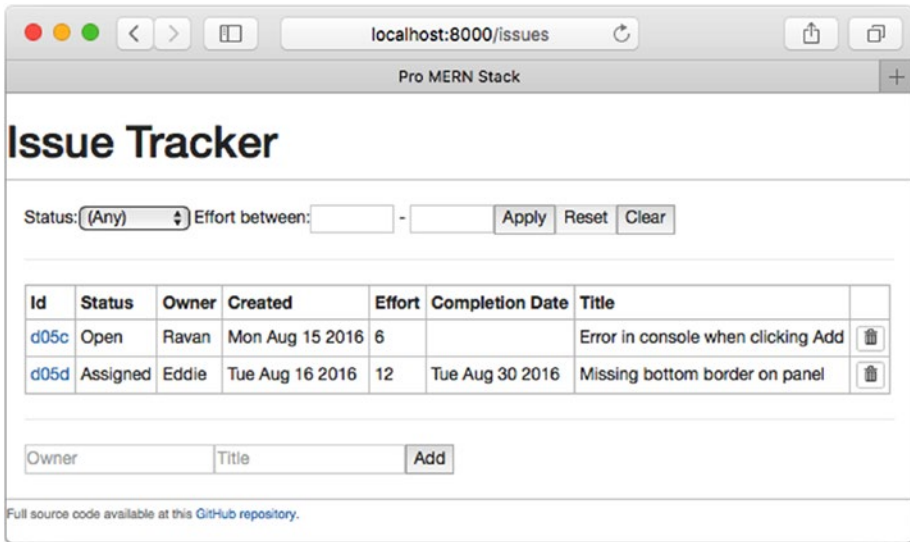


Figure 10-1. Initial Bootstrap-styled Issue List page

We'll deal with the alignment issues soon, but you can see that the changes that we made are indeed taking effect, and we can now start using other components and styles. The lack of margins around the header and the footer are due to Bootstrap's normalization of styles (also called style reset) to make it consistent across browsers.

Navigation

In this section, we'll explore what we can do with the header and also fix the footer alignment. It is common practice to have a header with a Navigation bar, one that lets the user navigate between various sections of the application. In our case, we don't have too many sections (yet), so apart from the main Issue List page, we'll add placeholders for the rest.

Let's keep the application name in its original place, after which we can have two tabs or links, one for the Issue List and another as a placeholder for Reports. The Edit page doesn't really belong in the navigation bar because it's not something that the user can go to directly; they need to navigate *from* the Issue List only. Let's also, on the right side, have an action item for creating a new issue (we'll move the in-page Add form to a Modal in later sections), and an extended drop-down menu for any other action you may have in mind. For now, let the extended menu have just one action, say, Logout (which does nothing for now).

We can use React-Bootstrap's `Navbar` and within that, a `Navbar.Header` and `Navbar.Brand` component for the application name, as described in the documentation. Following that, we need a `Nav` for the left-aligned items, and another for the right-aligned items using the `pullRight` attribute. In the right-aligned set, let's add a `NavItem` and a `NavDropdown` with one menu item for Logout. As for the left-aligned `Nav`, we have a decision to make. We can use `NavItem` from React-Bootstrap, which takes a `href`. But we really need React-router's `Links` instead, which include their own `<a>` elements causing the styling to mess up if we use it within the `NavItem`.

The recommended way to break this impasse is to use the package `react-router-bootstrap`, which provides a wrapper called `LinkContainer` acting as the React-router's `Link`, at the same time letting its children have their own styling. We can have a `NavItem` without a `href` inside the `LinkContainer`, and let it deal with the path to the route. Let's install the package to use the `LinkContainer`:

```
$ npm install --save-dev react-router-bootstrap
```

Since it contains React components and JavaScript code, we must also change the webpack configuration so that it gets included in the vendor bundle rather than the application bundle. This is shown in Listing 10-4.

Listing 10-4. `webpack.config.js`: Changes to Include `react-router-bootstrap` in Vendor Bundle

```
...
module.exports = {
  entry: {
    app: './src/App.jsx',
    'react', 'react-dom', 'react-router', 'react-bootstrap',
    'react-router-bootstrap',
  },
  ...
}
```

We'll now create a new component called `Header`, place all of the Navigation items within it, and use it in the `App` component in place of the original header. As for the footer, we'll pull it into the contents `<div>` so that it maintains margins, and use Bootstrap typography elements `<small>` and `<h5>` to make it look as before. These changes are shown in Listing 10-5, Listing 10-6, and Listing 10-7.

Listing 10-5. `App.jsx`: Imports for `react-bootstrap` and `react-router-bootstrap`

```
...
import { Navbar, Nav, NavItem, NavDropdown, MenuItem, Glyphicon } from 'react-bootstrap';
import { LinkContainer } from 'react-router-bootstrap';
...
```


Listing 10-6. App.jsx: New Component, Header

```

const Header = () => (
  <Navbar fluid>
    <Navbar.Header>
      <Navbar.Brand>Issue Tracker</Navbar.Brand>
    </Navbar.Header>
    <Nav>
      <LinkContainer to="/issues">
        <NavItem>Issues</NavItem>
      </LinkContainer>
      <LinkContainer to="/reports">
        <NavItem>Reports</NavItem>
      </LinkContainer>
    </Nav>
    <Nav pullRight>
      <NavItem><Glyphicon glyph="plus" /> Create Issue</NavItem>
      <NavDropdown id="user-dropdown" ↵
        title={<Glyphicon glyph="option-horizontal" />} noCaret>
        <MenuItem>Logout</MenuItem>
      </NavDropdown>
    </Nav>
  </Navbar>
);

```

Listing 10-7. App.jsx: App Component Rewritten

```

const App = (props) => (
  <div>
    <Header />
    <div className="container-fluid">
      {props.children}
      <hr />
      <h5><small>
        Full source code available at ↵
        this <a href="https://github.com/vasansr/pro-mern-stack">
        GitHub repository</a>.
      </small></h5>
    </div>
  </div>
);

```

Note that we haven't used `eventKeys` for the `NavItems`, as described in the React-Bootstrap documentation. These are required if we either want to set the active item, or if we have a common handler for the selection of an item and we need to know which item generated the event. Since `LinkContainer` takes care of showing which link is active, and we don't have a common handler for the events from the menu items, we don't need event keys.

Using an icon and text for the right-aligned `NavItem` is straightforward, but it's not so for the drop-down. Let's take a closer look at the code for the drop-down:

```
...
<NavDropdown id="user-dropdown" ←
  title={<Glyphicon glyph="option-horizontal" />} noCaret>
...

```

`NavDropdown` is the component that displays the clickable portion of the drop-down. Normally, it has a caret (downward pointing arrow), and the title is usually a string. We've set the title as an icon (components are acceptable titles for `NavDropdown`), which already indicates that it's a dropdown or an extended menu. So, we must disable the default caret by using the `noCaret` property. The ID is required for accessibility.

When you test this set of changes, you can see a screen similar to Figure 10-2. Note that switching between Issues and Reports highlights the selected route in the `NavBar`. The actionable right-side items don't do anything as of now. They do not show any highlighting because they are not associated with any route: they are pure actions. We'll eventually add an `onClick` handler in each of those items. Clicking on Reports will show a Page Not Found message, but that's OK. We are yet to implement the page.

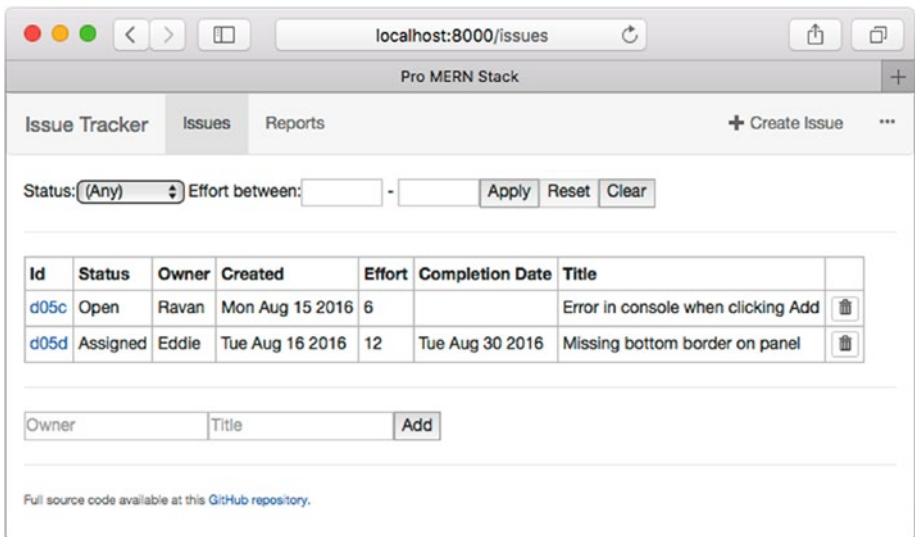


Figure 10-2. Header and footer styled with Bootstrap

Let's now test this on a mobile device and see how responsive the navigation bar is. One way to do this is to actually use a mobile phone and connect to the server that's running on your desktop. For that, you must compile the front end and back end, and then start the production server rather than use the development servers. An easier way is to use the mobile emulation mode of the web browsers to see how it looks. I found

that only Chrome accurately reflects what happens in a real mobile device; Safari (using the responsive mode under Develop menu) and Firefox (in the dev tools) only simulate a change of screen size. What you'd see on a real mobile device or on Chrome's mobile emulator is shown in Figure 10-3.

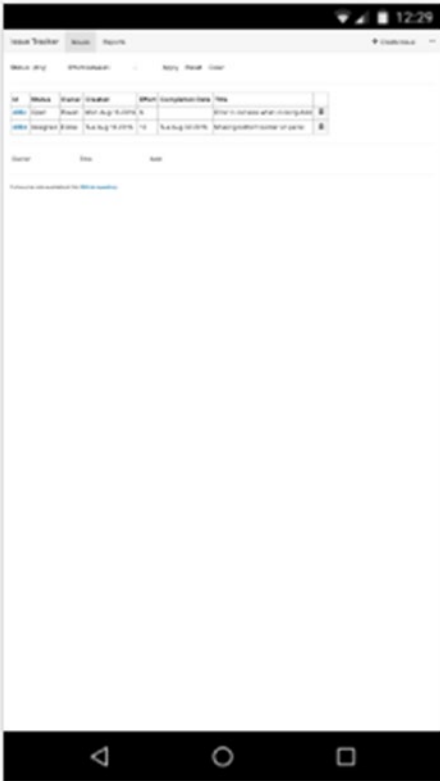


Figure 10-3. *The application in a mobile device: squished!*

This is not what we wanted to see. The screen looks squished or zoomed out. The reason this happens is that the mobile browser assumes that the page has not been designed for mobile screens, so it picks an arbitrary width that would be appropriate for a desktop, uses that, and then zooms out the page so that it fits the screen. We must tell the mobile browser that you know how to handle small screens, so that it doesn't do all that. The way to do so is by adding a meta tag in the main page. The change for this is shown in Listing 10-8.

Listing 10-8. index.html: Meta Tag for Adapting to Mobile Screens

```

...
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Pro MERN Stack</title>
...

```

What the above really means is that it's OK to set the viewport (the un-zoomed screen) to be the same as the device's width and also to start with a one-is-to-one scale. The effect of this important change is shown in Figure 10-4.

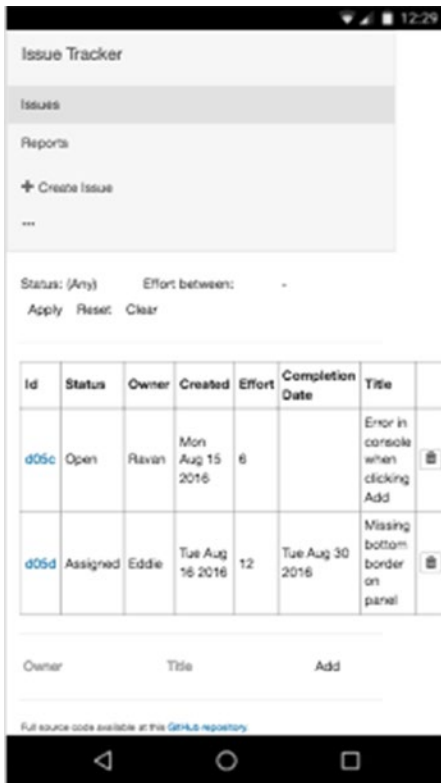


Figure 10-4. Viewport adapting to the device screen

The text is now in a readable font size. You can also see that the navigation bar has collapsed into a single column of choices rather than a wide horizontal list. The same could be simulated by changing the screen size of the browser on the desktop.

EXERCISE: NAVIGATION

1. Replace `LinkContainer` with a `NavItem` with a `href`, as given in the React-bootstrap examples, and switch between the two. Do you see a difference? Also, look at the network traffic while you switch.

Answer available at the end of the chapter.

Table and Panel

In this section, we'll use React-Bootstrap's table styles to replace our styles. Further, we'll make the Issue Filter a collapsible panel. This is so that when we add more filter fields and the section occupies more vertical space, we don't waste valuable real estate at the top of the screen.

Using React-Bootstrap's Table component is simple; we just need to replace the `<table>` with a `<Table>`. As for styling it, we can get rid of our own styles and class names, and instead use the Table component's property `bordered` to show borders. There are other useful properties such as `hover` and `responsive`, which are good to have as well. The property `hover` has the effect of highlighting a row when the user hovers the mouse over a row. To provide a horizontal scrollbar on smaller screens, we use the property `responsive`. To keep the table looking compact, we also need to use the `condensed` property.

To make the filter section collapsible, we need the Panel component to wrap the `IssueFilter` component. Once we do this, we can also get rid of the horizontal rules, as the panel itself has a border delineating the filter from the table. Panel supports the property `collapsible`, which makes the header (if there's one) clickable and toggles the visibility of the contents.

The changes for using a Panel and styling the table are shown in Listing 10-9.

Listing 10-9. `IssueList.jsx`: Changes for Table and Panel

```
...
import { Button, Glyphicon, Table, Panel } from 'react-bootstrap';
...
  <table className="bordered-table">
    <Table bordered condensed hover responsive>
  </table>
  </Table>
...
```

```

<div>
  <Panel collapsible header="Filter">
    <IssueFilter setFilter={this.setFilter} ←
      initFilter={this.props.location.query} />
  </Panel>
  <hr />
  <IssueTable issues={this.state.issues} ←
    deleteIssue={this.deleteIssue} />
  <hr />
  <IssueAdd createIssue={this.createIssue} />
</div>
...

```

The attributes `bordered`, `condensed`, and `hover` are pretty obvious, but it's worth pointing out the effect of the `responsive` attribute. In the previous section, you saw on mobile devices that the content within table cells is wrapped (see Figure 10-4). In addition, the screen width matched the smallest possible table width, leaving some gap in the navigation bar at the right. The `responsive` attribute of the table lets it behave differently on small screens. It prefers not wrapping cell contents; instead, it adds a horizontal scrollbar if the table is too wide to fit the screen. This is usually a better layout experience.

When you try out this set of changes, you'll find that it's not obvious that the panel header is clickable. Firstly, the cursor does not change to something that shows that it's clickable. Also, the only place you can click is on the text. For the sake of usability, what we'd really like is the cursor to indicate that it's clickable, and also let the entire header area be clickable. If you inspect the DOM using Safari's or Chrome's inspectors, you can see that there is an `<a>` element that is added by React-Bootstrap when the header is made collapsible.

Unfortunately, we don't have a way of configuring the panel to either not add the `<a>` (and let you specify a clickable node yourself for the header), or to tell it to fill the horizontal space. The only way we can do this is by using a style that makes the `<a>` a block element to fill the space and set the cursor. The changes for this style, and the removal of the table border styles which are no longer required, are shown in Listing 10-10.

Listing 10-10. `index.html`: Style Changes

```

...
table.bordered-table th, td {border: 1px solid silver; padding: 4px;}
table.bordered-table {border-collapse: collapse;}
input.invalid {border-color: red;}
div.error {color: red;}
.panel-title a {display: block; width: 100%; cursor: pointer; }
...

```

Ideally, we shouldn't have any styles in the header of the main page, because this is hard to discover and maintain. We'll have to live with this minor infraction, at least for now. The screenshot in Figure 10-5 shows how the screen looks with the filter collapsed.

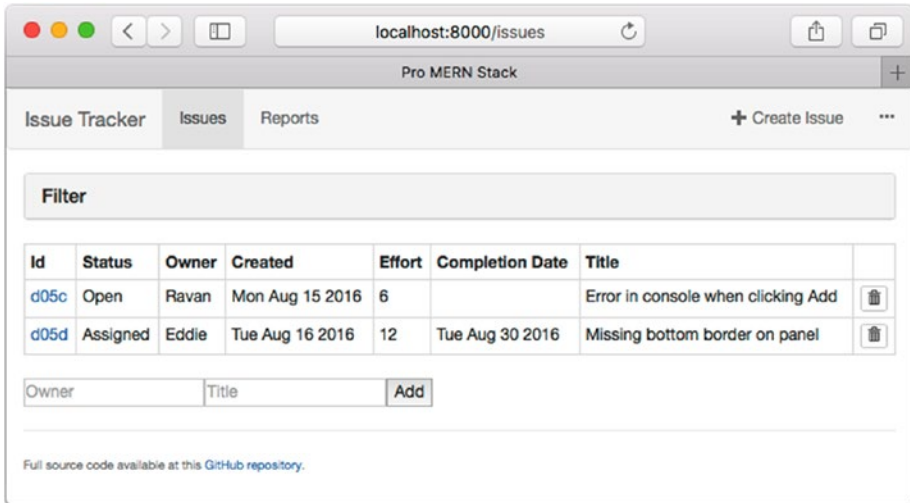


Figure 10-5. Collapsible filter panel and Bootstrap-styled table

Forms

We have three forms in the application, and we'll explore three different ways of displaying these forms. In the first approach, we'll also see how Bootstrap's Grid system is used to display pages responsively, that is, how they adapt to screen sizes.

Grid-Based Forms

The Bootstrap default way of displaying labels and inputs in a form is to have the input *below* the label and then another label and another input below it. For narrow screens or narrow sections within a page, this may work well. But if you think about a wide screen, it does seem like a lot of wasted real estate.

A better way to deal with this is to use the grid system of Bootstrap and let each field (and that includes the label) *float*, that is, occupy the space next to its precedent, or below its precedent if the width of the screen doesn't allow it. The Issue Filter is a good use case for this behavior, because we'd like to see it laid out horizontally, but on smaller screens, one below the other.

Bootstrap's grid system usually starts with a `<Grid>`, but we don't need this if the component is inside another container. The Issue Filter is inside a panel, which is already a container. So, we can directly add a `<Row>`, within which we can add `<Col>`s, which will hold the form fields. We'll refer to each `<Col>` as a cell in the following text.

The grid system works this way: the horizontal space is divided into a maximum of 12 columns. A cell can occupy one or more columns, and also a different number of columns at different screen widths. The cells wrap if you have more than 12 column-space of cells within a row. You only use a new row if you want to force a break in the flow of cells. When it comes to forms, the best way to use the grid system is to have a single row, and specify how many columns each form control (one cell) occupies at different screen widths.

Bootstrap then takes care of laying them out and deciding how to fit the cells at different screen widths. In the filter, we have three cells, of roughly equal width: status input, effort inputs, and the buttons. We don't want to break either the effort inputs or the buttons into multiple lines even on very small screens, so we'll treat them as one cell each.

Let's start with the smallest screen size: a mobile device. Let's use half the screen width per cell. This will mean that we'll have Status and Effort on one line, and the buttons in the next. This can be achieved by specifying `xs={6}` for each of the cells, that is, half the total available 12 columns. You may wonder how 3 cells of 6 columns each, totaling 18 columns, can fit a row of 12 columns. But the fact is that the grid system *wraps* the last 6 columns into another line (not row, mind you).

It's best to compare the fluid grid system with paragraphs and lines. Rows are like paragraphs rather than lines. A paragraph (row) can contain multiple lines. As the paragraph width (screen width) reduces, it will need more lines. It's only when you want to break two sets of sentences (sets of cells) that you really need another paragraph (row). Most people take some time to appreciate this aspect of the fluid grid system, because many popular examples show rows and columns in a *fixed* grid rather than a fluid one, and therefore lay out the screen in multiple rows.

Next, let's consider a slightly bigger screen: a tablet, in landscape mode. The property for this size is `sm`. Let's fill the screen width with all three cells in one line. We must use a width of 4 columns for each, thus specifying `sm={4}` for these cells. If we had more cells, then this too would wrap into multiple lines but since we have exactly three, this will fit the screen in one line.

On larger screens like desktops, we can let each cell continue to occupy 4 columns each, which doesn't require any more property specifications. But I think it looks ungainly if the form controls stretch too much, so let's reduce the width of the cells using `md={3}` and `lg={2}`. This will cause the trailing columns on larger screens to be unoccupied.

Now, for the contents of each cell in the grid. The most common usage is to use a `FormGroup` component that encloses a `FormLabel` for the label of the control and a `FormControl` component for the actual control. A `FormControl` has a `componentClass` property that specifies the component class of the actual element that is rendered, such as `input`, `select`, and `textarea`. It can even be our own custom class instead of the built-in HTML elements. The status drop-down fits this pattern, with the options as children of the `FormControl`.

The Effort inputs are not that straightforward since they're made up of two inputs. We can use an `InputGroup` to enclose the two `FormControls`, but by itself, it will cause the two inputs to show one below the other. The `InputGroup.Addon` component can be used to display the inputs next to each other, as well as show the dash between the two inputs.

We'll use a `ButtonToolbar` for the set of buttons in the last cell. There is no label for this, but to align the buttons with the controls, we need an empty label. The code for the new `render()` method of `IssueFilter` is shown in Listing 10-11.

Listing 10-11. `IssueFilter.jsx`: New `render()` Method, and Required Imports

```
...
import { Col, Row, FormGroup, FormControl, ControlLabel, InputGroup,
  ButtonToolbar, Button } from 'react-bootstrap';
...
render() {
  return (
```



```

<Row>
  <Col xs={6} sm={4} md={3} lg={2}>
    <FormGroup>
      <ControlLabel>Status</ControlLabel>
      <FormControl
        componentClass="select" value={this.state.status}
        onChange={this.onChangeStatus}
      >
        <option value="">(Any)</option>
        <option value="New">New</option>
        <option value="Open">Open</option>
        <option value="Assigned">Assigned</option>
        <option value="Fixed">Fixed</option>
        <option value="Verified">Verified</option>
        <option value="Closed">Closed</option>
      </FormControl>
    </FormGroup>
  </Col>
  <Col xs={6} sm={4} md={3} lg={2}>
    <FormGroup>
      <ControlLabel>Effort</ControlLabel>
      <InputGroup>
        <FormControl value={this.state.effort_gte} ↵
          onChange={this.onChangeEffortGte} />
        <InputGroup.Addon>-</InputGroup.Addon>
        <FormControl value={this.state.effort_lte} ↵
          onChange={this.onChangeEffortLte} />
      </InputGroup>
    </FormGroup>
  </Col>
  <Col xs={6} sm={4} md={3} lg={2}>
    <FormGroup>
      <ControlLabel>&nbsp;</ControlLabel>
      <ButtonToolbar>
        <Button bsStyle="primary" onClick={this.applyFilter}>Apply ↵
        </Button>
        <Button onClick={this.resetFilter} ↵
          disabled={!this.state.changed}>Reset</Button>
        <Button onClick={this.clearFilter}>Clear</Button>
      </ButtonToolbar>
    </FormGroup>
  </Col>
</Row>
);
}

```

The screenshots for very small and small screen sizes are shown in Figure 10-6 and Figure 10-7, respectively.

