Tomas Alabes, Konstantin Tarkus

Isomorphic JavaScript Web Development

Universal JavaScript with React and Node

Packt>

Isomorphic JavaScript Web Development

Universal JavaScript with React and Node

Tomas Alabes
Konstantin Tarkus

BIRMINGHAM - MUMBAI

Isomorphic JavaScript Web Development

First published: October 2017

Production reference: 1091017

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN: 978-1-78588-976-9

www.packtpub.com

Credits

Authors
Tomas Alabes
Konstantin Tarkus

Reviewer
Andrew Kovalenko

Commissioning Editor
Wilson D'suoza

Acquisition Editor
Denim Pinto

Content Development Editor
Jason Pereira

Technical Editor
Prashant Mishra

Copy Editor
Charlotte Carneiro

Project Coordinator
Ritika Manoj

Proofreader
Safis Editing

Indexer
Rekha Nair

Graphics
Jason Monteiro

Production Coordinator
Melwyn Dsa

About the Authors

Tomas Alabes is a sr. software engineer building the Clouds at Oracle in Silicon Valley. Having worked as a full-stack engineer for more than seven years, he is also an avid blogger and passionate learner. He is always finding ways to improve himself and share his knowledge.

Konstantin Tarkus is a seasoned software engineer specializing in developing custom web and cloud applications for technology startups. For over 14 years, Konstantin has had extensive experience working with a diverse range of technological stacks such as PHP and MySQL, Azure, SQL Server, .NET, C#, Node.js, and JavaScript. He is the author of React Starter Kit, the most popular React app boilerplate, as well as many other open source projects on GitHub.

About the Reviewer

Andrew Kovalenko is a software developer, team leader, and blogger. He is a member of Jaybird Group, the web and mobile development firm in the United States/Ukraine. He has worked there since the beginning of the company and now has the position of team leader. His work there includes overseeing and implementing projects in a wide variety of technologies, with an emphasis on JavaScript, NodeJS, HTML5, and Cordova (PhoneGap). He leads several development groups that produced products for call centers, marketing companies, real estate agencies, telecommunication companies, healthcare, and many others. Recently he has been focused on learning mobile development in detail. As a result, he started the *BodyMotivator* project – a mobile fitness application. He is a believer in the future of JavaScript as a generic development language.

When he isn't coding, Andrey likes to hang out with his family and exercise at the local cross-fit gym. He is a fitness enthusiast, and he is trying to put all his software development efforts into making life healthier.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at "`https://www.amazon.com/dp/1785889761`".

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

Today, JavaScript is taking the world by storm. Having logic shared between your JavaScript frontend and backend makes your application simple to reason about and maintain. Use the techniques covered in this book to take your JavaScript application to the next level.

What this book covers

`Chapter 1`, *Getting Started with Isomorphic Web Apps*, describes what isomorphic apps are and how they differ from the conventional SPAs that are around. They will understand the main challenges that developers might face while building isomorphic apps. Once the intro is done, they will get their hands dirty by setting up the development environment in order to start creating their isomorphic application.

`Chapter 2`, *Creating a Web UI with React*, describes how to start with building a web UI by using idiomatic JavaScript and React. They will begin creating components for the application and will learn how to organize the flow of data between the components for effective development.

`Chapter 3`, *Working with CSS and Media Assets*, describes how to style their UI components and bundle CSS and graphics into them. They will learn to effectively configure Webpack to bundle their assets.

`Chapter 4`, *Working with Browsersync and Hot Module Replacement*, describes how you can configure server-side rendering for your app, effectively optimizing your website for search engines (SEO) and improving the initial page load time.

`Chapter 5`, *Creating a GraphQL Server*, describes how to implement a GrapgQL server, based on Node, Express, and SQL. They will understand how it works in relation to the traditional RESTful API structure. They will learn to receive, validate, and modify data on the server.

`Chapter 6`, *Fetching Data with Relay*, explains how to effectively use Relay to query and fetch application data, and how to make it work with the application router.

`Chapter 7`, *Implementing Routing and Navigation*, describes how you can implement routing and navigation from scratch, and also how to integrate an existing library to fit into your isomorphic web app.

`Chapter 8`, *Authentication and Authorization*, gets into the intricacies of securing the application. Readers will learn how to implement a token-based authentication and access control for an isomorphic application.

`Chapter 9`, *Testing and Deploying Your App*, explains how to configure unit and integration tests. It also contains a few recipes on how to deploy your app to a cloud hosting service.

What you need for this book

You need any OS with Node 6+ installed and NPM 4+.

Who this book is for

This book is for developers who want to improve their JavaScript application skills and build a unified JavaScript application.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, and user input are shown as follows: "Note that the majority of the preceding modules should be installed as `dev` dependencies, by using the `--save-dev` command-line argument."

A block of code is set as follows:

```
<script src="moment.js"></script>
<script>
  moment().format('MMMM Do YYYY, h:mm:ss a');
</script>
```

Any command-line input or output is written as follows:

```
$ npm install babel-core, bluebird, express \
              moment, react, react-dom --save
```

New terms and important words are shown in bold, for example, appear in the text like this: "Module bundlers such as **Browserify** or **Webpack** allow you to bundle and optimize JavaScript code for a specific environment."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Isomorphic-JavaScript-Web-Development`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/IsomorphicJavaScriptWebDevelopment_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Getting Started with Isomorphic Web Apps

Welcome to the book on building isomorphic web applications where we will discuss the main challenges associated with building isomorphic (also known as universal) apps and will walk through the process of building a simple web application so you can learn by example.

In this chapter, we will give a brief overview of isomorphic apps; you will learn what makes your application code isomorphic; see how simple it is to get your first isomorphic application running and what a good project structure for an isomorphic app could be.

By the end of the chapter, we will have a basic working web application, powered by Node.js, Express, and React 16.

To bring it all together, we will cover the following topics:

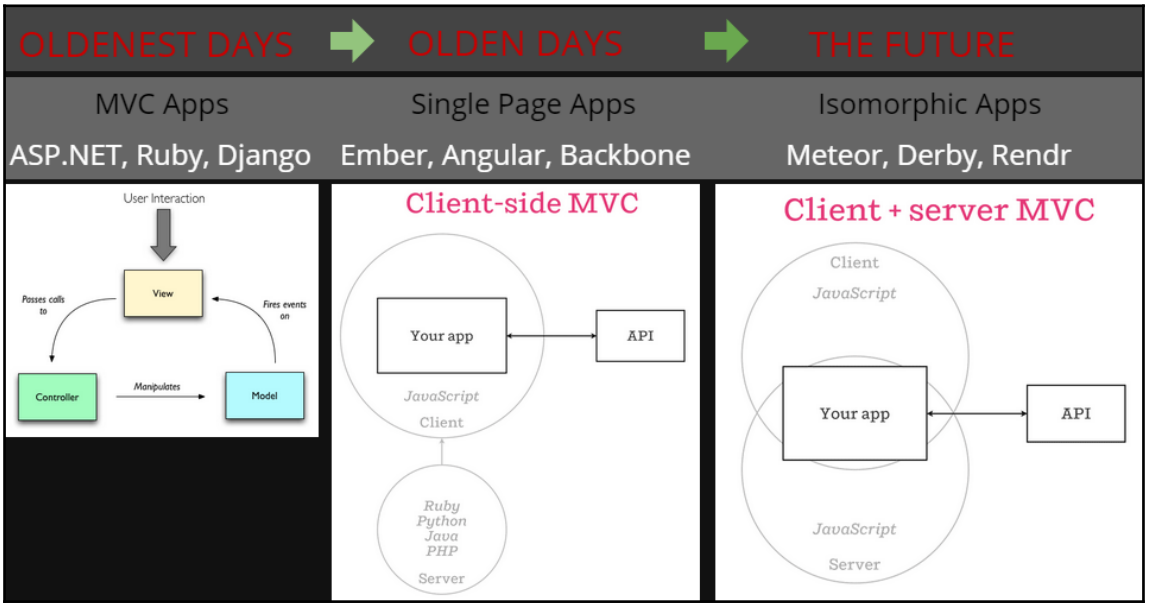
- What are isomorphic apps?
- How to write isomorphic JavaScript code?
- Introduction to React.js
- Sample application description
- What we need to get started
- Install project dependencies
- The basic project structure
- Creating the first React component
- How to render a React app on both client and server
- How to run and test our app locally

What are isomorphic apps?

The very name isomorphic describes this dual-purposes symmetry: *iso* means equal and *morphous* means shape.

The term isomorphic apps and the web development approach behind it were popularized by AirBnb in 2013. The main idea of this is to write a JavaScript application designed for the browser but at the same time; it must run on the server for generating HTML markup, which can dramatically improve the initial load time of such applications and naturally solve the problems associated with optimizing such web applications for search engines.

You may have noticed the trend over the last few years of how traditional server-side MVC applications evolved to client-side apps (aka single-page applications) and then to isomorphic apps:



Check out this great article published by AirBnb developers, which gives a thorough introduction to the isomorphic web application development approach:
`http://nerds.airbnb.com/isomorphic-javascript-future-web-apps/`.

Another essential benefit of isomorphic apps (other than initial page load speed and SEO) is an ability to reduce code duplication by sharing the code between client-side and server-side codebases. Since everything is written in JavaScript, it's easier to build cross-functional teams who know how to work on both client and server sides of the app.

What exactly can be shared between client and server? The presentation layer (UI), routing and navigation, data fetching and persistence, UI localization and internalization.

Is it required to have Node.js on the server in order to run isomorphic apps server side? No, you can execute JavaScript code on any platform, such as PHP, .NET, and Ruby. However, in this book, we're going to use Node.js and Express to keep things simple. Who is using this approach? Flickr, Instagram, Airbnb, and Asana, to name a few.

How to write isomorphic JavaScript code?

Isomorphic (aka universal) JavaScript code can be either environment agnostic or shimmed per environment. It means that it cannot contain browser-specific (window) or server-specific (`process.env`, `req.cookies`) properties. Alternatively, it must provide shims for accessing such properties so the module can expose a single API (`window.location.path` vs `req.path`).

Many modules in the npm repository already have this trait. For example, the `Moment.js` can run in both Node.js and browser environments as the following code demonstrates:

Server (Node.js):

```
import moment from 'moment';
moment().format('MMMM Do YYYY, h:mm:ss a');
```

Client (browser):

```
<script src="moment.js"></script>
<script>
  moment().format('MMMM Do YYYY, h:mm:ss a');
</script>
```

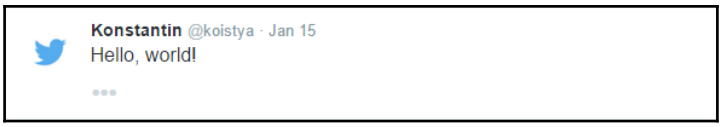
Module bundlers such as **Browserify** or **Webpack** allow to bundle and optimize JavaScript code for a specific environment. Later in this chapter, you will see how we use Webpack to generate two bundles from the same source code, one of which is optimized for running in a browser and another one is optimized for Node.js environment.

Introduction to React.js

Facebook's React library is a great choice for building isomorphic web UIs; it allows creating an in-memory representation of user interfaces, which looks like a tree of UI elements (components) and can be easily rendered either on a client or a server.

Let's do a simple exercise. If you were asked to create a JavaScript representation of a UI element such as a single tweet from `twitter.com`, how would you do it?

Here is how this UI element should look in a browser:



The HTML code for it would look similar to this:

```
<div class="tweet">
<div class="header">
<a class="account" href="/koistya">
<img class="avatar" src="/koistya/avatar.png" />
<strong class="fullname">Konstantin</strong>
<span class="username">@koistya</span>
</a>
<small class="time">Jan 15</small>
</div>
<div class="content">
<p class="text">Hello, world!</p>
<div class="footer">...</div>
</div>
</div>
```

Most likely, you would come up with a JavaScript object, which looks as follows (some code is omitted and replaced with `...` to keep it short):

```
const tweet = {
  node: 'div',
  class: 'tweet',
  children: [
    {
      node: 'div',
      class: 'header',
      children: [ ... ]
    },
    {
```

```
        node: 'div',
        class: 'content',
        children: [
          {
            node: 'p',
            class: 'text',
            children: 'Hello, world!'
          },
          ...
        ]
      }
    ]
  };
```

Having this object, you can easily write two `render()` functions, one for Node.js, which will traverse this object and build an HTML string, and another one for a browser which will also travers this object and build a DOM tree (for example, using `document.createElement(...)`).

In order to make this tweet object reusable, being able to display multiple tweets on a web page, you will want to convert it to a function, which will look similar to this:

```
function createTweet({ author, text }) {
  return {
    node: 'div',
    class: 'tweet',
    children: [
      {
        node: 'div',
        class: 'header',
        children: [...]
      },
      {
        node: 'div',
        class: 'content',
        children: [
          {
            node: 'p',
            class: 'text',
            children: text
          },
          ...
        ]
      }
    ]
  };
}
```

Now, you can construct a tweet object by passing some data to this function (props, in React terminology, and render it to HTML):

```
const tweet = createTweet({ author: ..., text: 'Hello, world!' });
const html = render(tweet);
```

`React.js` takes this to the next level by abstracting all the complexities of this approach from you and providing a simple to use API with just a few public methods. For example, instead of writing your own render function, you can use:

```
// Node.js
import ReactDOM from 'react-dom';
ReactDOM.hydrate(tweet, document.body);

// Browser
import ReactDOM from 'react-dom/server';
const html = ReactDOM.renderToString(tweet);
```

Since React 16, the react-dom package `render` method is called `hydrate`.

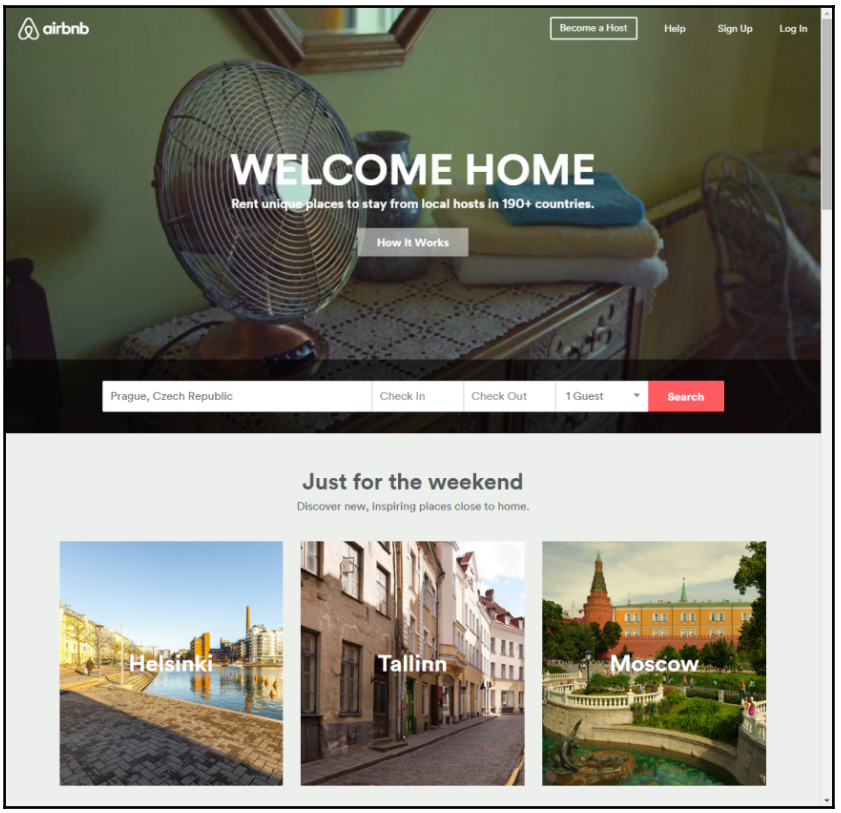
In a browser, React can detect changes made to the virtual representation of the UI, and synchronize it with the actual DOM in the most efficient way possible. So, you do not need to worry much (if at all) about mutating browser's DOM, but instead, can focus on creating UI elements of your app in a declarative manner (using JSX syntax) and managing application's state.

That's the library we are going to use for building a UI part of our isomorphic app.

Sample app description

Throughout the remaining chapters of the book, we are going to build a simple web application. This will be similar to the Airbnb site where users can either list their belongings they're willing to lend or find interesting items to rent posted by other users.

The interface of the app will be very similar to this one:



By the end of the book, we're going to build a minimal viable version (MVP) of this product, let's call it `rentalitems.com`. In the final chapter, we will go through the deployment process and publish the site live to make things even more interesting.

What we need to get started

We are going to build a sample isomorphic application by using JavaScript cross-stack. Therefore, the only requirement is to have the latest version of Node.js installed on your machine plus a text editor or IDE with ES6/ES2015 and JSX support. If you don't have Node.js installed, you can find the concrete steps on how to install it on the `https://nodejs.org` website.

WebStorm, Atom, or Sublime Text 3 are generally considered good option. Those can run on any platform. Just make sure that your text editor is properly configured to work with ES6/ES2015 JavaScript, JSX, and ESLint.

Optionally, you can also install the **React Developer Tools** extension for Google Chrome or Firefox to help you with debugging React apps in the browser.

Installing project dependencies

Let's go ahead and create an empty folder for our project, add the `package.json` file into it and install npm modules that we will need for the first chapter. You can either run the `npm init` command from a command line to generate a new `package.json` file or just create it manually and put `{ }` inside to make it a valid JSON.

The table here shows that where in the app each module is going to be used:

npm Module	Client	Server	Build	Test
babel			x	
babel-core	x	x		
babel-eslint				x
express			x	
del		x		
eslint				x
eslint-config-airbnb				x
eslint-plugin-react				x
extend			x	
gaze			x	
ncp			x	
moment	x	x		
react	x	x		
react-dom	x	x		
webpack			x	

Some of the packages are going to be used only by the client-side portion of the app, some, only for the server-side code (Node.js), and some will be shared between client and server; there are also packages that will not be used by the app directly but will help with bundling and optimization, testing, and debugging.

Some small explanation about these packages:

- `babel/babel-core`: JavaScript transpiler from new JavaScript syntaxes to JavaScript supported by the browsers chosen by you
- `express`: The web framework that we will use in this book for the server.
- `del`: This makes deleting files/folder easier
- `eslint*`: The JavaScript linter, works with the new js syntax
- `extend`: They extend a JavaScript object with others
- `gaze`: This watches for file system file changes
- `ncp`: This is async recursive file copying utility
- `moment`: This is the time library for JavaScript
- `react*`: No need for much explanation here
- `webpack`: The module bundler that we will use for our isomorphic app

To install these packages simply run:

```
$ npm install babel-core, bluebird, express \
              moment, react, react-dom --save
$ npm install babel babel-eslint babel-loader del eslint \
              eslint-config-airbnb eslint-plugin-react \
extend gaze ncp webpack --save-dev
```

Note, that the majority of the preceding modules should be installed as dev dependencies, by using the `--save-dev` command-line argument. The only packages that need to be installed as direct application dependencies are the ones that are supposed to be used at runtime by Node.js app (see Server column in the preceding table). It is also considered a good practice to use strict version numbers for the modules that application uses at runtime.

Now, the contents of your `package.json` file should look similar to this:

```
{
  "private": true,
  "dependencies": {
    "bluebird": "3.5.0",
    "express": "4.15.4",
    "moment": "2.18.1",
    "react": "16.0.0",
    "react-dom": "16.0.0"
  },
  "devDependencies": {
    "autoprefixer": "7.1.4",
    "babel-cli": "6.26.0",
```

```
    "babel-core": "6.26.0",
    "babel-eslint": "8.0.0",
    "babel-loader": "7.1.2",
    "babel-plugin-transform-runtime": "6.23.0",
    "babel-preset-node5": "12.0.1",
    "babel-preset-react": "6.24.1",
    "babel-preset-stage-0": "6.24.1",
    "del": "3.0.0",
    "eslint": "4.7.0",
    "eslint-config-airbnb": "15.1.0",
    "eslint-plugin-react": "7.3.0",
    "extend": "3.0.1",
    "gaze": "1.1.2",
    "ncp": "2.0.0",
    "webpack": "3.6.0"
  }
}
```

The basic project structure

Now, let's create the initial folder structure for our project, which will look like this:

```
.
├── /build/              # Folder for compiled output
├── /components/         # React components
├── /core/               # Core application code
├── /data/               # GraphQL data types
├── /node_modules/       # 3rd-party libraries and utilities
├── /public/             # Static files
├── /routes/             # Isomorphic application routes
├── /test/               # Unit and integration tests
├── /tools/              # Build automation scripts and utilities
├── .babelrc             # Babel configuration
├── .editorconfig        # Text editor configuration
├── .eslintrc            # ESLint configuration
├── .gitignore           # Files to exclude from SCM
├── client.js            # Client-side startup script
├── package.json         # Holds various project's metadata
└── server.js            # Server-side startup script
```

It is important to organize your source files in a way so that it will be easier for you to access them and reference one from one another. Avoid deeply nested folder structures. Group your files by purpose rather than by file types.

Sometimes, developers split the application's source code into client and server folders. With isomorphic apps that might be unnecessary because many of the components are shared between client-side and server-side code.

You can copy the dot files (`.babelrc`, `.editorconfig`, `.eslintrc`, and `.gitignore`) from the example source code accompanying this book into your project.

Creating the first react component

The next thing we're going to create is our first React component and then try to render it both client side and server side. Let's create `components/App.js` with the following content:

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

class App extends Component {
  constructor(props) {
    super(props);
    this.state = { time: null };
  }
  componentDidMount() {
    this.tick();
    this.interval = setInterval(this.tick.bind(this), 200);
  }
  componentWillUnmount() {
    clearInterval(this.interval);
  }
  tick() {
    this.setState({ time: new Date() });
  }
  render() {
    const time = this.state.time;
    const timeString = time && moment(time).format('h:mm:ss a');
    return (
<div>
<h1>Sample Application</h1>
<p>Current time is {timeString}</p>
</div>
    );
  }
}

export default App;
```

Since this component is using state, we extend it from the `React.Component` base class and set the initial state inside the `constructor()` method. Also we're using two of React's life cycle methods `componentDidMount()` and `componentWillUnmount()` to start the timer when the component is mounted to the browser's DOM and clear the timer before the component is unmounted. Inside the `render()` method, we're using the `Moment.js` library to format the date object to a user-friendly time string.

Note that we set the initial time state variable to null and not new `Date()`. This is required in order to make the first call to the `render()` method (during the initial rendering) return the exact same output, in-memory representation of the UI tree. When you render this component on the client side, in a browser, React will first check if the checksum of that UI tree matches what has been rendered on the server. If so, instead of generating an HTML page from scratch, it will pick up existing HTML from the DOM and just bind necessary DOM event handlers to it, effectively mounting the top-level React component (app in our case) into the DOM.

It's worth mentioning that in the current version of React, you cannot return multiple components from the `render()` method. For example, the following code will fail with the error `Adjacent JS elements must be wrapped in an enclosing tag`:

```
render() {
  return (
    <h1>Sample Application</h1>
    <p>Current time is {new Date().toString()}</p>
  );
}
```

In most cases, this is not a big deal. If you like, you can subscribe to issue *#2127* in the React repository on GitHub to track the status of this problem.

Rendering a React app on both client and server

Look at the following two code snippets showing how to render the same top-level React component on the client (in a browser) and on the server (in Node.js app):

In order to render the `App` component on the client, you write:

```
import ReactDOM from 'react-dom';
import App from './components/App';
ReactDOM.hydrate(<App />, document.getElementById('app'));
```

In order to render the same component on the server (in Node.js app), you write:

```
import ReactDOM from 'react-dom/server';
import App from './components/App';
const html = ReactDOM.renderToString(<App />);
```

Both methods will try to build an in-memory representation of the UI tree (aka virtual DOM) of the `App` component. The first one will compare that virtual DOM with the actual DOM inside the `<div id="app"></div>` HTML element and will modify the actual DOM to make it match the virtual DOM exactly. The second method will just convert the in-memory representation of the UI tree into HTML, which then can be sent to a client.

Now, let's see how a complete example for the client-side and server-side application code looks like. Go ahead and create `client.js` file with the following content:

```
import 'babel-core/register';
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';

function run() {
  ReactDOM.hydrate(<App />, document.getElementById('app'));
}

const loadedStates = ['complete', 'loaded', 'interactive'];

if (loadedStates.includes(document.readyState) && document.body) {
  run();
} else {
  window.addEventListener('DOMContentLoaded', run, false);
}
```

This ensures that the React application is only mounted when the HTML page was fully loaded into the browsers.

For the server-side app, let's create `server.js` file with the following content:

```
import express from 'express';
import React from 'react';
import ReactDOM from 'react-dom/server';
import App from './components/App';

const server = express();
const port = process.env.PORT || 3000;
server.get('*', (req, res) => {
  const title = 'Sample Application';
  const app = ReactDOM.renderToString(<App />);
```

```
  res.send(`<!doctype html>
<html>
<head>
<title>${title}</title>
<src script="client.js"></script>
</head>
<body>
<div id="app">${app}</div>
</body>
</html>`);
});

server.listen(port, () => {
  console.log(`App is listening at http://localhost:${port}/`);
});
```

It renders the App component to an HTML string, wraps it into a valid HTML5 document with `<head>` and `<body>` sections, and sends it to a client on all HTTP requests to the server.

We can go further and replace that ES7 string literal above with a React-based template in order not to worry about escaping HTML fragments. In order to do so, create `components/Html.js` file with the following content:

```
import React from 'react';

const Html = (props) =>
<html>
<head>
<meta charSet="utf-8" />
<meta httpEquiv="x-ua-compatible" content="ie=edge" />
<title>{props.title || ''}</title>
<meta name="description"
            content={props.description || ''} />
<meta name="viewport"
            content="width=device-width, initial-scale=1" />
<script src="client.js" />
</head>
<body>
<div id="app"
          dangerouslySetInnerHTML={{__html: props.children}} />
</body>
</html>;

export default Html;
```

Since this component doesn't use state, we don't need to extend it from `React.Component`, but instead, we can use a regular function which accepts a collection of properties as an argument and returns a React component. Make sure that all the tags in the mark-up used in your React components are properly closed and you use valid JSX attributes. For example, instead of `<meta charset="utf-8">`, you should write `<meta charSet="utf-8" />` and so on.

If you're new to JSX syntax, visit the following two pages to get more information about it:
`https://facebook.github.io/react/docs/jsx-in-depth`
`https://facebook.github.io/react/docs/jsx-gotchas`.

Another addition which we can make to the `server.js` file is to add `Express.js` static middleware to make it serve static files, such as `robots.txt` from the `/public` folder. Now, the final `server.js` file should look like this:

```
import path from 'path';
import express from 'express';
import React from 'react';
import ReactDOM from 'react-dom/server';
import Html from './components/Html';
import App from './components/App';

const server = express();
const port = process.env.PORT || 3000;

server.use(express.static(path.join(__dirname, 'public')));

server.get('*', (req, res) => {
  const body = ReactDOM.renderToString(<App />);
  const html = ReactDOM.renderToStaticMarkup(<Html
    title="My App"
    description="Isomorphic web application sample"
    body={body} />);
  res.send('<!doctype html>\n' + html);
});

server.listen(port, () => console.log(
  `Node.js server is listening at http://localhost:${port}/`
));
```

How to run and test our app locally

The next step is to configure build automation scripts, which will help us to bundle and optimize the source code so it can be deployed to a server. In addition, we'll need to set up a script, which will launch an HTTP server locally so we can test and debug our app.

It is considered a good practice to bundle the source code of a web application into distributable format, which will be optimized for running on the server and in a browser. You cannot just run `server.js` as it is on the server because it will contain many pieces that the current version of Node.js just doesn't understand, like, for example, `async/await` syntax.

In this book, we are going to use Babel and Webpack to help us transpile and bundle all the source code and other assets (images and fonts) into a distributable format and save the contents of the compiled output into the `/build` folder. The contents of the `/build` folder will look something like this:

```
.
├── /build/                     # Folder for compiled output
│   ├── /public/                # Static files
│   │   ├── client.js           # Client-side application bundle
│   │   ├── favicon.ico         # Application icon
│   │   └── robots.txt          # Search engine crawlers' settings
│   ├── package.json            # The list of npm modules
│   └── server.js               # Server-side application bundle
└── ...
```

It might also be a good idea to run your site locally from the `build` folder (as opposed to running it from source files), this way you can ensure that you're testing the exact same application which will eventually be deployed to a server.

Often, frontend developers use tools, such as Grunt or Gulp, to help with build automation, but in our case the majority of the work will be performed by Webpack, so we can just use plain JavaScript for automation scripts reducing the number of external dependencies in our project. Here is how the barebones of the build script (`tools/build.js`) may look written with plain JavaScript:

```
async function clean() {
  // TODO: Clean up the output directory
}

async function copy() {
  // TODO: Copy static files to the output directory
}
```

```
async function bundle() {
  // TODO: Bundle the source code with Webpack
}

async function build() {
  await clean();
  await copy();
  await bundle();
}

export default build;
```

We can improve it by making it writing to the console information about when a particular task (JavaScript function) started and ended, as well as how much time it took to perform any particular task. Let's add `tools/run.js` file with the following content:

```
function format(time) {
// return human readable string
return time.toTimeString()
             .replace(/.*(\d{2}:\d{2}:\d{2}).*/, '$1');
}
// run the async fn and time how much it took to complete
async function run(fn, options) {
  const start = new Date();
  console.log(`[${format(start)}] Starting '${fn.name}'...`);
  await fn(options);
  const end = new Date();
  const time = end.getTime() - start.getTime();
  console.log(
    `[${format(end)}] Finished '${fn.name}' after ${time} ms`
  );
}

if (process.mainModule.children.length === 0 && process.argv.length > 2) {
  delete require.cache[__filename];
  const module = process.argv[2];
  run(require('./' + module + '.js'))
    .catch(err => console.error(err.stack));
}

export default run;
```

In addition to the actual `run()` method, it contains code, which helps to execute our automation scripts using npm CLI.

Now, let's update our `tools/build.js` file to use the utility method `run()`:

```
import run from './run';

async function clean() {
  // TODO: Clean up the output directory
}

async function copy() {
  // TODO: Copy static files to the output directory
}

async function bundle() {
  // TODO: Bundle the source code with Webpack
}

async function build() {
  await run(clean);
  await run(copy);
  await run(bundle);
}

export default build;
```

Now, we can update the `package.json` file in the root of our source tree to include the following lines:

```
{
  ...
"scripts": {
"lint": "eslint components core data routes test tools",
"build": "babel-node tools/run build",
"serve": "babel-node tools/run serve"
  }
}
```

This will allow us to run JavaScript-based automation scripts from the `/build` folder by using npm CLI. For example:

```
$ npm run build
```

The next step is to create a configuration file, which will be used by Webpack. Let's create `tools/webpack.config.js` file with the following content:

