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.

V. Subramanian, *Pro MERN Stack*, DOI 10.1007/978-1-4842-2653-7_1

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 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)

V. Subramanian, *Pro MERN Stack*, DOI 10.1007/978-1-4842-2653-7_3

- 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 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.

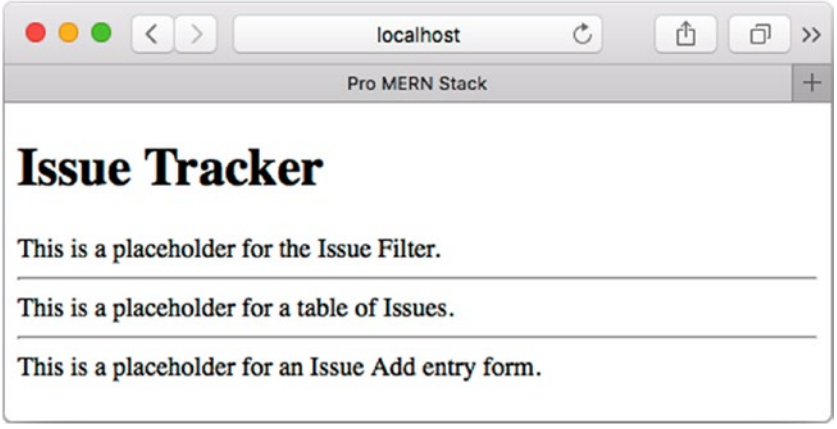

Figure 3-1. *Issue tracker by composing components*

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 containes 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.

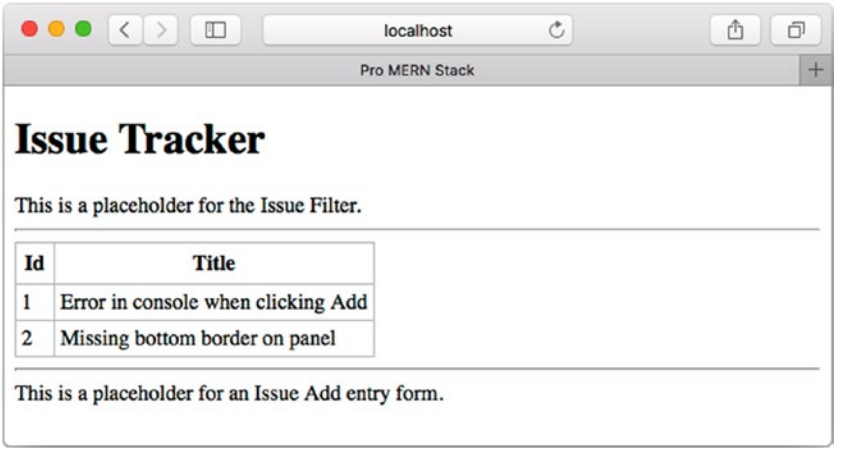

Figure 3-2. *Passing data to child components*

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 . 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 />
    </BordeWrap>
...
```

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_title~~children}</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.toDateString()}</td>
        <td>{issue.effort}</td>
        <td>{issue.completionDate ? ↵
        issue.completionDate.toDateString() : ''}</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.toDateString() : ''}</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 `toDateString()` 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.

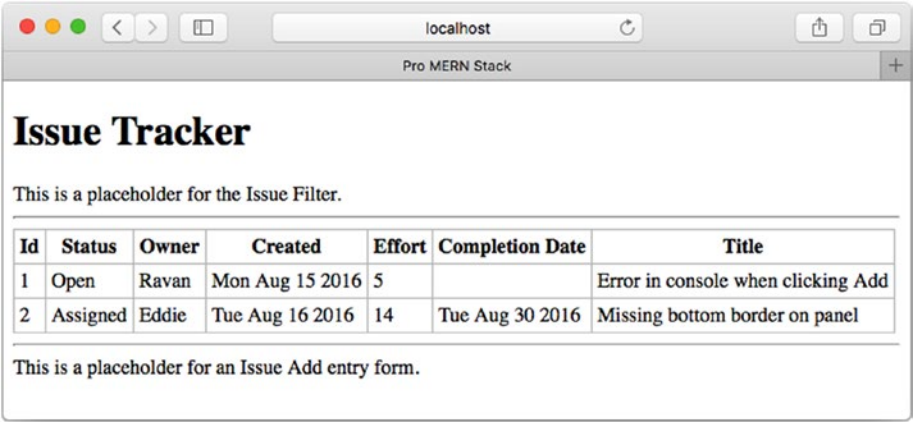

Figure 3-3. *Rows constructed programmatically from an array*

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 rerender 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 };
```