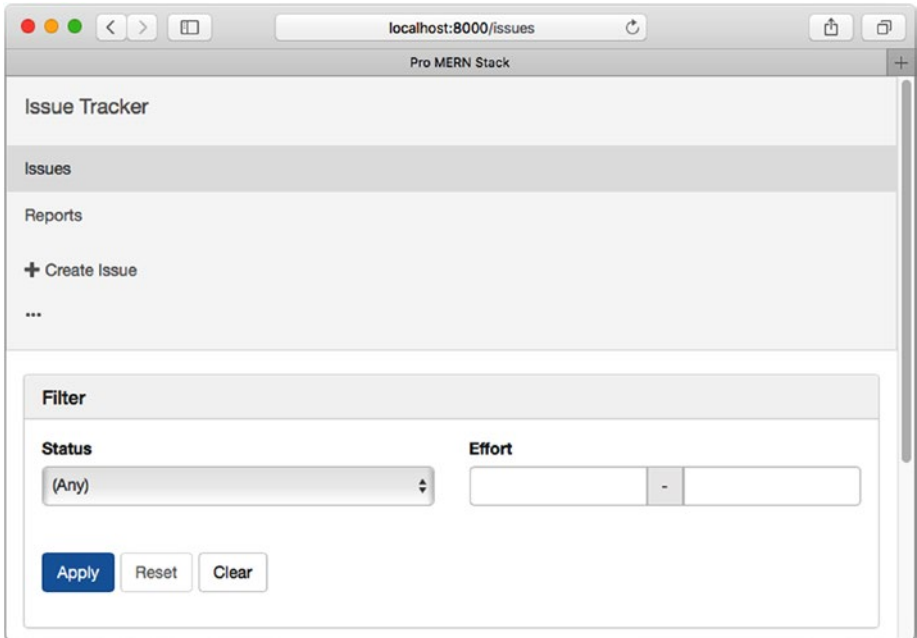


Figure 10-6. Filter form in a very small width screen

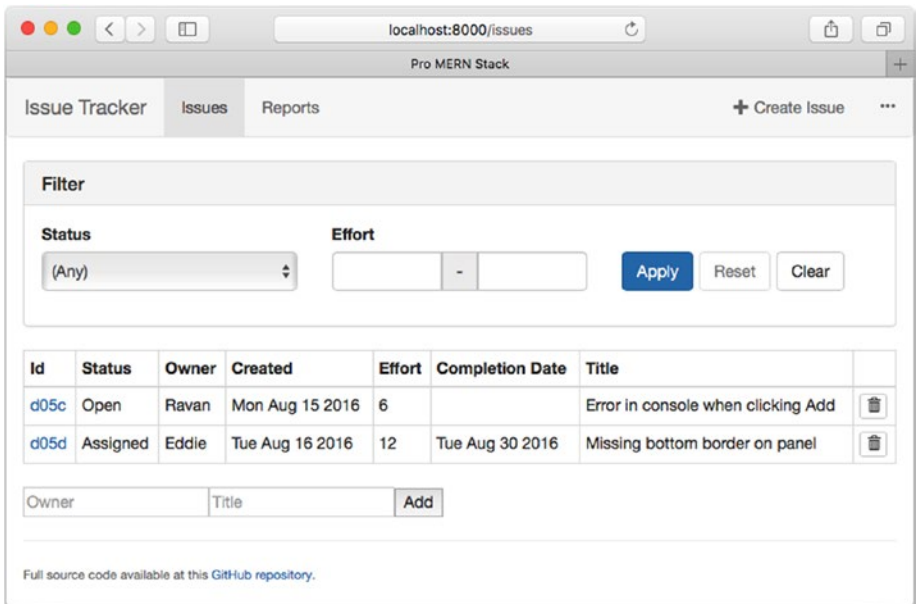


Figure 10-7. Filter form in a small width screen

EXERCISE: GRID-BASED FORMS

1. Let's say the cells are larger and you need the cells to (a) on very small screens, appear one below the other, (b) on small screens, have a max of two cells per line, (c) on medium sized screens, together fit the width, and (d) on large screens, together occupy two-thirds the screen width. What are the width specifications in this case?
2. Although great for mobile devices, the input controls look a bit oversized on the desktop browser. What can be done to make them look smaller? Hint: Look up the React-Bootstrap documentation and search for the `bsSize` property in the "Forms" section.

Answers are available at the end of the chapter.

Inline Forms

Sometimes we want the form controls next to each other, including the labels. This is ideal for small forms with two or three inputs that can all fit in one line and are closely related. This style will suit the Issue Add form, even though we don't have labels.

For the grid-based forms, we didn't have to enclose the controls or the groups within a `<Form>`, since the default behavior of the groups was a vertical layout (one below the other, including labels). For inline forms, we need a `<Form>` with the `inline` property around the controls. This also comes in handy because we need to set the other attributes of the form: name and submit handler.

Since we don't need labels, we can just place the two form controls and a button one after the other, without a `FormGroup`. The code for the new `IssueAdd` component's `render()` method is shown in Listing 10-12.

Listing 10-12. `IssueAdd.jsx`: New `render()` Method and Required Imports

```
...
import { Form, FormControl, Button } from 'react-bootstrap';
...
render() {
  return (
    <div>
      <Form inline name="issueAdd" onSubmit={this.handleSubmit}>
        <FormControl name="owner" placeholder="Owner" />
        {' '}
        <FormControl name="title" placeholder="Title" />
        {' '}
        <Button type="submit" bsStyle="primary">Add</Button>
      </Form>
    </div>
  );
}
```

```

    </Form>
  </div>
);
}

```

We had to manually add spaces using `{ ' ' }` since JSX strips out spaces and newlines. The effect of this change is shown in Figure 10-8. At this point, you should test to ensure that the add form and the filter forms work as they used to.

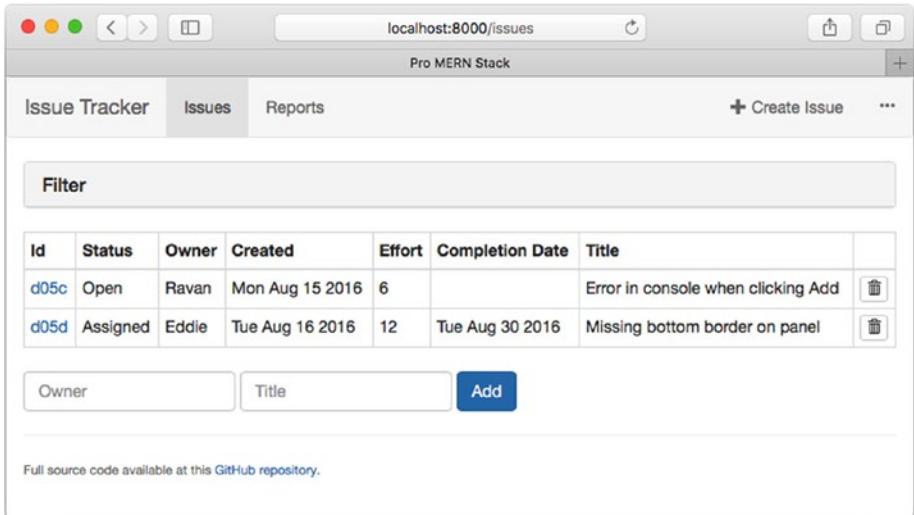


Figure 10-8. *IssueAdd form using inline forms*

EXERCISE: INLINE FORMS

1. Say we want to make the form controls look smaller. Where can we set the `bsSize` property? Hint: In the React-Bootstrap documentation, look in the “Inline Forms” section.
2. The widths of the two controls are identical, and it seems we have no control over this because the `bsSize` property only affects the height. If we want to show a smaller Owner input and a wider Title Input, what can be done? After setting the size, what difference do you see in very small screens?

Answers are available at the end of the chapter.

Horizontal Forms

The next type of form we will explore is the Horizontal Form, where the label appears to the left of the input, but each field appears one below the other. Typically, the input fills the parent container until the right edge, giving it an aligned look. Let's change the Issue Edit page to use a horizontal form, since this form has a lot of fields and this kind of form will suit it. Let's also use the validation states that Bootstrap provides rather than our own rudimentary method to display validation error in the Date input.

To lay out a horizontal form, first we need a horizontal property on the `<Form>` component, and the usual `FormGroups` that we have used before. But that's not all; we also need to specify how much width the label and the input will occupy. For this, we need to enclose the `<ControlLabel>` and the `<FormControl>` with `<Col>`s and specify the column width. Since we want it to fill the screen, we won't use different widths for different screen sizes, just one specification for the small screen width that splits the label and the input in some proportion. The grid system will use the same ratio for bigger screen widths. As for the very small screen width, it will cause it to collapse to a single column. Let's choose a 3-9 split between the two columns.

Enclosing the `<FormControl>` with a `<Col>` works OK, but for a `<ControlLabel>`, this does not have the intended effect of right-aligning the label. The suggested method in the Bootstrap documentation is to set the `componentClass` of the `<Col>` to `ControlLabel` instead. This has the effect of rendering a single element with the combined classes of a `ControlLabel` and a `Col` for rather than a label within a `<div>`.

Recall that we can specify our own component classes for the `FormInput` as well; this lets us use the custom components `NumInput` and `DateInput` where we need them. To show the invalidity of an input, we need to set the `validationState` to `error` in the `FormGroup`. The reason we need to do it in the form group rather than the input itself is that this makes the label stand out and adds other adornments. We'll just set the `validationState` to the string `error` if we find that the state variable `invalidFields` contains this field's name.

The code for the new `render()` method and the required import changes are shown in Listing 10-13.

Listing 10-13. IssueEdit.jsx: Changed `render()` Method using Horizontal Form

```
...
import { Link } from 'react-router';
import { FormGroup, FormControl, ControlLabel, ButtonToolbar, Button,
  Panel, Form, Col } from 'react-bootstrap';
...
render() {
  const issue = this.state.issue;
  const validationMessage = Object.keys(this.state.invalidFields).length
  === 0 ? null
    : (<div className="error">Please correct invalid fields before
      submitting.</div>);
  return (
```

```

<Panel header="Edit Issue">
  <Form horizontal onSubmit={this.onSubmit}>
    <FormGroup>
      <Col componentClass={ControlLabel} sm={3}>ID</Col>
      <Col sm={9}>
        <FormControl.Static>{issue._id}</FormControl.Static>
      </Col>
    </FormGroup>
    <FormGroup>
      <Col componentClass={ControlLabel} sm={3}>Created</Col>
      <Col sm={9}>
        <FormControl.Static>
          {issue.created ? issue.created.toDateString() : ''}
        </FormControl.Static>
      </Col>
    </FormGroup>
    <FormGroup>
      <Col componentClass={ControlLabel} sm={3}>Status</Col>
      <Col sm={9}>
        <FormControl
          componentClass="select" name="status" value={issue.status}
          onChange={this.onChange}
        >
          <option value="New">New</option>
          <option value="Open">Open</option>
          <option value="Assigned">Assigned</option>
          <option value="Fixed">Fixed</option>
          <option value="Verified">Verified</option>
          <option value="Closed">Closed</option>
        </FormControl>
      </Col>
    </FormGroup>
    <FormGroup>
      <Col componentClass={ControlLabel} sm={3}>Owner</Col>
      <Col sm={9}>
        <FormControl name="owner" value={issue.owner}
          onChange={this.onChange} />
      </Col>
    </FormGroup>
    <FormGroup>
      <Col componentClass={ControlLabel} sm={3}>Effort</Col>
      <Col sm={9}>
        <FormControl
          componentClass={NumInput} name="effort"
          value={issue.effort} onChange={this.onChange}
        />
      </Col>
    </FormGroup>
  </Form>
</Panel>

```

```

<FormGroup validationState={this.state.invalidFields. ↵
  completionDate ? 'error' : null}>
  <Col componentClass={ControlLabel} sm={3}>Completion Date</Col>
  <Col sm={9}>
    <FormControl
      componentClass={DateInput} name="completionDate"
      value={issue.completionDate} onChange={this.onChange}
      onValidityChange={this.onValidityChange}
    />
    <FormControl.Feedback />
  </Col>
</FormGroup>
<FormGroup>
  <Col componentClass={ControlLabel} sm={3}>Title</Col>
  <Col sm={9}>
    <FormControl name="title" value={issue.title} ↵
      onChange={this.onChange} />
  </Col>
</FormGroup>
<FormGroup>
  <Col smOffset={3} sm={6}>
    <ButtonToolbar>
      <Button bsStyle="primary" type="submit">Submit</Button>
      <LinkContainer to="/issues">
        <Button bsStyle="link">Back</Button>
      </LinkContainer>
    </ButtonToolbar>
  </Col>
</FormGroup>
</Form>
{validationMessage}
</Panel>
);
}
}

```

We enclosed the form within a `Panel` to make it look more like a form and also give it a header. Also, instead of a plain `Link` to go back to the Issue list, we used a `Button` with a link style; this lets us align the button and the link. Enclosing this within a `LinkContainer` ensures that `React-Router` gets hold of the `href` and manipulates the route correctly. For the non-editable fields `ID` and `Created Date`, we used `<FormControl.Static>`.

The code is testable at this point, but you will find that the date field is not filling the screen. It also looks quite differently styled from the other inputs; see Figure 10-9.

Figure 10-9. Issue Edit page, with the date input not looking good

The reason is that we set the class for the input within `DateInput` to either `null` or `invalid`, depending on the validation state. Bootstrap would have normally set a class for the input, and our setting, especially the `null`, overwrites it.

One option is to replace the `className` with `this.props.className`. But we're unsure what other properties are being passed through, so it is safer to use `{...this.props}`, as we did for the `NumInput`, with one change: we'll make a copy of the props and delete the `onValidityChange` property, since it is an invalid property for `<input>`. The changes to `DateInput` are listed in Listing 10-14.

Listing 10-14. `DateInput.jsx`: Changes for Passing Through Props

```
...
render() {
  const className = (!this.state.valid && !this.state.focused) ? 'invalid'
    : null;
  const value = (this.state.focused || !this.state.valid) ? this.state.value
    : this.displayFormat(this.props.value);
  const childProps = Object.assign({}, this.props);
  delete childProps.onValidityChange;
  return (
```



```

<input
  type="text" size={20} name={this.props.name} className={className}
  value={value}
  type="text" {...childProps} value={value}
  placeholder={this.state.focused ? 'yyyy-mm-dd' : null}
  onFocus={this.onFocus} onBlur={this.onBlur} onChange={this.onChange}
/>
);
}
...

```

Figure 10-10 shows the new Issue Edit form, with a validation error.

The screenshot shows a web browser window with the URL `localhost:8000/issues/57e14da9ca2d380662d9d05d`. The page title is "Issue Tracker" and the breadcrumb is "Issues". There is a "Create Issue" button in the top right. The main content is the "Edit Issue" form. The form fields are:

- ID:** 57e14da9ca2d380662d9d05d
- Created:** Tue Aug 16 2016
- Status:** Assigned (dropdown menu)
- Owner:** Eddie (text input)
- Effort:** 12 (text input)
- Completion Date:** 2016-12-250 (text input with a red border and a red 'x' icon, indicating a validation error)
- Title:** Missing bottom border on panel (text input)

At the bottom of the form, there are two buttons: "Submit" (blue) and "Back" (light blue). Below the buttons, a red message says: "Please correct invalid fields before submitting." At the very bottom of the page, there is a link: "Full source code available at this GitHub repository."

Figure 10-10. Issue Edit form with Bootstrap horizontal forms

Alerts

In this section, we'll convert all of the messages shown to the user to Bootstrap-styled alerts.

Validations

The first candidate to convert to a Bootstrap-styled alert is the validation message, as seen in Figure 10-10. We want this to look aligned and styled like the rest of the page. Also, we'll be subtler. Since the form field itself shows that something is wrong, we won't display the error message until *after* the user clicks Submit. We'll also let the user dismiss the message after they've seen it.

React-Bootstrap's `<Alert>` component works well for this. It has different styles to show a message, and it also has the ability to show a Close icon. But its visibility needs to be handled by us: we should show or not show the message based on a state variable in `IssueEdit`. But since the Close icon itself is part of `<Alert>`, to let that icon make the message disappear, we have to pass in a handler that modifies the visibility state. We can use the `onDismiss` property to achieve this.

By default, the message won't be shown (the initial showing state is false). When the user clicks on Submit, we'll force the message to be shown (by setting the state to true). We'll also hide the message if there are no errors. Thus, a combination of there being errors and the visibility state will show the message.

For alignment, let's make the validation message appear right below the button toolbar, inside a form group. The changes to `IssueEdit` for the validation message handling are shown in Listing 10-15.

Listing 10-15. `IssueEdit.jsx`: Changes for Using Alert for a Validation Message

```
...
import { FormGroup, FormControl, ControlLabel, ButtonToolbar, Button,
  Panel, Form, Col, Alert } from 'react-bootstrap';
...
  this.state = {
    issue: {
      _id: '', title: '', status: '', owner: '', effort: null,
      completionDate: null, created: null,
    },
    invalidFields: {}, showingValidation: false,
  };
  this.dismissValidation = this.dismissValidation.bind(this);
  this.showValidation = this.showValidation.bind(this);
...
  onSubmit(event) {
    event.preventDefault();
    this.showValidation();
  }
...

```

```

    showValidation() {
      this.setState({ showingValidation: true });
    }

    dismissValidation() {
      this.setState({ showingValidation: false });
    }
...
render() {
  const issue = this.state.issue;
  const validationMessage = Object.keys(this.state.invalidFields).length
    === 0 ? null
    : (<div className="error">Please correct invalid fields before
      submitting.</div>);
  let validationMessage = null;
  if (Object.keys(this.state.invalidFields).length !== 0 &
    && this.state.showingValidation) {
    validationMessage = (
      <Alert bsStyle="danger" onDismiss={this.dismissValidation}>
        Please correct invalid fields before submitting.
      </Alert>
    );
  }
...
    </FormGroup>
    <FormGroup>
      <Col smOffset={3} sm={9}><validationMessage></Col>
    </FormGroup>
  </Form>
  {validationMessage}
...

```

This is a good time to get rid of the styles in `index.html` that we had used to show validation errors and style the messages. This is shown in Listing 10-16.

Listing 10-16. `index.html`: Obsolete Styles Removed

```

...
<style>
  input.invalid {border-color: red;}
  div.error {color: red;}
  .panel-title a {display: block; width: 100%; cursor: pointer; }
</style>
...

```

When you try this out, you should see validation message for invalid entries in the date fields, as in Figure 10-11. Do also try out correcting the error and submitting the form to ensure the positive path works as before.

The screenshot shows a web browser window with the address bar displaying 'localhost:8000/issues/57e14da9ca2d380662d'. The page title is 'Pro MERN Stack'. The form contains the following fields:

- Owner:** Eddie
- Effort:** 12
- Completion Date:** 2016-12-244 (This field has a red border and a red 'x' icon on the right, indicating it is invalid.)
- Title:** Missing bottom border on panel

Below the fields are two buttons: 'Submit' (blue) and 'Back' (grey). At the bottom of the form, there is a red alert message box that says 'Please correct invalid fields before submitting.' with a red 'x' icon on the right.

Figure 10-11. Validation message using alerts

Results

Let's now look at result messages, that is, the reporting of successes and failures. Since we need to do this in multiple pages, let's create a new custom component for this that can be reused in any page that needs to show result messages.

These messages are intended to be unobtrusive, so let's make them disappear after a few seconds automatically rather than make the user close them. We'll also let the messages overlay the page as well as transition in and out like the Toast messages in the Android OS.

We'll use a visible state, just as in the previous section, that determines whether the alert is shown or not. To achieve an automatic close of the alert, we must install a timer that calls the `onDismiss` handler after a few seconds. We must also reinstall the timer when any property (for example, the message) changes. To achieve a smooth fading-in transition, we will enclose the `<Alert>` inside a `<Collapse>` component that's provided by React-Bootstrap for this purpose and use its `in` property to control the visibility.

The new component will support the properties `showing` (to control visibility), `onDismiss` (the dismiss handler), `bsStyle` (to control the style, such as success and error), and `message` (the message to be shown). Let's put the new component with all this in a new file called `Toast.jsx`, the code for which is shown in Listing 10-17.

Listing 10-17. `Toast.jsx`: New Reusable Alert Component

```
import React from 'react';
import { Alert, Collapse } from 'react-bootstrap';

export default class Toast extends React.Component {
  componentDidUpdate() {
    if (this.props.showing) {
```

```

        clearTimeout(this.dismissTimer);
        this.dismissTimer = setTimeout(this.props.onDismiss, 5000);
    }
}

componentWillUnmount() {
    clearTimeout(this.dismissTimer);
}

render() {
    return (
        <Collapse in={this.props.showing}>
            <div style={{ position: 'fixed', top: 30, left: 0, right: 0,
                textAlign: 'center' }}>
                <Alert
                    style={{ display: 'inline-block', width: 500 }} ←
                    bsStyle={this.props.bsStyle}
                    onDismiss={this.props.onDismiss}
                >
                    {this.props.message}
                </Alert>
            </div>
        </Collapse>
    );
}
}

Toast.propTypes = {
    showing: React.PropTypes.bool.isRequired,
    onDismiss: React.PropTypes.func.isRequired,
    bsStyle: React.PropTypes.string,
    message: React.PropTypes.any.isRequired,
};

Toast.defaultProps = {
    bsStyle: 'success',
};

```

Note that we're using inline styles rather than a class and style specification in a CSS file or `<style>` section. This is actually in keeping with the philosophy of React: everything required for a component is within the component. I find that maintaining this component is easier if the style specification is inline, mainly because I don't have to search for the CSS file where the class is defined and make a change to, say, the width of the alert. Some have taken this approach to the next level (search for CSS in JS on the Internet to find some interesting discussions on this topic), and until very recently, Material-UI fully embraced this approach.

But this technique has its limitations. For one, you need to make a distinction between *themes* and individual *styles*. Style consists of things like alignment, margins, sizes, etc., and is localized. Themes, on the other hand, need to be controlled globally. If you do adopt the inline-styles approach, you need global variables that can be referred to by the inline styles.

For example, if the position from top is something that someone other than the owner of the component needs to be able to change, it can't be an inline style. This can happen because a designer (who is different from the programmer) is in charge of these decisions. Or, it could be that component itself is part of a library, and the user of the library needs this flexibility. The hard part is deciding which style elements need to be controlled externally and how to account for them. I suggest you start with inline styles and as your project gets bigger, you'll realize which are the elements that need global or external control, and you can then start separating them out into theme CSS in JS or add class names as you deem fit.

For now, we'll leave it as inline styles because that seems most convenient (compare this to the class we had for styling the validation errors, and almost forgot removing the unnecessary styles from `index.html`).

To use the new component, we'll make changes in the `IssueEdit` and `IssueList` components. First, we'll place the `Toast` component somewhere in the page (it doesn't matter where because its position is fixed); add state variables for its visibility, message, and style; and methods to show and dismiss the alert message. Then, we'll replace all the alerts with `Toast` messages. These changes for `IssueEdit` are shown in Listing 10-18.

Listing 10-18. `IssueEdit.jsx`: Changes Replacing Alert with Toast

```
...
import Toast from './Toast.jsx';
...
constructor() {
  super();
  this.state = {
    issue: {
      _id: '', title: '', status: '', owner: '', effort: null,
      completionDate: null, created: null,
    },
    invalidFields: {}, showingValidation: false,
    toastVisible: false, toastMessage: '', toastType: 'success',
  };
  this.dismissValidation = this.dismissValidation.bind(this);
  this.showValidation = this.showValidation.bind(this);
  this.showSuccess = this.showSuccess.bind(this);
  this.showError = this.showError.bind(this);
  this.dismissToast = this.dismissToast.bind(this);
  this.onChange = this.onChange.bind(this);
  this.onValidityChange = this.onValidityChange.bind(this);
  this.onSubmit = this.onSubmit.bind(this);
}
```

```

...
  onSubmit(event) {
...
    this.setState({ issue: updatedIssue });
    alertthis.showSuccess('Updated issue successfully.');
```

});

} else {

response.json().then(error => {

alert**this.showError**(`Failed to update issue: \${error.message}`);

});

}

}).catch(err => {

alert**this.showError**(`Error in sending data to server: \${err.message}`);

});

}

```

...
  loadData() {
...
    this.setState({ issue });
    });
    } else {
      response.json().then(error => {
        alertthis.showError(`Failed to fetch issue: ${error.message}`);
      });
    }
    }).catch(err => {
      alertthis.showError(`Error in fetching data from server: ⚡
        ${err.message}`);
    });
  }
...
  showSuccess(message) {
    this.setState({ toastVisible: true, toastMessage: message, ⚡
      toastType: 'success' });
  }

  showError(message) {
    this.setState({ toastVisible: true, toastMessage: message, ⚡
      toastType: 'danger' });
  }

  dismissToast() {
    this.setState({ toastVisible: false });
  }
...
    </Form>
    <Toast
```

```

        showing={this.state.toastVisible} ←
        message={this.state.toastMessage}
        onDismiss={this.dismissToast} bsStyle={this.state.toastType}
      />
    </Panel>
  ...

```

There are similar changes in `IssueList`, except that we don't have any success messages, so we've skipped the `showSuccess` method. The changes for `IssueList` are shown in Listing 10-19.

Listing 10-19. `IssueList.jsx`: Changes Replacing Alert with Toast

```

...
import Toast from './Toast.jsx';
...
constructor() {
  super();
  this.state = {
    issues: [],
    toastVisible: false, toastMessage: '', toastType: 'success',
  };

  this.createIssue = this.createIssue.bind(this);
  this.setFilter = this.setFilter.bind(this);
  this.deleteIssue = this.deleteIssue.bind(this);
  this.showError = this.showError.bind(this);
  this.dismissToast = this.dismissToast.bind(this);
}
...
showError(message) {
  this.setState({ toastVisible: true, toastMessage: message, toastType:
    'danger' });
}

dismissToast() {
  this.setState({ toastVisible: false });
}
...
loadData() {
  fetch(`/api/issues${this.props.location.search}`).then(response => {
    if (response.ok) {
      ...
    } else {
      response.json().then(error => {
        alert(this.showError(`Failed to fetch issues ${error.message}`));
      });
    }
  });
}

```



```

    }).catch(err => {
      alertthis.showError(`Error in fetching data from server: ${err}`);
    });
  }
...
createIssue(newIssue) {
  fetch('/api/issues', {
    ...
  }).then(response => {
    if (response.ok) {
      ...
    } else {
      response.json().then(error => {
        alertthis.showError(`Failed to add issue: ${error.message}`);
      });
    }
  }).catch(err => {
    alertthis.showError(`Error in sending data to server: ${err.message}`);
  });
}
...
render() {
  return (
    <div>
      ...
      <IssueAdd createIssue={this.createIssue} />
      <Toast
        showing={this.state.toastVisible} ←
        message={this.state.toastMessage}
        onDismiss={this.dismissToast} bsStyle={this.state.toastType}
      />
    </div>
  );
}
...