```
import path from 'path';
import extend from 'extend';

const common = {
```

```
  stats: {
    colors: true,
    chunks: false
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        include: [
          path.join(__dirname, '../components'),
          path.join(__dirname, '../core'),
          path.join(__dirname, '../data'),
          path.join(__dirname, '../routes'),
          path.join(__dirname, '../client.js'),
          path.join(__dirname, '../server.js')
        ],
        loader: 'babel-loader'
      }
    ]
  }
};

const client = extend(true, {}, common, {
  entry: path.join(__dirname, '../client.js'),
  output: {
    publicPath: '/',
    path: path.join(__dirname, '../build/public'),
    filename: client.js'
  }
});

const server = extend(true, {}, common, {
  entry: path.join(__dirname, '../server.js'),
  output: {
    path: path.join(__dirname, '../build'),
    filename: 'server.js',
    libraryTarget: 'commonjs2'
  },
  target: 'node',
  node: {
    console: false,
    global: false,
    process: false,
    Buffer: false,
    __filename: false,
    __dirname: false
  },
  externals: /^[a-z][a-z\/\.\-0-9]*$/i
```

```
});

export default [client, server];
```

- At the top of the file, we have `const common = { ... }` variable containing a shared configuration, which is used in both client-side and server-side bundle configurations.
- The `stats` property let us configure, what information is going to be printed to the console when Webpack is running.
- The `modules.loaders` property allows configuring the source code loaders/transpilers for different file types. In our case, we say that all `.js` files from the `components`, `core`, `data`, and `routes` folders, as well as `client.js` and `server.js` files must be transpiled by Babel, using `babel-loader` npm module, which itself pickups settings from the `.babelrc` file in the root of our project's source tree.

Then, we use this common configuration object as a base for client and server bundle configurations in order to avoid code repetition since most of the configuration settings will be the same for client-side and server-side bundles as you will see in future chapters.

In the client bundle configuration, we say that the entry point of the client-side app is the `/client.js` file, and the resulting application bundle should be saved to the `/build/public/client.js` file.

Similarly, in the server bundle configuration, we say that the entry point of the server-side app is the `/server.js` file, and the resulting application bundle should be saved to the `/buid/server.js` file.

The important difference is that for the server bundle configuration, we must specify the `output.libraryTarget` property to be equal to `'commonjs2'`; the `target` property should be equal to `'node'`; the `node` property should contain the list of Node.js environment variables, which must not be mocked; the externals property should be equal to `/^[a-z][a-z\/\.\-0-9]*$/i`, which tells Webpack not to include any source code referenced from the `node_modules` folder into the bundle (`build/server.js`).

Now, let's implement `clean()`, `copy()`, and `bundle()` methods in our build script, which should now look like this:

```
import del from 'del';
import webpack from 'webpack';
import Promise from 'bluebird';
import run from './run';
import webpackConfig from './webpack.config';
```

```
async function clean() {
  await del(['build/*', '!build/.git'], { dot: true });
}

async function copy() {
  const ncp = Promise.promisify(require('ncp'));
// we copy everything inside public and package.json to the build folder
await ncp('public', 'build/public');
  await ncp('package.json', 'build/package.json');
}

function bundle({ watch }) {
  return new Promise((resolve, reject) => {
    let runCount = 0;
    const bundler = webpack(webpackConfig);
    const cb = (err, stats) => {
      if (err) {
        return reject(err);
      }

      console.log(stats.toString(webpackConfig[0].stats));

      if (++runCount === (watch ? webpackConfig.length : 1)) {
        return resolve();
      }
    };

    if (watch) {
      bundler.watch(200, cb);
    } else {
// run webpack normally
   bundler.run(cb);
    }
  });
}

async function build(options = { watch: false }) {
  await run(clean);
  await run(copy);
  await run(bundle, options);
}

export default build;
```

Lastly, let's create the `tools/serve.js` script, which will build the project and tell Webpack to watch for modifications in the source files so that as soon any of the source files are modified, Webpack will instantly update the `/build/server.js` (server) and `/build/public/client.js` (client) bundles. In addition to that, it will automatically restart the Node.js server whenever `/build/server.js` file is changed.

The source code of that script looks like this:

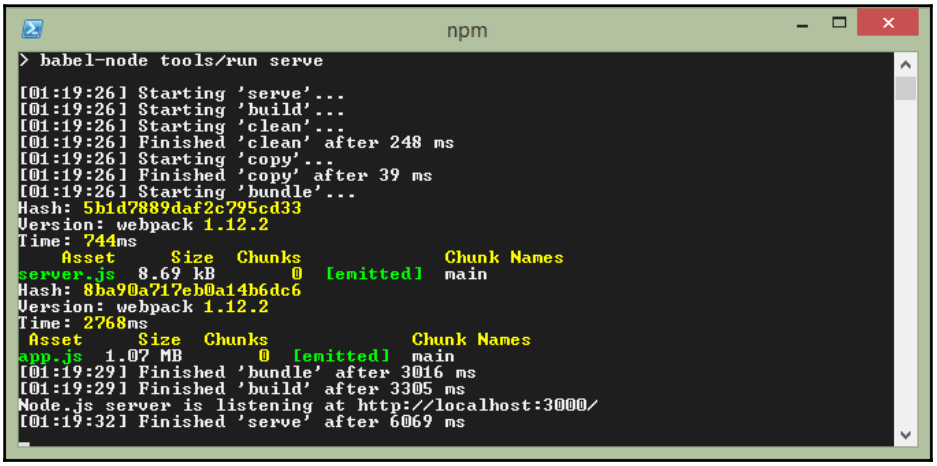
```
import path from 'path';
import cp from 'child_process';
import Promise from 'bluebird';
import build from './build';
import run from './run';

async function serve() {
  const watch = true;
  const app = path.join(__dirname, '../build/server.js')
  const gaze = Promise.promisify(require('gaze'));
  await run(build, { watch });
  await new Promise((resolve, reject) => {
    function start() {
      const server = cp.spawn(
        'node',
        [path.join(__dirname, '../build/server.js')],
        {
          env: Object.assign(
            { NODE_ENV: 'development' },
            process.env
          ),
          silent: false
        }
      );

      server.stdout.on('data', data => {
        process.stdout.write(new
Date().toTimeString().replace(/.*(\d{2}:\d{2}:\d{2}).*/, '[$1] '));
        process.stdout.write(data);
        if (data.toString('utf8').includes('Node.js server is listening
at')) {
          resolve();
        }
      });
      server.stderr.on('data', data => process.stderr.write(data));
      server.once('error', err => reject(err));
      process.on('exit', () => server.kill('SIGTERM'));
      return server;
    }
```

```
    let server = start();

    if (watch) {
  // when the server.js file changes, we will restart the server
    gaze('build/server.js').then(watcher => {
        watcher.on('changed', () => {
          server.kill('SIGTERM');
          server = start();
        });
      });
    }
  });
}

export default serve;
```

Now, you can run the app by executing the following command from a console:

<pre>**$ npm run serve**

If everything is alright, you should be able to see the following output in the console window:

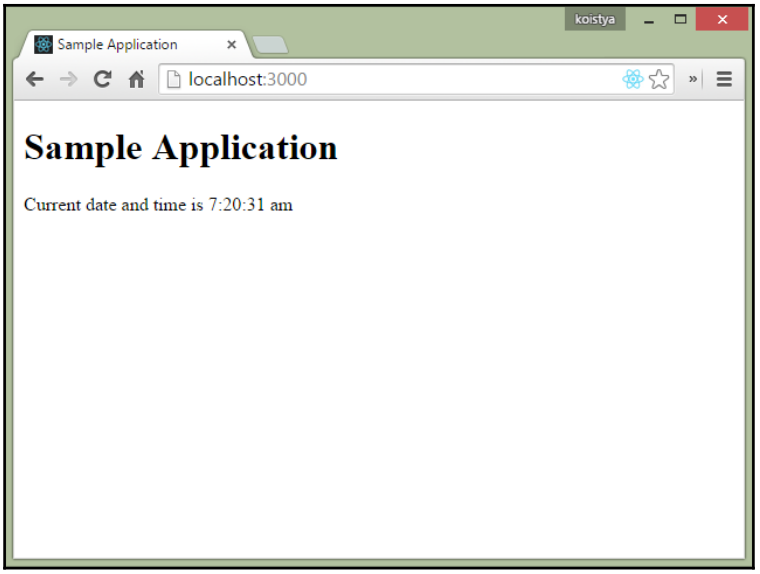


From this output, you can see that Webpack created two application bundles: `server.js`, which took 744 ms to build and is 8.69 kB in size (that's because it doesn't contain any code from the referenced npm modules), and `client.js` bundle, which took 2768 ms to compile and it's 1.07 MB. That's because the client-side bundle contains the debug version of the source code referenced by the client-side app, namely, react, react-dom, and moment modules.

If you try to make changes to any of the source files, Webpack will recompile the bundles, but this time it will take much less time-about 50ms for the server-side bundle and about 500 ms for the client-side bundle. That's because it caches unchanged modules and output files between compilations.

In a large application, the initial compilation can take up to 30 seconds and the incremental updates about 2-3 seconds. If you notice, that your app compiles longer than that, this could mean that something is wrong with your build configuration.

Now, if you open `http://localhost:3000/` URL in your browser, you should see how our app looks like:



Congratulations! You just successfully completed the first step of creating a basic React application as well as making sure that it renders both client-side and server-side. Now, you can compare your solution to the example source code, which accompanies this book and double check if you correctly completed all the steps described in this chapter.

Summary

In this chapter, we have given a basic overview of isomorphic apps. We explained what tools are needed to write JavaScript isomorphic apps. In the next chapter, we will drill into how to compose web UIs using React.

2
Creating a Web UI with React

Now that we got our feet wet with the isomorphism of JavaScript and node, Webpack as our building tool, testing and we have the base for our project; we need to start creating the **user interface** (**UI**) for it!

In this chapter, you will learn how to split UI into components, how to style those components, avoid shared CSS in favor of sharing and reusing components, and how to render different screens (or pages) of our app given a certain URL or path.

The following is on overview of what will be covered in this chapter:

- Component-based UI development
- Stateful versus stateless React components
- Grouping UI components in a project
- Breaking the UI into a component hierarchy
- Building a static version with React
- Implementing a basic isomorphic router

Component-based UI development

If you start working on a medium-size to large web application project for the first time, you will soon realize that sharing CSS and other assets throughout the project may lead to constant bugs, and in general, makes the development process harder than it should be. Imagine when someone from your team adds a global CSS rule in order to add some new functionality on the site but accidentally breaks a couple of existing pages. Even if you start small, as soon as your app becomes more complex, you will face this issue when you need to rethink your web UI architecture and project structure in order to make things more manageable and easy to maintain.

Many companies use a component-based approach to web UI development-an approach where instead of referencing shared styles, media assets, localization files, and so on, developers create standalone components, which can be developed and tested in isolation. These components can be shared and reused across the app as units or modules. Each component contains all the source files required to render a specific UI element. It may contain JavaScript code, CSS styles, images, unit tests, documentation, and other files if necessary. With the help of module bundlers (for example, Webpack and **Browserify**), the application containing such components can be compiled into a distributable format before deployment to a web server.

Q: Should I put all the source files, including CSS, images, icons, and fonts for each component into a separate folder?
A: Yes, that makes total sense. This may look like over-engineering at the first glance, but after you get familiar with this structure, you're going to really appreciate it. Editing UI will be more enjoyable and there will be less confusion when on-boarding new developers to your team. Note that simply building your web UI with React doesn't mean that you're using a component-based development approach.

You might be already familiar with **BEM**-a methodology for creating modular web applications developed by Yandex back in 2007. It describes the way developers can build complex web UIs out of small blocks (components). This methodology was widely adopted by other companies and to this day remains a well-known development approach in the frontend development community. To give you a couple of examples, check out the SUITE CSS framework by Twitter and the **Material Design Light** (**MDL**) framework by Google, both heavily inspired by BEM.

If you want to learn more about this methodology from the inventors, visit `https://en.bem.info/` and `http://getbem.com/faq/`.

The BEM methodology is as it is described on the official site, is not fully applicable to what you are going to learn in this book, but still, we're going to borrow some of its core principles and ideas be such as:

- Avoid shared CSS styles used in among different components. At first glance, this may look like a lot of code duplication, but in practice, your UI code will be much more maintainable and easy to refactor, yet you will be forced to think in terms of creating portable reusable components.
- Exclude the possibility of introducing conflicting CSS styles. For example, two or more components may define a classname `.title` and when you put them on the same web page, one or the other will take precedence. In BEM, the problem is often solved using namespaces, but we are going to tackle this problem with **CSS Modules** and Webpack.
- Find a way to group all the assets for each component, including unit tests and documentation, in their own place within a project. For example, when you delete a component from the project, you must be confident that you have not left any dormant CSS styles or other artifacts.
- Express intent and structure exclusively with CSS classes; avoid using element selectors. For example, you may choose either `<h1>` or `<div>` elements for some UI elements and, both should look the same if you apply class `.title` to them.
- Avoid both chained and nested CSS class selectors. For example, instead of writing `.item.selected { }` or `.item .selected { }`, it's better to have `.item--selected { }`.

You may ask: what about inline CSS, isn't it a new trend in UI development? While inline CSS approaches sound awesome in the context of developing SPA types of applications with React, they may not be as perfect in the context of isomorphic web apps. Because pre-rendered HTML generated on the server might be way too verbose with inline styles. However, nothing prevents you from using both external style sheets for the general look and feel of your app and inline CSS for dynamic functionality. Just find the right balance between these two.

If you're curious on what a pure inline CSS approach looks like, you can find great videos by Christopher Chedeau (aka Vjeux) on this subject; for example, `https://vimeo.com/116209150`.

Stateful versus stateless React components

The majority of UI elements of a web app tends to be stateless React components, which will be rendered the same way given the same input parameters. This is based on a well-known idea of pure functions.

The best way to write such components is by using simple functions accepting props, as the following example demonstrates:

```
function Checkbox({ name, label, ...other }) {
  return (
    <div className="field" {...other}>
      <input className="input" type="checkbox" name={name} />
      <label className="label">{label}</label>
    </div>
  );
}
```

If this list of function arguments looks foreign to you, you may want to read about destructing assignment on MDN:
`https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment`
`function ({ name }) { console.log(name); }`
vs
`function (props) { console.log(props.name); }`

It is considered a good practice to always validate input parameters of your React components either using **Flow** type checker or React's `PropTypes` constraints.

Since React 16, PropTypes are in a separate package called 'prop-types', not in the main react package anymore.

Let's add props validation to the React component from the previous example:

```
import React from 'react';
import PropTypes from 'prop-types';
function Checkbox({ name, label, ...other }) {
  return (
    <div className="field" {...other}>
      <input className="input" type="checkbox" name={name} />
      <label className="label">{label}</label>
    </div>
  );
}
```

```
Checkbox.propTypes = {
  name: PropTypes.string.isRequired,
  label: PropTypes.string.isRequired
};
export default Checkbox;
```

Flow is a static type checker developed at Facebook. It's designed to find type errors in JavaScript programs. Learn more about it at `http://flowtype.org`.

If a React component does not have the state, it's always better to write it as a pure function. This way React will be able to optimize the usage of such components internally. Also, your code will look simpler this way (as opposed to writing components with class-based syntax). In fact, you may want to have as many such components as possible.

The rest of your React components will have some internal state. These type of components are best described with the ES6 class-bases syntax as the following example demonstrates:

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

class Menu extends Component {
  constructor() {
    super();
    this.state = { hidden: false };
  }
  render() {
    return (
      <div className="menu"
        style={{display: this.state.hidden&& 'none'}}>
        ...
      </div>
    );
  }
}

export default Menu;
```

By the way, in the example earlier we are using inline CSS for dynamic functionality, which is a simpler approach than appending `.hide` CSS class to the element and having a CSS rule for it in an external file.

Adding props validation to such components is very easy to do with ES7 class property initializers:

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
class Counter extends Component {
  staticpropTypes = { initialCount: PropTypes.number };
  staticdefaultProps = { initialCount: 0 };
  constructor(props) {
    super(props);
    this.state = { count: props.initialCount };
  }
  tick = () => {
    this.setState({ count: this.state.count + 1 });
  };
  render() {
    return (
      <div onClick={this.tick}>Clicks: {this.state.count}</div>
    );
  }
}

export default Counter;
```

A good rule of thumb when working with React is to: try to keep as many of your React components as possible stateless. By doing so, you'll isolate the state to its most logical place and minimize redundancy, making it easier to reason about your application.

A common pattern is to create several stateless components that just render data and have a stateful component wrapping them that passes its state to its children via props. The stateful component encapsulates all of the interaction logic, while the stateless components take care of rendering data in a declarative way.

Stateful components do not normally render HTML DOM elements themselves directly. They're more like the controllers in MVC, and use other dumber components--stateless, the ones such as views on MVC--to actually render DOM elements.

Grouping UI components in a project

There are two extremes when it comes to the organization of UI components within a project, one of which is to put every single component into the same folder, regardless of when and how they are supposed to be used. The other extreme is to create many subfolders for each group of components.

Both approaches have their cons and pros. But, as you may guess, we advise you to have a habit of always thinking about what might be the most balanced approach, the golden mean, taking into consideration the size of your project, and other similar aspects.

Later in this book, we are going to split our components into two major groups and create a separate folder for each of them. Some components are either going to be shared (reused on multiple screens) or generic (not necessarily related to any particular screen or web page). We are going to put them into the `components` folder. Inside the `components` folder, we are always going to create a separate subfolder for each of these components.

This is what this `components` folder layout may look like:

```
├── /components/
│   ├── /Layout/
│   │   ├── /Layout.js
│   │   ├── /Layout.scss
│   │   └── /package.json
│   ├── /Navigation/
│   │   ├── /hamburger-icon.svg
│   │   ├── /Navigation.js
│   │   ├── /Navigation.scss
│   │   └── /package.json
│   ├── /Tweet/
│   │   ├── /Footer.js
│   │   ├── /Header.js
│   │   ├── /package.json
│   │   ├── /Tweet.js
│   │   └── /Tweet.scss
```

Notice that we only have one level of nesting inside the `components` folder. We intentionally avoid deeply nested folder structures, which may help with the developer experience--how easy the navigation through the project is going to be for you and your team members.

It is perfectly normal to split your shared React components into many small ones and still have them located in the same folder given that these smaller React components are going to be used locally. For example, the `Tweet` component in the preceding example (also mentioned in `Chapter 1`, *Getting Started with Isomorphic Web Apps*) contains `Footer` and `Header` sub-components located within the same `Tweet` folder. They will only be used by the parent `Tweet` component and not referenced directly from other places in the app.

You may also notice the `package.json` file in each of the `components` folders; it allows you to have less verbose references in your code: `import Navigation from '../Navigation';` vs `import Navigation from '../Navigation/Navigation.js';`

Here is what the `package.json` file for the `Navigation` component looks like:

```
{
  "private": true,
  "name": "Navigation",
  "main": "./Navigation.js"
}
```

Why not save React components to `index.js` files, thus avoiding the need for a `package.json` file in each component's folder?
With this approach, you may have problems with debugging and/or inconsistencies in your code.

We are also going to create components to render the web pages or screens of our app. We are going to place them in a separate folder called `routes`. Why `routes` and not pages or screens? This is because these components will also contain routing information and data fetching logic.

Here is an example of what the layout of the `routes` folder may look like:

```
├── /routes/
│   ├── /Home/
│   │   ├── /Home.js
│   │   ├── /Home.scss
│   │   └── /package.json
│   ├── /Search/
│   │   ├── /Search.js
│   │   ├── /Search.scss
│   │   └── /package.json
│   ├── /Dashboard/
│   │   ├── /Dashboard.js
│   │   ├── /Dashboard.scss
│   │   └── /package.json
```

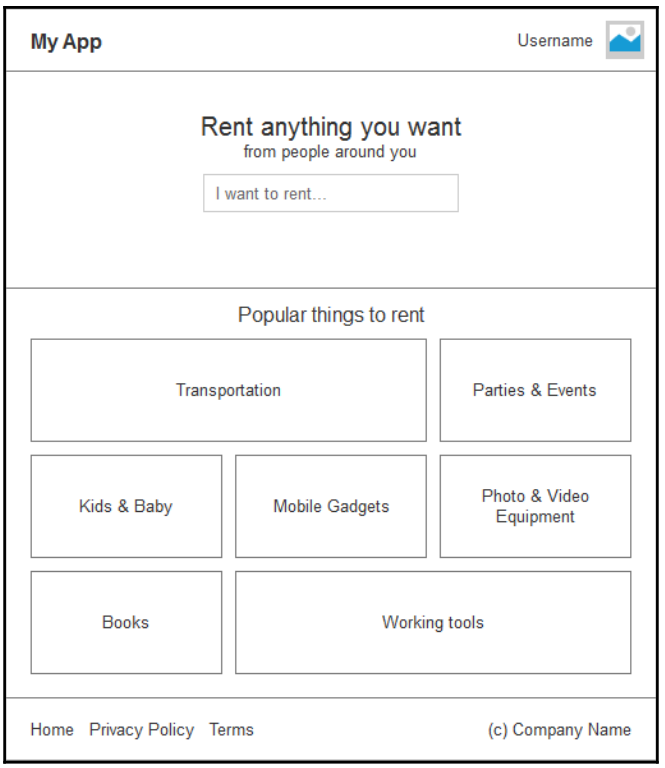
The preceding components are going to be used for the following web pages:

URL Route	**Page Title**	**Component**
`/`	Home	`/routes/Home`
`/search`	Search	`/routes/Search`
`/s/:category`	Search in `{Category}`	`/routes/Search`
`/dashboard`	User dashboard	`/routes/Dashboard`

Breaking the UI into a component hierarchy

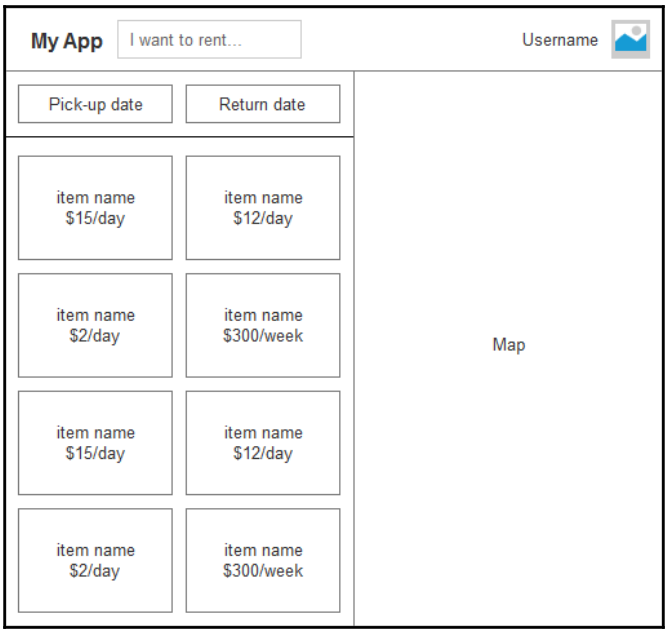
Let's look at what the app is going to be like. On the home page, there will be a large header with a background image. Inside the header, there will be the app name, navigation items, tagline, and a search box. In the main area, there will be a grid of tiles, where each tile represents a category of rental items. At the bottom, there is a regular footer with some links and follow us icons.

Here is the wireframe for the home screen of the app we are going to build:



Try to think about, how you would split this UI into components. Most likely, you will have a `Layout` component, which wraps other major components such as header, footer, and content area. You may also want to extract the page header and footer into separate components, though it's not absolutely necessary; we'll leave it up to you to decide. If at some point during the development you realize that some UI elements are duplicated across the app, you can always extract them from the `Layout` component into separate ones. Inside the content area, there will be the `Home` (home page) component, inside which you may have heading and tile components.

Now, let's see what the next screen looks like, which is the search page:



The header looks the same, so we can reuse the existing `Header` component and just add the `showSearchBox={true|false}` attribute to it in order to be able to hide the search box on the home page. This search page has a unique fluid layout, so we don't need to extract it into a shared component, but we still can have a local `Layout` component responsible for the positioning of the header, map, and search result components on the screen. The **Map** is located on the right half of the screen and is responsible for displaying Google Map and markers. The filter component on the left, right below the header, allows for tweaking search results. List and list item components (for the lack of a better name) help to display the search results.

How do you know what should be its own component? Just use the same techniques that, which you use for creating new functions and objects. One such technique is a single responsibility principle, that is, a component should ideally only do one thing. If it ends up growing, it should be decomposed into smaller sub-components.

Building a static version in React

Now, let's open the sample project from `Chapter 1`, *Getting Started with Isomorphic Web Apps*, and start hacking. First, we are going to drop the `App` component, because it was used solely for demonstration purposes and won't be used down the road in our app.

Then move `Html.js` into the `Html` subfolder for consistency with the other components we are going to create and add the `components/Html/package.json` file with the following contents:

```
{
  "private": true,
  "name": "Html",
  "main": "./Html.js"
}
```

Now, let's create the `Header` component (`components/Header/Header.js`, `components/Header/package.json`), which will looks similar to this:

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

function Header({ children }) {
  return (
    <header>
      <div>
        <span>My App</span>
        {
          !children &&
          <form><input type="search" /></form>
        }
      <div>
        <span>Username</span>
        <imgsrc="#" />
      </div>
    </div>
    {children}
    </header>
  );
}

Header.propTypes = {
  children: PropTypes.element
};

export default Header;
```

This `Header` component will be reused on both regular pages and the search page with a non-standard layout.

Now, let's create the `Layout` component:

```
import React from 'react';
import Header from '../Header';
import PropTypes from 'prop-types';

function Layout({ hero, children }) {
  return (
    <div>
      <Header>{hero}</Header>
      <main>
        {children}
      </main>
      <footer>
        <span>© Company Name</span>
      </footer>
    </div>
  );
}

Layout.propTypes = {
  hero: PropTypes.element,
  children: PropTypes.element.isRequired
};

export default Layout;
```

This `Layout` component accepts a `hero` object as a prop; it is used to pass an instance of the `Hero` component down to the layout hierarchy. This component displays product tagline in the middle of the large header element plus a search box. If this `hero` object is omitted, then a small search box will be displayed instead, next to the app name/logo.

Another prop the `Layout` component accepts is the `children` object, which is going to be rendered in the content section of the layout.

Don't worry about CSS styles for now; we are going to cover this topic in the next chapter.

Now, let's create a component for the home page itself. First, create `routes/Home` folder with the following files in it: `Hero.js`, `Home.js`, and `package.json`.

The `Hero` component will have the product tagline and a search box. Since it's going to be used only on the home page, we put it inside the `Home` (home page) component.

The basic source code of the `Hero` component looks as follows:

```
import React from 'react';

function Hero() {
  return (
    <div>
      <h2>Rent Anything You Want</h2>
      <p>From people around you</p>
      <form>
        <input
        type="search"
        placeholder="I want to rent..." />
        <button>Search</button>
      </form>
    </div>
  );
}

export default Hero;
```

The `Home.js` file will contain the markup for the home page but also the routing information:

```
import React, { Component } from 'react';
import Layout from '../../components/Layout';
import Hero from './Hero';

const path = '/';
const action = () =><Layout hero={<Hero />}><Home /></Layout>;

class Home extends Component {
  handleClick(event) {
    event.preventDefault();
    window.location = event.currentTarget.pathname;
  }
  render() {
    return (
      <div>
        <h2>Popular things to rent</h2>
        <div>
          <a href="/s/Tools" onClick={this.handleClick}>
            <span>Tools</span>
          </a>
          <a href="/s/Books" onClick={this.handleClick}>
```

```
            <span>Books</span>
          </a>
          ...
        </div>
      </div>
    );
  }
}

export default { path, action };
```

As you can see, this component exports a path variable, which is just a string (or, it can be a regular expression) which is going to be used by the `Router` component when it needs to find a corresponding component to render for any given URL path. If the router found a matching component (or, route in our terminology), it will try to execute its action method in order to get the actual instance of a `React` component to render.

Implementing a basic isomorphic router

We are going to cover routing and navigation in one of the later chapters in this book; but for now, let's see how a basic isomorphic router may look like.

Router is the core component in any web application, the main function of which is being able to match any specific URL string (for example `/search?q=canon`) to a corresponding action method, which is responsible for rendering a web page or screen. In its simplest form, a router is just a collection of routes plus a matching method. Each route is just a combination of a URL path string (or, regular expression pattern) and one or more action methods (actions). On the client side, it is used in a combination with a navigation component (for example, the one powered by HTML5 History API).

Let's add the `core/Router.js` file with the following contents:

```
// TODO: Register more routes here
const routes = [
  require('../routes/Home')
];

const router = {
  match(location) {
    const route = routes.find(x =>x.path === location.path);

    if (route) {
      try {
        returnroute.action();
```

```
        } catch (err) {
          returnroutes.find(x =>x.path === '/500').action();
        }
      } else {
        returnroutes.find(x =>x.path === '/404').action();
      }
    }
};

export default router;
```

We'll start with having just a single route (`Home`) in its internal collection just to make sure that everything works as expected and, down the road, you will register more routes in the router as you add them. It is possible to automate this at a build step.

Now, let's see how easy it is to integrate this router into our `client.js` and `server.js` startup scripts.

Here is an updated `client.js` file, that runs in a browser:

```
import 'babel-core/register';
import React from 'react';
importReactDOM from 'react-dom';
import Router from './core/Router';

function run() {
  const component = Router.match({
    path: window.location.pathname
  });
  ReactDOM.hydrate((component, document.getElementById('app'));
}

constloadedStates = ['complete', 'loaded', 'interactive'];

if (loadedStates.includes(document.readyState) &&document.body) {
  run();
} else {
  window.addEventListener('DOMContentLoaded', run, false);
}
```

Here goes `server.js` file, which runs in Node.js:

```
import 'babel-core/register';
import path from 'path';
import express from 'express';
import React from 'react';
importReactDOM from 'react-dom/server';
```

```
import Router from './core/Router';
import Html from './components/Html/Html';

const server = express();
const port = process.env.PORT || 3000;

server.use(express.static(path.join(__dirname, 'public')));

server.get('*', (req, res) => {
  const component = Router.match(req);
  const body = ReactDOM.renderToString(component);
  const html = ReactDOM.renderToStaticMarkup(<Html
    title="My App"
    description="Isomorphic web application sample"
    body={body} />);
  res.send('<!doctype html>\n' + html);
});

server.listen(port, () => console.log(
  `Node.js server is listening at http://localhost:${port}/`
));
```

Now if you run the app (`npm run serve` starts our local server) and open `http://localhost:3000/` in a browser, you should be able to see a rendered home page we created previously. Make sure that there are not any console errors and also view the source of the page in a browser, just to make sure that it was successfully pre-rendered on the server. If everything is OK, you can proceed to add the remaining pages/screens to the project.

You can try to create the remaining pages yourself, in the same way we created a route component for the home page of our sample app. For example, you need to add *not found* and error pages with `/404` and `/500` URL paths, respectively.