V. Subramanian, *Pro MERN Stack*, DOI 10.1007/978-1-4842-2653-7_4

```
    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 to 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.

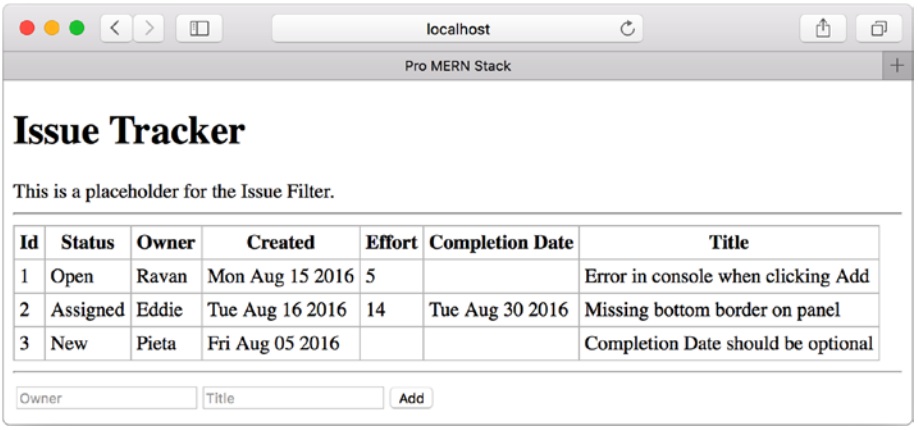

Figure 4-1. *Issue Tracker with user interactivity*

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`.

[1]This 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.toDateString()}</td>
    <td>{props.issue.effort}</td>
    <td>{props.issue.completionDate ? ➥
    props.issue.completionDate.toDateString() : ''}</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 8

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.

V. Subramanian, *Pro MERN Stack*, DOI 10.1007/978-1-4842-2653-7_8

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>
...
```

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.

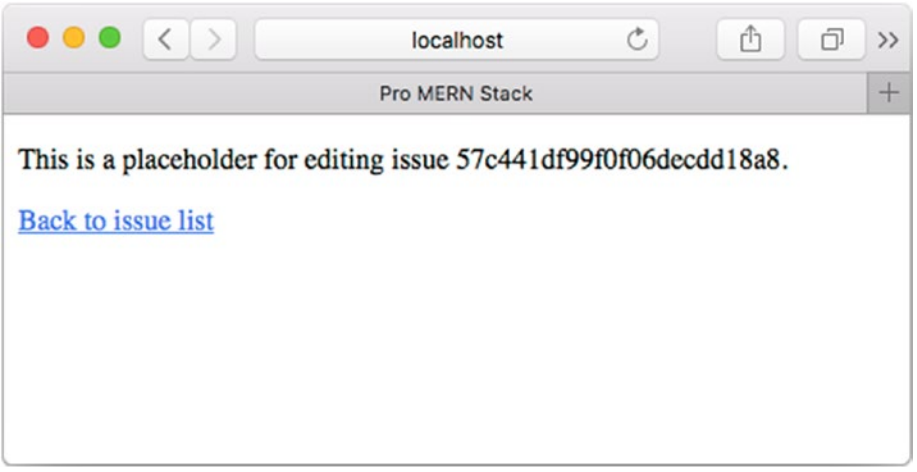

Figure 8-1. *Placeholder page*

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.

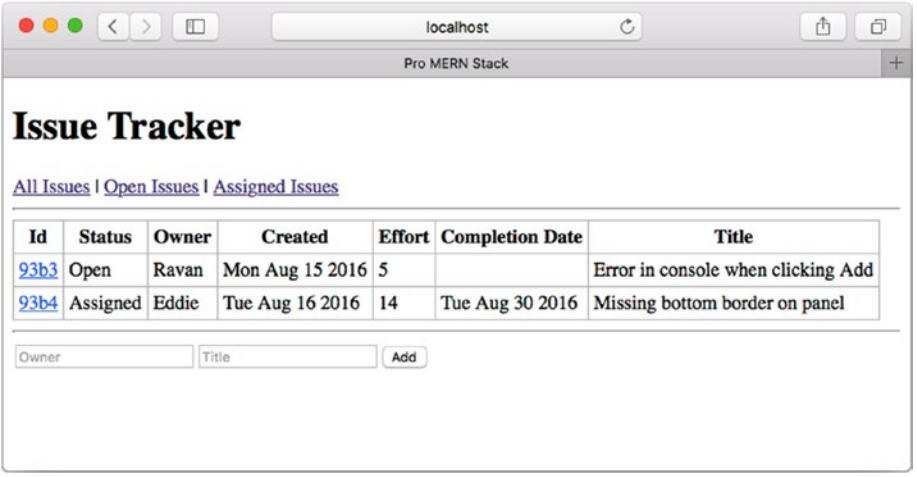

Figure 8-2. *Issue list with hardcoded filters as hyperlinks*

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, hashHistory, 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} />
...
```

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.

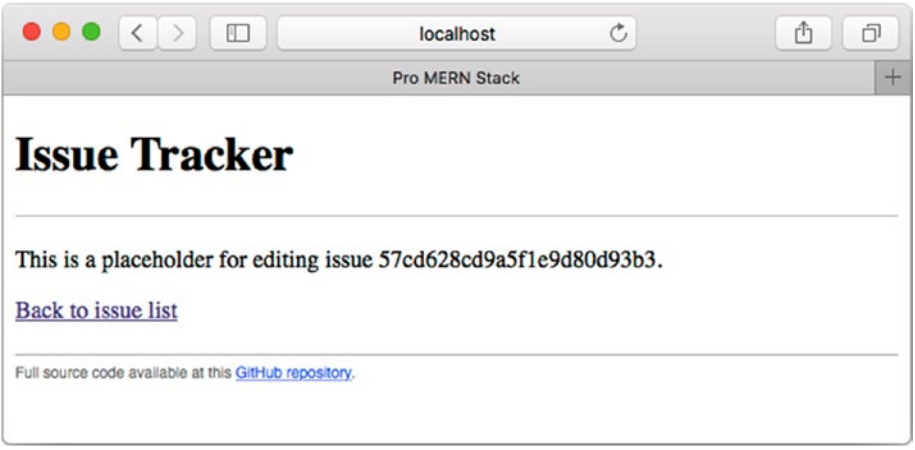

Figure 8-3. *Edit page, decorated*

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.