```

Now that we've rid the application of `alert()` messages, we can now remove the exception rule that we had in `.eslintrc` in the client-side code directory. This change is shown in Listing 10-20.

Listing 10-20. `src/.eslintrc`: Remove Exception for Alert

```

{
  "env": {
    "browser": true
  },
  "rules": {

```

```

    "no-alert": ["off"]
  }
}

```

An error message can be caused by clicking Add Issue with the input fields as blank. A success message can be tested by saving an issue in the Edit page; see Figure 10-12. An error message will look similar, with a red background instead.

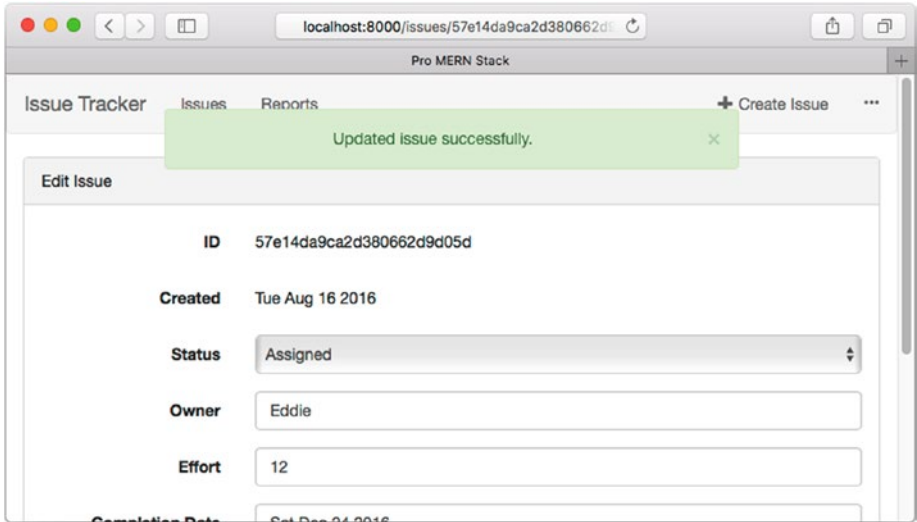


Figure 10-12. Successful Toast on saving an issue

Modals

In this section, we'll replace the in-page `IssueAdd` component with a modal dialog that is launched by clicking the `Create Issue` navigation item in the header. This is so that the user can create an issue from anywhere in the application, not just the `Issue List` page. Further, when the new issue is submitted, we'll show the newly created issue in the `Issue Edit` page, because this can be done regardless of where the dialog was launched from. Instead of a Modal dialog, the `Create Issue` can also be a separate page. But a Modal works better when the number of required fields is small; the user can quickly create the issue and later fill up more information if required.

When a Modal is rendered, it is rendered outside the main `<div>` of the DOM that holds the rest of the page. Thus, in terms of code placement, it can be placed anywhere in the component hierarchy. In order to launch or dismiss the Modal, the `Create Issue` navigation item is the controlling component. So, let's create a component that is self-contained: it displays the navigation item, launches the dialog and also controls its visibility, creates the issue, and routes to the issue edit page on a successful creation.

At the root of the component is a `<NavItem>`. In addition to the icon and text, this will also contain a `<Modal>` as one of its children. As discussed, placement of the `Modal` doesn't matter because when it is rendered, it is taken out of its original placement and rendered near the root of the DOM. The visibility of the `Modal` is controlled by a state variable: `showing`. The `<NavItem>`'s `onClick` will set this state to `true`, and the `Modal`'s `onHide` handler will set it to `false`.

Within the `Modal` are three sections: the header, the body, and the footer. We'll place a title in the header, and enable the Close button on the header using the `closeButton` property. The body will contain the form, a regular vertical form. The footer will contain a button toolbar for Submit and Cancel. To handle the submit, we'll move the code from `IssueList` to this component. On success, we'll navigate to the edit page of the newly created issue, based on its ID.

Finally, we also need to show error messages, so let's include a `Toast` within the `<NavItem>`, along with the required state variables and handlers as we've done in the previous sections. The entire code for this new component, called `IssueAddNavItem`, is shown in Listing 10-21.

Listing 10-21. `IssueAddNavItem.jsx`: New Component for Create Issue

```
import React from 'react';
import { withRouter } from 'react-router';
import { NavItem, Glyphicon, Modal, Form, FormGroup, FormControl,
  Controllabel, Button, ButtonToolbar } from 'react-bootstrap';

import Toast from './Toast.jsx';

class IssueAddNavItem extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      showing: false,
      toastVisible: false, toastMessage: '', toastType: 'success',
    };
    this.showModal = this.showModal.bind(this);
    this.hideModal = this.hideModal.bind(this);
    this.submit = this.submit.bind(this);
    this.showError = this.showError.bind(this);
    this.dismissToast = this.dismissToast.bind(this);
  }

  showModal() {
    this.setState({ showing: true });
  }

  hideModal() {
    this.setState({ showing: false });
  }
}
```

```

showError(message) {
  this.setState({ toastVisible: true, toastMessage: message, ←
    toastType: 'danger' });
}

dismissToast() {
  this.setState({ toastVisible: false });
}

submit(e) {
  e.preventDefault();
  this.hideModal();
  const form = document.forms.issueAdd;
  const newIssue = {
    owner: form.owner.value, title: form.title.value,
    status: 'New', created: new Date(),
  };
  fetch('/api/issues', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(newIssue),
  }).then(response => {
    if (response.ok) {
      response.json().then(updatedIssue => {
        this.props.router.push(`/issues/${updatedIssue._id}`);
      });
    } else {
      response.json().then(error => {
        this.showError(`Failed to add issue: ${error.message}`);
      });
    }
  }).catch(err => {
    this.showError(`Error in sending data to server: ${err.message}`);
  });
}

render() {
  return (
    <NavItem onClick={this.showModal}><Glyphicon glyph="plus" /> Create Issue
    <Modal keyboard show={this.state.showing} onHide={this.hideModal}>
      <Modal.Header closeButton>
        <Modal.Title>Create Issue</Modal.Title>
      </Modal.Header>
      <Modal.Body>
        <Form name="issueAdd">
          <FormGroup>
            <Controllabel>Title</Controllabel>

```

```

        <FormControl name="title" autoFocus />
      </FormGroup>
      <FormGroup>
        <Controllabel>Owner</Controllabel>
        <FormControl name="owner" />
      </FormGroup>
    </Form>
  </Modal.Body>
  <Modal.Footer>
    <ButtonToolbar>
      <Button type="button" bsStyle="primary" ←
        onClick={this.submit}>Submit</Button>
      <Button bsStyle="link" onClick={this.hideModal}>Cancel</Button>
    </ButtonToolbar>
  </Modal.Footer>
</Modal>
<Toast
  showing={this.state.toastVisible} message={this.state.toastMessage}
  onDismiss={this.dismissToast} bsStyle={this.state.toastType}
/>
</NavItem>
  );
}
}
}

IssueAddNavItem.propTypes = {
  router: React.PropTypes.object,
};

```

```
export default withRouter(IssueAddNavItem);
```

Let's take a closer look at injecting the router property. Firstly, we did not use `export default` as we would normally do in the class definition:

```
...
class IssueAddNavItem extends React.Component {
...

```

This means that the class is not exported. Instead, we imported `React-Router's withRouter` function, and used it to create a new component based on this component, and export that instead:

```
...
import { withRouter } from 'react-router';
...
export default withRouter(IssueAddNavItem);
...

```

This is similar to the `withRouter` usage within `App.jsx` to wrap `IssueList`, but this time we wrapped the component before exporting it. This ensures that the caller doesn't have to be aware of the fact that the component requires a router, and the usage of router is encapsulated within the component.

To use this new component, we'll modify the navigation bar, and use it in place of the `<NavItem>`. This change is shown in Listing 10-22.

Listing 10-22. `App.jsx`: Use `IssueAddNavItem` Instead of `NavItem`

```
...
import IssueAddNavItem from './IssueAddNavItem.jsx';
...
<Nav pullRight>
  <NavItem><Glyphicon glyph="plus" /> Create Issue</NavItem>
  <IssueAddNavItem />
  <NavDropdown id="user-dropdown" ↵
    title=<Glyphicon glyph="option-horizontal" /> noCaret>
...

```

Finally, we can now remove the `IssueAdd` component itself (no listing shown), and delete the code in `IssueList` that displays the inline form and the handler for adding an issue (these changes are shown in Listing 10-23).

Listing 10-23. `IssueList.jsx`: Removal of Issue Add Code

```
...
import IssueAdd from './IssueAdd.jsx';
...
this.createIssue = this.createIssue.bind(this);
...
createIssue(newIssue){
  fetch('/api/issues', {
    ...
  });
}
...
<IssueTable issues={this.state.issues} ↵
  deleteIssue={this.deleteIssue} />
<IssueAdd createIssue={this.createIssue} />
<Toast
...

```

Figure 10-13 shows the Create Issue modal dialog.

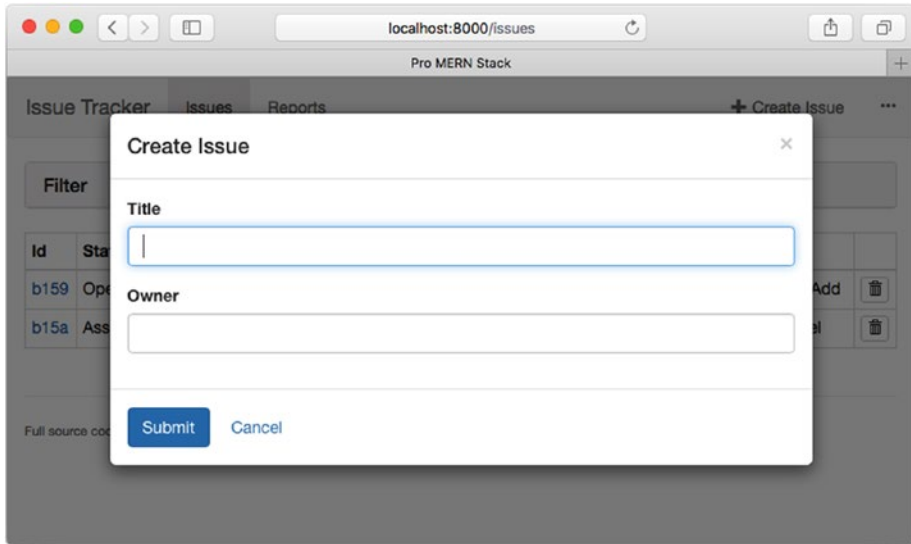


Figure 10-13. Create Issue Modal dialog

EXERCISE: MODALS

1. In the inline form version, we had to clear out the inputs after the form Add button was clicked. We didn't have to do the same in the Modal. Why? Hint: Inspect the DOM and locate the Modal both when it's visible and hidden.
2. We added a property called `keyboard` to the Modal. What effect does this have? Hint: Look in the React-Bootstrap documentation under Modals.

Answers are available at the end of the chapter.

Summary

Adding styles and themes to an application in a MERN stack is no different from any other stack because the important part is the CSS and how styles are handled by various browsers. And they don't vary depending on the chosen stack. Bootstrap, a pioneer in this area, gave us browser independence and a responsive behavior out of the box. React-Bootstrap replaced the separate JavaScript code that dealt with the Bootstrap elements, and gave us self-contained components.

We could have used Material-UI or any other framework to achieve what was required, but the take-away from this chapter should be a peek into how we can design our own reusable UI components if and when required

At this point in time, the application may look complete except for some advanced features. But if the application were something that we must let search engine bots index the pages naturally, we need to be able to serve the pages right from the server, as they would appear finally. In the next chapter, we'll learn how to do this and, more importantly, how to do this using the *same* code base on the client as well as the server.

Answers to Exercises

Exercise: Navigation

1. A plain `NavItem` with a `href` looks OK, but it doesn't automatically highlight the *current* route. This is because React-Bootstrap doesn't know which of the links is active. If we need to do it, we must handle it ourselves by using `activeHref` attribute of the `Nav` component, or setting `active` property of the `NavItem` by matching it with the URL. `react-router-bootstrap` automated all this in the `LinkContainer` component.

Another consequence of not using `Link` or `LinkContainer` is that a regular `<a href...>` causes the page to reload rather than replace the contents section of the app. This is because React Router does not know about the transition, and the browser does the reload of the page for the new URL that has been selected.

Exercise: Grid-Based Forms

1. If the cells were larger, a specification of `xs={12}` `sm={6}` `md={4}` `lg={3}` would work best. On very small and small screens, you'll see multiple lines, and on medium and large screens, the cells will fit into a single line.
2. To make the controls look smaller, we can use the `bsSize="small"` property on the `FormGroups`. But this does not work for buttons; we must specify the property for each button instead.

Exercise: Inline Forms

1. The `bsSize="small"` property can only be set on a `FormGroup`. If we want smaller looking controls, we must wrap the form controls within a `FormGroup` component, without a `ControlLabel`. For the button, we can set the style in the `Button` component itself.

2. To specify an exact width, you must use an inline style, such as `style={{ width: 300 }}`. Without a width specification, the control fills the width of the screen on very small screens. With a width, it takes the width specified on all screen sizes. In effect, if we do set the width, it's better to set the size on all controls rather than some.

Exercise: Modals

1. Every time the Modal is dismissed, the component is destroyed, and recreated whenever the Modal is launched. This is different from a `<Collapse>`, which uses a `display: none` style to hide itself. Thus, on recreation of the Modal element, the input fields are recreated as well and have no initial value. Whereas in the inline form, the input fields were not recreated, so we had to clear them out ourselves.
2. The property `keyboard` on the Modal lets the user press the Escape key to dismiss the modal dialog.



Server Rendering

In this chapter, we'll explore another cornerstone of React, the ability to generate HTML on the server in addition to being able to render directly to the DOM. This lets you create *isomorphic* applications, that is, applications that use the same code base on the server as well as the client to do either task: render to the DOM or create HTML.

One of the benefits of server rendering is that it improves the performance and thus the user experience. But the real need for it is when you want your application to be indexed by search engines. Search engine bots typically start from the root URL (/) and then traverse all the hyperlinks present in the HTML that the root URL returns. They do not execute JavaScript to fetch data via Ajax calls and look at the resulting DOM. So, if you want pages from your application to be properly indexed by search engines, you need to serve the exact same HTML that will result *after* the Ajax call in `componentDidMount()` calls.

For example, if a request is made directly to `/issues`, you must return HTML with the list of issues in the table prepopulated. The same goes for all other pages that can be bookmarked or have a hyperlink pointing to them. At the end of this chapter, the Issue Tracker application will have both the Issue List page and the Issue Edit page capable of being rendered on the server, using the *same* code base, that is, isomorphic.

A note of caution: not all applications need server rendering. If your application does not need to be indexed by search engines, the complexity introduced by server rendering can be avoided. The performance benefits alone do not justify server rendering.

Basic Server Rendering

The key API that React provides to let you render at the server is `renderToString()`. Although the API itself is a simple one, the changes that we need to do to use it are not. So, before we add server rendering capabilities to all the components in the Issue Tracker application, let's use a simple Hello World component so that we get familiar with the fundamentals. The code for the very basic Hello World component is shown in Listing 11-1, and we'll use this component to start exploring server rendering.

Listing 11-1. HelloWorld.jsx: Hello World Component for Server Rendering

```
import React from 'react';

export default function HelloWorld() {
  return (
    <h1>Hello World!</h1>
  );
}
```

Initiating the render is different on the client and the server, so let's create a new entry point that is specific to the client alone, and place it in a new directory, `client`. The entry point, `Client.jsx`, is shown in Listing 11-2.

Listing 11-2. Client.jsx: New Entry Point for Client

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

import HelloWorld from '../src/HelloWorld.jsx';

const contentNode = document.getElementById('contents');
ReactDOM.render(<HelloWorld />, contentNode);

if (module.hot) {
  module.hot.accept();
}
```

On the server side, we can call `renderToString()` to render the same component, `HelloWorld`. This will give us an HTML string representation of the rendered component. But we also need to place it inside the `<div>` with ID `contents` in `index.html` before we send it to the client. So, let's make a *template* out of `index.html` that can accept the contents of the `<div>` and return the complete HTML. Powerful templating languages such as `pug` (earlier known as `jade`) can be used for this, but our requirement is quite simple, so we'll just use the ES2015 template strings feature. Let's place this function in a file called `template.js` under the server directory. The entire code for this file is shown in Listing 11-3.

Listing 11-3. `template.js`: A Templated Version of `index.html`

```
export default function template(body) {
return `<!DOCTYPE HTML>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Pro MERN Stack</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="/bootstrap/css/bootstrap.min.css" >
}`
```

```

<style>
  .panel-title a {display: block; width: 100%; cursor: pointer; }
</style>
</head>

<body>
  <div id="contents">${body}</div> ←
  <!-- this is where our component will appear -->
  <script src="/vendor.bundle.js"></script>
  <script src="/app.bundle.js"></script>
</body>

</html>
`;
}

```

As you can see, the function just returns the original contents of `index.html`, with the variable `${body}` within the `<div>` contents. We can now delete the original file, `index.html` (there's no listing shown for this).

As for server-side routing, we can no longer match all requests and return `index.html` from the static directory. We should be able to match specific paths and return the components rendered using the new template. Let's put all the server rendering and routing code in a separate file under the server directory and call it `renderedPageRouter.jsx`. Note that we're assigning it a `.jsx` extension since we'll be putting JSX code to initiate the render, in the form of the `<HelloWorld>` component. The contents of this new file are shown in Listing 11-4.

Listing 11-4. `renderedPageRouter.jsx`: New Server-Side Rendering and Routing File

```

import React from 'react';
import { renderToString } from 'react-dom/server';
import Router from 'express';

import HelloWorld from '../src/HelloWorld.jsx';
import template from './template.js';

const renderedPageRouter = new Router();

renderedPageRouter.get('*', (req, res) => {
  const html = renderToString(<HelloWorld />);
  res.send(template(html));
});

export default renderedPageRouter;

```

React's `renderToString()` API is available not directly from `react-dom` but from a subdirectory called `server`, so the `import` statement used that subdirectory. Server rendering is as simple as calling this API to get the HTML for the rendered component, and then using the template to build the complete HTML to send back to the client.

For now, we'll using a `"*"` match, that is, something that matches any request. Unlike the previous express routes that took the form `app.get()`, etc., we are now creating a *router* instance to which we can attach multiple routes as we did in the app. The Express Router documentation says you can think of a Router instance as a mini-application capable of performing only routing and middleware functions. Since the router behaves like a middleware instance, you can just use it in an application's `use()` method.

And that's just what we'll do. We'll replace the catch-all route in `server.js` with a `use()` method for the new router. The changes in `server.js` are shown in Listing 11-5.

Listing 11-5. `server.js`: Changes to Separate Out Server Rendering

```
...
import path from 'path';
...
import renderedPageRouter from './renderedPageRouter.jsx';
...
app.get('*', (req, res) => {
  res.sendFile(path.resolve('static/index.html'));
});
app.use('/', renderedPageRouter);
...
```

To put all this to work, we also need a few changes to the build process. First, we need a change to the webpack configuration: the entry point for initiating the application bundle is now `Client.jsx`. Further, the dev-server needs to proxy *all* requests to the back-end server, not just API calls. This is because we'll be receiving calls to other URLs such as `/issues` as well, and we will want to return a prefilled page rather than the contents of `index.html`. This also means that static files in the `static` directory will no longer be served by the dev-server directly; instead, they will be routed to the Express server. This is really OK because all the static content now will come from libraries such as bootstrap, which will be cached by the browser anyway. See Listing 11-6.

Listing 11-6. `webpack.config.js`: Changes for New Entry Point and dev Server

```
...
entry: {
  app: './src/App.jsx',
  app: './client/Client.jsx',
  vendor: ['react', 'react-dom', 'whatwg-fetch'],
},
...
devServer: {
```

```
...
proxy: {
  '/api/*': {
    '**': {
      target: 'http://localhost:3000',
    },
  },
},
...

```

At this point, the `start_hook` way of starting the server works, with just one change to include the `react` preset (since we now have React components on the server side as well), as shown in Listing 11-7.

Listing 11-7. `start_hook.js`: Include React Preset

```
require('babel-register')({
  presets: ['es2015-node4', 'react'],
});

require('./server.js');
```

But we'll also need to tell `nodemon` to watch for changes to the `src` directory in addition, because any change in that directory also affects the server side code now onwards.

For production, we'd either have to include the `react` preset when compiling the server-side files using `babel`, or create a bundle just like in the client side. It turns out that it's not enough to compile the server-side files; we also need to compile all the React components, since they are also now part of the dependency tree. Thus, creating a bundle works best, even for the server. But we'll leave this for a later section and use the `start hook` method of running the server for now, and remove the other methods. All these changes are in `package.json`, as shown in Listing 11-8.

Listing 11-8. `package.json`: Changes for `start-hook` to Restart on Client Code Changes, Disable `babel` Method

```
...
"scripts": {
  "start": "nodemon -w dist dist/server.js",
  "compile-server": "babel server --presets es2015-node4 --out-dir dist ~
  --source-maps",
  "watch-server": "babel server --presets es2015-node4 --out-dir dist ~
  --source-maps --watch",
  "start-hook": "nodemon -e js,jsx -w server server/start_hook.js",
  "start-hook": "nodemon -e js,jsx -w server,src server/start_hook.js",
  "compile": "webpack",
  "watch": "webpack-dev-server --hot --inline",
  "dev-all": "npm run watch & npm run watch-server & npm start",
  "dev-all-hook": "npm run watch & npm run start-hook",
  ...

```

Now, browsing to *any* URL (since we matched all routes) should show a message like in Figure 11-1. Further, changes to `HelloWorld.jsx` should reflect in the browser automatically.

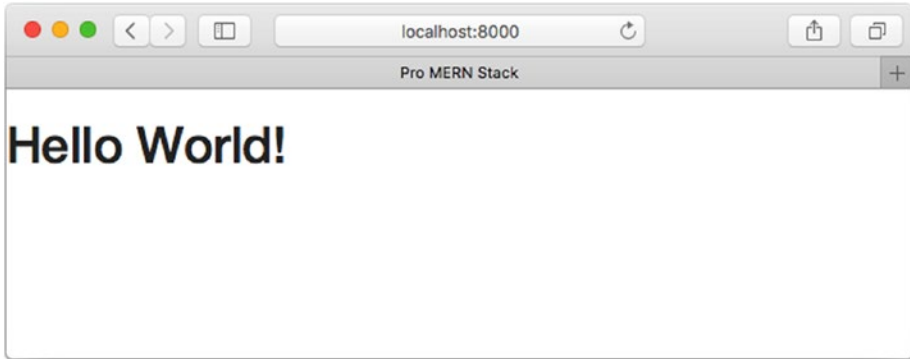


Figure 11-1. *Hello World rendered from server*

To ensure that you are getting the message prefilled from the server, take a look at the Network tab of the developer console in the browser and inspect the contents of initial response. Alternatively, you can use `curl` to make a request to the local server and see that the string `Hello World!` is indeed part of the response HTML.

Handling State

The Hello World component is trivial. It doesn't have things like state and data fetch from the server like the React components that need data from the server. (`IssueList` and `IssueEdit`). So, let's simulate what really happens in those components. Essentially, they start with an empty state. They then make an Ajax call to get the initial state and rerender the view with the state when the initial state is available.

We'll modify `HelloWorld` so that it takes in a message as a state parameter and displays that while rendering. Further, we'll start with an empty state and fill the state using an asynchronous call in the `componentDidMount()` method. Now, for server rendering, let's just hard-code this value to the same as that in the client. Let's pass this state via props to the component when rendering from the server whereas, when rendering from client, we'll assume that the asynchronous call will do the needful. Therefore, we won't pass any props from the client, but we'll set the default to an empty state.

The new component is shown in Listing 11-9.

Listing 11-9. HelloWorld.jsx: Stateful Component with Props for Server Rendering Initial State

```
import React from 'react';

export default class HelloWorld extends React.Component {
  constructor(props) {
    super(props);
    this.state = Object.assign({}, this.props);
  }

  componentDidMount() {
    setTimeout(() => {
      this.setState({ addressee: 'Universe' });
    }, 100);
  }

  render() {
    return (
      <h1>Hello {this.state.addressee}!</h1>
    );
  }
}

HelloWorld.propTypes = {
  addressee: React.PropTypes.string,
};