Here is an example of the page `NotFound` component:

```
import React  from 'react';

const path = '/404';
const action = () =><NotFound />;

functionNotFound() {
  return (
    <div>
      <h1>Page Not Found</h1>
      <p>Sorry, but the page you were trying to view does not exist.</p>
    </div>
```

```
    );
}

export default { path, action };
```

Register this route in the `core/Router.js` file:

```
const routes = [
  require('../routes/Home'),
  require('../routes/NotFound')
];
```

Now if you open the site and click on books link on the home page, you should be able to see `NotFound` pages similar to this:

You may ask why not simply integrate some existing third-party library? One reason is that for the basic routing you only need to write just a few lines of code to make it work. Another reason is knowing how to implement the routing from scratch may help you to choose an appropriate routing and navigation solution for your next project from a variety of third-party libraries available via npm.

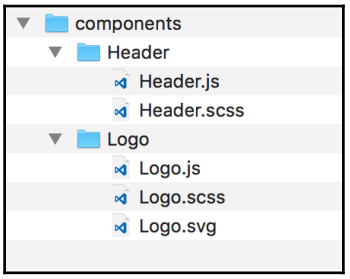
Summary

We walked through the process of creating a basic web interface using the `React.js` library and component-based approach to UI development. We mentioned a few major ideas from the BEM methodology and discussed how you can benefit from this method. You learned about different types of React components (stateful versus stateless) and saw an example of how to group them within a project. The code samples provided in this chapter may have some pieces omitted for brevity. It is highly recommended to open the example source code accompanying this chapter, read the code, and make sure you can reproduce the same steps described in the book on your own. In the next chapter, we are going to cover topics related to CSS styles as well as tips and tricks for working with Webpack, PostCSS, and CSS modules.

3

Working with CSS and Media Assets

As we discussed in the previous chapter, the ideal directory structure for your `components` is when each major UI element is located in its own folder, for example:



It might not be obvious how to concatenate all these files during build process making the app ready for deployment. However, thanks to module bundlers, such as Webpack and Browserify, this task is actually quite simple. You just need to learn how to configure these tools. There might be some caveats, but after completing this chapter, you should be more or less comfortable configuring bundling and optimization tools yourself.

Additionally, this project structure raises many concerns regarding modularization of CSS and media assets (images, icons, and fonts). For example, you may bump into this issue when it's not clear how to avoid conflicts between CSS selectors with the same names used across multiple components, or how to avoid problems with CSS inheritance.

Here is an example of conflicting CSS selectors:

```
// Header.scss
.name {
  color: black;
}
```

```
// Logo.scss
.name {
  color: gray;
}
```

In this chapter, we will cover:

- Inline styles in React components
- How to write modular CSS with CSS modules, PostCSS, and Webpack
- How to configure the Webpack module bundler for working with CSS, images, and other resources referenced inside React components

Inline styles in React components

First, one solution for the problem of classes clashing is using inline styles, that is, setting styles with an element's `style` property instead of a separate CSS file, given the following HTML and CSS snippet:

```
// CSS style sheet
.root {
  color: white;
  background-image: url(bg.png);
  -webkit-transition: all .5s;
  transition: all .5s;
}

// HTML fragment
<div class="root">My Component</div>
```

We could rewrite it with React and inline styles as follows:

```
const style = {
  root: {
    color: 'white',
    WebkitTransition: 'all .5s',
    transition: 'all .5s'
  }
};

functionMyComponent({ imageUrl }) {
  style.root.backgroundImage = `url(${imageUrl})`;
  return<div style={style.root}>My Component</div>;
}
```

You just specify styles with an object whose key is the camel case version of the style name, and whose value is the style's value, usually a string. The camel case is used to make it consistent with other JavaScript code, making it easier to type. Vendor prefixes other than `ms` should begin with a capital letter (in our case, `WebkitTransition` instead of webkit-transition).

The component from the previous example will be rendered to the following HTML string:

```
<div style="color:white;background-image:url(bg.png);-webkit-transition:all
.5s;transition:all .5s">My Component</div>
```

At first glance, this approach a look like a bad design, but keep in mind that you don't write these inline styles in HTML but in JavaScript / JSX, which is an essential difference. In addition to solving the CSS selector name conflicts problem mentioned previously, this approach also solves many problems adherent to CSS itself such as:

- Everything is global. Selectors are matched against everything in the DOM.
- CSS grows over time. Developers are afraid of their own CSS. Deleting obsolete CSS rules is hard, as you don't know if it's absolutely safe to do that.
- Non-deterministic resolution. When the styling differs, depending on the order in which CSS files are asynchronously injected into the page.

On the other hand, this approach may not play well with isomorphic apps, because HTML generated using inline styles might be way too verbose. Also, the inline styles approach has its own limitations. For example, simple states such as `:hover` / `:focus` / `:active` and media queries are much easier to implement in CSS.

We are not going to use inline styles as a replacement to the traditional CSS, but if you are curious to learn more about it, watch this presentation by Christopher Chedeau (Facebook):
`http://blog.vjeux.com/2014/javascript/react-css-in-js-nationjs.html`.

If you're still wanting to give it a try, you may want to look into the Free Style library, that allows the use of inline style objects, but enables generating regular CSS from them. Moreover, it allows processing such styles with `autoprefixer`.
`https://github.com/blakeembrey/free-style`
`https://github.com/blakeembrey/react-free-style`.

Getting started with CSS modules

Another way to modularize your CSS is using CSS modules. This allows styling your React components with regular CSS (or Sass, Less, Stylus), without the worry about issues related to conflicting class selectors and it allows you reduce global styles to a minimum and even eliminate them completely.

With CSS modules, everything you write in CSS files is local by default; each file is compiled separately, so you can use simple class selectors with generic names--no need to worry about polluting the global scope.

Let's say we're building a simple button component, which may have different states (default, disabled, in-progress, and so on). Before CSS modules, we would need to use the BEM naming convention, to make sure that our CSS classes don't cause problems with other UI elements:

components/Button/Button.scss

```
.Button { /* all styles for default state */ }
.Button-icon { /* styles for a child element */ }
.Button--disabled { /* overrides for disabled state */ }
.Button--inProgress{ /* overrides for in-progress state */ }
```

components/Button/Button.js

```
import React from 'react';

function Button({ disabled, inProgress, children }) {
  letclassName = 'Button';
  if (inProgress) {
    className += ' Button--inProgress';
  } else if (disabled) {
    className += ' Button--disabled';
  }
  return (
    <button className={className}>
      <iclassName="Button-icon" />
      {children}
    </button>
  );
}

exportdefault Button;
```

Props validation logic is omitted to keep the code samples short.

Now if you render `<Button disabled={true}>Save</Button>`, it will output:

```
<button class="Button Button--disabled"><i class="Button-
icon"></i>Save</button>
```

An integral part of this convention is that it should also enforce unique names for all application's components. Overall, this allows writing modular CSS code but requires more effort from developers to stick to this naming convention discipline.

With CSS modules, we no longer need to worry about prefixing our CSS class names and making sure that each new component's name is unique. Therefore, we can rewrite the component earlier as follows:

`components/Button/Button.scss`

```
.common { /* all styles for default state */ }
.icon { /* styles for a child element */ }
.disabled { /* overrides for disabled state */ }
.inProgress{ /* overrides for in-progress state */ }
```

`components/Button/Button.js`

```
import React from 'react';
import s from './Button.scss';

function Button({ disabled, inProgress, children }) {
  letclassName = s.common;
  if (inProgress) {
    className += ' ' + s.inProgress;
  } else if (disabled) {
    className += ' ' + s.disabled;
  }
  return (
    <button className={className}>
      <iclassName={s.icon} />
      {children}
    </button>
  );
}

export default Button;
```

Now if you render `<Button disabled={true}>Save</Button>`, it will output:

```
// in debug mode
<button class="Button_common_7Uc Button_disabled_T2i"><i
class="Button_icon_mYi"></i>Save</button>

// in release (aka production) mode
<button class="7Uc T2i"><i class="mYi"></i>Save</button>
```

The value of the `styles` variable inside this component will be equal to:

```
{
root: 'Button_common_7Uc',
  icon: 'Button_icon_mYi',
  disabled: 'Button_disabled_T2i'
}
```

CSS modules generate unique names for all the class selectors used in the code. You can customize the pattern that used to generate these class names. As an example, for development you may want to use a pattern such as `[name]_[local]_[hash:base64:3]`, and for production `[hash:base64:3]`. Further in the chapter, you will learn how to configure CSS modules.

Another interesting feature of CSS modules is its ability to compose CSS class selectors. Let's take this regular CSS code snippet as an example:

```
// CSS style sheet
.common {
  border: 1px solid black;
  border-radius: 3px;
}
.default {
  color: blue;
}
.alternate {
  color: green;
}

// JSX fragment
<div className={s.common + ' ' + s.default}>Default</div>
<div className={s.common + ' ' + s.alternate}>Alternate</div>
```

Using CSS modules composition, it can be rewritten as follows:

```
// CSS style sheet
.common {
  border: 1px solid black;
  border-radius: 3px;
}
.default {
  composes: common;
  color: blue;
}
.alternate {
  composes: common;
  color: green;
}

// JSX fragment
<div className={s.default}>Default</div>
<div className={s.alternate}>Alternate</div>
```

This will render the same HTML, CSS, as the earlier example:

```
<div class="common default">Default</div>
<div class="common alternate">Alternate</div>
```

In a similar way, you can compose CSS classes from external files. This is what it looks like:

```
// components/colors.scss
.primary {
  color: red;
}
.secondary {
  color: blue;
}

// components/Button/Button.scss
.common { /* common styles */ }
.default {
  composes: common;
  composes: primary from '../colors.scss';
}

// components/Button/Button.js
...
<button className={s.default}>Save</button>
...
```

This will render:

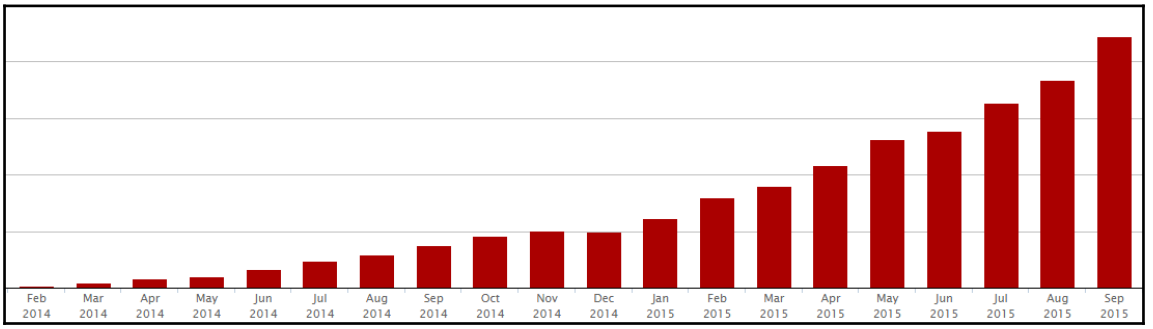
```
// CSS
.colors_primary_Fk7 { color: red; }
.colors_secondary_wUo{ color: blue; }
.Button_common_Ctk{ /* common styles */ }
.Button_default_twD { }

// HTML
<button class="colors_primary_Fk7
Button_common_CtkButton_default_twD">Save</button>
```

With CSS modules, you and your team can leverage your current knowledge of CSS, but become vastly more comfortable and more productive.

Getting started with PostCSS

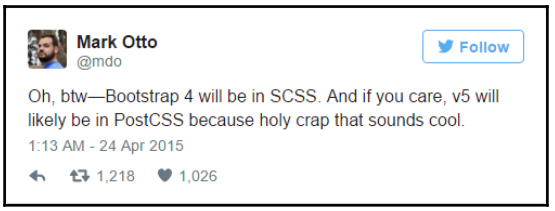
PostCSS is *de facto* becoming a standard for processing source CSS files. It has been growing in popularity at breakneck speed. More and more people are beginning to understand what it offers and how they can take advantage of it. By the way, CSS modules is powered by PostCSS under the hood.

Downloads of PostCSS per month:



Even if you're new to PostCSS, most likely you have been already using it without knowing it, assuming you're using `autoprefixer` in your projects, which itself is just one of the many plugins to PostCSS.

Interestingly, Mark Otto, the author of Bootstrap CSS mentioned on Twitter:

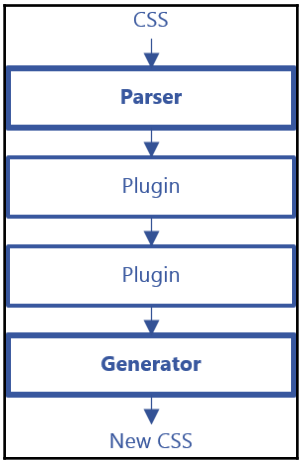


So, what is PostCSS? On the project's home page, it's described as:

> *PostCSS is a tool for transforming styles with JS plugins. These plugins can support variables and mixins, transpile future CSS syntax, inline images, and more.*

The tool itself is a JavaScript module that parses CSS into an **abstract syntax tree** (**AST**); passes that AST through any number of plugin functions; and then compiles that AST back into a CSS string, which you can output to a file.

The principle of operation:



It's important to understand that PostCSS is not just yet another CSS transpiler similar to Sass and LESS. It's intended to solve more interesting problems, such as injecting browser specific prefixes and polyfills, lint CSS code, extend CSS syntax, and quickly implement new ideas, such as CSS modules.

You can play with PostCSS straight from a browser on `http://codepen.io`.

PostCSS doesn't transform CSS itself. If you pass a CSS file through it, it will output exactly the same CSS string even if you haven't configured any plugins for it, that's where all the magic happens.

Let's walk through the notable plugins for PostCSS. One of them is `postcss-use`. It allows explicitly stating what other plugins must be used within a CSS file.

Here is an example:

```
@use postcss-center;

.root {
  top: center;
}
```

If you try to write this `top: center` property in another file, PostCSS will throw an exception unless the `postcss-center` plugin is registered globally.

As a rule of thumb, for a typical web application project, where you work with other developers, you want to register most of the plugins using the `@use` statement and have only a few globally registered plugins such as `autoprefixer`, `postcss-import`, and so on.

Another interesting plugin is `postcss-autoreset`, which helps with automatic conditional rule resets, making sure that your component doesn't accidentally inherit any styles from the parent components. It can be used in combination with `postcss-initial`.

Here is what it looks like:

```
.root {
  all: initial;
  margin: 1em;
}
```

The next plugin in our list is `postcss-assets`. It allows getting image sizes and inlining files as demonstrated here:

```
.icon {
  background: inline(icon.png);
  width: width(icon.png);
  height: height(icon.png);
}
```

`postcss-property-lookup` plugin allows referencing properties like this:

```
.icon {
  width: 32px;
  Height: @width;
}
```

`postcss-contrast` plugin allows changing text color depending on background color contrast:

```
.nav {
  background: #dd3735;
  color: contrast(@background);
}
```

Most likely, you already know `autoprefixer`, which automatically adds vendor prefixes to CSS rules using values from `caniuse.com`:

```
// before// after
:fullscren { }          :-webkit-full-screen { }
                        :-moz-full-screen { }
                        :-ms-fullscreen { }
:fullscreen { }
```

The `postcss-font-magitian` plugin will make you forget about writing `@font-face` rules by hand. Moreover, it has a big database of existing fonts (for example, Google Fonts). You can just write `font-family: Roboto` in your code and this plugin will inject an appropriate `@font-face` ruleset for it.

`stylelint` plugin for PostCSS allows ensuring that the CSS code follows a predefined style guide.

If you use CSS flex box layout in your code, you may take advantage of the `postcss-flexbugs-fixes` plugin, which injects hacks to improve compatibility with different browsers. For example, `flex: 1` becomes `flex: 1 1 0%` to fix a bug in IE11.

The last plugin we want to mention is `precss`, which allows using variables, conditionals, and other goodies from Sass in your CSS code, making it possible to remove the dependency on Sass compiler in your project. This is just a collection of a bunch of other plugins, but as a rule of thumb, you always want to use an existing collection instead of maintaining your own list of PostCSS plugins.

Getting started with Webpack loaders

There is not yet an agreed convention on how JavaScript modules should reference or embed resource files. For example, if you want to reference a JSON file in your JavaScript code you would normally write something like this:

```
import fs from 'fs';
const text = fs.readFileSync(__dirname + './data.json', 'utf8');
const data = JSON.parase(text);
```

There are two issues with this code. One is that the `fs` module is not isomorphic; you won't be able to run this code in a browser. Another issue is that this code is quite verbose.

Ideally, you want to write something like this instead:

```
import data from './data.json';
```

Webpack solves this problem through loaders. A loader is just a Node.js compatible JavaScript function, which accepts a source string (or object), manipulates the originally loaded content, and returns a JavaScript string or some arbitrary `content` object. It may also have side effects, for example, manipulating files on disk. Loaders can be chained; the last loader should return a JavaScript string.

This is how a basic `loader` function for referencing (loading) JSON files may look like:

```
function(source) {
  var data = JSON.parse(source);
  return 'module.exports = ' + JSON.stringify(data) + ';';
}
```

Now if you register this `loader` function in the Webpack's configuration file, you will be able to reference `.json` files in your code as if they were normal JavaScript.

The best part is that in most cases you don't even need to write any custom loaders, but instead you can choose from a wide range of existing loaders published on `https://www.npmjs.com/`.

If you open the `tools/webpack.config.js` file (Webpack configuration) from the previous chapter, you will see that it already contains one `loader`, which looks like this:

```
{
  module: {
    rules: [
      test: /\.js$/,
      include: [ ... ],
      loader: 'babel-loader'
    ]
  }
}
```

This instructs Webpack to check all the referenced (required) files, and if any of them end with `.js` and are located inside the specified folders (see `include` and `loader` property), Webpack will pass the contents of these files to the `babel-loader` npm module. This module exports a function similar to one demonstrated previously.

A loader may accept parameters in a form of a query string or a `query` property:

```
{
  test: /\.js$/,
  loader: 'babel-loader?presets[]=es2015&presets[]=react'
}
```

Alternatively:

```
{
  test: /\.js$/,
  loader: 'babel-loader',
  query: { presets: [ 'es2015', 'react' ] }
}
```

Sometimes, you need to register more than one loader for the same file type. For example, you can instruct Webpack to pass all the referenced `.md` files first through the `markdown-loader`, then through the `html-loader`. The later one will convert the markdown text string to the HTML, and the first one will convert HTML string into JavaScript code, at the same time replacing referenced local resources to `require(...)` statements. This way you can easily embed the contents markdown files into the application bundle (as an alternative to requesting these files at runtime with Ajax).

The order in which you specify loaders in Webpack config matters. This is how this configuration will look for markdown files:

```
{
  test: /\.md$/,
  include: [path.join(__dirname, '../content')],
  loader: 'html-loader!markdown-loader'
}
```

Alternatively:

```
{
  test: /\.md$/,
  include: [path.join(__dirname, '../content')],
  loaders: ['html-loader', 'markdown-loader']
}
```

You can also reference loaders by their shorthand names. The preceding example could be rewritten either as `loader: 'html!markdown'` or `loaders: ['html', 'markdown']`. The loader name convention and precedence search order are defined by `resolveLoader.moduleTemplates` within the Webpack configuration API.

Now, you can try to create a markdown file (`content/hello.md`) with the following contents:

```
# Welcome to Markdown!
[hello](./hello.png)
```

You should be able to reference it in your JavaScript source code as follows:

```
import html from './content/hello.md';
```

or

```
const html = require('./content/hello.md');
```

Webpack will replace that line of code with something equivalent to this before bundling the rest of the code:

```
const html = '<h2>Welcome to Markdown!</h2><p><imgsrc="' +
require('./content/hello.png') + '" alt="hello" /></p>';
```

You will also need to register `file-loader` for `.png` files in order to make this code sample work. `file-loader` will copy `hello.png` file into the output (build) folder during bundling and replacing `require('./content/hello.png')` with a correct URL string.

To learn more about using loaders visit:
`https://webpack.js.org/concepts/loaders/`

You might also want to check the documentation for each of the loaders mentioned so far:
`https://github.com/babel/babel-loader`
`https://github.com/webpack/json-loader`
`https://github.com/peerigon/markdown-loader`
`https://github.com/webpack/html-loader`
`https://github.com/webpack/file-loader`.

Configuring Webpack for images and CSS

The problem we're trying to solve by being able to write modular CSS is having all the CSS, images, and other resource files split across many React `components` folders. When module bundler creates the application bundle (`build/public/client.js` in our case), it should be able to resolve all the CSS files, images, and other assets used in the app and either inject them into the application bundle itself or copy to the output (`build/public`) folder during compilation.

In order to solve this problem, let's install the following npm modules:

- `postcss`: Pre-processor for CSS files
- `postcss-import`: PostCSS plugin that allows referencing other CSS files in your code, for example, `@import '../variables.scss'`
- `postcss-loader`: PostCSS loader for Webpack
- `precss`: PostCSS plugin that enables Sass syntax in CSS files
- `autoprefixer`: PostCSS plugin that appends vendor-specific CSS rules
- `css-loader`: Webpack loader that parses CSS and enables CSS modules
- `style-loader`: Webpack loader that allows injecting CSS into DOM at run-time
- `file-loader`: Webpack loader that allows referencing arbitrary files (images and so on) in your JavaScript code and copying them to the output
- `url-loader`: Webpack loader that allows embeding files into the bundle as a `base64` string

Execute the following command to install the required dependencies:

```
npm install postcsspostcss-import postcss-loader precss --save-dev
npm install autoprefixer --save-dev
npm install css-loader style-loader file-loader url-loader --save-dev
```

Now let's open the project from `Chapter 2`, *Creating a Web UI with React,* and edit the`tools/webpack.config.js` file to add instructions on how Webpack should handle CSS files and images.

Add loaders for `.scss` and images files, plus configuration settings for PostCSS loader:

```
const common = {
  /* other configuration settings, removed for brevity */,
  module: {
    loaders: [
      /* other Webpack loaders, removed for brevity */,
      {
        test: /\.(png|jpg|jpeg|gif|svg|woff|woff2)$/,
        loader: 'url-loader?limit=10000'
      },
      {
        test: /\.(eot|ttf|wav|mp3)$/,
        loader: 'file-loader'
      },
      {
        test: /\.scss$/,
        include: [
          path.join(__dirname, '../components')
        ],
        loader: 'style-loader!css-loader?modules&' +
          'localIdentName=[name]_[local]_[hash:base64:3]!' +
          'postcss-loader'
      }
    ]
  },
  postcss: function plugins(bundler) {
    return [
      require('postcss-import')({ addDependencyTo: bundler }),
      require('precss')(),
      require('autoprefixer')(),
    ];
  }
};
```

As you can see, `url-loader` is used to inject small images (under ~10 KB) and commonly used fonts (`woff` and `woff2`) into the application bundle. Other images, fonts, and audio files will be processed by `file-loader`.

The settings for PostCSS loader are provided via `postcss` property, which is just a list of PostCSS plugins. The order of the plugins matters. With the earlier settings, our `.scss` file will be processed by the `postcss-import` plugin first, then by the `precss` plugin, and finally by the `autoprefixer` plugin with default settings.

One caveat you may bump into is that `style-loader` should only be used in the client-side bundle configuration, as it makes no sense injecting CSS into DOM on the server. The Node.js app will go through an exception during server-side rendering if you attempt to run the code containing CSS processed by `style-loader`. Another thing is that you need to use `css-loader/locals` for the server-side bundle configuration.

You may want to check the `css-loader` documentation page located at: `https://github.com/webpack/css-loader`.

To remove `style-loader` from the server-side bundle configuration as well as replace `css-loader` with `css-loader/locals`, you can add the following code snippet just before `export default [client, server]` in `tools/webpack.config.js` file:

```
server.module.loaders
  .filter(x =>x.loader.startsWith('style-loader!css-loader'))
  .forEach(x => {
    x.loader = 'css-loader/locals' + x.loader.substr(23)
  });
```

Now, let's create `components/Layout/Layout.scss` file with the following contents, just to make sure that everything works as expected:

```
.root {
  color: red;
}
```

Reference this CSS file inside `components/Layout/Layout.js` file, as shown here:

```
import React from 'react';
import s from './Layout.scss';
import Header from '../Header';
import PropTypes from 'prop-types';
function Layout(/* ... */) {
  return (
```

```
      <div className={s.root}>
        {* ... *}
      </div>
    );
}

Layout.propTypes = {/* ... */};

export default Layout;
```

Some code is omitted in order to keep the samples short. You can find the full version of this layout component in the example source code accompanying this book. See the `chapter-03/components/Layout` folder.

ESLint, our javascriptlinting tool, will complain about the `s` variable name being too short. You can add this variable name to exceptions by appending the following rule to the ESLint configuration file (`.eslintrc`): `"id-length": [2, {"exceptions": ["x", "s"]}]`.

Also let's remove the `async` attribute from the `<script src="client.js" async />` tag, which you can find in the `components/Html/Html.js` file. This will eliminate the flash of the unstyled content (FOUC) issue during the initial page load. We will put it back in later chapters after you learn how to extract CSS from React applications for server-side rendering.

Now, when you build the project (by either running `npm run build` or `npm run serve`), Webpack will load the contents of the `Layout.scss` file, process it first by `postcss-loader`, then by `css-loader`, and inject the resulting CSS string into the client-side bundle (`build/public/client.js`). When you open the site in a browser, the code injected into the client-side bundle by `style-loader` will insert CSS styles referenced inside the `Layout` component into the DOM at runtime.

At this point, if you view the source code of the home page, it should look similar to this:

```
<html>
<head>
<meta charset="utf-8">
<meta http-equiv="x-ua-compatible" content="ie=edge">
<title>My Application</title>
<meta name="description" content="...">
<meta name="viewport" content="...">
<script src="client.js"></script>
<style type="text/css">
.Layout_root_1rN {
color: red;
```

```
}
</style>
</head>
<body>
<div id="app">
<div class="Layout_root_1rN" data-reactid="..." data-react-checksum="...">
<!-- ... -->
</div>
</div>
</body>
</html>
```

Let's see how Webpack handles images referenced in CSS. Copy some image file to the `components/Layout` folder and reference it in the `Layout.scss` file as demonstrated here:

```
.root {
  background: url(./bg.jpg) center / cover;
}
```

Now when you build the project, you see that Webpack creates one more file inside the output folder, for example, `build/public/f17853017415f4def9d23508feff521a.jpg`. This is done by `file-loader`, which by default calculates the MD5 hash of the target image file, copies it to the output folder with a new name, and updates the URL string of that file in CSS so that the preceding code will look like this in the final CSS (assuming that `bg.jpg` images is larger than 10 KB):

```
.root {
  background: url(/f17853017415f4def9d23508feff521a.jpg) center / cover;
}
```

If the referenced image is smaller than 10 KB, it will be embedded right into the output CSS code as a `base64` string; this is done by `url-loader`.

As you may guess, the file names generated by `file-loader` are easily customizable via the `name` query parameter. For example, you may force `file-loader` to use the original names for copied files by updating a section of `loader` in `webpack.config.js` as follows:

```
{
  test: /\.(png|jpg|jpeg|gif|svg|woff|woff2)$/,
  loader: 'url-loader?limit=10000&name=[path][name].[ext]'
},
{
  test: /\.(eot|ttf|wav|mp3)$/,
  loader: 'file-loader&name=[path][name].[ext]'
}
```

To learn more about configuring `file-loader` and `url-loader` visit:
`https://github.com/webpack/file-loader`
`https://github.com/webpack/url-loader`.

Sharing common settings across multiple CSS files

The `precss` plugin mentioned previously allows you to use Sass variables and mixins in your CSS code. While this may sound like a great feature, in practice, you may want to keep the usage of variables and (especially) mixins to the very minimum, in favor of a better composition of React components. This will keep your code more maintainable. There are some legit cases though where you need to have shared variables. For example, you may want to have the `$primary-color` variable containing the primary base color used by many UI elements in your app.

Let's create the `components/variables.scss` file for such variables. It may look similar to this:

```
/* Colors */
$color-primary: #0275d8;
$color-success: #5cb85c;
$color-info:    #5bc0de;
$color-warning: #f0ad4e;
$color-danger:  #d9534f;

/* Shadows */
$shadow-2dp: 0 2px 2px 0 rgba(0, 0, 0, .14),
             0 3px 1px -2px rgba(0, 0, 0, .2),
             0 1px 5px 0 rgba(0, 0, 0, .12);

/* Animations */
$animation-duration-default: .2s;
$animation-curve-default: cubic-bezier(.4, 0, .2, 1);
```

Here is an example of how you can use these shared variables from components' CSS:

```
@import '../variables.scss';

.root {
  box-shadow: $shadow-2dp;
  transition-timing-function: $animation-curve-default;
  transition-duration: $animation-duration-default;
}
```

Using the `.scss` file extension, instead of `.css`, we ensure that this code is properly highlighted in most text editors and IDEs.

You may want to learn more about `precss` plugin by visiting its homepage located at `https://github.com/jonathantneal/precss`. It contains some code samples and the list of PostCSS plugins behind it. As an alternative to `precss` you may also want to check out `postcss-cssnext` plugin.

You may ask, why not use the original Sass compiler? It might work less efficiently, because this way you will end up parsing CSS twice, first by Sass and then by PostCSS. This can make a difference if you run the bundler in a watch mode so that whenever you make changes to the source files, you expect the browser to reload the page (or portion of the page) as fast as possible, preferably under a second. See the next chapter on how to enable this functionality.

Another reason for not using the fully featured Sass compiler is that it will force you to keep CSS simple; you don't want to overload it with dynamic stuff, calculations, and so on. That sort of thing is much better to implement with JavaScript. The most things you may ever need in CSS (such as variables, partial imports, and so on) should be handled just fine by PostCSS and a few plugins such as `precss` and `autoprefixer`.

Summary

In this chapter, you learned how to use inline styles inside React components. You can use them occasionally in places where it's easier to add some dynamic functionality this way. For the core styles of our app, we have chosen the CSS modules approach powered by Webpack and PostCSS. It allows writing modular CSS more easily than a naming convention-based approach (BEM, SMACSS, and so on). Keep in mind that the CSS module doesn't solve all the pitfalls when it comes to writing maintainable CSS code, you may want to stick to some other ideas from BEM, such as using class selectors in favor of element selectors, avoid nested selectors, and so on. You also learned about PostCSS, and how you can instantly benefit from using it. We covered some notable plugins for PostCSS that can make your job as a frontend engineer easier. You learned how to use Webpack loaders and how to configure everything in order to be able to write modular and maintainable CSS.

4

Working with Browsersync and Hot Module Replacement

Frontend development usually involves a considerable degree of trial and error. A naive web development workflow may look like this:

1. Open your web app in a browser.
2. Write or edit a few lines of code.
3. Hit the browser's refresh button (or F5) and return to step 2.

