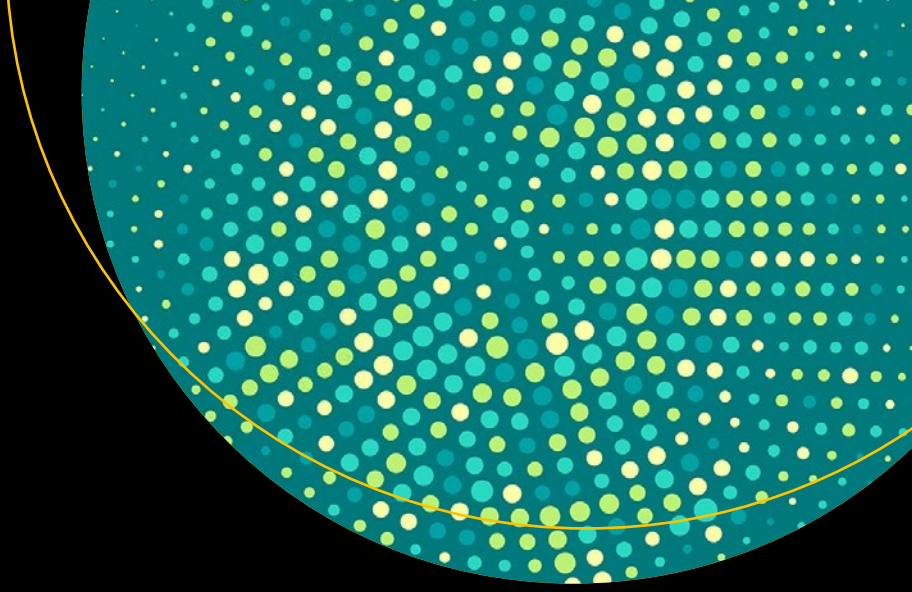

Pro MERN Stack

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

Vasan Subramanian

Apress®

Pro MERN Stack

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

Vasan Subramanian

Apress®

Pro MERN Stack

Vasan Subramanian
Bangalore, Karnataka, India

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

Library of Congress Control Number: 2017933833

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

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

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

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

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

Printed on acid-free paper

To Sandeep and Fazle.

Contents at a Glance

Contents

About the Author

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

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

CHAPTER 1

Introduction

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

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

What Is MERN?

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

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

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!