HelloWorld.defaultProps = {
  addressee: '',
};
```

The state variable has a single key, `addressee`, which we are using to parametrize whom we're saying Hello to. We'll pass 'Universe' as the addressee from the server, and in the simulated Ajax call (which is just a `setTimeout()` call). The corresponding changes in server rendering is shown in Listing 11-10.

Listing 11-10. renderedPageRouter.jsx

```
renderedPageRouter.get('*', (req, res) => {
  const initialState = { addressee: 'Universe' };
  const html = renderToString(<HelloWorld {...initialState} />);
  res.send(template(html));
});
```

On testing this, you can see that the page appears correctly, but there's a flash (if you miss this, you can try increasing the milliseconds parameter in the `setTimeout()` call). If you inspect the network traffic and look at the HTML response, you can indeed

see that the server is sending out ‘Universe’ as part of the HTML response, but the initial rendering shows only a blank addressee! You can confirm this by commenting out `componentDidMount()` method; you’ll find that the addressee is always blank.

On closer inspection, you’ll also see an error in the console, as in Figure 11-2.

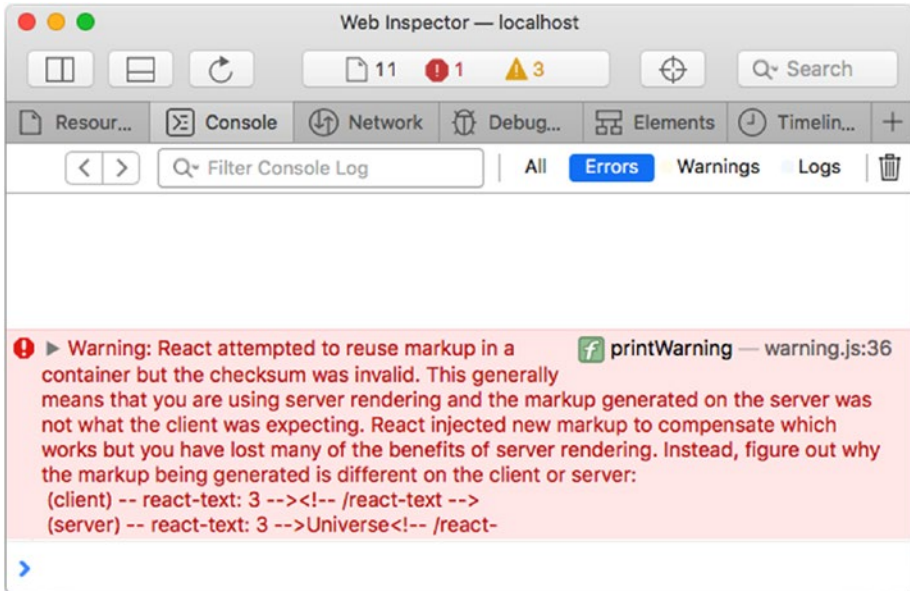


Figure 11-2. Error in console and the initial state from the server is rejected

As the error indicates, it appears that React tried to render the component on the client side, but found that this resulted in something different from what was sent from the server. What’s more, it *rejected* the server rendering and replaced it with what it thought was the correct DOM generated using `ReactDOM.render()` at the client. We’ll address this in the next section.

Initial State

The error in the previous section really means this: we need to ensure that the initial rendering on the client matches the page rendered at the server. The most common way to do this is to send the initial state also to the client along with the rendered page. Now, when the component is rerendered for the first time at the client, it uses this state, which results in the same DOM as is sent from the server so that React doesn’t see a problem with it.

To send the state to the client, we need to create a `<script>` section in the template and set a variable in that script that holds the initial state. The client rendering can now pass this state as the props to `HelloWorld`, just like it is done on the server. The first

change we'll make is to the template, which will now take two parameters: one for the body and another for the script. The changed template function is shown in Listing 11-11.

Listing 11-11. template.js: Changes to Include a Script in the Template

```
...
export default function template(body, initialState) {
  ...
  <body>
    <div id="contents">${body}</div> ⚡
    <!-- this is where our component will appear -->
    <script>window.__INITIAL_STATE__ = ${JSON.stringify(initialState)}; ⚡
    </script>
    <script src="/vendor.bundle.js"></script>
    <script src="/app.bundle.js"></script>
  </body>
  ...
}
```

We've named the initial state variable as something unique (with leading and trailing double underscores) because it has to reside in the global namespace. Now, while rendering on the server, we can pass the initial state to the template to generate HTML that includes it as a script. This change is shown in Listing 11-12.

Listing 11-12. renderedPageRouter.jsx: Changes for Including Initial State in the Template

```
...
renderedPageRouter.get('*', (req, res) => {
  const initialState = { addressee: 'Universe' };
  const html = renderToString(<HelloWorld {...initialState} />);
  res.send(template(html, initialState));
});
...
```

In `Client.jsx`, when we carry out the initial render, we can pass in the initial state that we received as part of the script. The changes for this are shown in Listing 11-13, which includes an ESLint rule exception since we have underscores around the variable name.

Listing 11-13. Client.jsx: Pass Initial State as Props from Client Also

```
...
/* eslint no-underscore-dangle: 0 */
ReactDOM.render(<HelloWorld {...window.__INITIAL_STATE__} />, contentNode);
...
```

On testing this set of changes, you'll find that the error message and the flash are now gone. A look at the response from the server using the network inspection tool will confirm that in addition to the `<div>` being populated, you also have a script that holds the initial state.

■ **Note** The use of a script will not work if there is any string in the initial state that has `</script>` in it, because this will end up terminating the script tag. An alternative some people use to work around this is to use another `<div>`, which is hidden to hold the contents of the stringified initial state. This string must to be parsed using `JSON.parse()` before use, though.

Server-Side Bundle

For production use, we cannot use the require hook. We need to precompile the server code. The babel compiler that we used until now is not convenient, since it does not manage dependencies, which are now getting larger. So, let's use webpack as we did in the client code to bundle the server code as well. We'll use a different configuration file, `webpack.server-config.js`, for this because the server configuration for webpack has to be different.

One difference between the client and the server is that we had to bundle the third-party libraries in a separate vendor bundle, whereas on the server we have these libraries already installed. We don't really need a bundle for them. Also, as in the client, we included two bundles in `index.html`, but there is no way to do that in the server because there is no encompassing `index.html`. We have to start the server with a single script: the application code bundle.

In the production environment, having the entire code in a single bundle or multiple bundles will not make any difference because the code will be loaded only once. But during development, creating the entire bundle every time we change the server code (or even the client code, since it is now part of the bundle) is going to be a hassle. So, let's separate the vendor and application code even for the server.

Just like specifying which files can go into a separate bundle, webpack lets you declare some dependencies as *external*. This tells webpack that these dependencies don't need to be included in the bundle, and the regular `require()` statement will resolve it as an installed module dependency. Rather than specify each external module that the server code depends on, we'll just use a pattern to say that anything that is not imported as a local dependency (that is, starting with `./`) can be considered an external. We'll use a simple `a-z` regex for this to say that any import statement with the module name starting with the characters `a-z` is an external module.

We also have to include a `target` entry to indicate that the output is meant to be run using Node.js (the default is `browser`, that's why we didn't have to do this for the client-side bundle), and a `resolve.extensions` parameter to indicate that `.jsx` files also are dependencies since the default does not include it. The webpack server configuration file is shown in Listing 11-14.

Listing 11-14. `webpack.server-config.js`: Webpack Configuration for Bundling Server

```
module.exports = {  
  target: 'node',  
  entry: './server/server.js',
```

```

output: {
  path: './dist',
  filename: 'server.bundle.js',
  libraryTarget: 'commonjs',
},
resolve: {
  extensions: ['', '.js', '.jsx'],
},
externals: [/^[a-z]/],
module: {
  loaders: [
    {
      test: /\.jsx$/,
      loader: 'babel-loader',
      query: {
        presets: ['react', 'es2015-node4'],
      },
    },
    {
      test: /\.js$/,
      exclude: /node_modules/,
      loader: 'babel-loader',
      query: {
        presets: ['es2015-node4'],
      },
    },
  ],
},
devtool: 'source-map',
};

```

The module loaders are similar to the client-side webpack configuration, so they need no explanation. To compile using this using npm command line, we need to change the server compile scripts in `package.json` from using babel to webpack. We also need to change the entry point for the start script from `server.js` to the generated bundle: `server.bundle.js`. Finally, since React and other libraries are now dependencies that we'll require at runtime, we need to move them from `devDependencies` to regular dependencies. These changes, all part of `package.json`, are shown in Listing 11-15.

Listing 11-15. `package.json`: Changes for Server-Side Bundling

```

...
"scripts": {
  "start": "nodemon -w dist dist/server.bundle.js",
  "compile-server": "webpack --config webpack.server-config.js",
  "watch-server": "webpack --config webpack.server-config.js --watch",
...

```

```

"dependencies": {
  ...
  "object-assign": "^4.1.0",
  "react": "^15.3.1",
  "react-bootstrap": "^0.30.6",
  "react-dom": "^15.3.1",
  "react-router": "^2.7.0",
  "react-router-bootstrap": "^0.23.1",
  ...
"devDependencies": {
  ...
  "react": "^15.3.1",
  "react-bootstrap": "^0.30.6",
  "react-dom": "^15.3.1",
  "react-router": "^2.7.0",
  "react-router-bootstrap": "^0.23.1",
  ...

```

■ **Note** Due to a bug in npm dependency evaluation, if you are using npm version less than 3, you will find that some packages need `object-assign`, but it's not resolved. You will need to install `object-assign` module manually. This is the case for Node.js version 4.x, which comes with an npm version less than 3. For Node.js version 6, this is not required.

Now, you should be able to use `compile-server` and `watch-server` npm run commands to compile the server-side code and test the Hello World component.

Back-End HMR

Earlier, the server-side and client-side code were independent, and each set of changes would only cause one particular bundle to be rebuilt. But now, any client-side code change will cause both the server and the client bundles to be rebuilt. Thus, you'll see that the server is restarted even for all client-side code changes.

This causes a problem in the smooth development process we had until now: while the browser reloads the new modules due to the client bundle changes, the server is also restarting. If there were API calls being made during the browser reload, they may fail because the server is temporarily down. In the `start-hook` method of running the server, the problem is compounded since the time taken is much longer: the server is down while compiling in addition to restarting.

Wouldn't it be great if we could use the hot module replacement (HMR) mechanism that we used in the browser also on the server? This is indeed possible, but the changes

are not the same as in the browser. That's because, with the browser, we don't have to manage an existing state when the code is replaced. With the Node.js server, the request handler for the HTTP server needs to be removed and a new one instantiated and added with the new code.

To be able to do this, we first need to separate out the code that creates the server from the rest of the Express code, the real request handler. Let's call this new file `index.js`. The contents of this file are shown in Listing 11-16.

Listing 11-16. `server/index.js`: New File for Starting the Server and HMR

```
import SourceMapSupport from 'source-map-support';
SourceMapSupport.install();
import 'babel-polyfill';
import http from 'http';

import { MongoClient } from 'mongodb';

let appModule = require('./server.js');
let db;
let server;

MongoClient.connect('mongodb://localhost/issuetracker').then(connection => {
  db = connection;
  server = http.createServer();
  appModule.setDb(db);
  server.on('request', appModule.app);
  server.listen(3000, () => {
    console.log('App started on port 3000');
  });
}).catch(error => {
  console.log('ERROR:', error);
});

if (module.hot) {
  module.hot.accept('./server.js', () => {
    server.removeListener('request', appModule.app);
    appModule = require('./server.js');    // eslint-disable-line
    appModule.setDb(db);
    server.on('request', appModule.app);
  });
}
```

Instead of using `app.listen()`, we are now creating the server using the `http` built-in module of Node.js. We need three steps now, `http.createServer()`, `server.on()`, and `server.listen()`, instead of the single statement `app.listen()`. The two methods are identical, as explained in the Express documentation. We need

this new method so that we can keep a handle to the server and reinstall a new request handler when needed. The following statement installs the request handler:

```
server.on('request', <request handler>);
```

Since multiple request handlers can be installed on a server, we also need a way to remove an installed handler:

```
server.removeListener('request', <request handler>);
```

In the `accept()` method of `module.hot`, the first parameter indicates the dependent module, the replacement of which you need to handle. When that module is replaced, the second parameter, the callback, is called. There is only one dependent module, which will give us a function for setting the DB connection, and another, the request handler, an `express app()`. We'll soon look at the modified `server.js` that will provide them.

In the callback that's called when the module (or any if *its* dependents) is modified, all we did was remove the old request handler and install the new one. Since we don't want to create a new database connection on a module reload, we kept the connection as a global variable in this file itself and supply it to the new module, both at the beginning and whenever it is replaced.

The next set of changes is in `server.js`. This includes removal of the code that started the server, and also a new function called `setDb()` to accept the database connection as a variable. We'll export the `express app` and this new function. The changes to `server.js` are shown in Listing 11-17.

Listing 11-17. `server.js`: Changes for Separating Out HTTP Server Code from the App

```
...
import SourceMapSupport from 'source-map-support';
SourceMapSupport.install();
import 'babel-polyfill';
...
import { MongoClient, ObjectId } from 'mongodb';
...
MongoClient.connect('mongodb://localhost/issuetracker').then(connection => {
  db = connection;
  app.listen(3000, () => {
    console.log('App started on port 3000');
  });
}).catch(error => {
  console.log('ERROR:', error);
});
...
function setDb(newDb) {
  db = newDb;
}

export { app, setDb };
...
```

Next, the server webpack configuration needs a few changes to enable HMR. The first is an additional entry point that injects the code that listens for module changes. We can do this by polling at a certain interval to see if there are changes. An alternative is to use the signal method, which sends a signal to a listening socket in the running server. This is a little more involved for setting up, so we will use the poll method.

The other change is to include the plugin that provides this feature. The changes to the webpack configuration are shown in Listing 11-18.

Listing 11-18. webpack.server-config.js: Changes for Back-End HMR

```
...
const webpack = require('webpack');

module.exports = {
  target: 'node',
  entry: './server/server.js',
  entry: ['./server/index.js', './node_modules/webpack/hot/poll?1000'],
  ...
  plugins: [
    new webpack.HotModuleReplacementPlugin(),
  ],
  ...
}
```

In the entry point, we could have used just `webpack/hot/poll` rather than specify the path within `node_modules`. But it so happens that the entry point cannot be an external module; it has to be part of the bundle for it to work. Since the pattern for deciding externals depends on the import statement *not* starting with `./`, we did just that. There are other mechanisms for achieving this; a popular one is using the package `webpack-node-externals`, which lets you create a list of node modules excluding one or more specific modules. The query string to the entry point tells the poller how frequently (in milliseconds) to look for changes.

Finally, in `package.json`, the start script for the server needs to change so that we no longer use nodemon. We can directly start the server, and it will poll for changes via the HMR mechanism. This change is shown in Listing 11-19.

Listing 11-19. package.json: Start the Server Without nodemon

```
...
"scripts": {
  "start": "nodemon -w dist dist/server.bundle.js",
  "start": "node dist/server.bundle.js",
}
...
```

Now, you can use the `dev-all npm` command to start all three processes: the webpack dev server to watch for client changes, the webpack watcher for the server, and the server starter. This is also a good time to test the HMR by changing the Hello World

string to something else, and checking if the server reloads the new module. You should see something like this in the console:

```
[HMR] Updated modules:
[HMR]   - 13
[HMR]   - 10
[HMR]   - 6
[HMR]   - 1
[HMR] Update applied.
```

Note that a change in `index.js` itself will fail because we haven't installed any handler to handle this change. We need to restart the server if we change this file. That's a good reason to keep the contents of this file to a minimum.

EXERCISE: BACK-END HMR

1. Change the `module.accept()` code to accept changes to self, that is, without any parameters or handlers, as we did in `Client.jsx`. What happens when a file is changed? Can you explain the behavior?

Answers are available at the end of the chapter.

Routed Server Rendering

Although server rendering worked well for a simple Hello World component, applying the same changes to the Issue Tracker application is not that simple. The reason is that the application uses React Router, which poses the following challenges:

1. On the server, we can't use React-Router's Router directly because it requires a real DOM. Also, browser history doesn't make sense here: there is no browser and therefore no history.
2. The initial state that we need to inject depends on the main component that's being rendered. In the case of `IssueList`, we need the list of issues, whereas in the case of `IssueEdit`, we need a single issue.
3. The component that needs the initial state is nested deep in the hierarchy; we cannot directly set the props for these components from the client rendering code.

A look at React-Router's documentation under the "Server Rendering" sections tells us that the makers of React-Router have thought about this and have specific solutions. To

address the first point, we need a different component to deal with server-side rendering called `RouterContext`. This is similar to `Router`, but is meant for the server. Also, we need to run the matching ourselves using the `match()` API. Both `Router` (on the client side) and `match()` on the server side expect the same set of routes. Rather than repeat the code, we'll move the contained routes into a separate file. The new file under `src` directory is shown in Listing 11-20.

Listing 11-20. `Routes.jsx`: Separate the Routes from `App.jsx` into a New File

```
import React from 'react';
import { Route, IndexRedirect, withRouter } from 'react-router';

import App from './App.jsx';
import IssueList from './IssueList.jsx';
import IssueEdit from './IssueEdit.jsx';

const NoMatch = () => <p>Page Not Found</p>;

export default (
  <Route path="/" component={App} >
    <IndexRedirect to="/issues" />
    <Route path="issues" component={withRouter(IssueList)} />
    <Route path="issues/:id" component={IssueEdit} />
    <Route path="*" component={NoMatch} />
  </Route>
);
```

Note that we had to use an `IndexRedirect` rather than a `Redirect` directly under `Router`, so that we could put all the necessary elements under one top-level route for `"/"`. Now, we can remove the routes from `App.jsx`. We can also delete other code we have already moved to `Client.jsx` and make it a pure component. These changes are listed in Listing 11-21.

Listing 11-21. `App.jsx`: Remove Routing and Entry Point Code

```
...
import ReactDOM from 'react-dom';
import { Router, Route, Redirect, browserHistory, withRouter } from  'react-router';
...
import IssueList from './IssueList.jsx';
import IssueEdit from './IssueEdit.jsx';
...
const contentNode = document.getElementById('contents');
const NoMatch = () => <p>Page Not Found</p>;
...
```

```

const RoutedApp = () => (
  <Router history={browserHistory}>
    <Redirect from="/" to="/issues"/>
    <Route path="/" component={App}>
      <Route path="issues" component={withRouter(IssueList)} />
      <Route path="issues/:id" component={IssueEdit} />
      <Route path="*" component={NoMatch} />
    </Route>
  </Router>
);

ReactDOM.render(<RoutedApp />, contentNode);

if (module.hot) {
  module.hot.accept();
}
...
export default App;

```

The last line to export the component is now required since it is referred to by the routes specification. `Routes.jsx` can now be used by server rendering to create a match and act upon the results of the match. But before we do this, let's deal with the second challenge: the initial state depends on *which* component (`IssueEdit` or `IssueList`) is being rendered. For now, we'll take advantage of the fact that the URLs of the page and the APIs are identical except for the `/api` prefix. We'll just make a call to the API to fetch the initial state data, like this:

```

...
fetch(`http://localhost:3000/api${req.url}`)
...

```

For example, if the call to the server is `/issues`, we'll make a call to `/api/issues` to fetch the data for the initial state. But the `fetch()` API is not available on Node.js; it's meant only for the browser. Fortunately, there's an npm module called `isomorphic-fetch` that makes this API available on Node.js via the `request` package, and `whatwg-fetch()` on browsers. Let's replace `whatwg-fetch` with the new package:

```

$ npm uninstall --save whatwg-fetch
$ npm install --save isomorphic-fetch

```

We also need to replace `whatwg-fetch` in `webpack.config.js` in the vendor section. This change is shown in Listing 11-22.

Listing 11-22. webpack.config.js: Changes to Replace whatwg-fetch with isomorphic-fetch

```
...
  vendor: [
    'react', 'react-dom', 'react-router', 'react-bootstrap', ←
    'react-router-bootstrap',
    'whatwg-fetch', 'isomorphic-fetch', 'babel-polyfill',
  ],
...
```

The last challenge was to be able to set the initial state property deep inside the component hierarchy. To achieve this, we'll use React's Context feature, which lets us make a set of data available in all the descendants of a component.

■ **Note** The context feature of React is an experimental one, and the recommendation from the authors of React is to use state management libraries such as Redux and Mobx. But this is a complex topic and is also tightly coupled with the Flux architecture, which at the moment is not relevant for server rendering. In production applications, you may choose to build your own global store to store the initial state, or use Redux or Mobx in place of Context.

Let's briefly explore how Context works. The parent component declares its context variables that are available to all descendants using a static variable, `childContextTypes`. This is very much like `propTypes`, which declares the names and types of props variables that are available to this component. As for the actual value of the context variables, the parent component defines a method `getChildContext()`, which needs to return an object, with each context variable as the key and its value as the value of the context variable.

As for accessing the variables exposed by a parent, within each descendant component you need to declare the context variables that are being *received*. You do this using the static variable `contextTypes`, again very similar to `propTypes`. Now, the descendant component can access these context variables using `this.context.<variable>` anywhere in the component's methods. For the constructor alone, just like props is not part of this, context is also passed into the constructor's arguments, so you need to declare the constructor as `constructor(props, context)`.

We'll first create the parent component that will wrap Router or RouterContext and provide the initial state to all descendants. Let's call this component `ContextWrapper`, the code for which is shown in Listing 11-23.

Listing 11-23. ContextWrapper: New Component to Inject Initial State to All Descendants

```
import React from 'react';

export default class ContextWrapper extends React.Component {
  getChildContext() {
    return { initialState: this.props.initialState };
  }

  render() {
    return this.props.children;
  }
}

ContextWrapper.childContextTypes = {
  initialState: React.PropTypes.object,
};

ContextWrapper.propTypes = {
  children: React.PropTypes.object.isRequired,
  initialState: React.PropTypes.object,
};
```

We've used a single context variable called `initialState`. We'll let the callers and components determine its contents. We're now ready to modify both the server and the client rendering entry point to use the wrapper and set the initial state in it. The new code for `renderedPageRouter` on the server is very similar to the code in the React Router's server rendering documentation. Since most of the code is rewritten, Listing 11-24 shows the new file in its entirety.

Listing 11-24. `renderedPageRouter.jsx`: Changes for Dealing with Routes

```
import React from 'react';
import { renderToString } from 'react-dom/server';
import { match, RouterContext } from 'react-router';

import Router from 'express';

import template from '../template.js';
import routes from '../src/Routes.jsx';
import ContextWrapper from '../src/ContextWrapper.jsx';

const renderedPageRouter = new Router();
```

```

renderedPageRouter.get('*', (req, res) => {
  match({ routes, location: req.url }, ⚡
    (error, redirectLocation, renderProps) => {
      if (error) {
        res.status(500).send(error.message);
      } else if (redirectLocation) {
        res.redirect(302, redirectLocation.pathname + redirectLocation. search);
      } else if (renderProps) {
        fetch(`http://localhost:3000/api${req.url}`).then(response => ⚡
          (response.json()))
          .then(data => {
            const initialState = { data };
            const html = renderToString(
              <ContextWrapper initialState={initialState} >
                <RouterContext {...renderProps} />
              </ContextWrapper>
            );
            res.status(200).send(template(html, initialState));
          })
          .catch(err => {
            console.log(`Error rendering to string: ${err}`);
          });
      } else {
        res.status(404).send('Not found');
      }
    }
  ));
});

export default renderedPageRouter;

```

The main change from sending out a hard-coded `<HelloWorld>` component is the call to `match()`, which is passed a set of React-Router routes. On a successful match, it calls the supplied callback with a possible error, redirect, or `renderProps`. The handling of the error and redirect are straightforward. As for `renderProps`, it contains the routes and some other data that is to be sent to the `RouterContext` to complete the rendering. Before we rendered to a string, we needed the data, which we got using the `fetch()` API on the API-ized version of the request URL. We used this data as initial state and passed it on to `ContextWrapper`.

We have to continue to use “*” in the Express router match, since we really don’t know if the requested URL has a match or not. This is done by the `match()` call, and if it doesn’t return a `renderProps`, we know that React-Router couldn’t match the request, and we can return a 404.

The client-side entry point for rendering is similar: we’ll use the context wrapper again, but this time we’ll get the initial state from the global `__INITIAL_STATE__` variable. Also, we will use a `Router` in the client side to wrap the routes instead of using `match()` and a `RouterContext`. The new `Client.jsx` is shown in Listing 11-25.

Listing 11-25. Client.jsx: Replaced Hello World with Issue Tracker Application

```

import 'babel-polyfill';
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router';

import routes from '../src/Routes.jsx';
import ContextWrapper from '../src/ContextWrapper.jsx';

const WrappedApp = (props) => (
  <ContextWrapper {...props}>
    <Router history={browserHistory} >
      {routes}
    </Router>
  </ContextWrapper>
);

const contentNode = document.getElementById('contents');
/* eslint no-underscore-dangle: 0 */
ReactDOM.render(<WrappedApp initialState={window.__INITIAL_STATE__} />, ←
contentNode);

if (module.hot) {
  module.hot.accept();
}

```

Finally, we need to use the global initial state in the components `IssueList` and `IssueEdit` to set *their* initial state. The change for `IssueList` also contains the replaced reference to `whatwg-fetch`. These changes are shown in Listing 11-26 and 11-27.

Listing 11-26. `IssueList.jsx`: Changes to Get Initial State from Global Variable