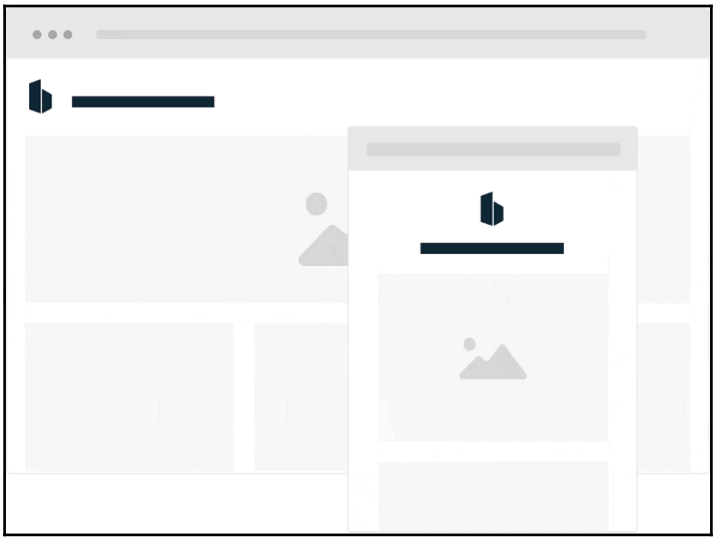
This approach just won't scale, especially if you want to test your web app in multiple browsers and on different devices. Instead, you need to have the ability to instantly see the result of your work--how the web app looks in a browser and on mobile devices. If you don't test early, you may end up spending more time on fixing bugs later on.

In this chapter, you will learn how to improve the development workflow by configuring and using tools such as Browsersync and Webpack's Hot Module Replacement.

Getting started with Browsersync

Browsersync is a free Node.js-based tool that acts as an intermediary between web application server and a browser. It can look for changes you make to a web page and automatically update the browser (similar to **LiveReload**). In addition to that, it can work across multiple devices, including physical ones, synchronizing user interactions such as scrolls and click on events.

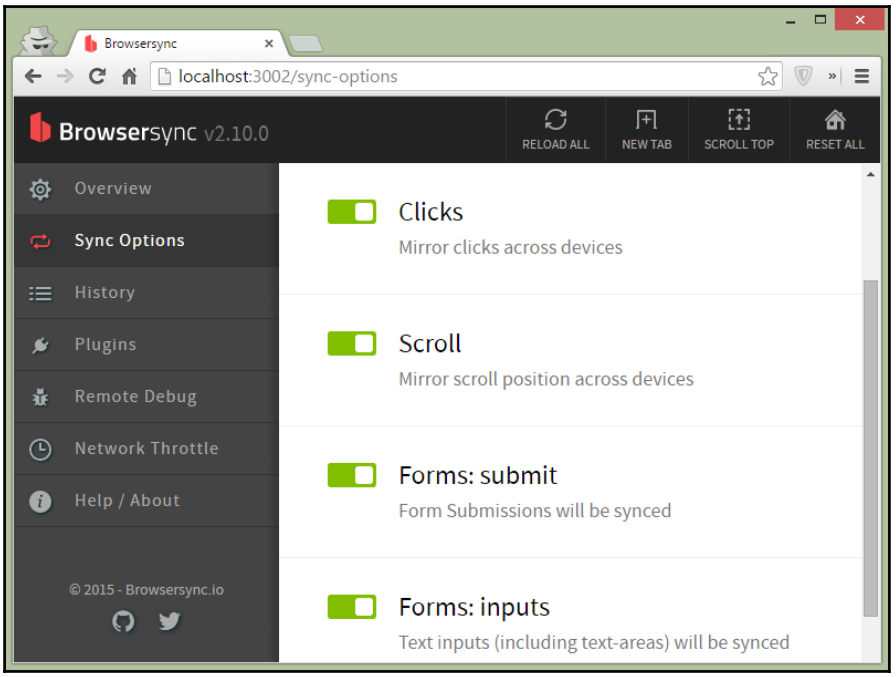
For example, you can launch a regular browser and iOS emulator; open your web app on both desktop and mobile browsers:



Then while you interact with the site in the desktop browser, all the user actions are going to be mirrored across these two browsers. When you modify an input field in your desktop browser, the same string will appear on the iOS device. Goodbye, on-screen keyboards!

Browsersync allows customizing which user actions should be mirrored and which should not. You can tweak that from either a configuration file or a nice UI that Browsersync provides.

This is what a Browsersync admin panel looks like:



As you navigate through the site on any device, the paths you visit are tracked by Browsersync. You can view this list and push one of the previously visited URLs to all devices in one click.

Another interesting feature of Browsersync is that it enables remote debugging on connected devices, somewhat similar to the Inspect Element mode in Chrome Developer Tools but not limited to a single browser's tab. Also, similar to Chrome Developer Tools, Browsersync allows you to outline all the HTML elements on the page on all connected devices, add shadows to the elements or CSS grid overlay. This is extremely helpful for debugging the alignment of elements and fixing any layout bugs you may have in your HTML/CSS.

If you want to test your web app on a slow internet connection, you can just enable network throttling in Browsersync settings, and this will do the trick. It is much easier than other options, especially when you test your app on different physical devices (iPhone/iPad, Android, Windows Phone, and so on)

How to install

It might make sense to install Browsersync as a local npm dependency in your project (it is similar to how you install Babel or Webpack). This way, it will be easier for other developers in your team to use this tool and make sure that everyone on the team is using the up-to-date version of it.

Open a terminal window and run:

```
npm install browser-sync --save-dev
```

If you're running **macOS** and encounter problems when trying to install Browsersync, it's almost always because you have problems with npm permissions. Check their documentation on how to resolve this issue: `https://docs.npmjs.com/getting-started/fixing-npm-permissions`. On Windows, installation issues may be related to `node-gyp` and its dependencies such as Visual C++ runtime libraries. You can find more information on how to resolve these issues on the `node-gyp` project's homepage: `https://github.com/nodejs/node-gyp`.

Now, let's see what the script launching Browsersync may look like. You could also run it directly from a command line, but sense that eventually we'll have more than just a couple of parameters that we need to pass into it; having a separate file is more convenient. In our example, Browsersync will work as a proxy to the Node.js/Express app (`server.js`).

In its simplest form, Browsersync can be launched via JavaScript API as follows:

```
const bs = require('browser-sync').create();
bs.init({ proxy: 'localhost:3000' });
```

It assumes that there is an HTTP server running locally at port `3000`. If you execute this script, it will print something like this to the console:

```
[BS] Proxying: http://localhost:3000
[BS] Access URLs:
 --------------------------------------
       Local: http://localhost:3001
    External: http://169.254.80.80:3001
 --------------------------------------
          UI: http://localhost:3002
 UI External: http://169.254.80.80:3002
 --------------------------------------
```

With default settings, it will also open a browser window pointing to `http://localhost:3001` in our case (it uses the nearest free IP port starting from the default one-port `3000`). That's the URL that you will use for debugging the app in a browser. There is also the Browsersync control panel available at `http://localhost:3002`, where you can enable or disable remote debugger and things like that. The other two URLs are intended to be used when you need to access your app from an external network. Pretty neat, huh?

There are about 40 configuration options that you can use to customize Brwosersync for your needs. For example, to prevent opening a new browser window when Browsersync starts you can add an `open` option as follows:

```
const bs = require('browser-sync').create();
bs.init({ proxy: 'localhost:3000', open: false });
```

If you want to easily share your web app hosted on your local development environment with internet users, you can add a tunnel option as follows:

```
const bs = require('browser-sync').create();
bs.init({ proxy: 'localhost:3000', tunnel: 'my-app' });
```

This way, your web app will be available at `http://my-app.localtunnel.me`.

For the full list of Browsersync options visit `http://www.browsersync.io/docs/options/`.

You can also plug an `Express.js` such as middleware (filters where the requests go through before it gets you your request handling logic) to the Browsersync server as follows:

```
const bs = require('browser-sync').create();
bs.init({
  proxy: {
    target: 'localhost:3000',
middleware: (req, res, next) => {
console.log(req.url);
next();
}
  }
});
```

This way, we can plug in `webpack-dev-middleware` and `webpack-hot-middleware` helping to improve a debugging experience. The first one takes a Webpack compiler targeting `web` platform, tweaks a compiler's `outputFileSystem` option to use an in-memory file stream, starts the compiler in a watch mode, and begins listening for incoming HTTP requests. If the requested URL matches any of the bundles created by the compiler in memory, it outputs the corresponding bundle to a client, otherwise, it does nothing and lets the Node.js server (Browsersync in our case) handle the request.

Here is what a simplified version of `webpack-dev-middleware` looks like (so, you can get a better idea of how it works internally):

```
import MemoryFileStream from 'memory-fs';

export default function(compiler, options) {
  const fs = compiler.outputFileStream = new MemoryFileStream();

  // start the compiler in a watch mode
  compiler.watch(...);

  // Express.js/Connect compatible middleware function
  return async (req, res, next) => {
    // if compilation is in progress, wait until it completes
    // ...
    const filename = getFileNameFromUrl(req.url);
    const stat = await fs.stat(filename);
    if (!stat.isFile()) {
      return next();
    }
    const content = await fs.readFile(filename);
    // set HTTP headers
    // ...
    res.end(content);
  };
}
```

Here is the usage:

```
import webpack from 'webpack';
import webpackDevMiddleware from 'webpack-dev-middleware';
import webpackConfig from './webpack.config';

const bundler = webpack(webpackConfig);
const middleware = webpackDevMiddleware(bundler, {/* options */});
```

For the list of available options, refer to `webpack-dev-middleware` documentation located at `https://github.com/webpack/webpack-dev-middleware`.

Along with `webpack-hot-middleware` and a couple of Webpack configuration tweaks, this enables the **Hot Module Replacement** (**HMR**) functionality that will automatically inject updated **CommonJS** modules into the running JavaScript application. Ultimately, you will be able to edit the source code of your app and instantly see the changes you make to the UI in a browser window without refreshing the entire page. You will learn more about HMR later in this chapter.

Unfortunately, the current version of `webpack-dev-middleware` doesn't work well with multi-bundle Webpack compilers, containing bundles targeting both tThe Web and Node.js platforms. One way of solving this issue is to fork `webpack/webpack-dev-middleware` repo on GitHub, patch the middleware within the forked repo, and install the patched version of this middleware.

You can install an already patched version of this module by running:

```
npm install koistya/Webpack-dev-middleware --save-dev
```

This command will install the `webpack-dev-middleware` module from the `https://github.com/koistya/webpack-dev-middleware` repository.

Patching an existing library this way may sound extreme, but unfortunately, not all modules published on npm are truly isomorphic. Sometimes, you will need to come up with a workaround similar to this one. More on that later.

Let's install the second middleware required for enabling HMR:

```
npm install webpack-hot-middlware --save-dev
```

Now, we are ready to configure Hot Module Replacement for our app. Let's move the code launching Node.js serve into a separate file called `tools/runServer.js`:

```
import path from 'path';

import cp from 'child_process';

// Should match the text string used in server.js
const RUNNING_REGEXP = /Server is listening at http:\/\/(.*?)\//;

let server;
 const webpackConfig = require('./webpack.config);
const { output } = webpackConfig.find(x => x.target === 'node');
const serverPath = path.join(output.path, output.filename);
```

```
function runServer(cb) {

  function onStdOut(data) {
    const match = data.toString('utf8').match(RUNNING_REGEXP);
    process.stdout.write(data);

    if (match) {
      server.stdout.removeListener('data', onStdOut);
      server.stdout.on('data', data => {
        process.stdout.write(data));
      }
      if (cb) {
        cb(null, match[1]); // e.g. cb(null, 'localhost:3000')
      }
    }
  }

  if (server) {
    server.kill('SIGTERM');
  }
  server = cp.spawn('node', [serverPath], {
    env: Object.assign({ NODE_ENV: 'development' }, process.env),
    silent: false
  });

  server.stdout.on('data', onStdOut);
  server.stderr.on('data', data => process.stderr.write(data));
}

export default runServer;
```

This helper function launches the Node.js server (`server.js`) in a child process and notifies the caller when the server is ready.

Now, let's update `tools/webpack.config.js` to include Webpack's `OccurenceOrderPlugin` in the common section of the configuration file:

```
const common = {
  ...
plugins: [
new webpack.optimize.OccurenceOrderPlugin(true)
]
  ...
};
```

This will help with debugging when you will need to check the source code of compiled bundles.

Next, create a `tools/start.js` script, which will run Webpack with HMR, Node.js server, and Browsersync. The barebones of this script looks as follows:

```
async function start() {
  await run(require('./clean'));
  await new Promise(resolve => {
    const bundler = webpack(webpackConfig);
    const middleware = [/* HMR middleware etc. */];
    bundler.compilers
      .find(x => x.options.target === 'node')
      .plugin('done', () => {
        runServer((err, host) => {
          cont bs = require('browser-sync').create();
          bs.init({
            proxy: { target: host, middleware }
          }, resolve);
        });
      });
  };
}
```

This code executes `tools/clean.js` script at first, which cleans up the build output folder (`./build`). Then, it initializes `webpack-dev-middleware` middleware, which instantly starts the Webpack compiler in a watch mode. Then it subscribes to the server-side bundle complete event (`compiler.plugin('done', ...)`), inside which it first starts the Node.js server in a child process; then after the Node.js server is ready to accept a new HTTP request it launches Browsersync as a proxy on top of it.

One more important thing to do is to patch Webpack configuration for client-side bundles by injecting `HotModuleReplacementPlugin`, `NoErrorsPlugin` and `webpack-hot-middleware/client` modules. Also, notice that `webpack-hot-middleware` should only be applied to the compilers with target mode `web` (or, `undefined`).

The complete version of the `tools/start.js` script looks as follows:

```
import path from 'path';
import webpack from 'webpack';
import webpackDevMiddleware from 'webpack-dev-middleware';
import webpackHotMiddleware from 'webpack-hot-middleware';
import { static as staticMiddleware } from 'express';
import run from './run';
import runServer from './runServer';
import webpackConfig from './webpack.config';

async function start() {
  await run(require('./clean'));
```

```
  await new Promise(resolve => {
webpackConfig.filter(x => x.target !== 'node').forEach(x => {
x.entry = [x.entry, 'webpack-hot-middleware/client'];
x.plugins.push(new webpack.HotModuleReplacementPlugin());
x.plugins.push(new webpack.NoErrorsPlugin());
});

    const bundler = webpack(webpackConfig);
const middleware = [
staticMiddleware(path.join(__dirname, '../public')),
webpackDevMiddleware(bundler, {
stats: webpackConfig[0].stats
}),
...(bundler.compilers
.filter(compiler => compiler.options.target !== 'node')
.map(compiler => webpackHotMiddleware(compiler)))
];
let handleServerBundleComplete = () => {
runServer((err, host) => {
if (!err) {
const bs = require('browser-sync').create();
bs.init({
proxy: { target: host, middleware }
}, resolve);
handleServerBundleComplete = () => runServer();
}
});
};

    bundler.compilers
      .find(x => x.options.target === 'node')
      .plugin('done', () => handleServerBundleComplete());
  });
}

export default start;
```

Now, if you add it to `package.json/scripts`, you will be able to run this script via `npm start` command. This is what the modified `package.json` file looks like:

```
{
  ...
  "devDependencies": {
    ...
"browser-sync": "^2.10.0",
"webpack-dev-middleware": "koistya/Webpack-dev-middleware",
"Webpack-hot-middleware": "^2.5.1"
  },
```

```
  "scripts": {
    ...
"start": "babel-node tools/run start"
  }
}
```

Just to verify that everything works as expected if you run `npm start` in your console window, it should print something like this:

```
[02:56:37] Starting 'start'...
[02:56:37] Starting 'clean'...
[02:56:37] Finished 'clean' after 275 ms
webpack built cc73cac49e83ce7175d7 in 3016ms
Hash: cc73cac49e83ce7175d7f8702ccb07d80b742bc2
Version: webpack 1.12.9
Child
    Hash: cc73cac49e83ce7175d7
    Version: webpack 1.12.9
    Time: 3016ms
        Asset    Size  Chunks             Chunk Names
    client.js  882 kB       0  [emitted]  main
Child
    Hash: f8702ccb07d80b742bc2
    Version: webpack 1.12.9
    Time: 2504ms
        Asset     Size  Chunks             Chunk Names
    server.js  16.1 kB       0  [emitted]  main
webpack: bundle is now VALID.
[02:56:42] Node.js server is listening at http://localhost:3000/
[BS] Proxying: http://localhost:3000
[BS] Access URLs:
 ------------------------------------
       Local: http://localhost:3001
    External: http://169.254.80.80:3001
 ------------------------------------
          UI: http://localhost:3002
 UI External: http://169.254.80.80:3002
 ------------------------------------
[02:56:44] Finished 'start' after 7263 ms
```

Additionally, in the browser's console, you should be able to see:

```
[HMR] Connected
```

This means that everything works as expected, and we can proceed to the next topics. If the output of the console window differs from the before, you may want to check the source code in the `chapter-04` folder and compare that to what you have in your project.

Getting started with Hot Module Replacement

HMR is an opt-in feature, meaning that you must explicitly specify what exactly should happen when new versions of modules are becoming available at runtime in the code.

Try to start the app by running `npm start` and edit the `components/Layout/Layout.js` file; you should see the following output in the browser's console window:

```
[HMR] bundle rebuilding
[HMR] bundle rebuilt in 445ms
[HMR] Checking for updates on the server...
[HMR] The following modules couldn't be hot updated:
      (Full reload needed)
[HMR]  - ./components/Layout/Layout.js
```

That's the expected behavior because, at this moment, the code doesn't provide instructions to HMR on what to do with the new `./components/Layout/Layout.js` module. When this module is loaded into the active runtime, HMR first checks whether this module contains a call to the `module.hot.accept(...)` method, and if it does, tries to execute it. If this method is not found, HRM traverses the dependency tree from the `Layout.js` module up to the top-most component. If one of the parent components up the dependency tree is able to plug the newly received module into the runtime (via `module.hot.accept(...)` method), the Hot Module Replacement operation will be successfully completed, otherwise HMR will report that the module could not be reloaded at runtime and a full application reload is needed.

Before we proceed, you may want to check the official documentation on HMR in Webpack documentation at the following URL:
`https://webpack.github.io/docs/hot-module-replacement`.

Now, let's see how it works. Create a new file, for example, `components/Layout/test.js` with the following contents:

```
if (typeof document !== 'undefined') {
  setTimeout(() => {
    document.body.style.backgroundColor = 'red';
  }, 2000);
}
```

Then, reference it inside the `components/Layout/Layout.js` file as follows:

```
import './test';
```

Now when you load the page in a browser, it should change the background color to red in a couple of seconds. If you try to edit the `test.js file`, for example, changing `red` to `blue`, you will receive a warning message in the browser's console:

```
[HMR] The following modules couldn't be hot updated:
[HMR]  - ./components/Layout/test.js
```

In order to fix that, all you have to do is to append the following code snippet at the end of the `test.js` file, which follows HMR convention:

```
if (typeof document !== 'undefined') {
  setTimeout(() => {
    document.body.style.backgroundColor = 'red';
  }, 3000);
}

if (module.hot) {
module.hot.accept();
}
```

It will tell HMR that the current module (`test.js`) can be safely replaced with the new one as soon as it becomes available. The `if (module.hot) { ... }` wrapper statement is needed because `module.hot` (HMR API) might not be available at runtime, for example, in the production build of the app.

Now if you reload the app and try to edit the `test.js` file, for example, by changing `red` to `blue`. You should be able to see how the background of the webpage is changing from red to blue.

Since this module doesn't export anything, we can't just call `module.hot.accept()`, which will instruct HMR to replace the module without the notification of parents.

Does this module have side effects? Yes, the call to the `setTimeout(...)` function creates a side effect which we need to remove before replacing the module with a new one. In order to do so, we need to use the `module.hot.dispose(...)` API method as demonstrated here:

```
let timeout;

if (typeof document !== 'undefined') {
timeout = setTimeout(() => {
    document.body.style.backgroundColor = 'red';
  }, 3000);
}

if (module.hot) {
```

```
  module.hot.accept();
module.hot.dispose(() => clearTimeout(timeout));
}
```

Alternatively, we can place HMR instructions to any of the parent components. Let's see how it works. Replace the contents of the `components/Layout/test.js` file with the following:

```
export default `
  body {
    background-color: 'red';
  }
`;
```

Then, update `components/Layout/Layout.js` to include the following code snippet at the end of the file:

```
let style;

function insertCss(css) {
  let elem = document.createElement('style');
  elem.textContent = css;
  document.head.appendChild(elem);
  return {
    remove: () => {
      document.head.removeChild(elem);
    }
  };
}

if (typeof document !== 'undefined') {
  style = insertCss(require('./test'));
}

if (module.hot) {
  module.hot.accept('./test', () => {
    style = injectCss(require('./test'));
  });
  module.hot.dispose(() => {
    style.remove();
  });
}
```

This works similarly to the previous example when you can edit the test.js file and see changes in the browser window. However, this time, Hot Module Replacement is handled by the parent (`Layout.js`) component.

Summary

By the end of this chapter, you should have learned how to install and configure Browsersync and Hot Module Replacement (HMR) as well as launch the Node.js app for development purposes. The configuration described in this chapter is optimized for isomorphic web application development and should dramatically improve your development workflow. Now, you should be able to edit UI elements faster as you can instantly see the result of your work in a browser and on mobile devices and spend less time on debugging.

In the next chapter, you will learn about many different aspects related to rendering a web application on the server.

5
Rendering React Components on the Server

So far, we've spent lots of time on setting core project structure and configuring development tools. Starting with this chapter, we will dig into the essence of isomorphic web application development, specifically server-side rendering (SSR). By the end of the chapter, you should be comfortable writing components in a way that they can be easily rendered inside a browser and in the Node.js app without any issues. To be more specific, here is the list of topics we're going to cover:

- The core concepts of server-side rendering
- Problems and troubleshooting
- Passing the component's state from server to client
- Working with the React context feature
- Setting the page title and other metadata
- Working with third-party libraries that are not isomorphic
- Fetching data from the server

The core concepts of server-side rendering

As you saw `Chapter 1`, *Getting Started with Isomorphic Web Apps*, the core concept of server-side rendering is very simple. The `React` library provides just two API methods for that:

- `ReactDOMServer.renderToString(ReactElement)`
- `ReactDOMServer.renderToStaticMarkup(ReactElement)`

They both accept a single React element as they input a parameter and return an HTML string. You can use either of these two methods on the server to generate HTML code for your JavaScript app and send the markup down on the initial request for faster page loads and to allow search engines to crawl your web app for SEO purposes.

Most of the time, you will use the `renderToString()` method as it not only generates HTML markup but also appends some metadata to the HTML (in a form of `data-reactid` attributes). This makes it possible to reuse generated markup on the client so that when you bootstrap the same JavaScript app in a browser, React will just verify that the checksum of the generated, on the server markup matches the checksum of the in-memory UI tree (virtual DOM), and if so, it will preserve that HTML and only attach event handlers.

Before we proceed to more advanced examples, let's refresh your memory on what basic rendering looks like.

Here is how a React.js web page can be pre-rendered on the server:

```
import express from 'express';
import React from 'react';
importReactDOM from 'react-dom/server';
import Html from './components/Html';
importTestPage from './components/TestPage';

const server = express();

server.get('/', (req, res) => {
const body = ReactDOM.renderToString(<TestPage />);
const html = ReactDOM.renderToStaticMarkup(
<Html title={TestPage.title} body={body} />
);
res.send('<!doctype html>\n'+ html);
});

server.listen(3000);
```

The following is the client-side code for bootstrapping this same page in a browser:

```
import React from 'react';
importReactDOM from 'react-dom';
importTestPage from './components/TestPage';

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

Note the difference between these two samples. In the client-side code, we reference the `ReactDOM` utility from the `react-dom` package, and in the server-side code, we reference it from the `react-dom/server` package. The latter package is mostly supposed to be used on the server, except maybe for some rare use cases. It also does work in a browser environment, but if you don't reference `react-dom/server` in the client-side code, this package won't be included in the compiled output, keeping your JavaScript app bundle small.

Another difference is that on the server, we need to wrap generated by the `renderToString()` markup into a valid HTML document by appending the `<html>`, `<head>`, `<body>`, and so on, tags. Instead of plugging in Jade, Handlebars, or some other templating engine for that, we can use React and its `renderToStaticMarkup()` method for this task (see `components/Html` in the example source code accompanying this book).

The generated on the server markup should contain the`<script>` tag referencing JavaScript app bundle; in our case `<script src="/app.js"></script>`, which will be loaded from`/build/public/app.js` at runtime. It should also contain an HTML element where the app is going to be mounted on the client, in our case, `<div id="app">...</div>`.

What does mount mean? When you call the `React.render(ReactComponent, DOMElement, callback)` method, first React creates an in-memory representation of the UI component passed as the first argument to this method. Then, it checks if the provided HTML element contains a child element with the `data-reactid` and `data-react-checksum` attributes, validates the checksum, and either uses all the HTML markup inside the container element as it is, or, if checksums do not match, traverses the existing DOM tree and adds/updates/removes HTML elements to make it fully identical to the virtual DOM.

You could mount the top level React component directly into the `<body>` element as follows:

```
ReactDOM.render(<TestPage />, document.body)
```

While this may work okay in your particular case, in general, it's considered a bad practice because third-party libraries and browser extensions may manipulate the `document.body` element directly leading to subtle reconciliation issues.

The location of the `<script>` tag inside the HTML document does matter, often developers put all the `<script>` tags at the bottom of the page right before the closing the `</body>` tag. If you choose to do so, you need to be aware that browsers may start downloading scripts only after HTML content was fully loaded. This may have a negative impact on the performance.

Another approach is to add the `async` attribute to the`<script>` element and put it inside the head section, right before the closing `</head>` tag, for example:

```
<!doctype html>
<html>
<head>
<title>My App</title>
<scriptasyncsrc="/app.js"></script>
</head>
<body>
<div id="app">...</div>
</body>
</html>
```

In this case, browsers will start downloading the app bundle asynchronously, while the rest of the HTML page is being loaded. This is a good thing for performance, but you need to remember that the JavaScript code may start executing before the DOM is ready. So, you need to check that first and if DOM is not ready, subscribe to the `DOMContentLoad` event as the following example demonstrates:

```
import React from 'react';
importReactDOM from 'react-dom';
importTestPage from './components/TestPage';

function run(cleanup = true) {
if (cleanup) {
document.removeEventListener('DOMContentLoaded', run);
window.removeEventListener('load', run);
}
ReactDOM.render(<My Page />, document.getElementById('app'));
}

if (document.readyState !== 'loading') {
run(false);
} else {
document.addEventListener('DOMContentLoaded', run);
window.addEventListener('load', run); // fallback
}
```

Another question you may ask is why not to have `<head>` and `<body>` elements under React's management? Because of the browser limitations, React doesn't play well with manipulating elements in the `<head>` section. For example, if you need to set the page title and description on the client side, it's better to do it with vanilla JavaScript as follows:

```
document.title = 'New Title';
document.querySelector('meta[name=description]')
.setAttribute('content', 'New Description');
```

Troubleshooting

There is a common issue during development when the markup generated on the server differs from what React should render on the client. As an example, consider the following `React` component:

```
import React from 'react';

functionCurrentTime() {
return<p>Current time (timestamp in ms): {Date.now()}</p>;
}

export default CurrentTime;
```

If you put this component on your React web page and reload the page in a browser, you should see a warning in the browser's console similar to this one:

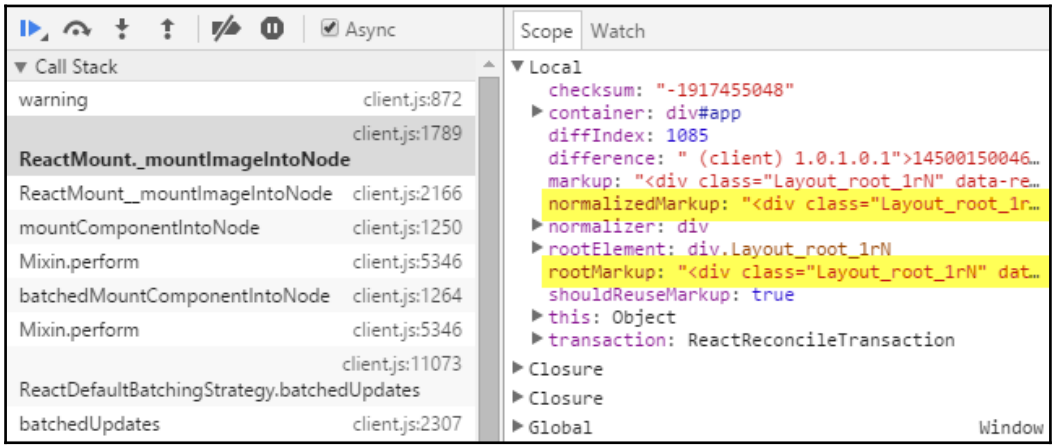
```
    Warning: React attempted to reuse markup in a container but the
checksum was invalid. This generally means that you are using server
rendering and the markup generated on the server was not what the client
was expecting. React injected new markup to compensate which works but you
have lost many of the benefits of server rendering. Instead, figure out why
the markup being generated is different on the client or server:
     (client) 1.0.1.0.1">1450010451418</span></p></div
     (server) 1.0.1.0.1">1450010450804</span></p></div
```

Not that React displays messages similar to this one only in the development environment (when an application bundle containing React is not optimized for production deployment).

It means that React will remove pre-rendered HTML markup during mounting and will rebuild the DOM from scratch. As you might guess, this will impact the performance of your app and how fast it's going to be bootstrap in a browser. As a rule of thumb, you need to spot such warnings early and fix the code to get server-side and client-side markup back in sync.

The error message contains a short code snippet that may give you a hint about what React component caused this problem. Unfortunately, it's quite concise and it's not always obvious to what exact place in the HTML document this snippet is referring to. Near the error (or warning) message in the browser's console, there should be displayed the name of the file and the line number that caused this error or warning message to be printed. You can just click on it, navigate to the corresponding line, put a breakpoint on that line, and reload the page. Now, the browser will stop execution on this exact line, and you can retrieve the full version of the client-side and server-side markups from the **Call Stack** in browser's dev tools.

The screenshot here demonstrates where you can find the full version of the markup in the browser's **Call Stack** (Chrome Dev Tools in this case):



Copy the value of `normalizedMarkup` and `rootMarkup` variables that, respectively, correspond to the client-side and server-side markups. Insert them, let's say, to `https://www.diffchecker.com/`, and it will show you all the places that differ in these two HTML strings. Having this information, you can tweak your React components so that they always render the same output regardless of whether they are used in the browser or server environments.

Passing the component's state from server to client

In the `CurrentTime` example component earlier, the timestamp value should be generated only on the server (during server-side rendering) and somehow passed to the client so that the same value can be reused when the `CurrentTime` component is mounted on the client side.

Try to think for a moment, how would you solve this problem? To make your life easier you can make use of the `ExecutionEnvironment` utility that comes with the `fbjs` package. It can be used like this:

```
import React from 'react';
import{ canUseDOM } from 'fbjs/lib/ExecutionEnvironment';

functionCurrentTime() {
if (canUseDOM) {
// TODO: Use the time generated on the server
} else {
time = Date.now();
}
return<p>Current time (timestamp in ms): {time}</p>;
}

export default CurrentTime;
```

The `fbjs` package is already used by React internally, so it should not increase the size of your application bundle. If you look at what's inside this `ExecutionEnvironment` utility class, you will find how it checks whether the application is running in a browser environment (`canUseDOM` flag) or not:

```
constcanUseDOM = !!(
typeof window !== 'undefined' &&
window.document&&
window.document.createElement
);
```

This might be the easiest way to perform this test in your app and should work well unless you create `window` and `document` global variables on the server.

Do you want to try to complete the earlier code sample to make the `CurrentTime` component isomorphic (meaning that it can operate in both Node.js and browser environments without any issues)? If not, this is what the isomorphic version of this component may look like:

```
import React from 'react';
import { canUseDOM } from 'fbjs/lib/ExecutionEnvironment';

functionCurrentTime() {
constelem = canUseDOM&&
document.querySelector('.time[data-time]');
consttime = elem ? +elem.dataset.time : Date.now();
return (
<pclassName="time" data-time={time}>
      Current time (timestamp in ms): {time}
</p>
  );
}

export default CurrentTime;
```

If the rendering of this component occurs on the server, it will generate a new timestamp value and render inside the `data-time` attribute. On the client, this component will check if the app is running in a browser, which would mean that this component was already pre-rendered on the server and exists in the DOM. It would then grab the value of the `time` (timestamp) variable from the `data-time` attribute and use it for rendering the exact same markup as the one pre-rendered on the server.

This is not the only way of solving a problem with the isomorphic rendering of this component. Depending on the size of state that needs to be passed from server to client, in addition to the `data-` HTML attributes, you can also use `<script>` or `<script type="application/json">` for embedding the application's state into an HTML document, thus passing it to the app running in a browser.

For example, you can pull the list of items from the database on the server, serialize them and inject into the HTML document using the`<script>` tag as follows:

```
<script id="state-items"type="application/json">
  [{ "id": 1, "Item A"}, { "id": 2, "Item B" }]
</scripts>
```

When you serialize a JavaScript object into a JSON string, it might be a good idea to sanitize the result before embedding it in a `<script>` tag:

```
JSON.stringify(obj)
.replace(/<\/script/g, '<\\/script')
.replace(/<!--/g, '<\\!--');
```

Then on the client (in the code that executes in a browser), you can de-serialize that data and pass it down to your React component via props:

```
constelem = document.getElementById('state-items');
const items = JSON.parse(elem.innerHTML);
elem.parentNode.removeChild(elem); // cleanup
const component = <List items={items} />;
```

Working with the React context

When you build an isomorphic app, it may be handy to use React's `context` feature. You can use it to pass context-related data from the top-level React component throughout the component tree down to all the child components that explicitly require that data. This way, you won't need to manually pass the data down to the child components at every level.

You can learn more about this feature in the React documentation: `https://reactjs.com/docs/context`.

Most of the time you are better off using props and/or Flux-like stores to pass data between components and use the React context only when it's hard to pass data in a traditional way. A good use case for that feature would be passing down the currently logged-in user object, the current language, or theme information. If you're building a multi-tenant website, it would be a good idea to pass the client/site information via context.

First, let's see how to access that data from inside React components. Let's assume that the context data was already provided at one of the top-level React components. This data is available via the `this.context` property inside stateful React components, and as the second argument provided to the stateless React components during rendering.

This is what a stateful React component accessing `user` object from the context looks like:

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
classWelcomeMessage extends Component {
staticcontextTypes = {
user: PropTypes.shape({
name: PropTypes.string.isRequired
    })
  };
render() {
const user = this.context.user;
return<p>Welcome, {user ? user.name : 'Guest'}!</p>;
  }
}

export default WelcomeMessage;
```

The following is the stateless functional component counterpart of the earlier component:

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
functionsWelcomeMessage(props, { user }) {
return<p>Welcome, {user ? user.name : 'guest'}!</p>;
}

WelcomeMessage.contextTypes = {
user: PropTypes.shape({
name: PropTypes.string.isRequired
  })
};

export default WelcomeMessage;
```

Pay attention to the `contextTypes` static property; it is used to explicitly specify what data from the context should be available to the component during rendering. In the preceding example, regardless of how many objects are stored in the context, only the `user` object will be available to that component at runtime. If you don't specify the `contextTypes` property, the context variable will be an empty object.

Now, let's see how to make the user object available via React's `context`. Create a new component called `Context`:

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
class Context extends Component {
staticchildContextTypes = {
```

```
user: PropTypes.shape({
name: PropTypes.string.isRequired
    })
  };
getChildContext() {
return { user: this.props.user };
  }
render() {
returnReact.Children.only(this.props.children);
  }
}

export default Context;
```

As you can see, all you have to do, is to set the `childContextTypes` static property that enforces context object validation during development (similar to how you use `propTypes`) and declares the `getChildContext()` instance method. This method will be called whenever the `Context` component is being mounted or updated. The returned value is then used by React to pass down the component tree.

The following is an example on how to use this component:

```
<Context user={{ name: 'Tarkus' }}><TestPage /></Context>
```

The `TestPage` component will be able to access the `user` object even though it wasn't explicitly passed to it via props.

Now let's see how to set this `user` object during server-side rendering. We are going to cover authentication in one of the later chapters, but for now, let's mock authentication middleware in our `server.js` file as follows:

```
server.use((req, res, next) => {
if (typeofreq.query.admin !== 'undefined') {
req.user = { name: 'Tarkus '};
  } else {
req.user = null;
  }
next();
});
```

If you append the `?admin` query string to the URL in a browser, this middleware will set the `req.user` property to a user object, otherwise, `req.user` is going to be `null`.

As the next step, let's update our server-side logic to pass the user object from the `req.user` variable down into our React application:

```
server.get('*', (req, res) => {
const state = { user: req.user };
const component = Router.match(req, state);
const body = ReactDOM.renderToString(component);
const html = ReactDOM.renderToStaticMarkup(<Html
title="My App"
description="Isomorphic web application sample"
body={body}
state={state} />);
res.send('<!doctype html>\n' + html);
});
```

We're using a basic router here, so let's update it as well to make sure that the state variable is passed down to the page component. Here it is:

```
import React from 'react';
import Context from '../components/Context';

const routes = [
require('../routes/Home'),
require('../routes/Test'),
require('../routes/NotFound')
];

const router = {
match(location, state) {
let component;
const route = routes.find(x =>x.path === location.path);

if (route) {
try {
component =route.action(location);
     } catch (err) {
component =routes.find(x =>x.path === '/500').action();
      }
    } else {
component =routes.find(x =>x.path === '/404').action();
   }

return<Context {...state}>{component}</Context>;
  }
};

export default router;
```

We should also update the `Html` component to make sure that the application's state is rendered inside the `<script>` tag:

```
import React from 'react';
import PropTypes from 'prop-types';
function Html({ title, description, body, state }) {
return (
<html>
<head>
<metacharSet="utf-8" />
<metahttpEquiv="x-ua-compatible" content="ie=edge" />
<title>{title}</title>
<meta name="description" content={description} />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<script asyncsrc="client.js" />
</head>
<body>
<div id="app" dangerouslySetInnerHTML={{__html: body}} />
<scriptdangerouslySetInnerHTML={{__html:'window.AppState=' +
JSON.stringify(state) }}></script>
</body>
</html>
  );
}

Html.propTypes = {
title: PropTypes.string.isRequired,
description: PropTypes.string.isRequired,
body: PropTypes.string.isRequired,
state: PropTypes.object.isRequired
};

export default Html;
```

Now when the app is bootstrapped in a browser, the `window.AppState` variable will contain the application's state generated on the server and passed to the client via the `<script>` tag, embedded into the HTML document.

The final peace of the puzzle would be grabbing this state variable in the client side code (`client.js`) and passing it to the `Router.match(location, state)` method the same way we did it on the server. Let's update the `run()` method inside the `server.js` file to look as follows:

```
function run() {
constlocation = { path: window.location.pathname };
const component = Router.match(location, window.AppState);
ReactDOM.render(component, document.getElementById('app'));
}
```

How to set the page title and meta tags

When React renders a page (screen), we need to somehow pass page metadata, such as page title, back to the caller. Look at this page component for example:

```
functionPrivacyPage() {
return (
<div>
<h1>Privacy Policy</h1>
<p>Coming soon</p>
</div>
  );
}
```

It has a title and some text. Wouldn't it be good to pass this exact title up to the rendering logic, so it could set `<title>Privacy Policy</title>` on the server (during server-side rendering) and call `document.title = "Privacy Policy"` on a client?

To solve this problem, we can use the `context` feature described previously. This is what the new `PrivacyPage` component would look like:

```
functionPrivacyPage(props, { page }) {
page.title = 'Privacy Policy';
return (
<div>
<h1>{page.title}</h1>
<p>Coming soon</p>
</div>
  );
}
```

```
PrivacyPage.contextTypes = {
page: PropTypes.shape({ title: PropTypes.string }).isRequired
};

export default PrivacyPage;
```

Now, let's update the router to pass a new page object each time we need to render a new page (screen):

```
const router = {
match(location, state) {
let component;
const page = {
title: 'My Application',
description: 'Isomorphic web application sample',
status: 200
};
const route = routes.find(x =>x.path === location.path);

if (route) {
try {
component = route.action(location, state);
      } catch (err) {
component = routes.find(x =>x.path === '/500').action();
page.status = 500;
      }
    } else {
component = routes.find(x =>x.path === '/404').action();
page.status = 404;
    }

return [
<Context {...state} page={page}>{component}</Context>,
page
];
  }
};
```

The default values should probably go from a configuration file, but to keep things short, we will set them right inside the router, where the page object is being initialized.

Next, we need to update the `Context` component to pass the `page` object as a context variable:

```
class Context extends Component {
staticchildContextTypes = {
page: PropTypes.shape({
title: PropTypes.string,
```

```
description: PropTypes.string,
status: PropTypes.number
}),
user: PropTypes.shape({
name: PropTypes.string.isRequired
    })
  };
getChildContext() {
return {
page: this.props.page,
user: this.props.user
    };
  }
render() {
returnReact.Children.only(this.props.children);
  }
}
```

The last thing we need to do is to update our server-side and client-side code which calls the `Router.match(...)` method and renders the output component. The rendering function inside `server.js` will look as follows:

```
server.get('*', (req, res) => {
const state = { user: req.user };
const[component, page] = Router.match(req, state);
const body = ReactDOM.renderToString(component);
const html = ReactDOM.renderToStaticMarkup(<Html
title={page.title}
description={page.description}
body={body}
state={state} />);
res.status(page.status).send('<!doctype html>\n' + html);
});
```

The client-side portion of the rendering logic will look like this:

```
function run() {
const state = window.AppState;
constcontaine = document.getElementById('app');
const location = { path: window.location.pathname };
const[component, page] = Router.match(location, state);
ReactDOM.render(component, container, () => {
document.title = page.title;
document.querySelector('meta[name=description]')
.setAttribute('content', page.description);
});
}
```

Right after the returned router component finished rendering and is mounted into the DOM, the preceding callback function will update the page title and description to the newest values. We do not yet have the functionality to navigate between pages on the client; this will be covered in later chapters.

Working with third-party libraries

When developing isomorphic apps you may bump into this issue when a third-party UI library that you need to include into your project does not support server-side rendering. If you try to include it in any of your React components, the `ReacDOMServer.renderToString()` will throw an exception when you try to render your app on the server. This often happens because a third-party library uses global variables such as `window`, `document`, and `navigator`, which is not available in the Node.js environment.

Often, rendering such components on the server doesn't give you much benefit. For example, it can be a WYSIWYG editor, an autocomplete box, and so on. Do you really need to pre-render that stuff on the server? If not, you just need to make sure that these components are referenced and initialized only on the client side.

The following is an example of a popular text editor `codeMirrorintegrated` into a React component:

```
classTextEditor extends Component {
componentDidMount() {
this.codeMirror = require('codemirror')
.fromTextArea(this.refs.input);
this.codeMirror.on('change', this.handleChange);
  }
componentWillUnmount() {
if (this.codeMirror) {
this.codeMirror.toTextArea();
    }
  }
handleChange = (editor) => {
if (this.props.onChange) {
this.props.onChange(editor.getValue());
    }
  };
render() {
return<textarea ref="input">{this.props.children}</textarea>;
  }
}
```

Notice that the `codemirror` module is loaded and initialized inside the `componentDidMount()` method (as opposed to referencing it at the top of the `TextEditor.js` file). This method is never called when you render a React app into the HTML string. Thus, the `codeMirror` text editor will be initialized only on the client (in a browser).

If you really need to render on the server a third-party library that is not natively isomorphic, for example, D3 charts, it is possible to do so with a `jsdom` library. However, this subject falls out of the scope of this book.

Fetching data from the server

Let's learn another good trick which may help you develop isomorphic components. Say that we need to have an HTTP client utility that will be used to request data from the server. The implementation of this utility will differ based on what environment it's targeting. In Node.js, it will use native `http` module, and on the client, it will use the browser's `XMLHttpRequest`. Or, if we want to work with a higher level of abstraction, we can use the `node-fetch` module on the server and the `whatwg-fetch` module on the client.

You can install these two npm modules by running the following:

```
npm install whatwg-fetch node-fetch --save
```

The first one uses `XHTMLHttpRequest` behind the scene and the later one-Node.js `http` module, both providing almost the same API.

Now, create two files, one for each of the environments we're targeting:

```
// core/fetch/fetch.client.js
import 'whatwg-fetch';

export default self.fetch.bind(self);
exportconst Headers = self.Headers;
exportconst Request = self.Request;
exportconst Response = self.Response;

// core/fetch/fetch.server.js
export * from 'node-fetch';
```

In addition to that, create `core/fetch/package.json` with the following contents:

```
{
  "private": true,
  "name": "fetch",
"main": "./fetch.server.js",
"browser": "./fetch.client.js"
}
```

Webpack will use it to determine which file of these two should be included into each of the bundles it generates. `fetch.client.js` will be compiled along with the client-side bundle(s) and `fetch.server.js` will be embedded into the server-side bundle. Now you can simply reference that `fetch` module we just created in our React component and forget that it works differently on the server and client behind the scene.

The code here demonstrates how this `fetch` module can be used:

```
import React, { Component } from 'react';
import fetch from '../../core/fetch';

class Test extends Component {
state = { data: 'loading...' };
async componentDidMount() {
try {
const response = await fetch('/api/test');
const data = await response.text();
this.setState({ data });
    } catch (err) {
this.setState({ data: 'Error ' + err.message });
    }
  }
render() {
return<p>Server response: {this.state.data}</p>
  }
}

export default Test;
```

Note that the `componentDidMount()` method is marked as `async`. It allows using async/await syntax inside, making the code easier to read as opposed to chaining with the `.then()` and `.catch()` methods.

If you're new to the WHATWG Fetch API, you can start by checking out the following blog post: `https://jakearchibald.com/2015/thats-so-fetch/`.
Find the full Fetch API at `https://fetch.spec.whatwg.org/`.

The React component from the previous example will send an HTTP request to the `/api/test` server endpoint as soon as it's mounted into the DOM. That's the easiest way to test Ajax functionality inside the React components. We are going to cover more advanced scenarios in later chapters. If you're curious what the RESTful API endpoint looks like, here it is:

```
// api/test.js
import express from 'express';

const router = express.Router();

router.get('/test', (req, res) => {
res.send({ message: 'Hello from RESTful API' });
});

export default router;
```

You can register this API middleware in the `server.js` file as follows:

```
server.use('/api', require('./api/test.js'));
```

There is a problem with the server-side version of the `fetch` module. What will happen if it tries to send an HTTP request to `/api/test` URL in the Node.js environment? It will throw this error: `only absolute URLs are supported`. In order to fix this problem, we can monkey patch the `node-fetch` module to prefix all relative URLs with `http://localhost:3000` as the following code sample demonstrates:

```
// core/fetch.server.js
import fetch, { Request, Headers, Response } from 'node-fetch';

functionlocalFetch(url, options) {
return fetch(url.startsWith('http') ?
url : 'http://localhost:3000' + url, options);
}

export { localFetch as default, Request, Headers, Response };
```

Now, our fetch module is truly isomorphic, and we can use it the same way in both client-side and server-side code.

Summary

In this chapter, we walked through the basics of rendering the React components on the server, learned how to debug potential issues related to server-side rendering, and got familiar with a bunch of tricks and techniques which should help you to master isomorphic web application development.

It's highly recommended that you play with the sample source code accompanying this book (see the `chapter-05` folder). Make sure that you can quickly accomplish the problems discussed in this book on your own, such as:

- How to serialize a component's state on the server, embed it into a HTML page and restore (dehydrate) on the client? See `components/CurrentTime`.
- How to pass the currently logged-in user object to React application? See `server.js`, `components/Html`, `routes/Test`.
- How to set the document's title and other meta tags on both server and client? See `server.js`, `components/Html`, `routes/Test`.
- How to include third-party UI components into your app that fail to render on the server?
- How to fetch data from the server so that the same code can be executed on both client and server? See `server.js`, `api/test.js`, `core/fetch`, `routes/Test`.

In the next chapter, you are going to learn how to create a server-side HTTP endpoint for streaming data into our app using GraphQL.

6
Creating Data API with GraphQL

In this chapter, you will learn how to create a data API server in compliance with GraphQL specification. Why GraphQL, and not a traditional RESTful API? Because, if you start developing a data API server following RESTful methodology and try to optimize it for your web and mobile client apps, at the end of the day, you will end up having something very similar to GraphQL, which is the technology used at Facebook to request and deliver data to mobile and web apps since 2012. It allows them to move fast, increase developer productivity and seems to be working brilliantly at Facebook scale and performance requirements, serving hundreds of billions of requests per day.

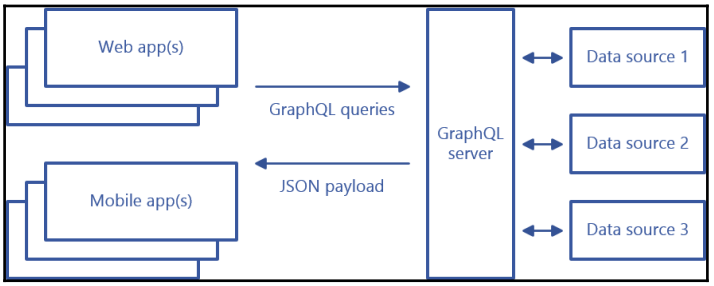
In 2015, it was open sourced in a form of GraphQL specification, a reference (runtime) library implemented in JavaScript, dev tools (GraphiQL IDE), and documentation that can be found at `http://graphql.org`.

The topics covered in this chapter are as follows:

- The basics of GraphQL
- GraphQL query language
- The GraphQL type system
- Implementing a GraphQL server with Node.js

The Basics of GraphQL

GraphQL is an application layer query language from Facebook. Despite the name and connotations it may imply, it has nothing to do with Graph databases; it is agnostic to how data is stored. For example, `https://github.com/graphql/swapi-graphql/` is a GraphQL server that is backed by the `swapi.co` REST API. The examples in `https://github.com/graphql/raphql-js` repository on GitHub are backed by in-memory JSON objects. In real-world applications, it's not uncommon that GraphQL types are backed by data stored in a number of backends, including types backed by SQL tables.



What does it look like? Let's say you have a webpage that displays a rental item, alongside a list of related ads in the same area:

The data required to render that page may look something like this:

```
{
  viewer: { name: 'Konstantin Tarkus' },
  offer: {
    id: 123,
    title: 'Children's Bike',
    location: { city: 'New York', state: 'NY' },
    picture: { url: '//cdn.com/123.png', width: 800, height: 90 },
    description: 'Loremipsum...',
    price: { hourly: 9, daily: 25 }
    author: { name: 'John' },
    related: [
      { id: ..., name: ..., picture: ... }, ...
    ]
  }
}
```

It contains the details of the currently logged in user: the `name`, `location`, `description`, and `price` information of the listing item, plus the list of similar ads within the same area.

While the shape of the data doesn't leave much room for uncertainty, there are many ways to implement data API endpoints that would stream that data into the client app, each with its own pros and cons.

With a traditional RESTful API, you could fetch that data by sending multiple requests to the server, at least one request per data type:

```
http://rentals.com/api/v1/user       - currently logged in user
http://rentals.com/api/v1/offer/123  - an ad with ID 123
http://rentals.com/api/v1/users/xxx  - author of the listing
http://rentals.com/api/v1/offers/xxx - info for each related ad
```

While this approach is very simple to implement on the server, it greatly complicates data fetching logic on the client, and, in many cases may be the cause of data communication issues related to performance, especially on mobile devices. Imagine when one of the requests fails; taking into consideration often unstable internet connections on mobile devices, how would you make sure that your code can handle it? Would you just retry the failed request, or the full batch?

Alternatively, you can create separate REST endpoints for each page/screen type of your app. This way, you will be able to fetch all the dataset required for any given page or screen with a single request, but this approach heavily increases coupling between the server and the client, making it error prone to carry out refactoring and adding new features.

Another issue with RESTful APIs is that it might be tricky to control the shape of data being fetched, dealing with under- or over-fetching issues.

In contrast, the data for the preceding example of a web page can be fetched with a single request to a GraphQL API server, as follows:

```
const query = `{
  viewer { name },
  ad(id: 123) {
    id,
    title,
    location { city, state },
    picture { url, width, height },
    description,
    price { hourly, daily },
    author { name },
    related(first: 3) {
      id,
      name,
      picture { url, width, height }
    }
  }
}`;

fetch(`${window.location.origin}/graphql`, {
  method: 'post',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ query })
}).then(res =>res.json()).then(result => console.log(result));
```

Here, the `query` is just a text string that is sent to the `/graphql` endpoint via a HTTP GET or HTTP POST request.

The `fetch` utility used in the code sample earlier doesn't currently work in IE, Edge, Safari, and Opera Mini browsers. However, it can be easily polyfilled using the `whatwg-fetch` library from npm. For more information, visit `https://github.com/github/fetch`.

This request would resolve to something like this:

```
{
  "data": {
    "viewer": { "name": "Konstantin Tarkus" },
    "offer": { "id": 123, "title": "Children's Bike", ... }
  }
}
```

Or, if the GraphQL server fails to handle the request, for example, if one of the fields was mistyped, the response from the server will look as follows:

```
{
  "errors": [{
      "message": "Cannot query field \"foo\" on type \"Offer\".",
      "locations": [{ "line": 4, "column": 5 }]
  }]
}
```

GraphQL query allows fetching exactly the same fields that are required on the client; no more, no less. Moreover, the shape of the query fully matches the shape of the data being fetched from the server, making this query language highly efficient and effective compared to other languages such as SQL, MQL, Gremlin, and so on.

As you can see, the GraphQL query looks very similar to JSON or JavaScript object notation, with the major difference being that it contains only object fields and not field values.

Why not use JSON instead of inventing a new standard? Although the same query might be written in JSON, it will be much more verbose and less efficient to work with. Compare these two queries as an example:

A query written in JSON

```
{
  "query": {
    "viewer": {
      "name": null
    },
    "offer (id: 123)": {
      "title": null,
      "description": null
    }
  }
}
```

A query written in GraphQL

```
query {
  viewer {
    name
  },
  offer (id: 123) {
    title,
    description
  }
}
```

If you look at the GraphQL query example earlier, you will notice that there are just two top-level fields in it (`viewer` and `offer`), and all the extracted data goes around these two seeds. This pattern proved to be a key design choice for an efficient architecture and short query delays. If you need to fetch more data, most of the time, you just add more fields to the nodes of the existing query.

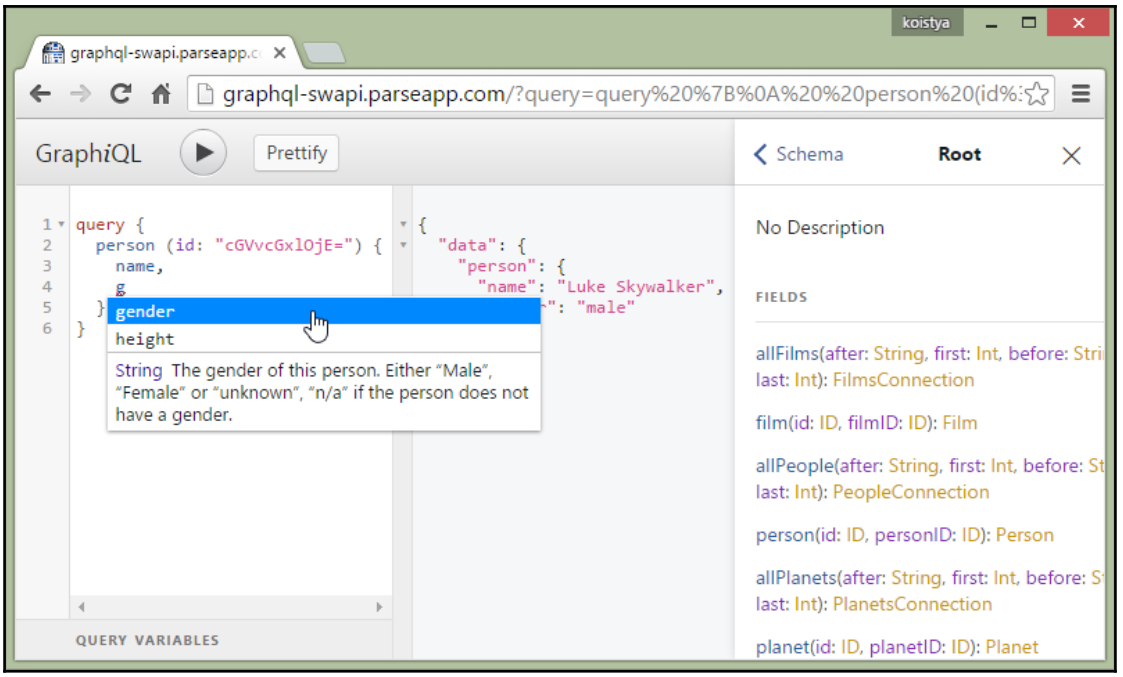
For example, to fetch the top five site notifications for the currently logged in user, assuming that the server supports it, you simply add the `notifications` field to the `viewer` node (selection set in GraphQL terminology):

```
{
  viewer {
    name,
    notifiactions { message }
  },
  offer (id: 123) { ... }
}
```

Requesting the whole selection set without specifying the list of its fields is not allowed. A query like `{ viewer { name, notifications } }` will return an error saying that `notifications` is a complex type but is being used as a scalar.

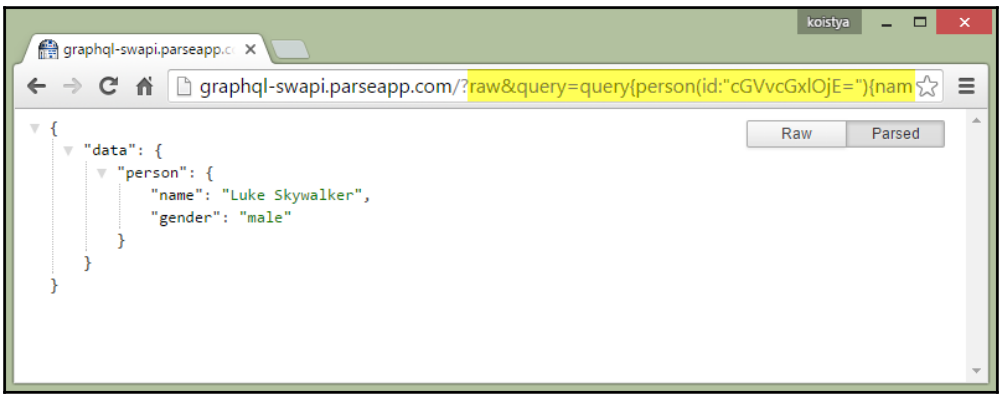
Since all the fields of a data shape requested by a client are explicitly stated within the GraphQL query, adding new fields to existing nodes on the server doesn't impact existing clients. This eliminates the need of versioning your data API server endpoint, which is not often the case with RESTful APIs. With GraphQL, there is no need to create a separate additional endpoint for mobile apps; moreover, different versions of the same mobile app will work fine with the same GraphQL API server. Adding a new feature should not break any of these apps.

The easiest way to try GraphQL, is by visiting `http://graphql.org/swapi-graphql/graphql-swapi.parseapp.com` website, which is a live demo of a GraphQL server built on top of the existing `swapi.co` REST API:



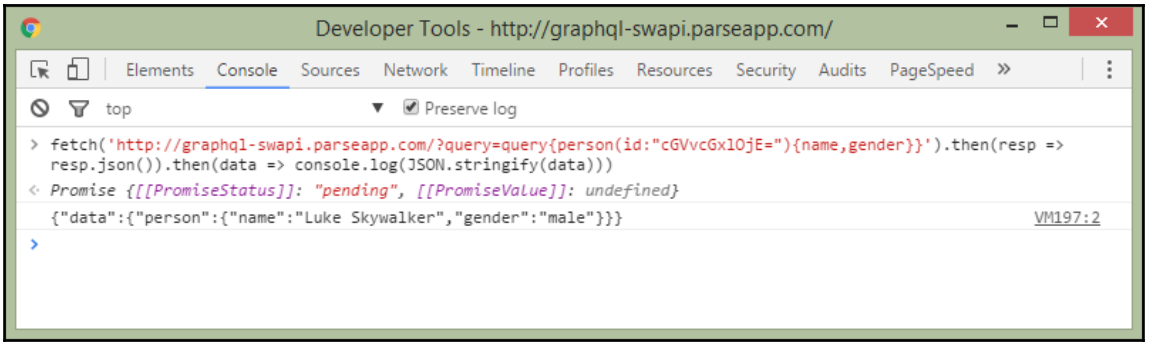
What you see there, is a GraphiQL IDE interface, where, on the right-hand side you can find the API documentation, automatically generated based on what capabilities the GraphQL server exposes. On the left-hand side, there is a GraphQL editor with auto-complete and syntax highlighting, and, in the middle section, you can see the result of query execution.

You can add the `raw` parameter to the query string in the browser's URL window, to view the result of the query as plain JSON:



Another way to execute the same query is by typing the `fetch(...)` command in the browser's console window, for example:

```
fetch('http://graphql-swapi.parseapp.com/?query=query{person(id:
 "cGVvcGxlOjE="){name,gender}}').then(resp =>resp.text()).then(data =>
console.log(data))
```



One of the design goals for GraphQL was to make it intuitively simple to learn and use by frontend developers. Try to play some more with the demo site earlier and see on your own whether you will be able to fetch the data available on the server even without reading any documentation to GraphQL query language. Right after that, we are going to explore some of the query language features that might not be obvious.

GraphQL query language

Let's take a closer look at this query here to see what parts it consists of:

```
query PersonQuery {
  person(id: "cGVvcGxlOjE=") {
    name,
    gender,
    homeworld {
      name
    }
  }
}
```

The first line defines an operation named `PersonQuery` that queries data from the server, where the `query` keyword is one of the three currently supported operations: query, mutation, and subscription.

Specifying the `query` keyword and an operation name (`PersonQuery` in this case) is only required when a GraphQL document defines multiple operations. We, therefore, could have written the previous query with a shorthand version:

```
{
  person(id: "cGVvcGxlOjE=") {
    name,
    gender,
    homeworld {
      name
    }
  }
}
```

The `person`, `name`, and `homeworld` keywords are all query fields, where `name` and `gender` fields are considered to be leaf nodes of the query, requesting scalar values such as strings, Boolean values, and numbers; and `person` and `homeworld` are the fields that correspond to complex data types on the server. The `id` field, as you can guess, is an argument to the `person` field, and the `cGVvcGxlOjE=` string is an input object to the `id` argument.

Before the GraphQL server can execute a query sent from a client app, it validates that query against the schema that the GraphQL server is built upon. Like in this example, the GraphQL server has a `query` field at the top (root) level of its schema, underneath that field it has `person` field, which is declared alongside with `id` argument of type `ID`, and underneath that field, it has `name`, `gender`, `homeworld`, and other fields.

If you mistyped any of the fields or haven't specified a field argument and input value that is marked as required on the server, you will get an error response from the server similar to this one:

```
{
  "data": { "person": null },
  "errors": [{ "message": "must provide id" }]
}
```

Speaking of field arguments, you can hardcode these arguments into the query string:

```
var id = 'cGVvcGxlOjE=';
var query = `query { person(id: "${id}") { name } }`;
fetch('http://graphql-swapi.parseapp.com/', {
  method: 'post',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ query })
}).then(resp =>resp.json()).then(data => console.log(data));
```

In most cases, you want to use the HTTP POST method when sending queries to a GraphQL server in order to avoid receiving cached responses.

This approach may not work with large queries because string concatenation might be an expensive operation taking into consideration that field arguments must be correctly encoded and the client may repeatedly compile multiple such queries.

Luckily, GraphQL query language comes with the notion of query parameters that you can send alongside with a parametrized query to a GraphQL server as the following example demonstrates:

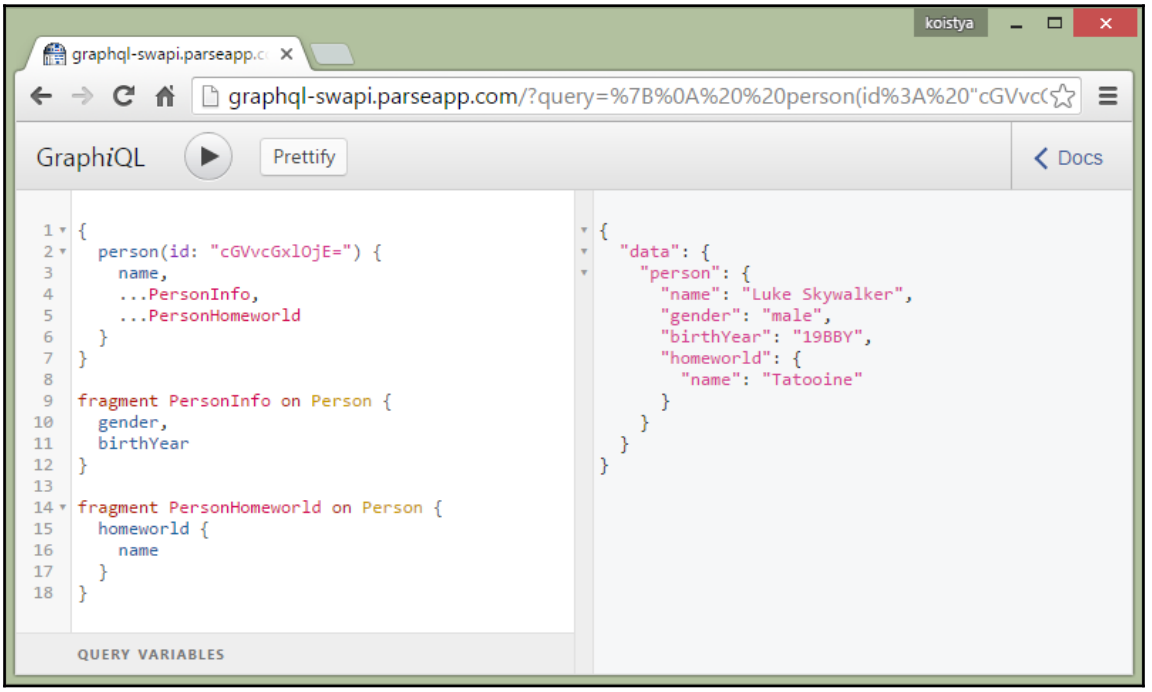
Here is the code sample that sends a parametrized GraphQL query to the server:

```
var query = `query PersonQuery($id: ID) {
  person(id: $id) {
    name,
    gender,
    homeworld {
      name
    }
  }
}`;
```

```
var variables = { id: 'cGVvcGxlOjE=' };
fetch('http://graphql-swapi.parseapp.com/', {
  method: 'post',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ query, variables })
}).then(resp =>resp.json()).then(data => console.log(data));
```

The query name `PersonQuery` in the earlier example is optional. The type of the query parameter should match to the type of the field argument where it's used as an input value, otherwise, the query won't pass validation to the server.

Another essential part of GraphQL language is an ability to compose a query out of query fragments as the following example demonstrates:



This is extremely useful when different UI components need to be able to specify their own data requirements in a form of query fragments that can be combined into a single GraphQL query operation. As a result, we are able to fetch data for all the UI components on a page (screen) using a single HTTP request to the server.

It is also possible to add field aliases that might be handy when you need to fetch the same field multiple times with different augments within the same query as the following example demonstrates:

```
{
  luke: person(id: "cGVvcGxlOjE=") {
    name,
    homeworld { name }
  },
  leia: person(id: "cGVvcGxlOjU=") {
    name,
    homeworld { name }
  }
}
```

yields:

```
{
  "data": {
    "luke": {
      "name": "Luke Skywalker",
      "homeworld": { name: "Tatooine" }
    },
    "leia": {
      "name": "Leia Organa",
      "homeworld": { name: "Alderaan" }
    }
  }
}
```

It is also possible to dynamically include or exclude some fields in/from the query result using `@include` / `@exclude` directives, for example:

```
query($full: Boolean!) {
  person(id: "cGVvcGxlOjE=") {
    name,
    gender@include(if: $full),
    homeworld@include(if: $full) {
      name
    }
  }
}
```

When the preceding query is being called with the `$full` query variable set to `false`, the `gender` and `homeworld` fields are going to be excluded from the query result.

At the time of this writing, `@include` and `@exclude` are the only two directives described in the GraphQL specification, but it's possible that more directives will be added in the future. Also, you can add custom directives as a way to extend the functionality of the GraphQL query language.

The GraphQL type system

At the core of GraphQL is a type system, which describes the capabilities of the data API server and is used to determine if a query is valid. At each level of a GraphQL query, a particular type applies describing not only what fields are available but what arguments you can supply to those fields, and what types are going to be resolved from those fields.

A GraphQL server's capabilities are referred to as that server's schema, that is defined in terms of the types and directives it supports. The fundamental unit of any GraphQL schema is a type. There are eight kinds of types in GraphQL: `Scalar`, `Enum`, `Object`, `Interface`, `Union`, `List`, `Non-Null`, and `Input Object`.

The `Scalar` types are primitive values that describe leaf nodes of the GraphQL schema; there are five built-in scalar types:

- `Int`: A signed 32-bit numeric non-fractional value
- `Float`: A signed double-precision fractional value
- String: Textual data represented as a UTF-8 character sequence
- `Boolean`: Can either be `true` or `false`
- ID: A unique identifier, serialized the same way as String

The `Object` types are lists of named fields, each of which has a specific type. Objects describe shapes of data trees, intermediate nodes of a GraphQL schema. At least one field must be specified when declaring a custom object type. The names of the fields must be unique; no two fields can share the same name. Object fields may have arguments, each of which has also a specific type that can be a scalar or a complex type. A field can be marked as deprecated. An `Object` type can be marked as a super-set of one or more interfaces; in that case, it should contain all the fields declared in the interfaces it implements.

The `Interface` type represents a list of named fields and arguments. The GraphQL object can then implement an interface, which guarantees that it will contain the specified fields.

The `Union` represents an object that could be one of the items in a list of GraphQL `Object` types. It's used to describe what types are possible as well as provide a function to determine which type is actually used when the field is resolved. When querying against a `Union`, you must narrow the type using a fragment; this ensures that your query will be resilient to the evolution of the type schema over time.

The `Enums` are a variant on the `Scalar` type, which represents a finite set of possible values. GraphQL serializes `Enum` values as strings; however, internally `Enums` can be represented by any kind of type, often integers.

The `Input Objects` are intended to be used when you need to specify a complex type for a field argument. The regular `Object` type is inappropriate for reuse here because objects can contain fields that express circular references or references to interfaces and unions, neither of which is appropriate to use as an input argument. For this reason, input objects have a separate type of the system. Similar to objects, they define a list of named fields and their types-`Scalars`, `Enums`, or other `Input Objects`. This allows arguments to accept arbitrarily complex structs.

`List` and `Non-Null` types are type modifiers (aka, markers in GraphQL schema language terminology); they wrap another type instance. For example, `List<String>` or `[String]` for short, denotes a list (collection) type where each item in the collection is of type `String`. `NonNull<String>` or `String!` for short, denotes a `String` type that cannot be `null`. It is also possible to combine these types, for example, `NonNull<List<String>>` or `[String]!` for short, denotes a collection type, that cannot be `null` and where each item of that collection is of the type `String`.

More detailed information about the GraphQL type system can be found at `http://facebook.github.io/graphql/#sec-Type-System`.

You can use the built-in scalar types to describe custom complex types, for example:

```
interface Node {
  id: ID!
}
type User implements Node {
  id: ID!,
  name: String,
  email: String,
  offers: [Offer]
}
type Offer implements Node {
  id: ID!,
```

```
  title: String,
  description: String,
  ...
}
enumOffersOrder {
  BEST_MATCH,
  NEAREST
}
```

What you see earlier is GraphQL schema language that is sort of a pseudo code used in GraphQL specification. Later in this chapter, you will learn how to declare a GraphQL schema alongside with custom GraphQL types in JavaScript.

In the code example earlier, `Node` is an `Interface` type; `User` and `Offer` are `Object` types that implement `Node` interface; the `id` field has a `Non-Null<ID>` type (a `Non-Null` wrapper type with an underlying `ID` type); `name` and `email` fields both have a type `String`; `offers` field has a type `List<Offer>` (a List wrapper type with an underlying `Offer` type); `OffersOrder` is an `Enum` type.

Now, you can use these custom types to define a root query (`Object`) type:

```
type Query {
  viewer: User,
  offer(id: ID!): Offer,
  offers(first: ID, order: OffersOrder): [Offer]
}
```

Finally, having this root query type and optionally mutation and subscription types, you can define a GraphQL `Schema` type:

```
type Schema {
  query: Query
}
```

In essence, this is also an `Object` type that must confirm to a GraphQL schema interface, enforcing you to declare at least the `query` field and optionally `mutation` and `subscription` fields.

Implementing a GraphQL server with Node.js

Now that you have the basic understanding about GraphQL type system, let's go ahead and implement a simple GraphQL server using project files from the previous chapter as a boilerplate for this exercise.

You can start by installing the following `npm` modules:

```
$ npm install graphql express-graphql sequelize sqlite3 --save-dev
```

The `graphql` module contains built-in data types, query validation, and execution logic. This module will be used for creating the GraphQL schema itself. The `express-graphql` module is a middleware that sits on top of Node.js/Express app and is backed by GraphQL schema created with the `graphql` npm module. It is capable of parsing bodies of HTTP requests and sending valid GraphQL responses over HTTP. Additionally, this module is capable of rendering a web page with GraphiQL IDE. Sequelize is a popular Node.js-based ORM that we are going to use for working with the database, to make our code less verbose. Finally, the `sqlite3` module is a Node.js driver for accessing SQLite database, though it might be any other database of your choice. For example, if you decide to use PostgreSQL instead, replace `sqlite3` with `pg` and `pg-hstore` npm modules, for Azure SQL Database use `tedious`.

Now create a `data` folder with `models`, `mutations`, `queries`, and `types` subfolders:

```
.
├── /data/
│   ├── /models/          # ORM models
│   ├── /mutations/       # GraphQL mutations
│   ├── /queries/         # GraphQL queries
│   ├── /types/           # Custom GraphQL types
│   └── /schema.js        # GraphQL schema
├── /server.js           # Node.js/Express
└── /package.json        # List of dependencies, project settings
```

Create a `data/schema.js` file that will export a GraphQL schema object (empty for now, we'll create the actual schema in a moment):

```
import { GraphQLSchema as Schema } from 'graphql';

const schema = new Schema({
  // TODO: Add schema configuration
});

export default schema;
```

Edit the `server.js` file to expose the GraphQL schema as an HTTP endpoint mounted at the `http://localhost:3000/graphql` URL with the help of the `express-graphql` middleware library:

```
importgraphql from 'express-graphql';
import schema from './data/schema';
...
app.use('/graphql', graphql({
  schema,
  graphiql: true,
  pretty: process.env.NODE_ENV !== 'production'
}));
```

For the full list of options for the `express-graphql` module, refer to `https://github.com/graphql/express-graphql`.

Now, we are ready to hack on the actual GraphQL schema. Before learning how to create GraphQL server backed by an SQL database, let's start with a hello world example. Here's the query and response that our GraphQL server should support:

GraphQL Query	GraphQL Response

`{`
`  greeting`
`}`
	`{`
	`  data: {`
	`  greeting: "Welcome, Guest!"`
	`  }`
	`}`

GraphQL Query	GraphQL Response

`{`
`  greeting(name: "John")`
`}`
	`{`
	`  data: {`
	`  greeting: "Welcome, John!"`
	`  }`
	`}`

We want to be able to request data for the `greeting` field from the server and optionally provide a `name` argument to that field. The server should return a text string as a result or an error if the query is mistyped.

In GraphQL schema language the schema, supporting that scenario, would look like this:

```
typeQuery {
  greeting(name: String): String
}
type Schema {
```

```
  query: Query
}
```

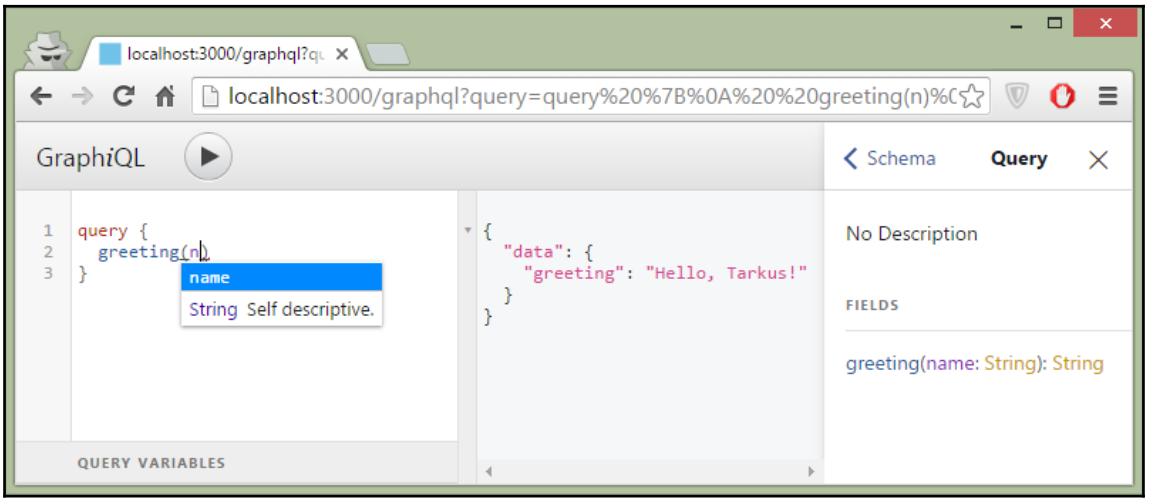
In order to translate it into JavaScript, update the `data/schema.js` file with the following contents:

```
import {
  GraphQLSchema as Schema,
  GraphQLObjectType as ObjectType,
} from 'graphql';

import greeting from './queries/greeting';

const schema = new Schema({
  query: new ObjectType({
    name: 'Query',
    fields: {
      greeting
    }
  })
});

export default schema;
```

In addition, create the `data/queries/greeting.js` file that will describe a data type of the `greeting` field used on the root query, the arguments it must support, and what exactly that field must return when queried:

```
import { GraphQLString as StringType } from 'graphql';

const greeting = {
  type: StringType,
  args: {
    name: { type: StringType },
  },
  resolve(_, { name }) {
    return `Welcome, ${name || 'Guest'}!`;
  },
};

export default greeting;
```

If everything is done correctly, now you should be able to start the server (by running `npm start` command from the console window) and navigate to the `http://localhost:3000/graphql` web page that should display GraphiQL IDE:



On the left-hand side, you can type your GraphQL query, after hitting *Ctrl* + *Enter*, the result of the query is going to be displayed in the middle section, and on the right-hand side, you have a nice documentation browser that is automatically generated based on the schema we've just created.

You can also execute the same query from code, bypassing Express middleware:

```
import { graphql } from 'graphql';
import schema from './data/schema';

graphql(schema, '{ greeting(name: "Tarkus") }')
  .then(result => console.log(result));

// =>{ data: { greeting: 'Welcome, Tarkus!' } }
```

This might be handy during server-side rendering when you need to fetch data before rendering the React app into an HTML markup string.

In order to improve your GraphQL schema, you may want to add descriptions to all the fields and field arguments, as the following example demonstrates:

```
const greeting = {
   type: StringType,
   description: 'Generates a welcome message',
```

```
    args: { name: StringType, description: 'First name' },
    resolve(_, { name }) { ... }
};
```

The resolve method may return a `Promise` object:

```
const greeting = {
   ...,
  resolve(_, { name }) {
    return new Promise(resolve =>
    setTimeout(resolve.bind(`Welcome, ${name}!`, 3000);
    });
  }
};
```

It will be correctly resolved to the actual value at runtime. This comes in handy when we need to write asynchronous code inside the resolve methods of GraphQL fields.

Backing GraphQL server by a SQL data store

Now let's see how to back our GraphQL server by a SQLite database and Sequelize ORM. Using an object-relational mapper library (ORM) seems to be a good choice for SQL data access, especially when you concerned about the speed of development, and the best part is that even with an ORM you can always fall back to writing raw SQL queries when needed.

You start by creating a `data/sequelize.js` file that initializes a new instance of Sequelize client:

```
importSequelize from 'sequelize';

constsequelize = new Sequelize('sqlite:database.sqlite', {
  define: { freezeTableName: true }
});

export default sequelize;
```

Now, create data models for each of the business entities of your app. For example, for a `User` entity you may want to create a model similar to this (`data/model/User.js`):

```
importDataType from 'sequelize';
import Model from '../sequelize';

const User = Model.define('User', {
```

```
    id: {
      type: DataType.INTEGER,
      primaryKey: true,
      autoIncrement: true
    },
    email: { type: DataType.TEXT, validate: { isEmail: true } },
    password: { type: DataType.TEXT },
    displayName: { type: DataType.TEXT }
  });

  export default User;
```

The `Offer` entity may look like this (`data/models/Offer.js`):

```
  importDataType from 'sequelize';
  import Model from '../sequelize';

  const Offer = Model.define('Offer', {
    id: {
      type: DataType.INTEGER,
      primaryKey: true,
      autoIncrement: true
    },
    slug: { type: DataType.TEXT },
    name: { type: DataType.TEXT },
    priceHourly: { type: DataType.REAL },
    priceDaily: { type: DataType.REAL },
    priceWeekly: { type: DataType.REAL }
  });

  export default Offer;
```

Same for other the remaining business entities (`Tag`, `Reservation`, `Notification`, and so on), you got the idea. After all data models were defined, you need to glue them together by configuring relationships (associations) between these models where appropriate. It might be a good idea to do that in a separate file (`data/models/index.js`) as the following code sample demonstrates:

```
  import User from './User';
  import Offer from './Offer';

  Offer.belongsTo(User, { as: 'author' });
  User.hasMany(Offer, { as: 'offers', foreignKey: 'authorId' });

  export { User, Offer };
```

This file will export all the models available in your app, so you could reference them from inside GraphQL schema files as follows:

```
import { User, Notification } from '../models';
```

One more thing that needs to be done is to add code that will establish a database connection when Node.js server starts and optionally feed the database with some fake/test data. Here is how you can do it with the popular `faker` library. First, update the `data/models/index.js` file by adding a `sync()` into it:

```
import faker from 'faker';
importsequelize from '../sequelize';
import User from './User';
import Offer from './Offer';
...

// Configure relationships (associations)
Offer.belongsTo(User, { as: 'author' });
User.hasMany(Offer, { as: 'offers', foreignKey: 'authorId' });
...

// Synchronizedb schema and data models
async function sync(options) {
  awaitsequelize.sync(options);
  if (options.force) {
    // Create fake users
    for (let i = 0; i< 50; i++) {
      constfirstName = faker.name.firstName();
      constlastName = faker.name.lastName();
      awaitUser.create({
        email: faker.internet.email(firstName, lastName),
        password: faker.internet.password(),
        displayName: faker.name.findName(firstName, lastName)
      });
      ...
    }
  }
}

export { default as { sync }, User, Offer, ... };
```

Then, import that files in `server.js` and makes sure that the `app.listen()` function launching Node.js server is executed only when a database connection was established:

```
import express from 'express';
importgraphql from 'express-graphql'
import models from './data/models';
```

```
...

models.sync({ force: process.env.NODE_ENV !== 'production' })
.catch(err =>console.error(err.stack))
.then(() => {
  app.listen(port, () => console.log(
      `Node.js server is listening at http://localhost:${port}/`
  ));
});
```

The `force` flag tells Sequelize that right after connecting to the database, it should compare database schema with the data models, drop and recreate tables that differ, and seed database with some reference (and/or fake) data. You need to be extra careful with this flag; it should not be used in production.

After having Sequelize data models in place, we must also create GraphQL types that may look similar but not exactly to the ORM data models. For example, the User data model contains a `password` field, which shouldn't be exposed to the corresponding User type. Let's go ahead and create `data/types/UserType.js`-a GraphQL type for the `Offer` entity:

```
import {
  GraphQLID as ID,
  GraphQLObjectType as ObjectType,
  GraphQLString as StringType,
  GraphQLNonNull as NonNull
} from 'graphql';

constUserType = new ObjectType({
  name: 'User',
  fields: {
    id: { type: new NonNull(ID) },
    email: { type: StringType },
    displayName: { type: StringType }
  }
});

export default UserType;
```

This corresponds to the following type in the GraphQL schema language:

```
type User {
  id: ID!,
  email: String
  displayName: String
}
```

Now, let's create a `data/queries/viewer.js` file that will describe the `viewer` field of the root query:

```
import User from '../models/User';
importUserType from '../types/UserType';

const viewer = {
  type: UserType,
  resolve({ user }) {
    returnUser.findById(user && user.id);
  }
};

export default viewer;
```

Don't forget to reference it in `data/schema.js` as the following example demonstrates:

```
import viewer from './queries/viewer';
import greeting from './queries/greeting';

const schema = new Schema({
  query: new ObjectType({
    name: 'Query',
    fields: {
      viewer,
      greeting
    }
  })
});
```

Wait, where does the `user` variable come from? The JaveScript implementation of GraphQL allows us to pass a root value (`rootValue`) to the query field resolver, which becomes available as the first argument of the `resolve()` methods of the top-level query fields, for example, `resolve(rootValue, args) { ... }` and we can also get access to it from child queries as well, using the third argument, for example, `resolve(parent, args, { rootValue }) { ... }`.

Let's update the `server.js` file to set that `rootValue` object:

```
app.use('/graphql', graphql({
  schema,
  rootValue: { user: 1 },
  graphiql: true,
  pretty: process.env.NODE_ENV !== 'production'
}));
```

For now, we'll just hard code the `{ user: { id: 1 } }` object as the root value to be passed to the query resolver, where `user` is supposed to contain information about the currently logged in user, decoded from an authorization token (more on that in `Chapter 8`, *Authentication and Authorization*).

If you've done everything correctly, by now you should be able to query the GraphQL server requesting information about the currently logged in user:

The `offer` field on the root query may look as follows:

```
import {
  GraphQLID as ID,
  GraphQLNonNull as NonNull
} from 'graphql';

import Offer from '../models/Offer';
importOfferType from '../types/OfferType';

const offer = {
  type: OfferType,
  args: { id: { type: new NonNull(ID) } },
  resolve(_, { id }) {
    returnOffer.findById(id);
  }
};

export default offer;
```

Unfortunately, more advanced topics fall out of the scope of this book. However, before finishing the chapter, we need to explain how to optimize nested and recursive queries.

Batching and caching

The simplest way to optimize nested and recursive queries is using caching and batching techniques. An alternative solution to optimize queries is using fields' ASTs, but that may become complicated.

Let's say we have a `viewer` field of a type `User` that type may contain `viewedOffers` field of type `List<Offer>` where each offer in the list has an `author` field of type `User`. This `viewer` field might be queried as follows:

```
{
  viewer {
    displayName,
    viewedOffers(first: 10) {
      title,
      author {
        displayName
      }
    }
  }
}
```

This query is supposed to return the currently logged in user, as well as the list of recently viewed by the user offers (ads) along with author information for each offer. Obviously, in order to execute this query, our GraphQL server may send many excess SQL requests to the database: `SELECT ... FROM User WHERE id = ?`, `SELECT ... FROM Offer WHERE id = ?`; one for each requested user and offer objects (regardless of how many authors have the same User ID). Fortunately, there is a very simple pattern allowing us to group related SQL requests together so that they become `SELECT ... FROM User WHERE id IN (?, ?, ?)`. It's based on Node's `process.nextTick()` feature and is successfully used at Facebook. Here is how it works. Instead of calling `User.findById(..)` directly inside the `viewer` field's `resolve()` method, let's extend the `User` model by adding the `.load(id)` method into it, that will work similar to `.findById(..)`, but will batch SQL requests under the hood (by calling `User.findAll({ where: { id: [1, 2, 3] } })`).

Update`data/models/User.js` with the following contents:

```
const batch = { keys: new Set(), task: null };

const User = Model.define('User', {
  ...
}, {
  classMethods: {
    load(id) {
      if (!batch.keys.size) {
        batch.task = new Promise((resolve, reject) => {
          process.nextTick(() => {
            this.constructor.prototype.findAll.call(this, {
              where: { id: Array.from(batch.keys) }
            }).then(data => {
              batch.keys.clear();
              resolve(data);
            }, reject);
          });
        });
      }
      batch.keys.add(id);
      returnbatch.task.then(data =>data.find(x => x.id === id));
    }
  }
});
```

Now, you can update the `data/queries/viewer.js` file to use the `User.load(...)` method instead of `User.findById(...)` as the following example demonstrates:

```
const viewer = {
  type: UserType,
  resolve({ user }) {
    returnUser.load(user && user.id);
  }
};
```

Also, do the same in all other places throughout the scheme where you need to retrieve a user object by its ID (for example, inside the `author` field's `resolve()` method). This will ensure that regardless of how many times the `User.load(id)` method is actually called during the same GraphQL query execution process, all user objects will be fetched using a single SQL request. In addition to that, it should be simple to cache the retrieved from the database entities in memory (or, in Redis) in order to further optimize the query execution process, so it could be used like this:

```
User.load(id, { cache: true });
```

There is an existing library called `DataLoader` that helps with batching and caching that you may want to use to optimize your GraphQL schema. Nevertheless, having an understanding of how batching and caching works under the hood should be a valuable skill to have.

Summary

With GraphQL, Facebook engineers promote a very good idea that there should be a clear boundary between the client and the server following common agreement. So, both parties can build and scale independently, with low coupling and high cohesion.

Isomorphic is not just about sharing code between the client and the server. It's beyond that. For example, with a GraphQL server, you can have the same data API endpoint serving both the web and multiple versions of mobile apps saving you lots of time and efforts.

By now, after completing this chapter, you should know how to write and execute GraphQL queries, via GraphiQL IDE, using Fetch API and directly by calling `graphql(..)` method on the server. You should become familiar with GraphQL type system and schema language. You should be able to create custom GraphQL schema backed by an SQL data store and using a Sequelize ORM and GraphQL JavaScript reference library. Finally, you should have an idea how to optimize your schema to work well under heavy load.

In the next chapter, we're going to look deeper into the routing and navigation part in the context of isomorphic web apps. We will define routes that will be reused in the server and in the client; we will protect them with the appropriate authorization and test them so we can be sure that we won't break it during our development of the application.

7
Implementing Routing and Navigation

A very important part of an application is routing. Server-side routing is one of the oldest things around. The client requests something through a URL and the server answers back with the requested resource. It might return a static asset as is, or execute some complex logic, gathering resources from external sources and generating the final data for the client to show. The way of doing this may greatly differ between technologies, but all web frameworks have a form of server-side routing.

Client-side routing is newer. When **single-page applications** (**SPAs**) started to win space in web development, client-side routers appeared to increase the navigation speed, improving the experience of the end user. The code to navigate to the next views was already there in the client, ready to be run. This is not free though; there is some overhead in the first load if you send a lot of data and also wait for that first render to finish. This can hurt the performance of the page if it is not handled correctly.

Many JavaScript libraries that are around manage client-side routing. Examples such as `router.js` (`https://github.com/tildeio/router.js`) and `crossroads.js` (`https://github.com/millermedeiros/crossroads.js`) are library agnostic, while others are tied to a specific framework. **React Router** (`https://github.com/ReactTraining/react-router`) can be used with React, Angular 1 and 2 have their own router, vue has `vue-router` (`https://github.com/vuejs/vue-router`), and so on.

Having our JavaScript rendering logic in the client let us handle navigation much faster with a simple replacement of markup and an optional call to the server if we need more information to feed the markup.

We have different ways of handling the routing on the client, and ways to handle the routing on the server. But in our isomorphic application, we must have one single way of defining the routes that should be reused by both the client and server.

In this chapter, we will :

- Pure server-side routing with Express
- Pure client-side routing using React Router
- Isomorphic routing
- Isomorphic routing with initial data

Let's start!

Pure server routing

Let's begin with the basics of server routing. All web frameworks have a way to define routing in the server. Routing just refers to the definition of application end points (URIs) and how they respond to client requests.

This mechanism can be as simple as returning an HTML or as complex as receiving data from the client, validating it, checking the authentication and authorization of the user, executing some complex business logic and answering with the correct response.