```

...
import 'whatwg-fetch';
import 'isomorphic-fetch';
...
constructor (props, context) {
  super (props, context);
  const issues = context.initialState.data.records;
  issues.forEach(issue => {
    issue.created = new Date(issue.created);
    if (issue.completionDate) {
      issue.completionDate = new Date(issue.completionDate);
    }
  });
}

```

```

    this.state = {
      issues: [],
      issues,
    ...
    IssueList.contextTypes = {
      initialState: React.PropTypes.object,
    };
    ...

```

Listing 11-27. IssueEdit.jsx: Changes to Get Initial state from Global Variable

```

...
constructor(props, context) {
  super(props, context);
  const issue = context.initialState.data;
  issue.created = new Date(issue.created);
  issue.completionDate = issue.completionDate != null ?
    new Date(issue.completionDate) : null;
  this.state = {
    issue: {
      id: '', title: '', status: '', owner: '', effort: null,
      completionDate: null, created: null,
    },
    issue,
  ...
  IssueEdit.contextTypes = {
    initialState: React.PropTypes.object,
  };
  ...

```

Testing this should show that server rendering works for both IssueList and IssueEdit; it automatically fetches the appropriate data at the server and creates the corresponding HTML.

But, if you start with one of the pages (that is, use browser refresh on that page) and then switch to the other, you'll see that things don't work that well. There are errors on the console; also, sometimes the page does not load properly when switched. I'll discuss these problems in the next section.

EXERCISE: ROUTED SERVER RENDERING

1. Is `componentDidMount()` required? Since the component is rendered on the server, why do we need to make an Ajax call once more to get the state? Hint: Comment out `componentDidMount()` handling and see what happens when switching between the Issue Edit and Issue List pages.

Answers are available at the end of the chapter.

Encapsulated Fetch

It should be obvious by now that the approach we took in the previous section has serious limitations. Here are a few things that we'll address in this section.

1. If we had API calls that don't match the URL, for example, if App had to fetch the currently logged-in user's name to show in the header, the initial state fetching will not work. We need a better way of finding out which API call to fetch data for the initial state, one that doesn't depend on the sameness of the API call and the requested URL.
2. Say we had more than one component that needs data for a single rendering. Using the same user name example above, the App component's data would need to be fetched in *addition* to either IssueEdit or IssueList data.
3. For every new component added, we need to change the server rendering entry point. This is not good, and makes the code maintenance a hassle. The knowledge of which data to fetch should be part of the component itself.
4. A component should not try to use some other component's initial data. That's what happened in the previous section, causing errors on the console when switching between the Issue Edit and Issue List pages.

To deal with the above, we need to encapsulate the fetch within the components that can have initial data. We also need a contract between all server-rendered components and the server rendering entry point that lets each component specify an identifier that identifies *its* initial data so that it knows that the data in the initial state is its own. Further, we'll let the components implement a data fetcher that can return the identified data, given the route's properties.

To facilitate this process, React-Router's `match()` provides us as part of `renderProps`, a list of components that are required to render the route. Note that this is a list of component classes, not instances, since the rendering has not yet begun; we've just matched a route. We could traverse this list of components that `match()` provides, and call a static function in each of them that returns you something that can fetch the data. We'll also make it so that the same function can be called from within component itself, to fetch the data after being mounted in the browser. Here's the function signature:

```
dataFetcher({ params, location, urlBase }) => Promise
```

From the server rendering entry point, we pass in the `params`, `location`, and the base URL of the API endpoint, and the component is expected to return a `Promise`, which will resolve to an object of the form:

```
{ Identifier: data }
```

The Identifier for each component needs to be unique. We can use a convention that the class name itself is the identifier. Further, if the component does not define the static data fetcher function, it means that there is no initial data requirement for that component.

Let's first make the changes to `renderedPageRouter` to consume the new contract. These changes are shown in Listing 11-28.

Listing 11-28. `renderedPageRouter.jsx`: Changes for Encapsulating Data Fetch

```
...
    } else if (renderProps) {
      fetch(`http://localhost:3000/api${req.url}`).then(response => ←
        (response.json()))
      .then(data => {
        const initialState = { data };
        const componentsWithData = renderProps.components.filter(c => ←
          c.dataFetcher);
        const dataFetchers = componentsWithData.map(c => c.dataFetcher({
          params: renderProps.params, location: renderProps.location,
          urlBase: 'http://localhost:3000',
        }));
        Promise.all(dataFetchers).then((dataList) => {
          let initialState = {};
          dataList.forEach((namedData) => {
            initialState = Object.assign(initialState, namedData);
          });
          const html = renderToString(
            <ContextWrapper initialState={initialState} >
...

```

We first constructed a list of components that have a data fetcher from the list of components provided in `renderProps`. Then, we created a Promise for each of these components by calling the data fetcher in that component. `Promise.all()` is used to resolve multiple promises and get the resolved result from all of them in an array. Thus, when all the fetches complete, `dataList` will have one element for each of the components. We need to create an initial state of the form `key (component identifier) - value (initial data for that component)` from this array, and that's what the `dataList.forEach()` does.

Next, we need changes in `IssueList` and `IssueEdit` that define these data fetchers, and also use the same function to fetch the data while loading the component dynamically. The data fetchers return a Promise that resolves to a *named* data element, so the name has to be unwrapped for the data to be used within the component. This is done both in the constructor (setting initial state) and the `loadData()` method, where the same data fetcher is called. These changes for `IssueList` and `IssueEdit` are shown in Listings 11-29 and 11-30, respectively.

Listing 11-29. IssueList.jsx: Changes for Using Named Data

```

...
export default class IssueList extends React.Component {
  static dataFetcher({ urlBase, location }) {
    return fetch(`${urlBase || ''}/api/issues${location.search}`).
      then(response => {
        if (!response.ok) return response.json().then(error => Promise.
          reject(error));
        return response.json().then(data => ({ IssueList: data }));
      });
  }

  constructor(props, context) {
    super(props, context);
    const issues = context.initialState.data.records;
    const issues = context.initialState.IssueList ?
      context.initialState.IssueList.records : [];
  }

  ...
  loadData() {
    fetch(`/api/issues${this.props.location.search}`).then(response => {
      if (response.ok) {
        response.json().then(data => {
          data.records.forEach(issue => {
            issue.created = new Date(issue.created);
            if (issue.completionDate) {
              issue.completionDate = new Date(issue.completionDate);
            }
          });
          this.setState({ issues: data.records });
        });
      } else {
        response.json().then(error => {
          this.showError(`Failed to fetch issues ${error.message}`);
        });
      }
    })
    IssueList.dataFetcher({ location: this.props.location })
    .then(data => {
      const issues = data.IssueList.records;
      issues.forEach(issue => {
        issue.created = new Date(issue.created);
        if (issue.completionDate) {
          issue.completionDate = new Date(issue.completionDate);
        }
      });
      this.setState({ issues });
    }).catch(err => {
  ...

```

Listing 11-30. IssueEdit.jsx: Changes for Using Named Data

```

...
export default class IssueEdit extends React.Component {
  static dataFetcher({ params, urlBase }) {
    return fetch(`${urlBase || ''}/api/issues/${params.id}`).then(
      (response => {
        if (!response.ok) return response.json().then(error => Promise.
          reject(error));
        return response.json().then(data => ({ IssueEdit: data }));
      })
    );
  }

  constructor(props, context) {
    super(props, context);
    const issue = context.initialState.data;
    issue.created = new Date(issue.created);
    issue.completionDate = issue.completionDate != null ?
      new Date(issue.completionDate) : null;
    let issue;
    if (context.initialState.IssueEdit) {
      issue = context.initialState.IssueEdit;
      issue.created = new Date(issue.created);
      issue.completionDate = issue.completionDate != null ?
        new Date(issue.completionDate) : null;
    } else {
      issue = {
        _id: '', title: '', status: '', owner: '', effort: null,
        completionDate: null, created: null,
      };
    }
  }

  ...
  loadData() {
    fetch(`/api/issues/${this.props.params.id}`).then(response => {
      if (response.ok) {
        response.json().then(issue => {
          issue.created = new Date(issue.created);
          issue.completionDate = issue.completionDate != null ?
            new Date(issue.completionDate) : null;
          this.setState({ issue });
        });
      } else {
        response.json().then(error => {
          this.showError(`Failed to fetch issue: ${error.message}`);
        });
      }
    });
  }
}

```

```

IssueEdit.dataFetcher({ params: this.props.params })
.then(data => {
  const issue = data.IssueEdit;
  issue.created = new Date(issue.created);
  issue.completionDate = issue.completionDate !== null ?
    new Date(issue.completionDate) : null;
  this.setState({ issue });
}).catch(err => {
...

```

In both cases, in the constructor, we have a check for whether the initial data contains data with the key as the name of the class; only then do we use it to initialize the state. Otherwise, we fall back to the original behavior of using a default data object, which is what will be used in the initial render of the object when newly mounted on the client.

Since the same function is used both at the server and the client, we had to add a check for `urlBase`, which is not supplied if it is a client-side function call. Also, we needed to prefix the class name to call static class functions, which is why you see `loadData()` having `IssueEdit.dataFetcher()` and `IssueList.dataFetcher()`.

And that's it. Finally, we have the application working seamlessly with multiple components loading their own data both on the server and the client. Testing this should show that switching between components works, as also do direct calls (using Refresh in the browser) to each page. Do test the server rendering with a filter applied in the Issue List page as well.

Summary

This chapter may have been a little heavy since we used complex constructs and patterns to implement server rendering. Hopefully, going back to the Hello World example eased out the complexity and helped you understand the fundamental concepts that allow us to do server rendering.

It must also be evident by now that React itself, not being a framework, does not dictate each of the additional parts that complete the application. React-Router helped us quite a bit with all that, and so did other packages like *isomorphic-fetch*. We also used *our* own pattern of data fetchers to tie the data requirement of each component when being rendered on the server.

In the next chapter, we won't focus on one single feature or concept; instead we'll learn about common features that most applications will need, which also lets us exercise the salient features of the technology choices that we've made.

Answers to Exercises

Back-End HMR

1. If we use `module.hot.accept()` without any parameters or callbacks, we'll see an error that the port 3000 is already in use. This is because any change will reload `index.js` itself, which will cause a starting of *another* HTTP server at the same port, 3000, and it will fail because we already have a server running on that port.

Giving `module.hot.accept()` a set of dependencies has the effect of saying "This is where the buck stops. I know how to handle changes in my dependents." Otherwise, it has the effect of saying, "I don't care; replace me."

Routed Server Rendering

1. If we remove the loading of data on `componentDidMount()`, we'll find that when switching between the two pages, the server does not render the page that is switched to. Here, the only way to get to the state data is via an Ajax call. An optimization opportunity exists: we should be able to detect that the component data exists and is available from the server and consume it. But once consumed, we should delete this data so that when the component is mounted again (purely in the UI), it is fetched instead of being used again. For the moment, the extra code does not seem worth it, so we'll leave it as is.



Advanced Features

In this chapter, we'll take a look at features common to many applications. These features cut across the different technologies (front end, back end, database) that the MERN stack consists of, and requires us to put together changes in all of them to make these features work.

MongoDB Aggregate

We had left a placeholder for reports in the Navigation bar in an earlier chapter. Let's now implement this page, where we'll show a table with a count of issues assigned to various owners categorized by status. We'll use a format that is popularly known as a cross-tab or a pivot table: a two-dimensional array or table with one axis labeled with the statuses and the other axis with owners.

Since the number of owners can be many, and there are a limited number of statuses, we'll line up the statuses on the horizontal axis (table row header), and use one row per owner to show the count of issues assigned to that owner.

The first decision we need to make is the design of the REST API. Since the report itself is not a resource, it doesn't fit into the usual CRUD pattern or the methods that we use for a typical resource. Some people tend to use a prefix to the resource endpoint, with `/stats/<resource>` as the common pattern. Others use a suffix instead, but this creates confusion with the resource ID and you must call the suffix a reserved word. I've also seen suggestions to make a completely new endpoint like `/issue_stats`, but then this (and the prefix strategy) can have only the GET method; the other methods don't make sense. It doesn't feel RESTful enough, since the resource is either not that concrete or some of the verbs become invalid.

What we'll do instead is to treat the report API as a *view* of the GET method itself on the Issues collection. The default view is the list of issues, and by supplying a parameter, we indicate that we need a summary of the issues rather than a full listing. This also falls in line with the commands used to do the same with a SQL database: you use SELECT for both getting a list or for getting a summary using GROUP BY. (I'll discuss the MongoDB way of doing this soon.) We can even use the filter to apply to the summary view.

Since all regular query parameters to the HTTP endpoint `/issues` indicate filters, we must use a special convention to indicate reserved words such as showing a summary (and later, to indicate the sort order, etc.). We'll use an underscore prefix to distinguish all these non-filter parameters. It doesn't make an immediate difference, but it avoids

conflicts in future if we were to add a filter on a field called `summary`. So, if we find a request parameter `_summary` in the List API, we'll return a summary instead of a listing.

To be able to test this, you first need lots of issues populated in the database. A simple MongoDB shell script to generate a set of issues randomly distributed across dates, owners, and statuses is shown in Listing 12-1.

Listing 12-1. `generate_data.mongo.js`: Mongo Shell Script to Generate Some Data

```
/* eslint-disable */
var db = new Mongo().getDB('issuetracker');

var owners = ['Ravan', 'Eddie', 'Pieta', 'Parvati', 'Victor', 'Violet'];
var statuses = ['New', 'Open', 'Assigned', 'Fixed', 'Verified', 'Closed'];

var i;
for (i=0; i<1000; i++) {
  var randomCreatedDate = new Date(
    (new Date()) - Math.floor(Math.random() * 60) * 1000*60*60*24);
  var randomCompletionDate = new Date(
    (new Date()) - Math.floor(Math.random() * 60) * 1000*60*60*24);
  var randomOwner = owners[Math.floor(Math.random() * 6)];
  var randomStatus = statuses[Math.floor(Math.random() * 6)];
  var randomEffort = Math.ceil(Math.random() * 20);
  var issue = {
    created: randomCreatedDate, completionDate: randomCompletionDate,
    owner: randomOwner, status: randomStatus, effort: randomEffort,
  };
  issue.title = 'Lorem ipsum dolor sit amet, ' + i;
  db.issues.insert(issue);
}
```

MongoDB provides the collection method `aggregate()` to summarize and perform various other read tasks on the collection using a *pipeline*, that is, a series of transforms on the collection before returning the result set. In fact, the default call to `aggregate()` is identical to a call to `find()`, that is, it returns the entire list of documents in the collection, without any manipulation (which reinforces our decision to use a variation of the GET method to fetch a summary).

The MongoDB aggregation pipeline consists of [stages](#). Each stage transforms the documents as they pass through the pipeline. For example, a `match` stage will act like a filter on the list of documents from the previous stage. To simulate a `find()` with a filter, you use a single `match` stage in the pipeline. To transform the document, you use a `project` stage, which, unlike the projection in `find()`, can even add *new* calculated fields to the document using expressions. Each stage does not have to produce a one-to-one mapping of the previous stage. The `group` stage is one such stage that produces a summary. The `unwind` stage is something that does the opposite: it expands array fields into one record for each array element. The same stage can appear multiple times; for example, you could start with a `match`, then `group`, and then another `match` to filter out some documents after the grouping.

To implement the Reports page, all we need is a match stage (based on the filter) and a group stage to summarize the matching documents. I'll only discuss the group stage in depth because it's the most commonly used. You can refer to the MongoDB Aggregation documentation for the other useful aggregation pipeline stages such as `unwind`.

The `aggregate()` method takes a single parameter, an array of pipeline stages. Each pipeline stage is an object with a single key indicating the type of the stage and the value holding the parameters for the stage. A match stage takes in the filter, as you would specify in a `find()` method as the value. Thus, the following command (issued in the MongoDB shell) will return all open issues:

```
> db.issues.aggregate([ { $match: { status: 'Open' } } ])
```

The group stage is a little more involved. It consists of a set of fields that need to be created, and a specification of how they need to be created. In the object specification, the key is the name of the field, and the value specifies how the field's value is to be constructed. The values are typically based on existing fields in the document (actually the results of the previous stage), and to refer to these fields, one needs to use a `$` prefix; otherwise it will be taken as a literal. The field `_id` is mandatory and has a special meaning: it is the value that the results are grouped by. Typically, you use an existing field specification for this. For the rest of the fields in the output, you can specify an aggregation function to construct their values.

For example, if you need the sum and average effort of all issues grouped by the owner, you use the following `aggregate()` command (which has a single stage in the pipeline):

```
> db.issues.aggregate([
  { $group: {
    _id: '$owner',
    total_effort: { $sum: '$effort' },
    average_effort: { $avg: '$effort' },
  } }
])
```

The above command will produce an output like this:

```
{ "_id" : "Parvati", "total_effort" : 1643, "average_effort" :
9.838323353293413 }
{ "_id" : "Violet", "total_effort" : 1694, "average_effort" :
10.204819277108435 }
{ "_id" : "Victor", "total_effort" : 1827, "average_effort" :
9.770053475935828 }
{ "_id" : "Pieta", "total_effort" : 1581, "average_effort" :
9.75925925925926 }
{ "_id" : "Eddie", "total_effort" : 1730, "average_effort" :
10.548780487804878 }
{ "_id" : "Ravan", "total_effort" : 1644, "average_effort" :
10.538461538461538 }
```

If the entire collection is to be grouped into a single value, you can use a literal value for the `_id`, typically `null`. Also, there is no special count aggregate function; you just need to sum the number 1 to get a count of all matched documents. Thus, another way to count the number of documents in a collection using the `aggregate()` method is

```
> db.issues.aggregate([ { $group: { _id: null, count: { $sum: 1 } } } ]])
```

This will result in the following output:

```
{ "_id" : null, "count" : 1002 }
```

For the reports page, we need two grouped by fields, the owner and the status. Multiple group fields are specified as an object instead of a string. So, to get a count of issues grouped by owner and status, this is the command to use:

```
> db.issues.aggregate([ { $group: {
  _id: { owner: '$owner', status: '$status' },
  count: { $sum: 1 },
} } ]])
```

This produces output like this:

```
{ "_id" : { "owner" : "Ravan", "status" : "Assigned" }, "count" : 23 }
{ "_id" : { "owner" : "Parvati", "status" : "Fixed" }, "count" : 30 }
{ "_id" : { "owner" : "Violet", "status" : "Assigned" }, "count" : 29 }
{ "_id" : { "owner" : "Ravan", "status" : "Closed" }, "count" : 21 }
...
```

Let's now use this query to generate the summary report in the List API. We need to check for the request parameter `_summary`; if it is undefined, we'll just return the list of issues as we did before. If the request is indeed for a summary, we'll use the filter to match the documents and the `aggregate()` method to get the results. Further, we'll transform the results to a friendlier output that can be consumed by callers more easily. We'll send back an object with one key for each owner, and the value for that owner holding the total count under each status. For example,

```
{
  "Ravan": {
    "Assigned": 23,
    "Closed": 21,
    "Fixed": 26,
    ...
  },
  ...
}
```

```

"Parvati": {
  "Fixed": 30,
  "Closed": 21,
  ...
}
...
}

```

This way, the caller can use `stats[owner][status]` to get to the count for an owner-status combination. The changes to GET API to achieve this are listed in Listing 12-2.

Listing 12-2. `server.js`: Changes to List API for Summary Output

```

...
app.get('/api/issues', (req, res) => {
  ...
  if (req.query._summary === undefined) {
    let limit = req.query.limit ? parseInt(req.query._limit, 10) : 20;
    if (limit > 50) limit = 50;
    db.collection('issues').find(filter).limit(limit)
      .toArray()
      .then(issues => {
        const metadata = { total_count: issues.length };
        res.json({ _metadata: metadata, records: issues });
      })
      .catch(error => {
        console.log(error);
        res.status(500).json({ message: `Internal Server Error: ${error}` });
      });
  } else {
    db.collection('issues').aggregate([
      { $match: filter },
      { $group: { _id: { owner: '$owner', status: '$status' }, count: { ←
        $sum: 1 } } } },
    ]).toArray()
      .then(results => {
        const stats = {};
        results.forEach(result => {
          if (!stats[result._id.owner]) stats[result._id.owner] = {};
          stats[result._id.owner][result._id.status] = result.count;
        });
        res.json(stats);
      })
      .catch(error => {
        console.log(error);
        res.status(500).json({ message: `Internal Server Error: ${error}` });
      });
  }
});

```

We added a `limit()` to the existing List API because we now have a lot of records, and it slows down the rendering if we show all of them in the Issue List page. Also, now that we're using a lot of variables with a dangling underscore, let's disable this rule in `.eslintrc`. This change is shown in Listing 12-3.

Listing 12-3. `.eslintrc`: Disable `no-underscore-dangle`

```
{
  "extends": "airbnb",
  "rules": {
    "no-underscore-dangle": ["error", { "allow": ["_id"] }],
    "no-underscore-dangle": "off",
    "no-param-reassign": ["error", { "props": false }]
  }
}
```

To test this API, you can use `curl` to fetch the URL `http://localhost:3000/api/issues?_summary`. To make use of this new API, let's create a new `IssueReport` component that displays the report. The code is similar to the other two components: we fetch the data using initial state or using an Ajax call. The rendering is also simple since we made the API output easy to consume. At the outer level, we need to iterate through all the keys returned by the API to generate one table row per owner. At the inner level, we need to iterate through all valid statuses and get its count from the API output and then display it in a table cell. The code for the new component is shown in Listing 12-4.

Listing 12-4. `IssueReport.jsx`: New Component to Show the Report

```
import React from 'react';
import { Panel, Table } from 'react-bootstrap';

import IssueFilter from './IssueFilter.jsx';
import Toast from './Toast.jsx';

const statuses = ['New', 'Open', 'Assigned', 'Fixed', 'Verified', 'Closed'];

const StatRow = (props) => (
  <tr>
    <td>{props.owner}</td>
    {statuses.map((status, index) => (<td key={index}>{props.
      counts[status]}</td>))}
  </tr>
);

StatRow.propTypes = {
  owner: React.PropTypes.string.isRequired,
  counts: React.PropTypes.object.isRequired,
};
```

```

export default class IssueReport extends React.Component {
  static dataFetcher({ urlBase, location }) {
    const search = location.search ? `${location.search}&_summary` : ←
    '?_summary';
    return fetch(`${urlBase || ''}/api/issues${search}`).then(response => {
      if (!response.ok) return response.json().then(error => Promise. ←
        reject(error));
      return response.json().then(data => ({ IssueReport: data }));
    });
  }

  constructor(props, context) {
    super(props, context);
    const stats = context.initialState.IssueReport ? context.initialState. ←
    IssueReport : {};
    this.state = {
      stats,
      toastVisible: false, toastMessage: '', toastType: 'success',
    };
    this.setFilter = this.setFilter.bind(this);
    this.showError = this.showError.bind(this);
    this.dismissToast = this.dismissToast.bind(this);
  }

  componentDidMount() {
    this.loadData();
  }

  componentDidUpdate(prevProps) {
    const oldQuery = prevProps.location.query;
    const newQuery = this.props.location.query;
    if (oldQuery.status === newQuery.status
      && oldQuery.effort_gte === newQuery.effort_gte
      && oldQuery.effort_lte === newQuery.effort_lte) {
      return;
    }
    this.loadData();
  }

  setFilter(query) {
    this.props.router.push({ pathname: this.props.location.pathname, query });
  }

  showError(message) {
    this.setState({ toastVisible: true, toastMessage: message, ←
      toastType: 'danger' });
  }
}

```

```

dismissToast() {
  this.setState({ toastVisible: false });
}

loadData() {
  IssueReport.dataFetcher({ location: this.props.location })
    .then(data => {
      this.setState({ stats: data.IssueReport });
    }).catch(err => {
      this.showError(`Error in fetching data from server: ${err}`);
    });
}

render() {
  return (
    <div>
      <Panel collapsible header="Filter">
        <IssueFilter setFilter={this.setFilter} ↵
          initFilter={this.props.location.query} />
      </Panel>
      <Table bordered condensed hover responsive>
        <thead>
          <tr>
            <th></th>
            {statuses.map((status, index) =><td key={index}><{status}</td>)}
          </tr>
        </thead>
        <tbody>
          {Object.keys(this.state.stats).map((owner, index) =>
            <StatRow key={index} owner={owner} ↵
              counts={this.state.stats[owner]} />
          )}
        </tbody>
      </Table>
      <Toast
        showing={this.state.toastVisible} message={this.state.toastMessage}
        onDismiss={this.dismissToast} bsStyle={this.state.toastType}
      />
    </div>
  );
}
}