For our pure server-side routing examples, we will use Express. It is one of the most popular web frameworks in the node ecosystem. If you have worked with web applications in node, you must have heard of it. Other popular frameworks are Hapi or Koa. All our examples should be easy enough to develop in these frameworks.

Enough introduction; let's get to coding.

Your package.json should have the following dependencies (we will use them as we make progress in the chapter):

```
"dependencies": {
    ...
    "express": "4.15.4",
    "react": "16.0.0",
    "react-redux": "5.0.6",
    "react-router": "4.2.0",
```

```
    "react-router-config": "1.0.0-beta.4",
    "react-router-dom": "4.2.2",
    "redux": "3.7.2"
    ...
  }
```

Express routing

In Express, a route method is derived from one of the HTTP methods and is attached to an instance of the `express` class.

The following code is an example of routes that are defined for the GET and the POST methods to the root of the app.

```
var express = require('express')
var app = express()

// GET method route
app.get('/', function (req, res) {
 res.send('GET request to the homepage')
})

// POST method route
app.post('/', function (req, res) {
 res.send('POST request to the homepage')
})
```

You can create chainable route handlers for a route path using `app.route()`. Because the path is specified at a single location, creating modular routes is helpful, as is reducing redundancy and typos.

The following examples create a router as a module, load a middleware function in it, define some routes, and mount the router module on a path in the main app:

```
// router.js
var express = require('express')
var path = require('path')
var router = express.Router()

// middleware that is specific to this router
router.use(function timeLog (req, res, next) {
 console.log('Time: ', Date.now())
 next()
})

// define the home page route, sending an HTML file
```

```
router.get('/', function(req,res){
 res.sendFile(path.join(__dirname+'/index.html'));
 //__dirname : It will resolve to your project folder.
});

// define the about route
router.get('/about', function (req, res) {
 res.send('About me')
})

//using params in the URL
router.get('/books/:bookId', function (req, res) {
 res.send(req.params)
})

// routers can use regular expressions
router.get(/.*author$$/, function (req, res) {
 res.send('/.*author$/')
})

module.exports = router
```

You can see that there are many ways to define a route, with specific URLs, with a regular expression or using IDs that will be resolved for your use when the request arrives.
The next step is to use our module router in our Express app:

```
var express = require('express')
var path = require('path')

var app = express()
var router = require('router') //the one just created

app.use(router);

app.listen(3000, function () {
 console.log('Express server running at localhost: 3000')
});
```

All the routes defined in the router now work in our application.

This is a very basic example of how server-side routing works in a particular web server such as Express, but the concepts in it are very similar in all web frameworks.

Although we won't use this kind of routing (because is not isomorphic), it's good to know how it would work in a non-isomorphic way.

Now, let's see what pure client-side routing would look like.

Pure client routing

Pure client-side routing came into picture when people started using the JavaScript functionality and state they had in the client to manage not only the state of the current view but also the navigation to other views.

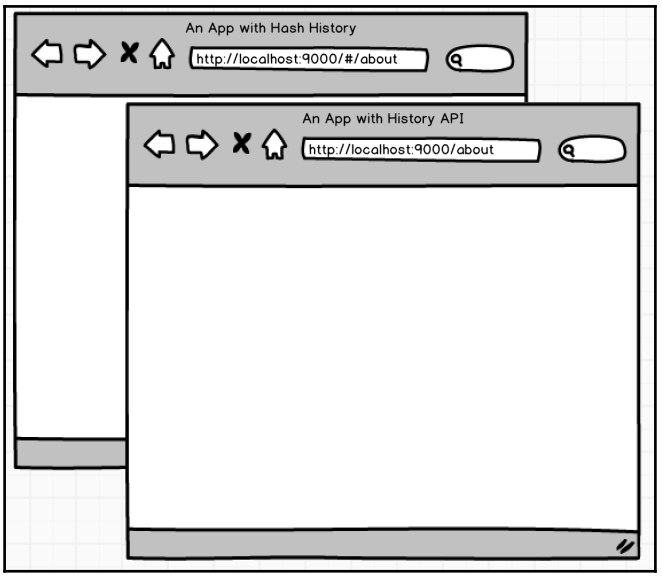
The good part of client-side routing is that it's fast, the code to render the view is there, and if the data is there too, then it's instantaneous. The bad part is that the initial load of the code is slower, as you need to load the current view and the code to navigate to the next ones. As always, it's a matter of choosing where each practice fits the best, and mix it if necessary, to get the best possible user experience.

Hash versus history API

The hash history works on older browsers (such as Internet Explorer 8 and 9) and doesn't require any server configuration. It uses hashes in the URL to define the routes, such as `example.com/#/home` and `example.com/#/about`.

If your application doesn't need to run on legacy browsers, where the browser's History API is not supported and you have the possibility to configure your server, the ideal approach is to use the browser history setup, which creates real URLs that look like `example.com/path`.

The browser history API setup can generate real looking URLs. The dom window object provides access to the browser's history through the history object. It exposes useful methods and properties that let you move back and forth through the user's history, as well as starting with HTML5, manipulate the contents of the history stack:



Some examples of the API can be:

```
//To move backward through history:
window.history.back();
window.history.forward();

//This will act exactly like the user clicked on the Back or Forward
buttons respectively in their browser toolbar.

//You can use the go() method to load a specific page from session history,
identified by its relative position to the current page (with the current
page being, of course, relative index 0).

//To move back one page (the equivalent of calling back()):
window.history.go(-1);
//To move forward a page, just like calling forward():
window.history.go(1);
```

These are the most common actions that can be done with the API; now let's see some other functionalities like seeing the history stack and pushing and replace its state.

```
//You can determine the number of pages in the history stack by looking at
the value of the length property:
var numberOfEntries = window.history.length;

//HTML5 introduced the history.pushState() and history.replaceState()
methods, which allow you to add and modify history entries, respectively.
These methods work in conjunction with the window.onpopstate event.
var stateObj = { foo: "bar" };
history.pushState(stateObj, "page 3", "bar.html");

//pushState() takes three parameters: a state object, a title (which is
currently ignored), and (optionally) a URL.

//state object — The state object is a JavaScript object which is
associated with the new history entry created by pushState().
//title – Ignored, maybe with future use
//URL — The new history entry's URL is given by this parameter

history.replaceState(stateObj, "page 3", "bar2.html");
```

What if I want to React to the state changes?

```
//A popstate event is dispatched to the window every time the active
history entry changes. If the history entry being activated was created by
a call to pushState or affected by a call to replaceState, the popstate
event's state property contains a copy of the history entry's state object.
window.onpopstate = function(event) {
 console.log("location: " + document.location + ", state: " +
JSON.stringify(event.state));
};

history.pushState({page: 1}, "title 1", "?page=1");
history.pushState({page: 2}, "title 2", "?page=2");
history.replaceState({page: 3}, "title 3", "?page=3");
history.back(); // log: "location: http://example.com/example.html?page=1,
state: {"page":1}"
history.back(); // log: "location: http://example.com/example.html, state:
null
history.go(2); // log: "location: http://example.com/example.html?page=3,
state: {"page":3}
```

But what happens if the user navigates to a deep nested URL in our application with client-side routing?

These URLs are dynamically generated at the browser; they do not correspond to real paths on the server, and since any URL will always hit the server on the first request, it will likely return a `Page Not Found (404)` error. This is where server-side rendering comes into play.

But first, let's see how to use the browser history API with React in the client.

React routing

Although we will explore only one option, React Router, it is important to know that other options exist as well. These are some examples:

- **react-router-component** (`https://github.com/STRML/react-router-component`)
- **react-mini-router** (`https://github.com/larrymyers/react-mini-router`)
- **Universal Router** (`https://www.kriasoft.com/universal-router/`)
- **router5** (`http://router5.github.io/`)
- **next.js** (`https://github.com/zeit/next.js/`)

We will use the React Router (`https://github.com/ReactTraining/react-router`) package, version *4*, the most popular option when dealing with React routing. The version *4* was released recently and includes many changes compared to the third version. We will use the fourth version to be up to date with it.
The changes done in version *4* can be seen at (`https://github.com/ReactTraining/react-router/blob/master/CHANGES.md`).

The official description for React Router is:

React Router keeps your UI in sync with the URL. It has a simple API with powerful features like lazy code loading, dynamic route matching, and location transition handling built right in.

We will start with a simple routing example done on the client side. This way, we will see how client-side routing works, and then extend it to the server.

```
// client.js

import React from 'react'
import { render } from 'react-dom'
import { BrowserRouter } from 'react-router-dom'

import App from './modules/App'

render((
```

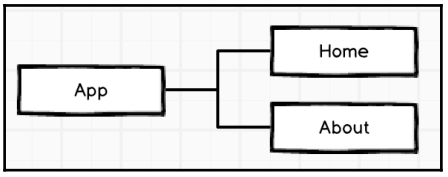
```
 <BrowserRouter >
 <App/>
 </BrowserRouter>
), document.getElementById('app'));
```

The router that our application will use is called `BrowserRouter`, this is a component that React Router exposes, and it's the router that uses the history API.

The React Router also provides other types of routers, depending on your needs from (`https://reacttraining.com/react-router/web/api/`):

- `HashRouter`: This uses the hash portion of the URL (that is, `window.location.hash`) to keep your UI in sync with the URL.
- `StaticRouter`: This is useful in server-side rendering scenarios when the user isn't actually clicking around, so the location never actually changes. It's also useful in simple tests when you just need to plug in a location and make assertions on the render output.
- `MemoryRouter`: This keeps the history of your URL in memory (does not read or write to the address bar). Useful in tests and non-browser environments.

Our routes will exist in our application file, but can be used anywhere we'd like.




```
//App.js
import React from 'react'
import {Link} from 'react-router-dom'
import {Route} from 'react-router'
import Home from './Home'
import About from './About'

export default class extends React.Component {
 render() {
   return (
   <div>
   <h1>Simple Navigation</h1>

   <nav>
     <ul>
       <li><Link to="/">Home</Link></li>
```

```
          <li><Link to="/about">About</Link></li>
        </ul>
      </nav>

      <Route exact path="/" component={Home}/>
      <Route path="/about" component={About}/>

    </div>
  )
  }
}
```

In this example you see two routes, one for the home components, with the `/` path, and another for the `About` component, with the `/about` path.
In previous versions of React Router, only `Home` would be rendered because it picks the first route that matches the path. In version *4*, both components will be rendered if the first route doesn't have the `exact` property, because it renders all the routes which apply to the path. In our current case, the `Home` component would be rendered only if the path matches exactly.

This is a simple example for client-side routing using `react-router`. As it is not our goal to explain React Router in depth, but to explain its extensibility in the server, we will explore how to manage these routes in the server.

React server rendering

React provides a way to render components to strings specially for the case of server-side rendering. This is done through the `react-dom` package, that manages the React to dom conversion. But, we don't need to just render a view, but the right view. That is where isomorphic routing comes into picture.

As we said in the beginning of the chapter, there are several options for routing in React, and we will continue using `react-router` for it.

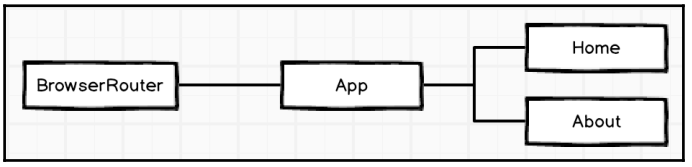
Let's go.

Rendering a view

Rendering on the server is a bit different. When we get a request, we need to figure out which is the right route and serve the resource, maybe along with initial client state data. The basic idea is that we wrap the app in a stateless `<StaticRouter>` instead of a `<BrowserRouter>`. The main difference between these two routers is that the location of the static router never changes, hence useful for our case, where we only need to compute the needed route and send the view back to the client.

We will see the client and the server's code, each one using a different router that serves its needs. The client `BrowserRouter` stays the same and the server static router handles the incoming requests routing:

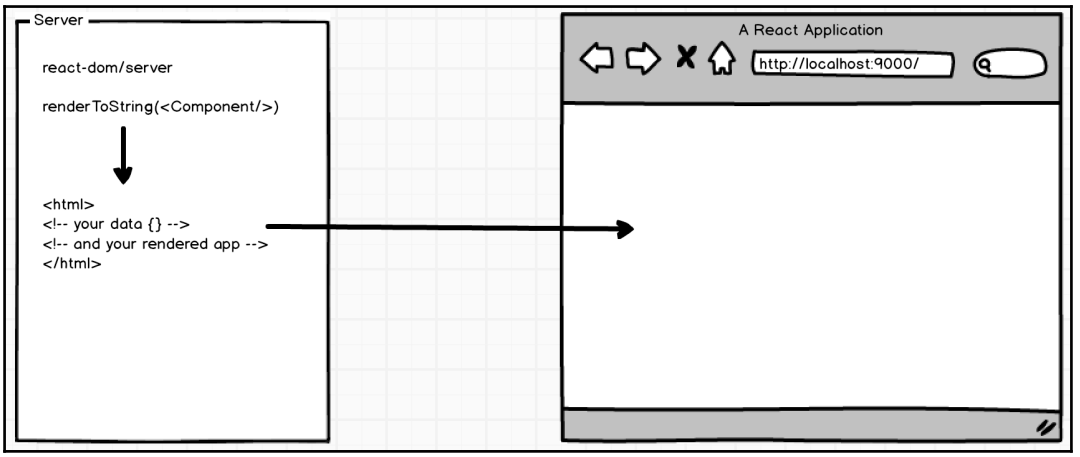
```
// --- client.js

import React from 'react'
import { render } from 'react-dom'
import { BrowserRouter } from 'react-router-dom'

import App from './modules/App'

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



Now, let's look at the server part.

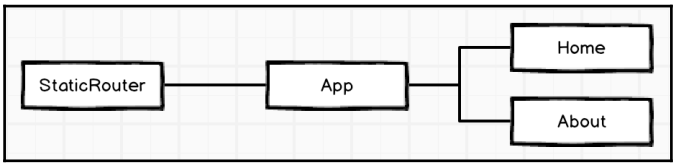
1. In the server, we map all requests (through the `app.get('*'` Express routing) to React's `renderToString` method from the `react-dom` package. You can map whatever pattern you like, instead of all GETs.
2. The component that React will render will be defined by the static router, that receives the requested URL and a context object that we will later explain.

3. The mentioned component markup will be passed to a simple function that generates an HTML with the markup in it. And that's what Express will send back to the client:




```
// --- server.js

import path from 'path';
import React from 'react'
import express from 'express'
// we'll use this to render our app to an HTML string
import { renderToString } from 'react-dom/server'
// and these to match the url to routes and then render
import { StaticRouter } from 'react-router'
import App from './modules/App'

const app = express();

// serve our static stuff
app.use(express.static(path.join(__dirname, 'public')));

app.get('*', (req, res) => {
 const context = {}
 const markup = renderToString(
 <StaticRouter
 location={req.url}
 context={context}>
 <App/>
 </StaticRouter>
 )
```

```
 res.set('content-type', 'text/html');
 res.send(renderPage(html));
});

// We are using this function that renders an HTML
// with markup passed as parameter
function renderPage(appHtml) {
 return `
 <!doctype html>
 <head>
 <title>Isomorphic Router Example</title>
 </head>
 <div id=app>${appHtml}</div>
 `
}
```



As we said, there is one thing we are not covering here, the `context` object. The context will help in the case the router triggers a redirect, for example, if the user wants to navigate to a protected URL. In these cases, we use it like this:

```
// --- server.js

// ...

app.get('*', (req, res) => {
 const context = {}
 const markup = renderToString(
 <StaticRouter
 location={req.url}
 context={context}>
 <App/>
 </StaticRouter>
 )

 if (context.url) {
 // Somewhere a `<Redirect>` was rendered
 res.redirect(302, context.url);
 } else {
 res.set('content-type', 'text/html');
 // here you can use whatever view rendering technique that you like
 res.send(renderPage(html));
```

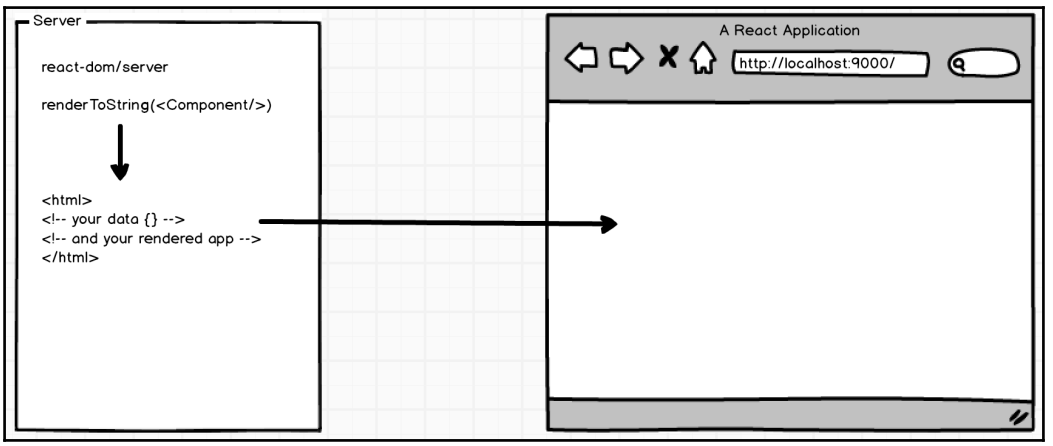
```
  }

});

// ...
```

When you render `<Redirect>` on the client, the browser history changes state and we get the new screen. In a static server environment, we can't change the app state. Instead, we use the context prop to find out what the result of rendering was. If we find a `context.url`, then we know the app redirected. This allows us to send a proper redirect from the server.
This redirect will make the router handle the next request as if nothing happened.

We saw how to send the right view to the client, depending on the requested URL, even redirecting to the right URL if needed. But, what about the initial state of the app?

Passing the state to the application

Once the markup is rendered by the browser, we still need the client state of the application, so the client side of the application can continue working transparently.
If we let the components in the client request that state to the server, then we will need an extra request for it, losing the advantage of having everything sent already cooked by the server. The whole idea of server-side rendering is improving the performance and loading time of your application, and an extra request would definitely not help. Using the initial response to send the client state would save that request:



Let's see how that can be done.

Initial state

The authors of `react-router` do not prescribe or lean toward any particular approach. There are many different approaches to this, and there's no clear best practice yet.

First, you need your routes defined separately. The routes can be defined in a separate object (or module), and the components can define a static function that will have the responsibility of fetching the necessary data for that component.

```
// routes.js
const routes = [
 { path: '/',
   component: App,
   loadData: App.loadData, //static function in the component class
   routes: [
   // child routes
     path: '/about',
     component: About
   ]
 },
 // more...
]
```

React Router exports the `matchPath` static function that it uses internally to match locations to routes. You can use this function to know which routes map to the request URL and then use the routes properties to, for example, determine what your data dependencies are and fetch them before rendering.

```
import { matchPath } from 'react-router'
import UserComponent from './UserComponent'

const match = matchPath('/users/123', {
 path: '/users/:id',
 component: UserComponent,
 exact: true,
 strict: false,
 loadData: UserComponent.loadData
})
```

But `matchPath`, at the moment, works to know if a location matches a single route, so you would need to iterate through all your routes manually and getting the necessary functions to ask the server for your data.

For more complex scenarios, you might like something more intelligent and integrated to the routing system of your application. That's were `react-router-config` comes in handy.

Using react-router-config

Another approach is using the `react-router-config` npm package, from the same authors of the `react-router`. This package exposes another function called `matchRoutes`, which will help you in matching all the right routes for a location.

A warning must be made though; the `react-router-config` package is now in version number four, but is beta and some of the children packages that we will see are also in development, so some changes may be necessary to make the following code work as intended, but hopefully not.

```
import { matchRoutes } from 'react-router-config'

const branch = matchRoutes([
 {
   path: '/users',
   component: User,
   loadData: User.loadData,
   routes: [{
     path: '/users/:id'
     component: Users,
     loadData: Users.loadData
  }]
 }
], '/users/123');

// this returns an array of the routes that apply
// [
// routes[0],
// routes[0].routes[1]
// ]
```

We are declaring the routes in an object, in a static way. The object can contain all the properties of a `route` component, and you can also add any property or function you need.

As you can see, we added the `loadData` property referencing to the static method of the component that loads the data from the server.

That function will be used in the client too, probably in the `componentDidMount` function, but only if there's no initial data, as we don't want to ask the server the initial state of the component it just sent again.

You can use this functionality in the server to get the initial data of your routes, like this:

```
import { routes } from './routes'
import { matchRoutes } from 'react-router-config'

// We define a function that will take a location and return the
// list of promises return by the loadData functions in the routes
const loadBranchData = (location) => {
 const branch = matchRoutes(routes, location);

 const promises = branch.map(({ route, match }) => {
 return route.loadData
        ? route.loadData(match)
        : Promise.resolve(null)
 });

 // we use the Promise API to return a promise
 // that will be resolved when all promises
 // are resolved
 return Promise.all(promises)
};

app.get('*', (req, res) => {

 loadBranchData(req.url).then((initialData) => {
 // The resolved "initialData" will be an array,
 // with one object for each promise resolution
 const context = {};
 const html = renderToString(
 <StaticRouter
 location={req.url}
 context={context} >
 <Root />
 </StaticRouter>
 );

 if (context.url) {
   // Somewhere a `<Redirect>` was rendered
   res.redirect(302, context.url);
 } else {
   res.set('content-type', 'text/html');
   // here you can use whatever view rendering technique that you like
   res.send(renderPage(html, initialData));
 }
 });

});
```

```
// We are using this function that renders an HTML and adds
// a global variable to the page through a <script> tag.
// That variable will be the one having your initial client state
function renderPage(appHtml, initialState) {
 return `
   <!doctype html>
   <head>
   <title>Isomorphic Router Example</title>
   </head>
   <div id=app>${appHtml}</div>
   <script type="text/javascript" charset="utf-8">
   window.__INITIAL_STATE__ = '${JSON.stringify(initialState)}';
   </script>
   <script src="/bundle.js"></script>
 `
}
```

This way, we find all the necessary `loadData` functions of the applicable routes, run them, and when we have the data ready, we create the global variable that will make it available in the client; finally, we render the found route with the `StaticRouter`.

In the client, you can take that initial state and handle it in the way that fits your application.

```
// we use the global variable in the server to get the initial state as a
JavaScript object
const initialData = JSON.parse(window.__INITIAL_STATE__);
// we delete it to make sure is not used anymore
delete window.__INITIAL_STATE__

render((
    <BrowserRouter >
    <Root initialData={initialData}/>
    </BrowserRouter>
), document.getElementById('app'));
```

In the case of having routes with sub-routes, you probably will end up passing the initial state through them, to feed the components with the passed initial data.

A component like the following might be helpful, where you pass the static defined routes, and it generates all the children routes too.

```
// wrap <Route> and use this everywhere instead, then when
// sub routes are added to any route it'll work
const RouteWithSubRoutes = (route) => (
    <Route path={route.path} render={props => (
        // pass the sub-routes down to keep nesting
```

```
            <route.component {...props} routes={route.routes}/>
        )}/>
    )
```

Then, you use it where the routes would go:

```
{routes.map((route, i) => (
    <RouteWithSubRoutes key={i} {...route}/>
))}
```

In addition, you can add the `initialData` as another prop if you need it, something like this:

```
{routes.map((route, i) => (
    <RouteWithSubRoutes key={i} {...route} {initialData} />
))}
```

This way, the sub-routes have access to the `initialData`, and you can use it to hydrate your client application (hydrate is a way of saying "give your app its state").

To sum up, with `react-router`, you get the initial state of your application depending on the route you need to render, you render the right view exposing that data as a variable for the client hence avoiding an extra request to the server.

That data should be passed to the different components that will check if it exists or not; in case it does, use it to render, if not, fetch it from the server (you can set a `initialStateLoad` variable to `true` and after loading for the first time set it to `false` for example).

How to use that initial data in the client really depends on your application. One very common pattern to use in React applications is called **redux**, which is based on having a central place for all your app states and having a unidirectional flow of changes. The state resides in what's called a Store, the views trigger changes through Actions, and the views are re-rendered through React with the new state from the store. That new state is calculated with a function called Reducer, which takes a state and an action, and returns a new state. Rinse, and repeat.

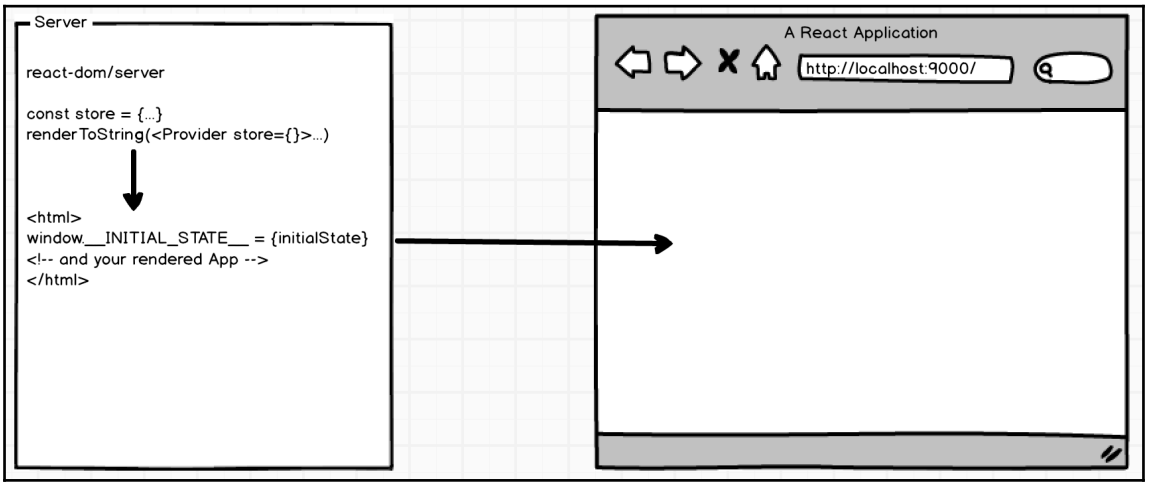
It provides a lot more than that but we will not go deep into it. But, we will see how it might help to hydrate your application state during your initial render if you choose to use it.

Using redux

Redux is a very popular architectural pattern in the React ecosystem. It's a way to manage the state of your application.

In a few words, it centralizes the application state in an object called store and uses a uni-directional pattern to React to changes in the view and model. If you haven't heard of it, I highly encourage you to take a look at the official documentation (`http://redux.js.org/`).

As the state becomes centralized in your store, the initial state of that store can be provided by the server more easily.



You wll need to create the store with the loaded data, and wrap the router with the `Provider` element (that receives the store as property).