IssueReport.propTypes = {
  location: React.PropTypes.object.isRequired,
  router: React.PropTypes.object,
};

```

```
IssueReport.contextTypes = {
  initialState: React.PropTypes.object,
};
```

Note that we used the `IssueFilter` as is; this lets us reuse the component's display (HTML) as well as its behavior (JavaScript). In many other frameworks, you may have to wire the two separately. For the status columns, we used a constant array of statuses to simplify the code and also to show the statuses in a logical order. Finally, to include the component in the UI, a small change to `Routes.jsx` is required to connect the component to the route `/reports`. This is shown in Listing 12-5.

Listing 12-5. `Routes.jsx`: Include Issue Report Component

```
...
import IssueReport from './IssueReport.jsx';
...
<Route path="/" component={App} >
...
<Route path="issues/:id" component={IssueEdit} />
<Route path="reports" component={withRouter(IssueReport)} />
...
```

The result of all the above changes can now be seen in the application UI, shown in Figure 12-1.

	New	Open	Assigned	Fixed	Verified	Closed
Ravan	27	32	23	26	27	21
Parvati	31	35	22	30	25	24
Violet	30	39	29	27	19	22
Eddie	23	30	26	28	27	30
Pieta	30	19	25	25	32	31
Victor	39	25	27	37	32	27

Full source code available at this [GitHub repository](#).

Figure 12-1. Reports page

Do test the new page by changing the filter as well as refreshing the page to let the server render it, with and without filters.

Pagination

Now that we have many documents in the database, we need a way to see all of them, yet not overwhelm the browser by sending them all in one shot. It's common to add pagination or an infinite scroll (more data is loaded when the user scrolls down) to address this requirement.

An infinite scroll appears slick at first glance, but it has its issues. Most infinite scroll implementations do not unload the scrolled-out data, so the DOM can get really big if the user keeps scrolling, affecting performance. The browser's back button, too, needs to be handled specially since the additional set of loaded items will no longer be loaded if the user clicked on an issue's link and wanted to come back to the list. Finally, if you want your pages to be indexed, you need to insert special invisible code that allows search engine bots to index all pages, because apart from the initially shown results, the rest of the pages won't have natural hyperlinks. This is not to say that infinite scrolls are bad, but that you need to evaluate if it's worth the effort required to implement it correctly.

We'll use pagination buttons, since they are simpler and let us quickly explore the concepts required to get them working end to end. We can use React-Bootstrap's `Pagination` component; it's a convenient component that seems to give us just what you want.

The List API needs to change to add pagination support. First, it needs to give us data for a given page, that is, starting from a certain index into the list. So, just like the `_limit` parameter in the API's query, we'll introduce an `_offset` parameter that determines where in the list of documents you need to start. The MongoDB cursor method `skip()` can be used to get the list of documents starting at an offset. As compared to a parameter that specifies which *page* to start on, we'll choose the granular offset because it reduces coupling between the API and the UI. With a `page` parameter, both the UI and the API need to implicitly agree upon a page size (or pass an additional parameter for that), whereas with an offset, the UI is free to determine the page size. We'll of course default the page size and ensure it is not too large in the back end to protect from unintended queries for large amounts of data or even hackers who can take advantage of this.

Whenever we use an offset into a list, we also need to ensure that the list is in the same order when queried multiple times. Without an explicit sort order, MongoDB does not guarantee this. The order of the documents *may* vary between two queries (although it appears as that it is always the order of insertion). To guarantee a certain order, we need to include a sort specification. Since the Object ID is a natural key to sort on (since it matches the insertion order), and it is an indexed field (that is, there is no penalty for requesting the list to be sorted in this order), we'll use it as the default sort key.

Since the `Pagination` component needs the total number of pages, we need to get the total count of documents that matches the filter. Here, the cursor method `count()` comes in handy. It returns the total count of documents that were matched, as if the `limit()` was not applied. We can use the same cursor to get the actual list of matching documents too.

The changes to the List API in `server.js` are shown in Listing 12-6.

Listing 12-6. server.js: Changes to List API to Support Pagination

```

app.get('/api/issues', (req, res) => {
  ...
  if (req.query._summary === undefined) {
    const offset = req.query._offset ? parseInt(req.query._offset, 10) : 0;
    let limit = req.query._limit ? parseInt(req.query._limit, 10) : 20;
    if (limit > 50) limit = 50;
    db.collection('issues').find(filter).limit(limit)
      .toArray()

    const cursor = db.collection('issues').find(filter).sort({ _id: 1 })
    .skip(offset)
    .limit(limit);

    let totalCount;
    cursor.count(false).then(result => {
      totalCount = result;
      return cursor.toArray();
    })
    .then(issues => {
      const metadata = { total_count: issues.length };
      res.json({ _metadata: metadata, records: issues });
      res.json({ metadata: { totalCount }, records: issues });
    })
  }
}

```

Instead of directly getting the array of documents using `toArray()`, we split the call into two stages. First, we got a cursor by filtering, sorting, skipping, and applying a limit. Then, we issued a `count()` request on the cursor. The argument `false` ensures that the effects of `skip()` and `limit()` will be ignored when returning the count, that is, the true count of documents with only the filter applied. After we got the count, we got the actual document list using `toArray()` on the same cursor. Finally, we made some cosmetic changes to the metadata that we return, which contains the total count of matched documents.

At this stage, you can test the new API by supplying an `_offset` parameter in a curl request to the endpoint, and optionally a `_limit` parameter.

Let's now use the new API in the UI to display the pagination strip. Of the many options that React-Bootstrap's Pagination component has, I like the Previous and Next buttons because they are the most used ones. Showing the boundary links (that is, the first and last page numbers) also helps since the last page number is an indication of how many documents or pages there are in the filtered results.

We also need to take a break from the one-to-one mapping of the front-end URL query parameters and the API call. This is because we need to translate a *page* in the UI to an *offset* in the API call. We'll do this translation in the `dataFetcher()` method. For now, we can hard code the page size using a constant. But it's easy to imagine (and implement) a way to let the user select the page size.

The changes are all in the `IssueList` component, shown in Listing 12-7.

Listing 12-7. `IssueList.jsx`: Changes for Pagination

```
...
import { Button, Glyphicon, Table, Panel, Pagination } from 'react-
bootstrap';
...
const PAGE_SIZE = 10;
...
static dataFetcher({ urlBase, location }) {
  return fetch(`${urlBase || ''}/api/issues${location.search}`). ←
  then(response => {
    const query = Object.assign({}, location.query);
    const pageStr = query._page;
    if (pageStr) {
      delete query._page;
      query._offset = (parseInt(pageStr, 10) - 1) * PAGE_SIZE;
    }
    query._limit = PAGE_SIZE;
    const search = Object.keys(query).map(k => `${k}=${query[k]}`).join('&');
    return fetch(`${urlBase || ''}/api/issues?${search}`).then(response => {
    ...
  }
}

...
constructor(props, context) {
  super(props, context);
  const issues = context.initialState.IssueList ? context.initialState. ←
  IssueList.records : [];
  const data = context.initialState.IssueList ? ←
  context.initialState.IssueList
  : { metadata: { totalCount: 0 }, records: [] };
  const issues = data.records;
...
  this.state = {
    ...
    totalCount: data.metadata.totalCount,
  };
...
  this.selectPage = this.selectPage.bind(this);
}
...
componentDidUpdate(prevProps) {
  ...
  && oldQuery._page !== newQuery._page) {
    return;
  }
  this.loadData();
...
}
```

```

selectPage(eventKey) {
  const query = Object.assign(this.props.location.query, {
    _page: eventKey });
  this.props.router.push({ pathname: this.props.location.pathname,
    query });
}
...
loadData() {
  ...
  .then(data => {
    ...
    this.setState({ issues, totalCount: data.IssueList.metadata.
      totalCount });
  })
  ...
  render() {
    ...
    <Panel collapsible header="Filter">
      <IssueFilter setFilter={this.setFilter}
        initFilter={this.props.location.query} />
    </Panel>
    <Pagination
      items={Math.ceil(this.state.totalCount / PAGE_SIZE)}
      activePage={parseInt(this.props.location.query._page || '1', 10)}
      onSelect={this.selectPage} maxButtons={7} next prev boundaryLinks
    />
    ...
  }
}

```

The key part of the change, apart from the `Pagination` component, is the `selectPage` handler, which uses the page number (event key) and pushes that as the `_page` parameter to the URL. This eventually comes back as a change in props, which we dealt with in `componentDidUpdate` to reload the data. To load the data, we used an API call that converts the `_page` parameter to an offset based on the page size, and also explicitly set the page size in the `_limit` parameter to the API.

The results of these changes are shown in the screenshot in Figure 12-2, the pagination bar.

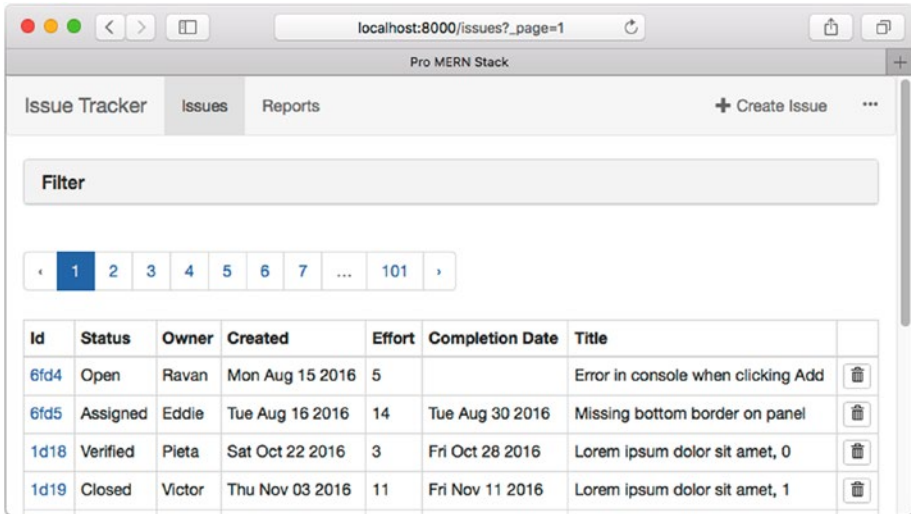


Figure 12-2. *Pagination*

The approach of using the same cursor to fetch the count was OK for small data sets, but you can't use it for larger data sets. The problem with React-Bootstrap's Pagination is that it needs the total count of pages or the total count of documents in the filtered set. The fact is that in any database, counting the number of matches is expensive; the only way you can do it is to visit *every* document to check whether it matches the filter (unless, of course, you have indexes for every possible filter combination, which either means limiting the kind of filters you want to allow the user, or spending enormous storage capacity for indexing all combinations).

I find that, practically, it isn't of much use to show the exact number of pages or count of matched records, when the result is possibly very large. If it's indeed hundreds of pages long, it is highly unlikely that the user will want to go to exactly the 97th page, or even the last page. In such cases, it's advisable to just show the Previous and Next links and not query the total count in every request. React-Bootstrap's Pager component will work well for this approach.

You should also be aware that React-Bootstrap's Pagination component uses buttons and not links. This means that search engines will not be able to reach and index other pages. Once again, the Pager component works better if you really need indexing on all pages.

Higher Order Components

In the Issue Tracker application, you can see that a fair bit of code was repeated across the main views for showing and managing Toast messages. We'll try and reuse this code in this section. The authors of React recommend Composition over Inheritance for reusing code across components.

One way to do this is to use a wrapper component that does all the message management and passes down the methods for showing error and success messages to the children, like this:

```
class ToastWrapper extends React.Component {
  constructor() {
    this.state = { toastVisible: false, ... }
  }
  showError() {
    this.setState({ toastVisible: true, ... });
  }
  render() {
    return (
      <div>
        {this.props.children}
        <Toast showing={this.state.toastVisible} .... />
      </div>
    );
  }
}
```

This way, both the display and the state management are localized within the wrapper, which can be used to wrap any of the components before use, like this:

```
...
<ToastWrapper><IssueEdit /><ToastWrapper>
...
```

Although this addresses the display, you also need to let the wrapped component be able to show error messages. You need to pass the `showError()` method as props to the wrapped component. React provides access to the children and also an iterator (since `this.props.children` itself is an opaque object) to iterate through it. You can use a clone of the wrapped components in the `render()` method and add additional properties for injecting the new methods like this:

```
render() {
  const childrenWithMessages = React.Children.map(this.props.children, child =>
    React.cloneElement(child, { showSuccess: this.showSuccess, ↵
      showError: this.showError }));
  return (
    ...
    {childrenWithMessages}
  );
}
```

`React.Children.map` takes in the children as a parameter and calls back a function with each child as an argument. You can clone this child using `React.cloneElement` with additional properties for the new methods that the child will receive.

This approach works, but since `this.props.children` is not necessarily a single element, you must use an iterator. An alternative approach is to use the Higher Order Components (HOC) design pattern, which is essentially a function that takes in a component and returns a wrapped component with additional functionality, thus enhancing it. You saw a glimpse of this pattern when we used the function `withRouter` to inject the router property into components that needed it.

Let's create our own HOC called `withToast` along similar lines. This function too will take in a component and return an enhanced component. The `render()` method will use a `<div>` and place the original component as well as a `Toast` component within it. Also, like in the previous examples, it will have state variables and methods to manage the message visibility. The code for this new file is shown in Listing 12-8.

Listing 12-8. `withToast.jsx`: A Higher Order Component to Add Toast Functionality

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

export default function withToast(OriginalComponent) {
  return class WithToast extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        toastVisible: false, toastMessage: '', toastType: 'success',
      };
      this.showSuccess = this.showSuccess.bind(this);
      this.showError = this.showError.bind(this);
      this.dismissToast = this.dismissToast.bind(this);
    }

    showSuccess(message) {
      this.setState({ toastVisible: true, toastMessage: message,
        toastType: 'success' });
    }

    showError(message) {
      this.setState({ toastVisible: true, toastMessage: message,
        toastType: 'danger' });
    }

    dismissToast() {
      this.setState({ toastVisible: false });
    }
  }
}
```

```

render() {
  return (
    <div>
      <OriginalComponent
        showError={this.showError} showSuccess={this.showSuccess} ↵
        {...this.props}
      />
      <Toast
        showing={this.state.toastVisible} message={this.state. ↵
        toastMessage}
        onDismiss={this.dismissToast} bsStyle={this.state.toastType}
      />
    </div>
  );
}
};
}

```

Let's take a closer look at some of the code in the new component.

```

...
return class WithToast extends React.Component {
...

```

The new function just had one return statement, and it returned a new component, which wrapped the original component. We moved all of the Toast management state variables and functions into this component, and we also passed in the `showError` and `showSuccess` handlers to the original component:

```

...
    <OriginalComponent
      showError={this.showError} showSuccess={this.showSuccess} ↵
      {...this.props}
    />
...

```

Unlike the `cloneElement` and iteration over children, this is more explicit, and it also forces the fact that only one component is wrapped. Now, we're ready to remove all the Toast-related code from the components that contained the repeated code. But there's one change that an HOC causes: any static functions that you used in the original component are no longer available in the wrapped component. We must explicitly copy the static functions from the original to the wrapped.

Let's first do this change in the Issue Edit page. The changes for this are shown in Listing 12-9.

Listing 12-9. IssueEdit.jsx: Changes for Reusing Toast Management

```

...
import Toast from './Toast.jsx';
import withToast from './withToast.jsx';
...
export default class IssueEdit extends React.Component {
...
  this.state = {
    ...
    toastVisible: false, toastMessage: '', toastType: 'success',
  };
...
  this.showSuccess = this.showSuccess.bind(this);
  this.showError = this.showError.bind(this);
  this.dismissToast = this.dismissToast.bind(this);
...
    this.setState({ issue: updatedIssue });
    this.props.showSuccess('Updated issue successfully.');
...
    response.json().then(error => {
      this.props.showError(`Failed to update issue: ${error.message}`);
...
    }).catch(err => {
      this.props.showError(`Error in sending data to server: ⚠
      ${err.message}`);
...
    }).catch(err => {
      this.props.showError(`Error in sending data to server: ⚠
      ${err.message}`);
...
  showSuccess(message) {
    this.setState({ toastVisible: true, toastMessage: message, toastType: ⚠
    'success'}));
  }

  showError(message) {
    this.setState({ toastVisible: true, toastMessage: message, toastType: ⚠
    'danger'}));
  }

  dismissToast() {
    this.setState({ toastVisible: false });
  }
...

```



```

render() {
  ...
  <Toast
    showing={this.state.toastVisible} message={this.state.
    toastMessage}
    onDismiss={this.dismissToast} bsStyle={this.state.toastType}
  />
  ...
  IssueEdit.propTypes = {
    ...
    showSuccess: React.PropTypes.func.isRequired,
    showError: React.PropTypes.func.isRequired,
  };
  ...
  const IssueEditWithToast = withToast(IssueEdit);
  IssueEditWithToast.dataFetcher = IssueEdit.dataFetcher;

  export default IssueEditWithToast;
  ...

```

Most of the changes are obvious: we removed all the methods for showing and dismissing the Toast, and also the state variables and the Toast component itself. We changed the `this.showError` and `this.showSuccess` messages to use the functions passed in through props instead, and also added `propTypes` for these functions. Finally, we removed the export of the class itself; instead, we included an export of the class created using `withToast()`. But before exporting, we had to copy the static functions.

Changes to `IssueList` and `IssueReport` are very similar, except that these components do not have a `showSuccess()` method, so we don't have to remove them. But the new wrapper lets us use `showError` if and when required without changes in the wrapper. These changes are listed in Listing 12-10 and 12-11, respectively.

Listing 12-10. `IssueList.jsx`: Changes for Reusing Toast Management

```

...
import Toast from './Toast.jsx';
import withToast from './withToast.jsx';
...
export default class IssueList extends React.Component {
...
  this.state = {
    ...
    toastVisible: false, toastMessage: '', toastType: 'success',
  };
...
this.showError = this.showError.bind(this);
this.dismissToast = this.dismissToast.bind(this);
...

```

```

showError(message) {
  this.setState({ toastVisible: true, toastMessage: message, toastType: '
    'danger' ' });
}

dismissToast() {
  this.setState({ toastVisible: false });
}
...
}).catch(err => {
  this.props.showError(`Error in fetching data from server: ${err}`);
...
fetch(`/api/issues/${id}`, { method: 'DELETE' }).then(response => {
  if (!response.ok) this.props.showError('Failed to delete issue');
...
render() {
  ...
  <Toast
    showing={this.state.toastVisible} message={this.state. '
    toastMessage}
    onDismiss={this.dismissToast} bsStyle={this.state.toastType}
  />
  ...
  IssueList.propTypes = {
    ...
    showError: React.PropTypes.func.isRequired,
  };
  ...
  const IssueListWithToast = withToast(IssueList);
  IssueListWithToast.dataFetcher = IssueList.dataFetcher;

  export default IssueListWithToast;
  ...

```

Listing 12-11. IssueReport.jsx: Changes for Reusing Toast Management

```

...
import Toast from './Toast.jsx';
import withToast from './withToast.jsx';
...
export default class IssueReport extends React.Component {
...
  this.state = {
    ...
    toastVisible: false, toastMessage: '', toastType: 'success',
  };
}

```

```

...
    this.showError = this.showError.bind(this);
    this.dismissToast = this.dismissToast.bind(this);
...
    showError(message) {
      this.setState({ toastVisible: true, toastMessage: message, toastType: ⚡
        'danger' });
    }

    dismissToast() {
      this.setState({ toastVisible: false });
    }
...
  }).catch(err => {
    this.props.showError(`Error in fetching data from server: ${err}`);
...
render() {
  ...
  <Toast
    showing={this.state.toastVisible} message={this.state. ⚡
    toastMessage}
    onDismiss={this.dismissToast} bsStyle={this.state.toastType}
  />
...
IssueReport.propTypes = {
  ...
  showError: React.PropTypes.func.isRequired,
};
...
const IssueReportWithToast = withToast(IssueReport);
IssueReportWithToast.dataFetcher = IssueReport.dataFetcher;

export default IssueReportWithToast;
...

```

The changes to `IssueAddNavItem` are not the same as the above, because the introduction of a `<div>` interferes with the display of the navigation item, which is an inline element. So, we'll look for a more appropriate component up the hierarchy and pass down the `showError` method via props from that component. The `Header` component seems ideal for this. The changes for this component are shown in Listing 12-12.

Listing 12-12. `App.jsx`: Changes in Header for Including Toast

```

...
import withToast from './withToast.jsx';
...

```

```

const Header = (props) => (
  ...
  <IssueAddNavItem showError={props.showError} />
  ...
  Header.propTypes = {
    showError: React.PropTypes.func.isRequired,
  };

  const HeaderWithToast = withToast(Header);

const App = (props) => (
  <div>
    <Header />
    <HeaderWithToast />
  ...

```

Instead of using Header directly in the App, we used a wrapped component. But Header itself does not use the showError() method; all we did was pass it through via props to IssueAddNavItem. The changes in IssueAddNavItem are similar to that of IssueList, etc., except that we don't wrap this component. We only make use of the showError method passed in via props. The changes to this component are in Listing 12-13.

Listing 12-13. IssueAddNavItem: Changes for Reusing Toast Management

```

...
import Toast from './Toast.jsx';
...
...
  this.state = {
    ...
    toastVisible: false, toastMessage: '', toastType: 'success',
  };
...
  this.showError = this.showError.bind(this);
  this.dismissToast = this.dismissToast.bind(this);
...
  showError(message) {
    this.setState({ toastVisible: true, toastMessage: message, toastType: '
    danger' });
  }

  dismissToast() {
    this.setState({ toastVisible: false });
  }
...

```

```

...
    response.json().then(error => {
      this.props.showError(`Failed to add issue: ${error.message}`);
    }).catch(err => {
      this.props.showError(`Error in sending data to server: ${err.message}`);
    })
  }
  render() {
    ...
    <Toast
      showing={this.state.toastVisible} message={this.state.toastMessage}
      onDismiss={this.dismissToast} bsStyle={this.state.toastType}
    />
  }
}
IssueAddNavItem.propTypes = {
  ...
  showError: React.PropTypes.func.isRequired,
  ...
}

```

At this point, you should be able to test each of the error messages and success messages in all these components that have been modified.

Search Bar

A search bar in most applications lets you find documents by just typing in some words. We'll implement this not like a search-filter, but instead as an autocomplete that finds all issues matching the words typed, and lets the user pick one of them to directly view. We'll add this search in the Navigation bar since the user should be able to jump to a particular issue, no matter which page they are viewing.

Assuming that the number of issues is large, it wouldn't perform well if we were to apply a filter criterion, say, a regex on all the issues. That's because to apply the regex, MongoDB would have to scan all the documents and apply the regex to see if it matches the search term. MongoDB's text index, on the other hand, lets you quickly get to all the documents that contain a certain term. A text index gathers all the terms (words) in all the documents, and creates a lookup table that, given a term (word), returns all documents containing that term (word). You can create such an index using the following MongoDB shell command:

```
>db.issues.createIndex({ title: "text" })
```

This will collect all the terms in the field `title` and create a text index. Now the documents can be searched using the usual `find()` method, with a special filter like this:

```
>db.issues.find( { $text: { $search: "lorem" } } )
```

This will *quickly* give you the list of documents that had the term “lorem” in it. Note that this does not go through all the documents in the collection to try to match the contents. Instead, it looks up the index for the term “lorem,” which already has the IDs of all documents containing this term.

Let’s first change the initialization script (and optionally rerun it, as well as the generation script) to create this index. If you don’t want to go through the reinitialization, you could just execute the `createIndex()` command manually. The changes to the initialization script are shown in Listing 12-14.

Listing 12-14. `init.mongo.js`: Add `createIndex` for a Text Index on Title

```
...
db.issues.createIndex({ created: 1 });
db.issues.createIndex({ title: "text" });
...
```

The next step is to use this index in the List API to find documents containing a given term. All we need to do is add a filter criterion if we find a request parameter for search. This is shown in Listing 12-15.

Listing 12-15. `server.js`: List API Changes to Include a Text Search Filter

```
...
if (req.query.ffmpeg_gte) filter.ffmpeg.$gte = parseInt(req.query. ↵
    ffmpeg_gte, 10);
if (req.query.search) filter.$text = { $search: req.query.search };
...
```

Now you should be able to test the new filter using curl to see if you get document matches. You’ll probably notice that MongoDB searches only for *whole* terms. I’ll discuss the impact of this a little later.

Let’s now install a package that provides an auto-complete: as you type in a word, it lists all the matching issues in a drop-down. It lets you pick one so that you can view that document. I’ve chosen a package called `react-select` that we’ll install using

```
$ npm install react-select --save
```

This component also requires that we include some CSS styles, which are part of the distribution of the package. So, as we did for Bootstrap, we’ll soft-link (or copy, if you’re using Windows) the CSS from its distribution directory to the application’s static directory:

```
$ ln -s ../node_modules/react-select/dist static/react-select
```

We’ll also have to include the stylesheet in the HTML that is generated using the template. This change is shown in Listing 12-16.

Listing 12-16. template.js: Include react-select Stylesheet

```
...
<link rel="stylesheet" href="/bootstrap/css/bootstrap.min.css" >
<link rel="stylesheet" href="/react-select/react-select.css" >
...
```

Now that we have the infrastructure in place, we can add the auto-complete component to the header's Navigation bar. We'll separate out the Header component for convenience into a separate file because we now have a lot of code that deals with the auto complete. In the header, we'll need to divide the space into three columns since the react-select component happens to be a `<div>` that will occupy the entire width of its parent. We'll use Bootstrap's `Col` components to do the division of space.

In the middle column, we'll place the `Select` component imported from `react-select`. We'll use the `Async` variant of this component since that's the one that loads the list from the server asynchronously. The important properties that this component needs are the functions that load the options asynchronously and the function that is to be called when an item in the drop-down is selected. With these functions implemented, Listing 12-17 shows the code for the new Header component.

Listing 12-17. Header.jsx: New Component Extracted Out of App.jsx, to Include the Search Bar

```
import React from 'react';
import { Navbar, Nav, NavItem, NavDropdown, MenuItem, Glyphicon, Col } from 'react-bootstrap';
import { LinkContainer } from 'react-router-bootstrap';
import { withRouter } from 'react-router';
import Select from 'react-select';

import IssueAddNavItem from './IssueAddNavItem.jsx';
import withToast from './withToast.jsx';

const Header = (props) => {
  function searchIssues(input) {
    if (input.length < 2) return Promise.resolve({ options: [] });

    return fetch(`/api/issues?search=${input}`).then(response => {
      if (!response.ok) return response.json().then(error => Promise.reject(error));
      return response.json().then(data => {
        const options = data.records.map(issue => ({
          value: issue._id,
          label: `${issue._id.substr(-4)}: ${issue.title}`,
        }));
        return { options };
      });
    });
  }
}
```

```

    }).catch(error => {
      this.props.showError(`Error fetching data from server: ${error}`);
    });
  });
}

function filterOptions(options) {
  return options;
}

function selectIssue(item) {
  if (item) props.router.push(`/issues/${item.value}`);
}

return (
  <Navbar fluid>
    <Col sm={5}>
      <Navbar.Header>
        <Navbar.Brand>Issue Tracker</Navbar.Brand>
      </Navbar.Header>
      <Nav>
        <LinkContainer to="/issues">
          <NavItem>Issues</NavItem>
        </LinkContainer>
        <LinkContainer to="/reports">
          <NavItem>Reports</NavItem>
        </LinkContainer>
      </Nav>
    </Col>
    <Col sm={4}>
      <div style={{ paddingTop: 8 }}>
        <Select.Async
          instanceId="search" placeholder="Search ..." autoload={false} ←
          cache={false}
          loadOptions={searchIssues} filterOptions={filterOptions} ←
          onChange={selectIssue}
        />
      </div>
    </Col>
    <Col sm={3}>
      <Nav pullRight>
        <IssueAddNavItem showError={props.showError} />
        <NavDropdown
          id="user-dropdown" title={<Glyphicon glyph="option-horizontal" ←
          />} noCaret
        >

```



```

        <MenuItem>Logout</MenuItem>
      </NavDropdown>
    </Nav>
  </Col>
</Navbar>
);
};

Header.propTypes = {
  showError: React.PropTypes.func.isRequired,
  router: React.PropTypes.object,
};

export default withRouter(withToast(Header));

```

There were three methods that we supplied to `Select.Async` as properties. The first is `loadOptions`, which makes an Ajax call to search issues based on the input text. The return value is an object with a key option that holds an array of matching options, with a value and a label. The value is the key to each item that is matched, so we used the `_id` of the issue for this. As for the label, we used the title of the issue, and this is what will be displayed in the drop-down that is shown after the user types in a few characters.

We also have a handler for filtering the options:

```

...
function filterOptions(options) {
  return options;
}
...

```

Normally, the `Select` component looks at the list of options that we gave it, and *further* filters it based on what the user has typed. To override this and tell it that the list has already been filtered based on the input text, we had to return the fetched array, no matter what the input was. This ensures that the list fetched from the server is shown in its entirety, without another filter applied by the `Select` component.

The property `instanceId` is needed to work around a bug in this module, which causes server-rendering issues if not supplied. The `onChange` handler uses the selected item's ID and routes the router to display the page that shows the selected issue.

Finally, since the header is now in a separate file, we'll need to remove this from the `App` component and import it instead. The new file is shown in Listing 12-18.

Listing 12-18. `App.jsx`: Rewritten After Separating Out the Header

```

import 'babel-polyfill';
import React from 'react';

import Header from './Header.jsx';

```

```

const App = (props) => (
  <div>
    <Header />
    <div className="container-fluid">
      {props.children}
      <hr />
      <h5><small>
        Full source code available at this <a href="https://github.com/
        vasansr/pro-mern-stack">
        GitHub repository</a>.
      </small></h5>
    </div>
  </div>
);

App.propTypes = {
  children: React.PropTypes.object.isRequired,
};

export default App;

```

To test it, it's best to add new issues manually with different words. The search is case insensitive and also does word stemming: you can type *missed* and it will match *miss*, *missing*, etc. But you'll find that you need to type a whole word (or more) for the text index to match. For example, if you type *con*, the issue containing the word *console* will not be fetched. This is due to the text index in MongoDB. This is the trade-off between faster searches and flexibility. If you really want the user to be able to type just *con* and the search to return the issue, you must use a regex-based search, but this cannot use an index. It will have to scan through the entire collection to find all the issues that match the regex.

A screenshot of the application's Issue List page after including the search bar in the header is shown in Figure 12-3.

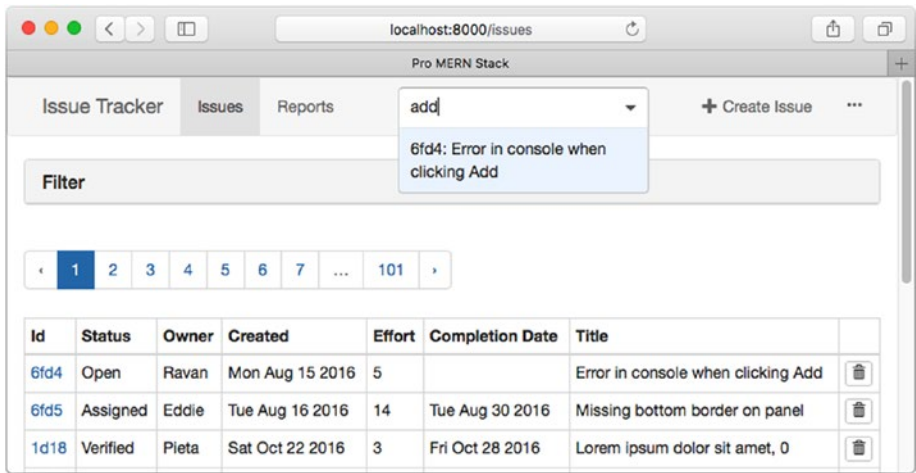


Figure 12-3. Search bar added to header

Google Sign-In

Most applications need to identify and authenticate users. Instead of having our own user creation and authentication mechanism, we'll integrate with one of the social sign-ins. We'll only implement one (Google Sign-In for Websites); this will serve as a good example for other integrations, since it uses the OAuth2 mechanism, which most other authentication integrations also use.

In this section, we'll only focus on the integration, and leave the actual access control and session tracking to the next section. Eventually, we'll make it so that users can view all information without signing in, but in order to make any change, they will have to sign in. We'll use a Modal dialog that lets the user sign in with Google from anywhere in the application. Once signed in, the application will let the user stay on the same page, so that they can perform the edit functions after having signed in.

The various options for integrating with Google Sign-In are listed in the "Guides" section at <https://developers.google.com/identity/sign-in/web/>. In the recommended method for integration listed there, the library itself renders the button and handles its enabled and signed-in states. Unfortunately, this doesn't work well with React because the Google library needs a handle to the button and needs it to be permanent. If you try to use the button within a React-Bootstrap modal, the library throws up errors since on closing the modal the button is destroyed and recreated when the modal is opened again. The library apparently doesn't like this. So, we must display the button ourselves by following the guide titled *Building a button with a custom graphic*.

As a preparatory measure, we need a Console Project and a Client ID to identify the application. Please follow the instructions in the guide to create your own project and client ID.

To use the client ID, we'll store it in a configuration file that will be required during runtime, but one that we cannot commit as part of the source code or include in any bundle during compilation. This is because we'd like the file to be changed

after deployment, and the contents can be different for different deployments (e.g., development, staging, production). Since this configuration will only be used in the client, we can simply create a JavaScript file in the `static` directory with the configuration parameters and include that in the `index.html` that is generated. The configuration file name is simply `config.js`. A sample configuration is available in the GitHub repository accompanying this book as `config.sample.js`, and the contents should be like in Listing 12-19, with the text `YOUR_CLIENT_ID` replaced by the client ID you generated in your project.

Listing 12-19. `config.js`: Runtime Configuration File with Client ID

```
window.config = {
  googleClientId: YOUR_CLIENT_ID.apps.googleusercontent.com',
};
```

We'll use this configuration parameter in a new component that encapsulates the entire sign-in and sign-out functionality. Just like we did for Create Issue, we'll encapsulate this inside a `NavItem`. When the user is signed in, we'll show a drop-down menu with the user's name and a menu item to sign out. When not signed in, we'll instead show a navigation item that lets the user sign in.

We'll initialize the authentication engine within `componentDidMount`, as it will be called only once, since the component is in the header and always visible. Upon clicking the Sign In navigation item, we'll show a modal dialog, with a single button for Google Sign-In. I used an image as per the branding guidelines found in the Google Sign-In developer guide within this button. For testing, you may use a plain button, but if you intend to put the application in production, you should follow the branding guidelines.

Upon clicking this button, we'll call the `auth2.signIn()` method. This method is not described in the guide, but you can find a description in the "Reference" section. On a successful sign in, we'll call a callback with the name of the user to the parent component, since this information will eventually have to flow down into all components that will need the user information. This component, too, will need the information on the signed-in state and the name of the user to determine the navigation item to display. The contents of the new component are shown in Listing 12-20.

Listing 12-20. `SignInNavItem.jsx`: New Component to Handle Google Sign-in

```
import React from 'react';
import { NavItem, Modal, Button, NavDropdown, MenuItem } from 'react-bootstrap';

export default class SignInNavItem extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      showing: false, disabled: true,
    };
    this.showModal = this.showModal.bind(this);
    this.hideModal = this.hideModal.bind(this);
  }
```

```

    this.signout = this.signout.bind(this);
    this.signin = this.signin.bind(this);
  }

  componentDidMount() {
    window.gapi.load('auth2', () => {
      if (!window.gapi.auth2.getAuthInstance()) {
        if (!window.config || !window.config.googleClientId) {
          this.props.showError('Missing Google Client ID or config file ↵
            /static/config.js');
        } else {
          window.gapi.auth2.init({ client_id: window.config.googleClientId ↵
            }).then(() => {
              this.setState({ disabled: false });
            });
        }
      }
    });
  }

  signin() {
    this.hideModal();
    const auth2 = window.gapi.auth2.getAuthInstance();
    auth2.signIn().then(googleUser => {
      fetch('/signin', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ id_token: googleUser.getAuthResponse(). ↵
          id_token }),
      }).then(response => {
        if (response.ok) {
          response.json().then(user => {
            this.props.onSignin(user.name);
          });
        } else {
          response.json().then(error => {
            this.props.showError(`App login failed: ${error}`);
          });
        }
      })
      .catch(err => {
        this.props.showError(`Error posting login to app: ${err}`);
      });
    }, error => {
      this.props.showError(`Error authenticating with Google: ${error}`);
    });
  }
}

```

```

signin() {
  const auth2 = window.gapi.auth2.getAuthInstance();
  fetch('/signin', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
  }).then(response => {
    if (response.ok) {
      auth2.signOut().then(() => {
        this.props.showSuccess('Successfully signed out.');
```

←

```
        this.props.onSignout();
      });
    }
  });
}

showModal() {
  if (this.state.disabled) {
    this.props.showError('Missing Google Client ID or config file /static/ ↵
    config.js');
```

←

```
  } else {
    this.setState({ showing: true });
  }
}

hideModal() {
  this.setState({ showing: false });
}

render() {
  if (this.props.user.signedIn) {
    return (
      <NavDropdown title={this.props.user.name} id="user-dropdown">
        <MenuItem onClick={this.signout}>Sign out</MenuItem>
      </NavDropdown>
    );
  }
  return (
    <NavItem onClick={this.showModal}>Sign in
      <Modal keyboard show={this.state.showing} onHide={this.hideModal} ↵
        bsSize="sm">
        <Modal.Header closeButton>
          <Modal.Title>Sign in</Modal.Title>
        </Modal.Header>
        <Modal.Body>
          <Button block disabled={this.state.disabled} ↵
            onClick={this.signin}>
            
          </Button>
        </Modal.Body>
      </Modal>
    </NavItem>
  );
}
```

```

        </Button>
      </Modal.Body>
    <Modal.Footer>
      <Button bsStyle="link" onClick={this.hideModal}>Cancel</Button>
    </Modal.Footer>
  </Modal>
</NavItem>
);
}
}
}

SigninNavItem.propTypes = {
  user: React.PropTypes.object,
  onSignin: React.PropTypes.func.isRequired,
  onSignout: React.PropTypes.func.isRequired,
  showError: React.PropTypes.func.isRequired,
  showSuccess: React.PropTypes.func.isRequired,
};

```

We expect the property `user` to be an object that has the keys `signedIn` and `name`. These keys let us choose between showing a drop-down or a navigation item in the `render()` method. As a precautionary measure, we also disabled the login button until the initialization has completed, and if it has failed, we show an error (it is very likely that on a fresh deployment, someone will miss the configuration file for the Google Client ID). Also, we're expecting a sign in and a sign out handler, as well as functions for showing success and error as part of the properties.

The `gapi` functions are available as part of Google's platform library, which we'll include as part of the `index.html` template. This change is shown in Listing 12-21.

Listing 12-21. `template.js`: Changes to Include Google API Platform Library, and Loading the Configuration

```

...
<script src="https://apis.google.com/js/api:client.js"></script>
...
<script src="/app.bundle.js"></script>
<script src="/config.js"></script>
...

```

To use the component, we'll include it in the header, in place of the placeholder that we had until now. Further, we won't let the header handle the sign-in or sign-out; instead, we'll pass on the information to *its* parent, `App`, so that the user information can be held at the topmost point possible. The changes to the `Header` component are shown in Listing 12-22.

Listing 12-22. Header.jsx: Changes for Including Sign-In Navigation Item

```

...
import { Navbar, Nav, NavItem, NavDropdown, MenuItem, Glyphicon, Col } from 'react-bootstrap';
...
import SigninNavItem from './SigninNavItem.jsx';
...
render()
  ...
    <NavDropdown id="user-dropdown" title={<Glyphicon glyph="option- ↵
horizontal" />}
      noCaret>
      <MenuItem>Logout</MenuItem>
    </NavDropdown>
    <SigninNavItem
      user={props.user} onSignin={props.onSignin} ↵
      onSignout={props.onSignout}
      showError={props.showError} showSuccess={props.showSuccess}
    />
  ...
Header.propTypes = {
  ...
  showSuccess: React.PropTypes.func.isRequired,
  onSignin: React.PropTypes.func.isRequired,
  onSignout: React.PropTypes.func.isRequired,
  user: React.PropTypes.object,
  ...

```

The App component will now have to be converted to a stateful component since it will hold the user information and the signed-in state. We'll also introduce methods to manipulate this state. The new rewritten App component is shown in Listing 12-23.

Listing 12-23. App.jsx: Rewritten as a Stateful Component to Hold User Information

```

import 'babel-polyfill';
import React from 'react';

import Header from './Header.jsx';

export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      user: { signedIn: false, name: '' },
    };
    this.onSignin = this.onSignin.bind(this);
    this.onSignout = this.onSignout.bind(this);
  }

```



```

onSignIn(name) {
  this.setState({ user: { signedIn: true, name } });
}

onSignout() {
  this.setState({ user: { signedIn: false, name: '' } });
}

render() {
  return (
    <div>
      <Header user={this.state.user} onSignIn={this.onSignIn} ↵
        onSignout={this.onSignout} />
      <div className="container-fluid">
        {this.props.children}
        <hr />
        <h5><small>
          Full source code available at this
          <a href="https://github.com/vasansr/pro-mern-stack">
            GitHub repository</a>.
        </small></h5>
      </div>
    </div>
  );
}
}

App.propTypes = {
  children: React.PropTypes.object.isRequired,
};

```

The `render()` method is almost the same, except for passing in the handlers as properties to `Header`. The effect of these changes can now be tested. Figure 12-4 shows the modal dialog. Clicking the button should open a new browser window with your accounts, and if it's the first time you are signing in, you will also be shown a message asking to grant permissions to the application. Upon successful sign-in, the menu item will be replaced with the first name of the user, which is obtained from the authentication.

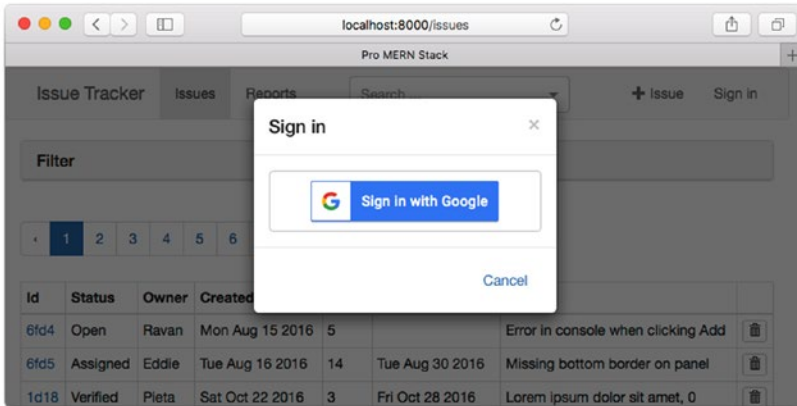


Figure 12-4. Google Sign-In modal dialog

If you want to support multiple social sign-ins such as Facebook and GitHub in addition to Google, you're probably better off using an npm package such as `passport`, which lets you quickly put together multiple sign-in options. But the `passport` integration is at the back end and works well if the result of the sign-in doesn't have to be the same page the user initiates it from. It takes the user through a series of redirects and ends up in a possibly new page as the result of a successful sign-in.

Session Handling

Just authenticating with Google is not enough; we'll need to do something with the authentication. First, we'll need to ensure that the credentials are validated at the back end, and also establish a session so that requests can be validated against credentials for all subsequent API calls. Session handling support is available for Express via the npm package `express-session`. Let's now install this:

```
$ npm install --save express-session
```

Using the package is quite simple. We install it as a middleware in the application using `app.use()`. A required option to create the middleware is the `secret`, which is used to encrypt cookies. We'll just use a random string for this purpose. The other options are based on recommendations in the `express-session` documentation.

We'll initialize a user session in a new route: `/signin`, which will take in a parameter in the request body, which is the Google Authentication token. This token will have to be revalidated using a call to a Google API as described in the sign-in developer guide. Once verified, we'll keep an object called `user` in the session to identify the user, and save just one variable: the user's name. In the future, we can save more information such as the

user's email ID, which is unique, say, if we add audit logs for saving who changed what in the issues. The counterpart route of `/signin` simply destroys the session. Let's also add a special API endpoint called `/api/users/me` that returns the currently signed in user details.

Although we'll implement checks in the UI to prevent edits unless the user is signed in, it is good practice to enforce this at the back end also, since the back end API can be accessed directly without the UI. A single route to `/app/all` does the job for us where we check if the methods are one of POST, PUT, or DELETE, and then return an error if we don't find a session. In reality, you may have finer-grained access based on roles such as *only administrators can create new users*, in which case you need to add such checks in each endpoint handler, and also save the user's roles and rights along with the session.

These changes are all in the server, as shown in Listing 12-24.

Listing 12-24. `server.js`: Changes for Session Handling

```
...
import session from 'express-session';
...
app.use(session({ secret: 'h7e3f5s6', resave: false, saveUninitialized: ←
true }));

app.all('/api/*', (req, res, next) => {
  if (req.method === 'DELETE' || req.method === 'POST' || req.method === ←
  'PUT') {
    if (!req.session || !req.session.user) {
      res.status(403).send({
        message: 'You are not authorized to perform the operation',
      });
    } else {
      next();
    }
  } else {
    next();
  }
});
...
app.get('/api/users/me', (req, res) => {
  if (req.session && req.session.user) {
    res.json(req.session.user);
  } else {
    res.json({ signedIn: false, name: '' });
  }
});

app.post('/signin', (req, res) => {
  if (!req.body.id_token) {
    res.status(400).send({ code: 400, message: 'Missing Token.' });
    return;
  }
});
```

```

fetch(`https://www.googleapis.com/oauth2/v3/tokeninfo?id_token=${req.body.id_token}`)
  .then(response => {
    if (!response.ok) response.json().then(error => Promise.reject(error));
    response.json().then(data => {
      req.session.user = {
        signedIn: true, name: data.given_name,
      };
      res.json(req.session.user);
    });
  })
  .catch(error => {
    console.log(error);
    res.status(500).json({ message: `Internal Server Error: ${error}` });
  });
});

app.post('/signout', (req, res) => {
  if (req.session) req.session.destroy();
  res.json({ status: 'ok' });
});
...

```

Now, from the sign-in component, at appropriate events, we'll need to send the sign-in and sign-out information to the server as well. For a sign-in, we'll retrieve the `id_token` using the `getAuthResponse()` method on the user object. With this token, we'll call the `/signin` API in the server, and on success, we'll propagate the user name to the parent component.

The sign out process is reversed: we first call `/signout` on the server, and on success, sign the user out with Google Authentication. The new `signin()` and `signout()` methods in the component are shown in Listing 12-25.

Listing 12-25. `SignInNavItem.jsx`: Send Sign In and Sign Out Events to the Server

```

...
signin() {
  this.hideModal();
  const auth2 = window.gapi.auth2.getAuthInstance();
  auth2.signIn().then(googleUser => {
    fetch('/signin', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ id_token: googleUser.getAuthResponse().id_token }),
    }).then(response => {
      if (response.ok) {
        response.json().then(user => {
          this.props.onSignIn(user.name);
        });
      }
    });
  });
}

```

```

    } else {
      response.json().then(error => {
        this.props.showError(`App login failed: ${error}`);
      });
    }
  })
  .catch(err => {
    this.props.showError(`Error posting login to app: ${err}`);
  });
}, error => {
  this.props.showError(`Error authenticating with Google: ${error}`);
});
}

signinout() {
  const auth2 = window.gapi.auth2.getAuthInstance();
  fetch('/signinout', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
  }).then(response => {
    if (response.ok) {
      auth2.signOut().then(() => {
        this.props.showSuccess('Successfully signed out. ');
        this.props.onSignout();
      });
    }
  });
}
...

```

At this stage, you'll find that the session information is persisted in the server, and also that all the edit operations return an error. Rather than let the user try this and face error messages, we'll just make these operations unavailable. We'll hide the Create Issue navigation item and the delete icons in the Issue List. In the Issue Edit page, we'll disable the Submit button. To be able to do all this, we'll assume that the user information is passed through to these components from the App component. The UI changes are shown in Listings 12-26, 12-27, and 12-28.

Listing 12-26. IssueEdit.jsx: Disable Submit Button if User Is Not Signed In

```

...
render()
...
    <Button bsStyle="primary" type="submit">Submit</Button>
    <Button bsStyle="primary" type="submit" disabled={!this. ←
      props.user.signedIn}>
      Submit
    </Button>

```

```
...
IssueEdit.propTypes = {
  ...
  user: React.PropTypes.object.isRequired,
};
...
```

Listing 12-27. Header.jsx: Hide Create Issue Button When User Is Not Signed In

```
...
{props.user.signedIn ? <IssueAddNavItem showError={props.showError} /> : null}
...
```

Listing 12-28. IssueList.jsx: Hide delete Icons When User Is Not Logged In

```
...
const IssueRow = (props) => {
  ...
  return (
    ...
    {props.deleteIssue ? (
      <td>
        <Button bsSize="xsmall" onClick={onDeleteClick}><Glyphicon ↩
          glyph="trash" /></Button>
        </td>
      ) : null
    )
  );
  ...
IssueRow.propTypes = {
  ...
  deleteIssue: React.PropTypes.func.isRequired,
};
...
function IssueTable(props) {
  ...
  return (
    ...
    {props.deleteIssue ? <th></th> : null}
  );
  ...
IssueTable.propTypes = {
  ...
  deleteIssue: React.PropTypes.func.isRequired,
};
...
class IssueList extends React.Component {
  ...
  render() {
    ...
    <IssueTable issues={this.state.issues} deleteIssue={this. ↩
      deleteIssue} />
```

```

    <IssueTable
      issues={this.state.issues}
      deleteIssue={this.props.user.signIn ? this.deleteIssue : null}
    />
...
IssueList.propTypes = {
  ...
  user: React.PropTypes.object.isRequired,
};
...

```

Since we already had `user` as a property in `Header`, we didn't have to add it as part of `propTypes`, but for the other two components, we added it. The user logged-in state in `IssueList` had to percolate down to the descendent components, and rather than pass the user information down, we made the function `deleteIssue` optional, indicating to the children that it is not supported when not passed.

To make the user information available in these components, we must stuff it in as props from `App`. But due to the usage of `React-Router`, we have direct access only to `Header`, not to the other children, which are dynamically mounted. We'll use the `React.cloneElement()` strategy discussed earlier to make the props available to all children without knowing which ones they are. Also, to make the user information available even when the user refreshes the browser (in this case, the `onSignIn` handler wouldn't have been called), we'll fetch the user information from the server using `/api/users/me` API as usual, in the `componentDidMount()` method in the `App` component.

With just these changes, you'll find that the application works, but for a signed-in user, there's a brief flash when the browser is refreshed: it first displays as if the user has not signed in, and when the Ajax call to `/api/users/me` completes, the UI is redrawn to reflect the fact that there is a signed-in user. We can fix this using the Server Rendering technique such that `App` also uses initial state from the context to initialize its state.

But that's not enough, because a call to `/api/users/me` from the server when constructing all the initial states will fail to fetch the currently logged-in user because it does not supply a cookie to identify the session. To fix this, we must pass any cookie from the UI *through* to all API calls when server-rendering a page. The first change for this is to extract the cookie from the request and pass it to all the data fetchers. This change is shown in Listing 12-29.

Listing 12-29. `renderedPageRouter.jsx`: Pass on Cookies to the API Calls

```

...
const dataFetchers = componentsWithData.map(c => c.dataFetcher({
  params: renderProps.params, location: renderProps.location,
  urlBase: 'http://localhost:3000', cookie: req.headers.cookie,
}));
...

```

Now, in the data fetcher of `App`, we'll use the cookie to add a header while making the fetch API call. Note that this needs to be done only when rendering on the server, so there's a check for existence of this parameter in the data fetcher call, which is not passed

in from `componentDidMount()` when rendering in the DOM. This change, and the other changes in App discussed above, are shown in Listing 12-30.

Listing 12-30. App.jsx: Changes for fetching user data with support for server rendering

```
export default class App extends React.Component {
  static dataFetcher({ urlBase, cookie }) {
    const headers = cookie ? { headers: { Cookie: cookie } } : null;
    return fetch(`${urlBase || ''}/api/users/me`, headers).then(response => {
      if (!response.ok) return response.json().then(error => Promise.reject(error));
      return response.json().then(data => ({ App: data }));
    });
  }

  constructor(props, context) {
    super(props, context);
    const user = context.initialState.App ? context.initialState.App : {};
    this.state = {
      user: { signedIn: false, name: '' },
    };
  }
  ...
  componentDidMount() {
    App.dataFetcher({ })
      .then(data => {
        const user = data.App;
        this.setState({ user });
      });
  }
  ...
  render() {
    const childrenWithUser = React.Children.map(this.props.children, child =>
      React.cloneElement(child, { user: this.state.user })
    );
    return (
    ...
      {this.props.children}
      {childrenWithUser}
    ...
    App.contextTypes = {
      initialState: React.PropTypes.object,
    };
    ...
  }
```


Note that we included the cookie in the data fetcher's `fetch()` call conditionally, as in these lines:

```
...
static dataFetcher({ urlBase, cookie }) {
  const headers = cookie ? { headers: { Cookie: cookie } } : null;
...

```

In a client-side call, the cookie would have been automatically included in the fetch call, hence we didn't have to explicitly pass the cookie as a parameter when calling the same method in `componentDidMount()`.

Also, we did not make the same change in other components' data fetchers because their results don't depend on the session. If they did, for example, if the list of issues depended on the signed-in user, say something like *My Issues*, then we would have to include the cookie in those data fetchers as well.

Summary

In this chapter, we explored a variety of techniques and concepts that we can use to implement features that make the application more usable.

We saw how to use MongoDB's aggregate framework to summarize or even expand the data that we can get from collections. We used a third-party React-specific library to search for issues based on a text index in MongoDB. Finally, we added sign-in and session capabilities to the application.

Let's wrap up by discussing what more can be done with the MERN stack in the next chapter. We'll stop adding features and code to the application and just look at other possibilities and libraries that could be useful, depending on how your specific application takes shape.

CHAPTER 13



Looking Ahead

I hope by now that I have succeeded in planting the fundamentals of the MERN stack in your mind. More importantly, I hope I have equipped you to take it to the next level, because this is by no means the end of the Issue Tracker application, or anything else you had in mind as a project. If you'd really tried answering the exercises in each chapter, you should now be familiar with where to get more information when needed.

There are many more features that can be added and more technologies that can be used to help you going forward. In this chapter, I'll touch upon some of the technologies that you may want to consider if you decide to use MERN in real-life projects. But this will only be a brief introduction to what's possible; we won't be adding any code to the application we've created until now. I'll also talk about the options you have and things you need to do to deploy. Finally, I'll introduce mern.io, which can be used to quickly start off a new project using a boilerplate.

But note that these new things may not necessarily suit your application. You need to evaluate them carefully if and when you hit a roadblock or want to automate some repetitive code as your application grows. The previous chapters in this book should have given you enough confidence that you can use the MERN stack and solve all problems *by hand*, or yourself. But in many a case, others have faced similar problems and created libraries to solve them. I suggest that you look for an existing solution, but wait until you are clear about what is it that you want solved.

Mongoose

As I'm sure you've noticed, there's a lot of repetitive code to convert dates to and from strings. We also had to add validation code to check the validity of an issue being added or updated. We did not add these validations on the client-side code, expecting the server to catch these errors. This is not a great user experience; one would expect the errors to be caught earlier and friendlier messages to be shown.

We encapsulated some of the conversions and validations in `issue.js` on the server, and we could easily share this with the client-side code, our own *model* of sorts. But we still need to repeat most of the code for all the CRUD operations on MongoDB across other models, if we had more than just the issues in the application.

An alternative to consider is the object-document mapping (ODM) package, `mongoose`. This is similar to object-relational mappers (ORMs) that you find for SQL

databases, except that there is in reality very little *mapping* that's happening. What mongoose provides instead is the following:

- *Schema definitions*: This is useful since MongoDB does not enforce a schema, unlike SQL databases. With SQL databases, a schema error will be automatically caught by the database, whereas we had to write our own schema validations. With mongoose, you can define schemas and have new documents automatically validated against it.
- *Data conversions*: Happily, mongoose automatically converts data such as a JSON date string into the native Date type.
- *Custom validators*: Note that mongoose built-in validators include type checking, required checking, minimum, maximum for integers, and lengths for strings. Over and above these validators, you can add custom validators such as email ID validators.
- *Isomorphic*: There's a browser component that lets the schema validations be used in the browser, although data conversions still need to be done manually.

For smaller projects such as the Issue Tracker, mongoose was probably not needed. You could easily extend and share `issue.js` to reuse code. But for larger projects with multiple people working in a team, using mongoose will definitely ease errors in the development process and also serve as documentation for the object schema, which is especially helpful for newcomers.

Flux

If you have read about React, it is very likely that you have also heard about Redux and/or the Flux pattern. Due to its popularity, it's very tempting to dive right in and start using it. But let's first look at opportunities for improvement before we find solutions for them.

When we added user sign-in, we found that we had to transfer user information and the sign-in action up the component hierarchy and back down to the components that had use for it. It seems a little wasteful for the components somewhere in between, that don't need to know the information. For example, the `IssueList` component by itself had little use for the user's name or signed-in state. All it did was pass the knowledge down to `IssueTable`.

Imagine a more deeply nested hierarchy, where only the leaf element needs this information. A great example a developer gave me was that in his application, the up-vote button needed to be disabled for comments or posts made by the currently signed-in user (for obvious reasons). And this button was a component, five levels deep in the hierarchy, and none of the intermediate components had any use for the signed-in user information.

If you run into many such cases where you find yourself having to traverse the component hierarchy needlessly, you really need to consider a *global* state, one that every component that is being rendered has access to and depends on for its display. Of course, all of these components will now be stateless components, pure functions, as I discussed

in Chapter 4. The state will be abstracted away into a place where every component has access to it, called a *store*. But anything global needs some restrictions or contracts to avoid havoc caused by uncontrolled changes to it. This is what the Flux architectural pattern tries to define.

Flux embraces a unidirectional data flow, so that all changes to the state are channeled through a dispatcher, which controls the order of changes and thus avoids infinite loops because of interdependencies. Although this pattern was invented by the same people who developed React (that is, Facebook), the pattern is not restricted to be used only in React. Here is a very succinct, yet complete, description of Flux that I'm quoting from Facebook's React blog (emphasis mine):

When a user interacts with a React view, the view sends an action (usually represented as a JavaScript object with some fields) through the dispatcher, which notifies the various stores that hold the application's data and business logic. When the stores change state, they notify the views that something has updated. This works especially well with React's declarative model, which allows the stores to send updates without specifying how to transition views between states.

Essentially, the pattern consists of formally defined *actions*, for example, Create Issue, initiated by the user. The action is *dispatched* to a *store*. This has the effect of affecting the store's state. Typically, a function called a *reducer* is used to describe the effect of an action on the state: the new state is a function of the current state and the action.

Redux and Mobx are two popular choices recommended by the authors of React that can be used for global state management, and they follow the concepts of the Flux pattern largely. The effect of these frameworks or the Flux pattern is that you will have to write a lot of *boilerplate* code, that is, code that looks very much like some other code, seemingly unnecessarily. For every user action, you must formally define an action, a dispatcher, and a reducer that, given the current state, returns the contents of a new state.

Let's take the action of deleting an issue as an example. You must define a set of formal actions, including a constant like `DELETE_ISSUE`. Then, you must define a reducer, a function that takes in various actions, their parameters, and returns a new state (a switch-case for each different action). Then, you must create a dispatcher, which converts the action and parameters into a real action, such as sending the request to the server.

All of this is worth it if the state machine in your application is very complicated, like say, if a delete request can be initiated from several different places in the UI (and also not the UI, as in some other user's action), and it has many other implications besides just removing a row in a table. Not many applications face this kind of complexity. I can assure you with reasonable confidence that there'll be a point in time when your application will grow big enough within a single page (imagine your Facebook page) and you'll *know* you need Redux or Mobx, and why. Until that point in time, it may be wiser to get things done using just the basics and fundamentals while continuing to learn about these awesome patterns.

Deployment

All this while, we've been using the development environment, although we did make distinctions between installing packages for development versus for production. Let's now look at the different things you may need to do when deploying the application for others to consume.

Firstly, you need to extricate the hard-coding of various parameters into configuration files, so that they can be changed depending on the deployment. You really want to keep the deployment flexible; for example, the MongoDB server may not be on the same machine as the Node.js web server, or the port, or the user and password to access the database. We started by using a configuration file for the Google client ID. It's a good idea to separate all other parameters that could be different when the application is really deployed. Things you may like to put in the configuration are the following:

- *MongoDB database connection parameters:* The server name, IP address, port, user, password.
- *The port on which the web server listens:* This could be port 80 if Express is handling requests directly, or some other port if there is a reverse proxy or a load balancer before the HTTP requests hit the actual server.
- *Location of static content:* JavaScript bundles, images, and possibly CSS files. They may reside on a CDN, so the prefix of where to get these from should be soft-coded in `template.js` and configured.
- If the traffic is to be encrypted using SSL, and if so, the location of the certificates and keys.

If you decide to not use a proxy and make the Express server directly face the Internet traffic, the first roadblock you will hit when deploying the application is that you will need root access on the server for Node.js to run. One option is to install Node.js and the application under `/root` and run it under root privileges. But this is dangerous because it exposes any vulnerabilities in the application or the underlying platform to hackers. It's always safer to use a reverse proxy such as apache or nginx to terminate the requests coming in via the Internet, and run the actual Node.js process as a non-root user. This way, even if a hacker gains access to the server via unknown security vulnerabilities in your code or Node.js, the maximum they can get is a regular user access. This is not so damaging as access as a root user, who gains complete control over the server.

You also want to think about keeping the different layers separate and possibly running on different machines. Running MongoDB on a separate server is very likely to happen, especially if you want to handle high availability and add clustering capabilities so that MongoDB is not running on a single server but multiple servers. Separating the configuration of where the database resides is almost a must. It's also a good idea to make the API server configurable as well, just in case you want the server rendering to be handed by a separate server. Thus, instead of hard-coding the URL Base in `renderedPageRouter.jsx`, you should move it to a configuration file.

Finally, you also need to think about how you copy a tested, certified version of the application (or even parts of it) to a production server. Lots of people directly pull

the code from their source control system on to the production servers, but I don't recommend this. Instead, I think you should have a small script that compiles all the required bundles, archives them (tar or zip) along with other static files that are required, and create a distribution that can be deployed. On the target deployment server, all you need to do is un-archive the distribution and install any new packages needed (just run `npm install`).

In the process, it's also advisable to mark the distribution with a version number that is traceable back to the exact replica of the source code from which the distribution was created. There are multiple ways to do this depending on the source control system, but most of them support a tagging system that can be used for this purpose. This lets you look at bug reports in production and relate them to the exact source code that caused the bug.

mern.io

Although you've learned how to build a MERN stack application from scratch, there are times when you don't want to go through the same steps for starting a new application. A lot of code that we wrote would have to be repeated. For example, most of the code for server rendering or all of the webpack configurations, and even `.eslintrc`, would have to be written again and again for each project that you want to initiate.

If you want to get a good head start on most of the popular practices followed in a MERN stack application, it's handy to start with a project that's already crafted with all of the boilerplate code, as well as an example set of objects that you can tweak or copy to get your job done quickly.

Enter `mern.io` (<http://mern.io>). This project includes a nifty utility that you can use to create the skeleton of a MERN application. The package itself is called `mern-cli`, a command-line utility that creates a MERN app. The commands to start using this are as follows:

```
$ npm install -g mern-cli
$ mern init my-new-app
```

You will find that you have an entire functional application under the directory `my-new-app`. To quickly see if it really works, all you need to do is navigate to the directory, install all the required packages (using `npm install`), and then run

```
$ npm start
```

If you point your browser to `http://localhost:8000`, you will see a functional application that can create and delete blog posts. The differences between what we've developed as part of this book and the application created by `mern-cli` are the following:

- `mern-cli` does not use Bootstrap or React-Bootstrap. It has its own style sheets for styling its content.
- The `mern-cli` application uses Redux and mongoose, two technologies that we discussed but did not use in the Issue Tracker application.

- There is no user sign-in code in the `mern-cli` application.
- The code is organized as *modules*, cohesive pieces of code that work together to expose a significant functionality. The only module in the `mern-cli` application is the Post module.
- The `mern-cli` application supports internationalization.
- The server side code is also modularized: the `mern-cli` application has separate folders for models, controllers, and routes.
- It uses CSS loaders to pack styles into the main bundle as opposed to dealing with them as separate files.

The downside is that the framework is opinionated: it has already picked technologies such as mongoose for the ODM and AVA for testing, choices that you may not be comfortable with. You may also not be in sync with the organization of the folders.

But the good part about using `mern-cli` is that all of the tooling is taken care of. You can focus on your application's components instead of building the infrastructure needed to run the application. The choices already made are probably good for you.

That's All, Folks!

I hope it has been a fun sail through the waters of the MERN stack for you, as it has been for me. I have learned a lot by just contemplating the programming model, the paradigm the new thinking that this new stack has opened my eyes to.

But I'm sure this is nowhere close to the end. In a few months, I am reasonably certain that things won't be the same. Who knows? Browsers may themselves adapt or incorporate the virtual DOM technique, making React mostly redundant! Or, you'll see a new framework (not a library) that anchors React as the View part in an MVC offering. The key thing is to keep looking for these new developments, yet be very, very analytical about why they do or do not work for *your* application and your team.

Here's to looking ahead to the *more* awesome future.

Index

■ A

- Airbnb rule, 145
- app.bundle.js, 118–120, 126, 128–129, 131, 148
- App.jsx, 108, 119, 124–125, 131, 161, 164, 168, 208–209, 211–212, 241
- App.jsx Rewritten with Router, 153
- Application programming interfaces (APIs), 69
 - Create API, 80–81, 90
 - error handling, 85–88, 91
 - using, 84–85, 91
 - List, 76, 79, 89
 - automatic server restart, 77
 - testing, 77–79
 - using, 82–84, 90

■ B

- babel-preset-es2015-node4, 136
- Bootstrap typography elements, 211
- Bundle, 120, 122–124

■ C

- Client-side libraries, 125
- CommonsChunkPlugin, 127
- Componentcomposition. *See* Composing component
- componentDidMount(), 58
 - lifecycle methods, 182
- componentDidUpdate() lifecycle methods, 178, 182, 186, 205
- Composing component, 40–41
- Content delivery network (CDN), 20

- Convenience, editing, 143
- Cross-tab, 275
- CRUD mapping, 70
- Custom validators, 320

■ D

- Data conversions, 320
- Debugging, 134
- Deployment, 322–323
- devServer, 129–130
- Document-oriented databases, 10–11
- Dynamic composition
 - CSS Styles, 50
 - in-memory array, 47
 - IssueList class, 48
 - IssueRow class, 49
 - IssueTable class, 48–49

■ E

- ECMAScript 2015 (ES2015)
 - App.jsx, 32
 - arrow function, 32
 - index.html, 31
 - package.json, 31
 - template string, 33
- Editing convenience, 143
- Error handling, 85–88, 91
- ESLint, 140–142
 - .eslintrc, 140–142, 145–147
- Express, 24, 26, 72
 - handler function, 73
 - request objects, 73–74
 - response objects, 74–75

Express (*cont.*)

- MERN, 9–10
- middleware, 75–76
- routing, 72
 - request matching, 72
 - route lookup, 73
 - route parameters, 73

■ F

- Facebook, 6
- File start_hook.js, 138
- Flux pattern, 320–321
- FormGroup component, 219, 243
- Forms
 - actual DOM, 180
 - back-end APIs, 173
 - buttons, 175
 - onChange events, 176
 - componentWillReceive
 - Props, 204
 - controlled components, 175
 - date input, 190, 192–193, 195
 - delete API, 200–203
 - edit page, 182, 185, 205
 - get API, 180–181
 - hard-coded filter, 179
 - HTML component, 174
 - initFilter, 177
 - input's type, 204
 - IssueAdd component, 174
 - IssueFilter.jsx, 176–178
 - issue.js, 196
 - IssueList.jsx, 179
 - in List API, 173–174, 204
 - number input, 187–190
 - PATCH method, 195
 - PUT method, 196
 - render() method, 175
 - server.js, 196
 - UI components, 186
 - Update API, 199–200
 - updateOne MongoDB
 - function, 196
 - virtual DOM, 180

■ G

- gapi functions, 307
- Glyphicon, 209
- Google Sign-In, 303–310

■ H

- Hello World component, 250
- Higher Order Components (HOC),
 - 288–291, 293–295, 297
- historyApiFallback, 167
- Horizontal form, 224
- Hot module replacement (HMR)
 - back-end, 256–260
 - middleware, 132–134

■ I

- index.html, 120, 126, 128, 207–208, 215,
 - 217, 230, 233
- Initial Bootstrap-styled Issue
 - List page, 210
- init.mongo.js, 100
- Isomorphic, 320
- isomorphic-fetch, 262
- IssueAdd class, 119
- IssueAdd component, 175
- IssueAdd.jsx, 119, 121–123, 125, 140, 222
- IssueAddNavItem.jsx, 238
- IssueEdit.jsx, 153, 155, 182, 190, 194,
 - 199, 229, 233
- IssueFilter component, 216
- IssueFilter.jsx, 122, 125, 158, 162, 219
- issue.js, 116, 136
- IssueList.jsx, 122–123, 125, 131, 145, 148,
 - 156, 158–159, 161, 165, 202, 209,
 - 216, 235, 241
- Issue Tracker application, 288–291,
 - 293–295, 297

■ J, K

- Jade, 246
- JavaScript, 11–13
- JavaScript modules, 208
- JavaScript Object Notation (JSON), 14
- JSX
 - automate, 29
 - react library, 30
 - separate script file, 27
 - transformation, 28, 120

■ L

- LinkContainer, 211–212, 216, 226, 243
- Loaders, 118, 120–121, 127

■ M

Material UI, 207, 232
 mern.io, 323–324
 MERN stack, 1–2
 components, 5
 Express, 9–10
 libraries, 13
 MongoDB, 10–11
 Node.js, 8–9
 React, 6–7
 React-Bootstrap, 12
 React-Router, 12
 tools and libraries, 12
 webpack tool, 12
 purpose, 13
 isomorphic, 14
 JavaScript, 13
 JSON, 14
 Node.js performance, 14
 not a framework, 15
 npm Ecosystem, 14
 Middleware function, 75–76
 Model-view-controller (MVC), 1
 module.exports, 115, 117, 121, 136
 MongoDB
 aggregate, 275–283
 App.jsx, 108
 async module, 105–106
 callbacks, 103
 collections, 94
 documents, 93
 generator and co module, 104–105
 insert method, 109–111
 in server.js, 107
 installation, 95
 limit(), 113
 MERN, 10
 document-oriented, 10–11
 JavaScript, 11
 NoSQL, 10
 schema-less, 11
 mongo shell, 95–98, 112
 Node.js Driver, 101–103
 promises, 104
 query language, 94
 schema initialization, 99–100, 112
 shell scripting, 99
 TCP connection, 113
 Mongoose, 319–320

■ N

NavDropdown, 211, 213
 Node.js, 21
 MERN, 7
 event-driven, 9
 modules, 8
 and npm, 8
 performance, 14
 NoSQL databases, 10
 npm, 22, 24
 nvm, 21

■ O

onBlur() handler, 187
 onChange event, 175–176, 180, 182, 184,
 186–189, 191–193, 204

■ P, Q

package.json, 117, 122, 131, 137, 139
 Pagination, 284–288
 PATCH method, 195–196
 Placeholder page, 155–156
 Project directory, 22
 PUT method, 195–196, 199

■ R

React, 55
 async state initialization, 58–59
 child to parent, communicating from,
 60–63, 68
 classes, 38–39
 composing component, 40–41
 designing components, 65
 communication, 66
 component hierarchy, 65
 stateless components, 66
 state *vs.* props, 65
 event handling, 60
 MERN, 5
 component-based, 6–7
 declarative, 6
 Facebook, 6
 isomorphic, 7
 no templates, 7
 setting state, 55–58, 67
 stateless components, 63–65

React-Bootstrap, 12
 CSS-in-JS and inline-style approach, 207
 grid-based Forms, 218–221, 243
 horizontal forms, 224–225, 227–228
 inline forms, 222–223, 243
 installation, 207, 209–210
 Material-UI, 243
 Modals, 237, 240, 242, 244
 navigation, 210, 213–215, 243
 results, 231–237
 table and panel, 216–218
 UI library, 207
 validations, 229–231

React components
 classes, 38–41
 Issue Tracker, 37
 passing data
 property, 42–44
 property validation, 44–45
 props.children, 45–46

React-specific issues, 145

render() method, 309

Representational state transfer (REST), 69
 APIs, 69
 HTTP methods as actions, 70–71
 JSON, 71
 resource based, 69

Routed server rendering, 260–261, 263–267

RouterContext, 261

Routing techniques
 App.jsx Rewritten with Router, 153
 browser history, 152, 167–168
 controlled forms, 169
 hash-based, 152
 IssueEdit.jsx, 153
 nested routes, 164–166
 parameters, 154–157, 169
 programmatic navigation, 161–163, 170–171
 properties, 151
 query string, 157, 159–160, 169
 React Router, 152
 React Router works, 153
 SPAs, 151
 webpack.config.js, 153

■ S

Schema definitions, 320
 Search bar, 297–299, 301–302

Server, 248
 server.js, 107–108, 110, 117, 133, 135–136, 145, 157, 168, 174, 181, 200

Server-less Hello World, 17–20

Server rendering, 245
 back-end HMR, 256–257, 259
 basic, 245–250
 encapsulated fetch, 268–272
 handling state, 250–252
 initial state, 252–253
 routed, 260–261, 263–267
 server-side bundle, 254–255

Server setup
 Express, 24–26
 Node.js, 21
 npm, 22, 24
 nvm, 21
 project directory, 22

Server-side bundle, 254–255

Server-side ES2015, 135–139

Server-side modules, 115–117

Session handling, 310–317

setDb(), 258

Single page applications (SPAs), 1, 151–152, 167

Structural issues, 143

Syntax consistency, 142

■ T, U

template.js, 246

Third-party libraries, 125

Toast.jsx, 231, 235

Transform, 120, 122–124

trymongo.js, 102–106

■ V

Validation function, 87

Vendor bundle, 211

Virtual DOM technique, 324

■ W, X, Y, Z

Webpack, 117–120
 webpack.config.js, 120–121, 127, 129–130, 132, 134, 140, 147, 153, 168, 211
 webpack-dev-server, 129
 Webpack tool, 12