```
//server.js
//...

import { combineReducers, createStore } from 'redux'
import { Provider } from 'react-redux'

app.get('*', (req, res) => {

    loadBranchData(req.url).then((initialData) => {
        const context = {};
        const reducer = combineReducers(/* your reducers */);
        const store = createStore(reducer, initialData);
        const html = renderToString(
            <Provider store={store}>
```

```
                <StaticRouter
                    location={req.url}
                    context={context} >
                    <Root />
                </StaticRouter>
            </Provider>
        );

        if (context.url) {
            // Somewhere a `<Redirect>` was rendered
            res.redirect(302, context.url);
        } else {
            res.set('content-type', 'text/html');
            // here you can use whatever view rendering technique that you
like
            res.send(renderPage(html, initialData));
        }
    });

});
//...
```

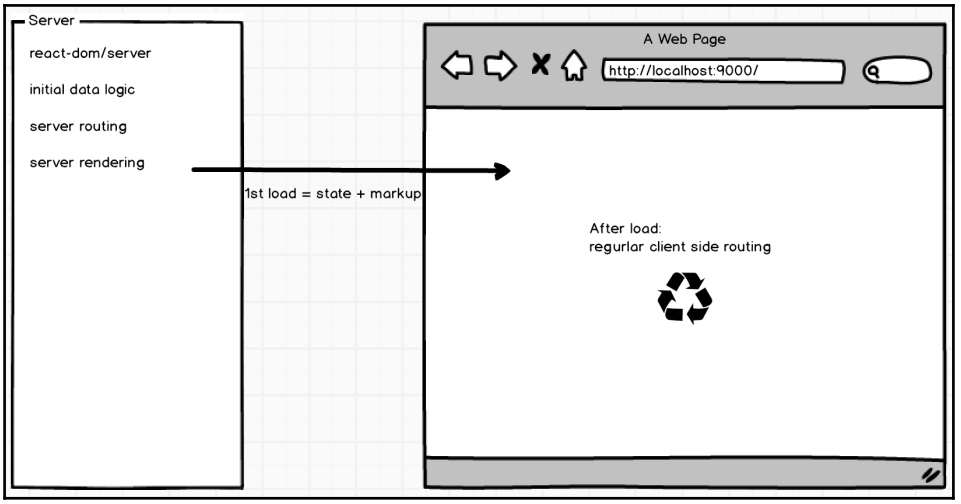
We use the same `loadBranchData` method to get the necessary information for our route and we create the store with it. You might need to massage the `initialData` to get the `store` created in the way you need it, but once that `store` is created with that data, then the way to inject it in your app is very straight forward, using the `Provider` element as always.

Then, you do the same in the client using the global variable with the initial state:

```
//client.js

//...
const initialData = JSON.parse(window.__INITIAL_STATE__);
delete window.__INITIAL_STATE__

// create your combined reducer that will manage state changes
const reducer = combineReducers(/* your reducers */);
// finally create the store that will hold your state
// the second argument is the initial state of the store
const store = createStore(reducer, initialData);

render((
    <Provider store={store}>
    <BrowserRouter >
    <Root initialData={initialData}/>
    </BrowserRouter>
```

```
    </Provider>
), document.getElementById('app'));
//...
```

In this way, hydrating your app state becomes easier and you don't need to pass the initial data through all your routes manually and let `redux` manage it for you.



Summary

We have seen how to have isomorphic routing using React Router, reusing the client routes we defined first and providing all the necessary initial data for the client application to start seamlessly. We saw how that initial data was managed without the framework and used one of the most popular state containers, redux.

This approach will unify your routing system and make your application load.

In the following chapter, we will see how to integrate authentication to our isomorphic routing, managing JWT tokens, and redirecting users if not allowed to navigate to a protected route and more.

Let's go!

8
Authentication and Authorization

This chapter gets into the intricacies of securing the application. You will learn how to implement a token-based authentication and access control for an isomorphic application using React.

Authentication and authorization are present in almost all middle to large web applications; it's a very important aspect of the functionality of your app. Since it is such an important matter, it is also very important to make it as simple as possible for you, the developer, to code and extend when needed.

You will have public routes and protected routes, routes that only authenticated users can see. You could also have routes depending on authorization, but in our case, it is just another layer of assertion. These routes will continue to be defined and used in an isomorphic way, and with the mechanisms that we will build around them, we will provide the necessary behavior to make our application work as expected in this matter.

The topics we will cover in this chapter are:

- Token-based Auth and Cookies
- What's a JWT?
- Using a JWT to manage:
    - Sign up
    - Log in
    - Request to a protected route
    - Log out
- Using a **High Order Component** (**HoC**) to manage React protected routes
- Include authorization information in the client hydration

Let's start talking about some important web application's authentication concepts before jumping to the implementation of the JWT usage.

Token-based authentication and cookies

There's a general misunderstanding when it comes to the relation between these two topics. There are people that think these two can only be used exclusively, such as token versus cookie, when both can be used together because both have different purposes.

We will see how they differ and how they can play together to help you manage the authentication of your application.

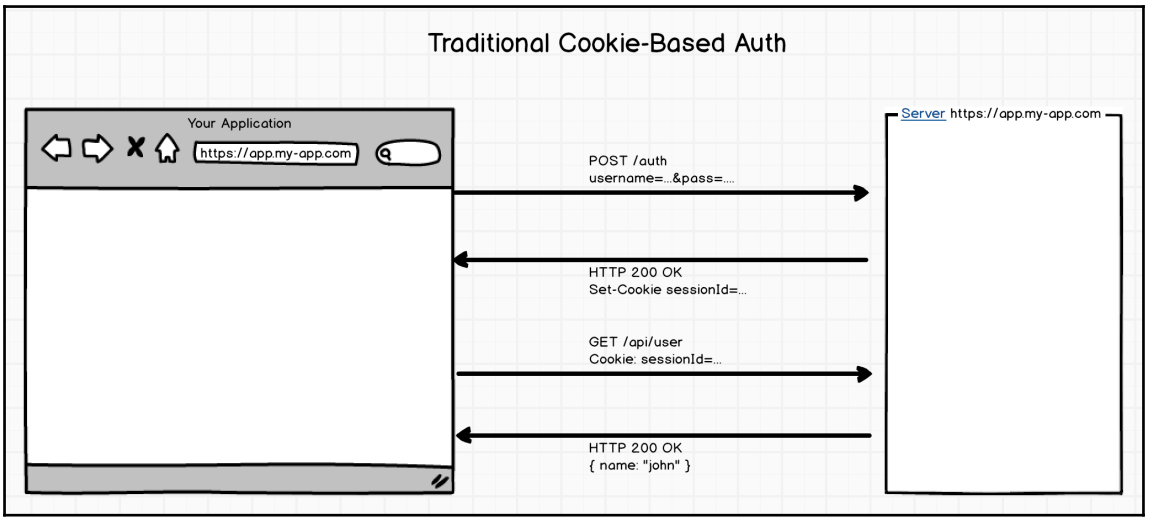
Cookies

Without going into the very basics of cookies, we will show the relevant parts of them with respect to authentication. Cookie-based authentication has been the default, tried-and-true method for handling user authentication for a long time.

Let's look at the flow of traditional cookie-based authentication:

1. The user enters their login credentials, usually username/email and password.
2. The server checks that the credentials are correct, and if the application needs a session, it creates it and stores it in a database, in memory or as a part of the following cookie (though it is the least common option).
3. A cookie with the session ID is placed in the users' browser.
4. On subsequent requests, the session ID is verified against the database, and if valid, the request is processed.
5. Once a user logs out of the application, the session is destroyed in both the client and server.

Here's a useful image to show this flow:



There are a couple of important flags that you must know if you are managing the session IDs with cookies: these are as follows:

- **httpOnly**: This means that the cookie won't be accessible through JavaScript, it is only present in the requests and hence to the server
- **secure**: The cookie will only be sent in secure requests, using HTTPS; this means that it uses SSL/TLS to protect the data of the application layer

These two flags will help you protect whatever is identifying your user's session.

Nowadays, the reason why the secure flag is a must is obvious; we must never let our session IDs fly in the open without encryption. This opens the door to **Cross-site Request Forgery** (**CSRF**) (`https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)`) attacks.

This type of attack is the most popular vulnerability for cookies; once they are stolen, they can be reused to impersonate you. But the most popular web frameworks use techniques to manage this through another piece of information called **Synchronizer** (CSRF) tokens, which helps the server to identify if it's really you who's doing the request. This approach needs the server state. Another approach that doesn't need server state is double submit cookie. A double submit cookie is defined as sending a random value in both a cookie and as a request parameter, with the server verifying if the cookie value and request value match. Again, it is rare that you need to handle this yourself; web frameworks or security libraries already provide a way to secure your application from this attack.

The `httpOnly` flag helps you prevent access to your cookies in case a malicious piece of code is able to run in the context of your application. By far, the most common way to do this is by **cross-site scripting** (**XSS**) (`https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`). But with this flag, the attacker wouldn't be able to get the cookie via JavaScript.

Another approach regarding cookies and sessions is having the session client-side in the cookie, but:

- There's a limit of the cookie size of 4 kb, and that can be a problem, if you need more space than that for your data.
- Also, one needs to be very careful what is stored there as it is visible and modifiable by the user. A good approach is to sign the cookie with a server secret; that way, the user won't be able to modify the content of the session in the cookie without you knowing (assuming that the user doesn't know your secret obviously).
- This last approach can guarantee the authenticity of the data (that it was generated by your site, and not someone else), and the integrity of the data (that it is all there and correct); it cannot guarantee freshness, that is, you are being sent back the last thing you sent to the client. This means that for some uses of session data, the cookie backend might open you up to replay attacks (`https://en.wikipedia.org/wiki/Replay_attack`). This means reusing cookies later to impersonate another user.

That being said, I wouldn't recommend this all-client-side session approach. Having the session ID in the cookie is the oldest practice and it's totally valid.

Now, let's see what the token can bring to the table.

Token-based authentication

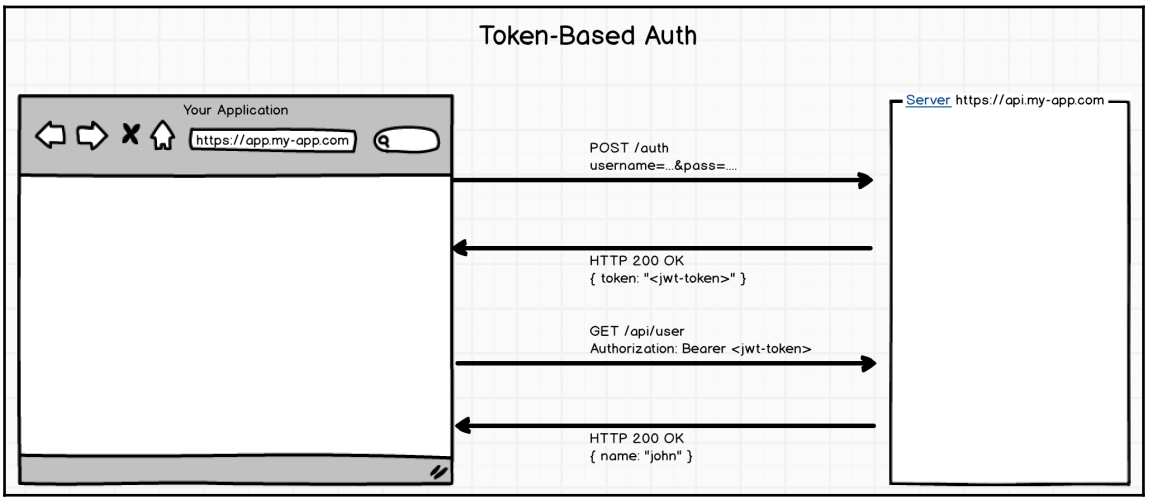
Token-based authentication has become more popular over the last few years due to the rise of **single-page applications** (**SPA**), web APIs, and the **Internet of Things** (**IoT**). When we talk about token-based authentication, we generally talk about **JSON Web Tokens** (**JWT**) (`https://jwt.io/introduction/`). Although there are different ways to implement tokens, JWTs have become the *de facto* standard.

The token is generally sent in every request to the server in an authorization header in the form of `Bearer {JWT}`, but can additionally be sent in the body of a POST request or even as a query parameter.

Let's see how this flow works:

1. Users enter their login credentials, usually username/email and password.
2. The server verifies if the credentials are correct and returns a signed token.
3. This token is stored on the client side, in a local/session storage or a cookie as well.
4. Subsequent requests to the server include this token as an additional authorization header or through one of the other aforementioned methods.
5. The server decodes the JWT, and if the token is valid, it processes the request.
6. Once a user logs out, the token is destroyed on the client side and no interaction with the server is necessary.

Here's a useful image to show this flow:



We can divide the type of JWT token as:

- **Stateless JWT:** A JWT token that contains the session data, encoded directly into the token.
- **Stateful JWT:** A JWT token that contains just a reference or ID for the session. The session data is stored on the server side.

Benefits of using a token-based approach:

- **Less overhead in requests**: No need of sending the token with all requests as with cookies, only the ones that need to be authenticated.
- **Decoupling**: You are not tied to a particular authentication scheme. The token might be generated anywhere; hence, your API can be called from anywhere with a single way of authenticating those calls.
- **CSRF**: If you are not relying on cookies, you do not need protection against cross-site requests (for example, it would not be possible for your site to generate a POST request and reuse the existing authentication cookie because there will be none).
- **Standard based**: Your API could accept a standard JSON Web Token (JWT) (`http://tools.ietf.org/html/draft-ietf-oauth-json-web-token`). This is a standard and there are multiple backend libraries (.NET (`https://www.nuget.org/packages?q=JWT`), Ruby (`https://rubygems.org/search?utf8=%E2%9C%93query=jwt`), Java (`https://code.google.com/archive/p/jsontoken/`), Python (`https://github.com/davedoesdev/python-jwt`), PHP (`https://github.com/firebase/php-jwt`)) and companies backing their infrastructure (for example, Firebase (`https://www.firebase.com/docs/web/guide/login/custom.html`), Google (`https://developers.google.com/identity/protocols/OAuth2ServiceAccount#overview`), and Microsoft).

There are also the following disadvantages:

- Programmatically, add the header to all necessary requests.
- If it is stored in the browser local/session storage, then it's accessible through JavaScript. This opens the door for stealing it in the case of an XSS attack. If it's stored in a cookie, as mentioned, CSRF protection needs to be added.
- It can be too big, if the payload is abused.
- The data in a stateless token can eventually go stale, and no longer reflect the latest version of the data in your database, that is, somebody might have a token with a role of the admin, even though you've just revoked their admin role. Because, the token is still the same.
- You are essentially powerless when dealing with stateful tokens, as you cannot kill a session without building a stateful infrastructure to explicitly detect and reject them, defeating the entire point of using stateless JWT tokens to begin with.

Tokens are a way to securely validate, identify, and store session data optionally. Cookies are a way of transporting it to/from the server. As we said, the browser's local storage and session storage are also options (using the authorization header to send it to the server).

We will use a stateless JWT token for this chapter because managing sessions in the server is not our goal in this chapter.

But what is a JWT really?

JSON Web Token (JWT)

We will start talking about JWT. JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA.

The two main properties of this standard are:

- **Compact**: Due to their small size, JWTs can be sent through a URL, POST parameter, or inside an HTTP header. But the payload can be as big as you want, so you have to have that in mind.
- **Self-contained**: The payload contains all the required information. This can help avoid the need to query the database for that information, optionally making the session stateless, as everything is in the token payload.

This token generation is something we must not do manually, but use a proven library that is capable of generating it. This is almost always a rule when using cryptography for security in your application. Good algorithms for cryptography are super difficult to create and should be left to specialists.

A good node library for generating this token is `jwt-simple`.

Using jwt-simple

The hello world example for this library is something like this:

```
var jwt = require('jwt-simple');
var payload = { foo: 'bar' };
var secret = 'xxx'; // NEVER REVEAL THIS!
```

```
// encode
var token = jwt.encode(payload, secret);

// decode
var decoded = jwt.decode(token, secret);
console.log(decoded); //=> { foo: 'bar' }
```

In this simple example, we will highlight three concepts that we will revisit during the rest of the chapter.

First, the payload. The token accepts what are called claims:

- **Registered claims:** These are possible values of the payload defined by the JWT Specification; we will see some of the possible values while creating the token for our application. These claims have a key and a purpose defined already.
- **Public claims:** These are keys defined by you, but should:

  > Either be registered in the IANA JWT Claims registry [...] or be a public name: a value that contains a collision-resistant name. In each case, the definer of the name or value needs to take reasonable precautions to make sure they are in control of the part of the namespace they use to define the claim name.
  >
  > (`http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html#PublicClaimName`)

- **Private claims:** These are also keys defined by you, but they don't guarantee to be a collision-resistant name.

We will use `jwt-simple` to generate and validate a JWT token and as a part of the payload we will use some registered claims.

An important note here, the secret. Your token is generated based on an application-based secret. This means that what makes a token yours and decodable by you is your secret. Never publish it, make it secret (duh), have it in a separate file and unversioned. We will talk about this secret a little bit more later.

Another option in the library is to define the algorithm to use to encode and decode the token. The options are:

- HS256
- HS384
- HS512
- RS256

The HMAC (HS*) family of algorithms are probably the most common algorithm for signed JWTs. In this library, HS256 is the default algorithm used. It's not one of the strongest algorithms, but it is not in the scope of this book to analyze all possible algorithms, we want simplicity for now. If you are interested in a more in-depth explanation of them, you can check at `https://auth0.com/blog/json-web-token-signing-algorithms-overview/`.

The specs define many more algorithms for signing. You can find them all in RFC 7518 (`https://tools.ietf.org/html/rfc7518#section-3`).

Enough theory about the token, let's look at how to authenticate a user in the server using it.

Server authentication

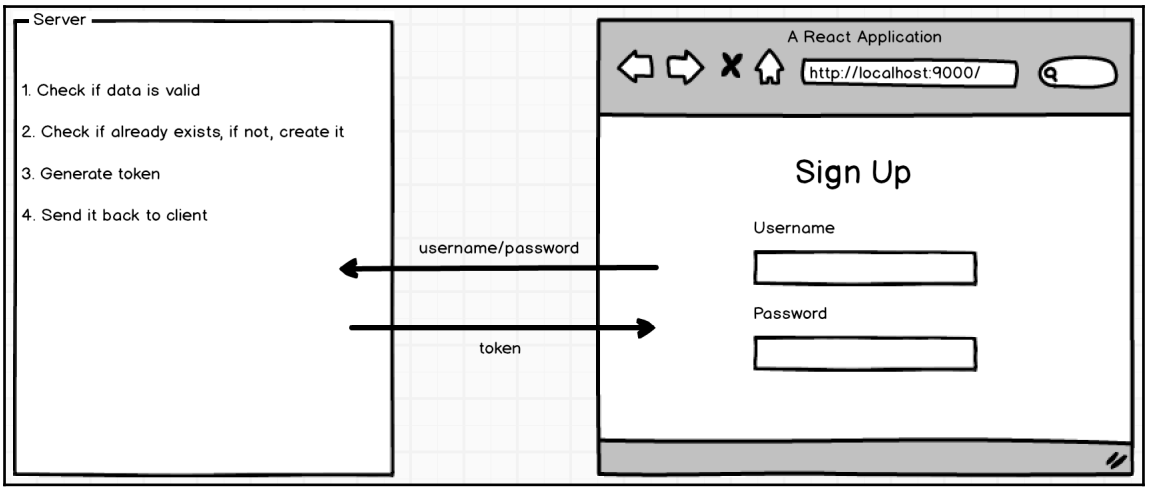
We will use Express in our server and `react-router` and redux in the client, as we did with examples in the previous chapter. We will cover:

- Users sign up
- Log in/out
- Protecting routes
- Handling navigation to protected routes without being logged in

Sign up

Signing up is the process of adding a new user to your system, to recognize that user in the future and provide him with exclusive permissions that non-signed-up users don't have. It's a super popular use case and a very important one in the web applications.

The basic idea of our sign up process will be like this:



In this way, we will add them to our database (not in the scope of the chapter) and then return a JWT token to continue using our application to the user.
For signing up, we will receive the user and password from the client. In this example, we will map the POST call to `/signup` to this functionality.

We will do it this way:

```
import express from 'express';
import bodyParser from 'body-parser';
import Authentication from './controllers/authentication';

const app = express();
app.use(bodyParser.json());

//...

app.post('/signup', Authentication.signup);

//...
```

Okay, that seemed simple enough.

One comment on the `bodyParser` package, that we see in the example, is used to parse the body from the request, in our case with the user information. It's express middleware, meaning that it will intercept the request and transform the body information into a JavaScript object in `req.body`.

This module also provides the following parsers:

- JSON body parser (`https://www.npmjs.com/package/body-parser#bodyparserjsonoptions`)
- Raw body parser (`https://www.npmjs.com/package/body-parser#bodyparserrawoptions`)
- Text body parser (`https://www.npmjs.com/package/body-parser#bodyparsertextoptions`)
- URL-encoded form body parser (`https://www.npmjs.com/package/body-parser#bodyparserurlencodedoptions`)

You can use the parser for different types of payloads. Let's move to signing up the user.

The signup functionality works as follows:

```
// controllers/authentication.js

import jwt from 'jwt-simple';
import {secret} from './secretFile';

// this function takes a user and generates a token for it
function tokenForUser(user) {
 const timestamp = new Date().getTime();
 // "iat" (issued at) claim identifies the time at which the JWT was
issued.
 // "sub" (subject) claim identifies the principal that is the subject of
the JWT.
 return jwt.encode({ sub: user.id, iat: timestamp }, secret);
}

exports.signup = function(req, res, next) {
 // we get the necessary data from the request
 const email = req.body.email;
 const password = req.body.password;

 if (!email || !password) {
  // in case there's something wrong with the user/pass, we return the
right HTTP status, 422: Unprocessable Entity
  return res.status(422).send({ error: 'You must provide an email and
password'});
 }
 // If a user with email does NOT exist => create and save user record
 const user = {
 email: email,
 password: password
 };
```

```
    // save user!
    // no implementation for example simplification

    // Respond to request indicating the user was created
    res.json({ token: tokenForUser(user) });
  });
```

Although we skipped the part interacting with the user model persistence, we can see that the logic for creating a token is pretty simple, thanks to `jwt-simple` and the express middleware capabilities.

We can also see that the token contains a couple of properties: `iat` and `sub`. Although we explain them in the code, let's take a look at what they mean and at other possibilities for the token data.

JWT token claims

In this example, we have first defined the function that will generate the JWT token from the user name, using the `sub` and `iat` properties of JWT and the secret.
These properties are two among many properties that the JWT specification (`https://tools.ietf.org/html/draft-ietf-oauth-json-web-token-32#section-4.1`) defines. To name a few:

- `sub`: The `sub` (subject) claim identifies the principal that is the subject of the JWT. The Claims in a JWT are normally statements about the subject.
- `iat`: The `iat` (issued at) claim identifies the time at which the JWT was issued.
- `iss`: The `iss` (issuer) claim identifies the principal that issued the JWT. The processing of this claim is generally application specific.
- `exp`: The `exp` (expiration time) claim identifies the expiration time on or after which the JWT *must not* be accepted for processing.
- `nbf`: The `nbf` (not before) claim identifies the time before which the JWT *must not* be accepted for processing.

More claims are available at the official specification, these are some of the most popular ones.

All these claims are optional, but help you in your application authentication logic. Pick the ones you think your application needs.

Now, let's revisit another important part of the encoding/decoding process of your token, the secret. We mentioned it earlier emphasizing its importance, but let's look at the minimum requirements it needs to have.

JWT token secret

The secret used in the token is your application secret that:

- Needs to be hard to guess
- Should never be public or visible to everyone
- Should not be exposed to anyone who shouldn't see it in your org

That's why a separate file is needed to store it; this makes it easier to manage its visibility.

After validating this secret, we validate and check the provided username and password, and finally create a new user model in the persistence layer. This last step was not implemented for the sake of the example's simplicity.

Finally, we send the new token based on the new user to the client. That token will be the proof that the user belongs to our application and it's logged in. That token will be used in the future to protect our routes against unauthenticated requests.

The persistence of that token is left to your discretion, knowing the pros/cons explained before in this chapter (cookies versus local storage). I would personally go with the cookie approach, but is up to you.

We have seen the steps that makes the sign up functionality in our application; they are as follows:

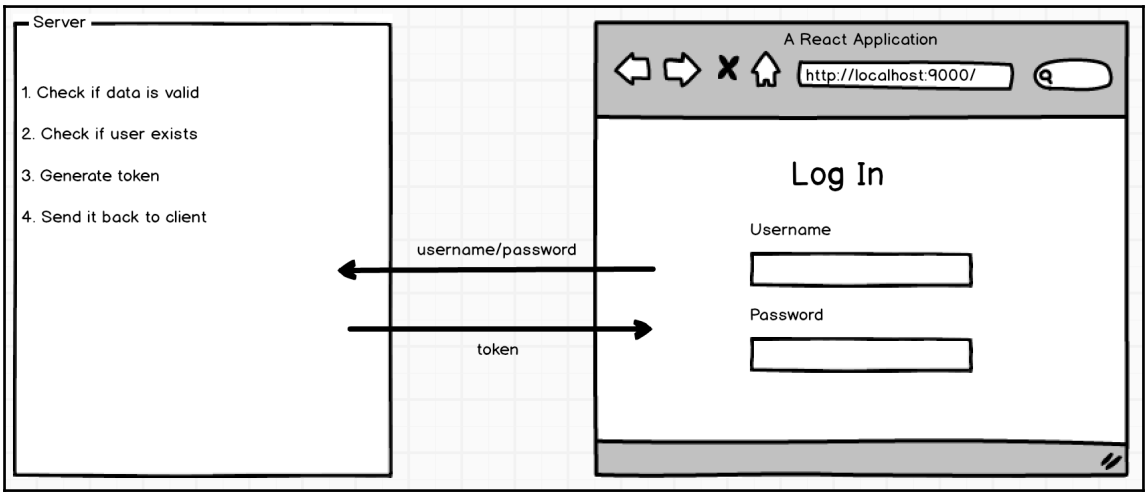
- Using `bodyParser` to take the information sent from the client
- Creating a JWT token using `jwt-simple`, the JWT-registered claims, and our private application token
- Simulating the creation of the user model
- Returning the token that will identify the user in the next requests

Now, it's time to check how a user can log in to our application.

Log in

Let's consider the use case where the user wants to log in. He's already signed up and he wants to access our application.

The basic idea of our login process will be like this:



This case is not far from the logic we used to sign up the user. We need to grab the provided user and password, check if they exist in the users, database, and in case it exists, return the token to that user. It is the token that will be received in the following requests from that user:

```
const jwt = require('jwt-simple');
const secret = require('./secretFile').secret;

// the function that returns a token from a user id
function tokenForUser(user) {
 const timestamp = new Date().getTime();
 return jwt.encode({
 sub: user.id,
 iat: timestamp
 }, secret);
}

exports.signin = function(req, res, next) {
 const email = req.body.email;
 const password = req.body.password;

 if (!email || !password) {
   // in case there's something wrong with the user/pass, we return the
right HTTP status, 422: Unprocessable Entity
   return res.status(422).send({
    error: 'You must provide an email and password'
   });
```

```
  }

  // Check e-mail and password against database

  if( /* user exists */ ) {
   //send the token that will be used for the following requests
   res.send({ token: tokenForUser(req.user) });
  } else {
 //not in our database, 401: Unauthorized
   res.status(401).send({
     error: 'error: username/password does not match an existing user'
   });
  }
 }
```

As we said, in this case, we again receive the username and password from the client. We check them against our database of users. If we find it, we generate a token and send it back to the client, meaning that the user is authenticated.

If we don't find the user/password, we return a `401: Unauthorized` message. This message should be handled in the client, showing a message with the classic user or password invalid.

Remember to not say what part of the data is invalid like password not found or user not found, as this provides the user with information that can be used to find existing users and try to engineer their passwords. Also, limit the amount of possible attempts to avoid brute-force attacks.

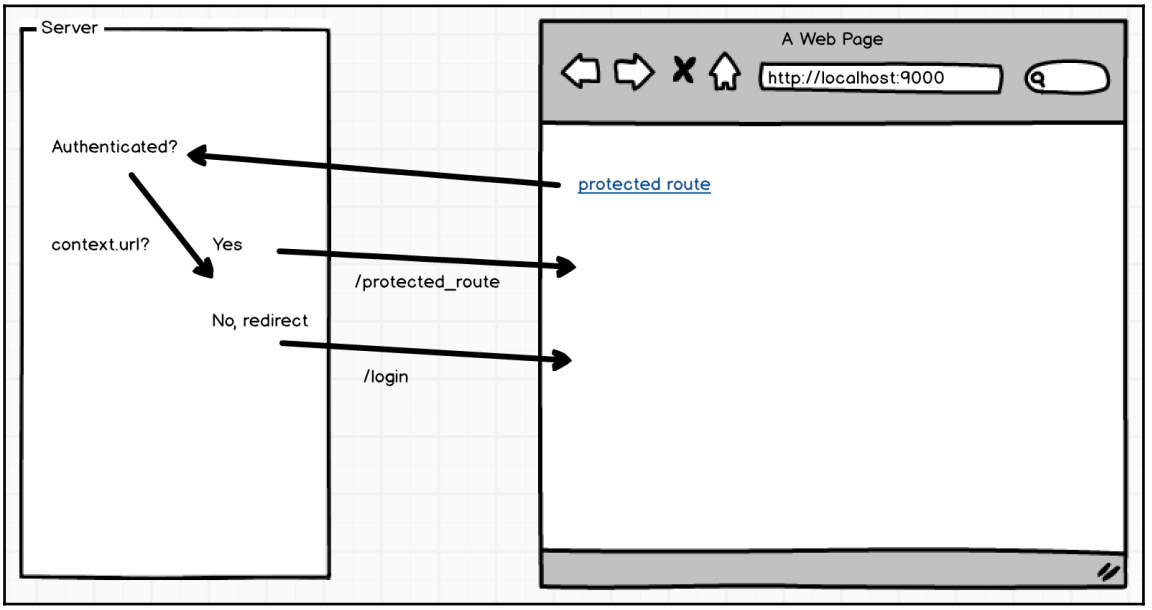
These are a couple of important practices around the log in that you need to take into account; I encourage you to do a little research about it to be sure you cover all of them. Password recovery and other ways of authentication are a couple of other topics to be researched.

Now, let's move to routing redirection; this happens when the user is found to not be authenticated, either because there's no token or the token is invalid.

Let's go!

Routing redirection

We have seen, where we set up server-side rendering, that a branch of our server logic checked for the presence of a `context` object, in the previous chapter; if that `context` existed, then a redirection was found:



That is a mechanism for `react-router` to indicate that a redirection component was rendered while rendering the application. That component will be useful, as it will let us know when we need to redirect the user to our log in page for example.

Remember?

```
import jwt from 'jwt-simple';
import secret from './secretFile';
import { createStore } from 'redux'
import { Provider } from 'react-redux'

app.get('*', (req, res) => {
 const context = {};
 // we make up a reducer that takes and returns the same state
 const store = createStore(state => state, {});
 const html = renderToString(
 <Provider store={store}>
 <StaticRouter
 location={req.url}
```

```
 context={context} >
 <Root />
 </StaticRouter>
 </Provider>
 );

 if (context.url) {
 // Somewhere a `<Redirect>` was rendered
 res.redirect(302, context.url);
 } else {
 res.set('content-type', 'text/html');
 // here you can use whatever view rendering technique that you like
 res.send(renderPage(html, initialData));
 }
});
//...
```

We passed `StaticRouter` a context object, which the router will populate if a `Redirect` is rendered in the process of rendering the components.

In the following sections:

- We will first see how we can use the concept of high order components (HoC) to facilitate the rendering of a component or redirection to a safe route
- Inject that authentication data into the server-side view so that the React Router can handle the routing accordingly
- And finally, send the right view to the client

Authenticating high order component

For protecting private routes in our application, we will take a clean and reusable approach; we will create a high order component that will wrap our protected routes.

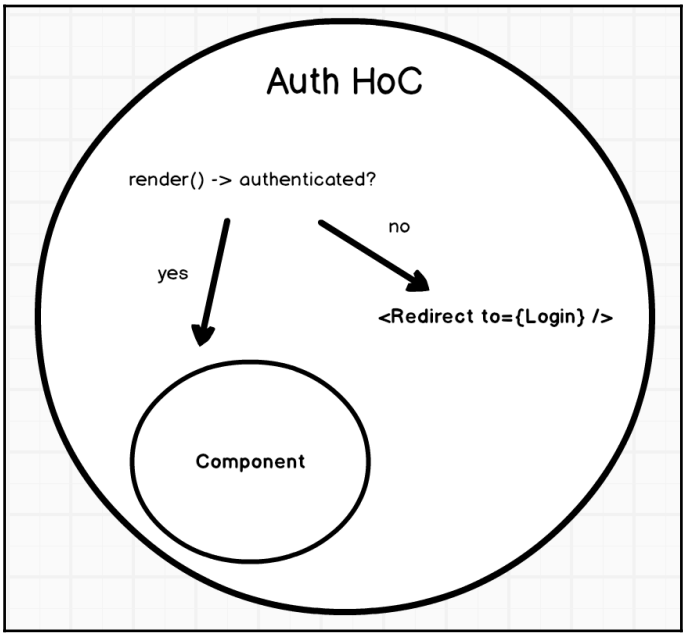
In this example we will:

- Use redux for state management, using its redux store
- Use an `authenticated` flag in our redux store, indicating if the user is authenticated or not:

```
import { Component } from 'react';
import { connect } from 'react-redux';
import { Redirect } from 'react-router-dom'
```

```
export default function(ComposedComponent) {
class Protected extends Component {
 render() {
   if (!this.props.authenticated) {
     return <Redirect to="/login"/>
   } else{
     return <ComposedComponent {...this.props} />
   }
  }
 }

 function mapStateToProps(state) {
   return { authenticated: state.auth.authenticated };
 }

 return connect(mapStateToProps)(Protected);
}
```

The main purpose of our component is to, when rendering, check if the user is authenticated using the information from the store. If it is, then render the component; if it is not, then render a `<Redirect />` element specifying where to redirect, probably to a login page:



With this high order component, your routes end up being something like this:

```
//App.js
import React from 'react'
import {Link} from 'react-router-dom'
import {Route} from 'react-router'
import Home from './Home'
import About from './About'
import Protected from './Protected'

export default React.createClass({
 render() {
   return (
     <div>
     <h1>Simple Navigation</h1>

    <nav>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
      </ul>
    </nav>

    <Route exact path="/" component={Home}/>
    <Route path="/protected" component={Protected(About)}/>

    </div>
   )
 }
})
```

If you define your `routes.js` like this, it should be as follows:

```
// routes.js
import App from './App'
import Protected from './Protected'

const routes = [
 { path: '/',
   component: App,
   loadData: App.loadData, //static function in the component class
   routes: [
     // child routes
     path: '/protected',
     component: Protected(About)
   ]
 },
```

```
 // more...
]
```

Just with this change, if a user wants to go to `/about` clicking on a `Link` without being logged in, the `Redirect` is rendered in the client, triggering a redirection to the safe route, the log in page.

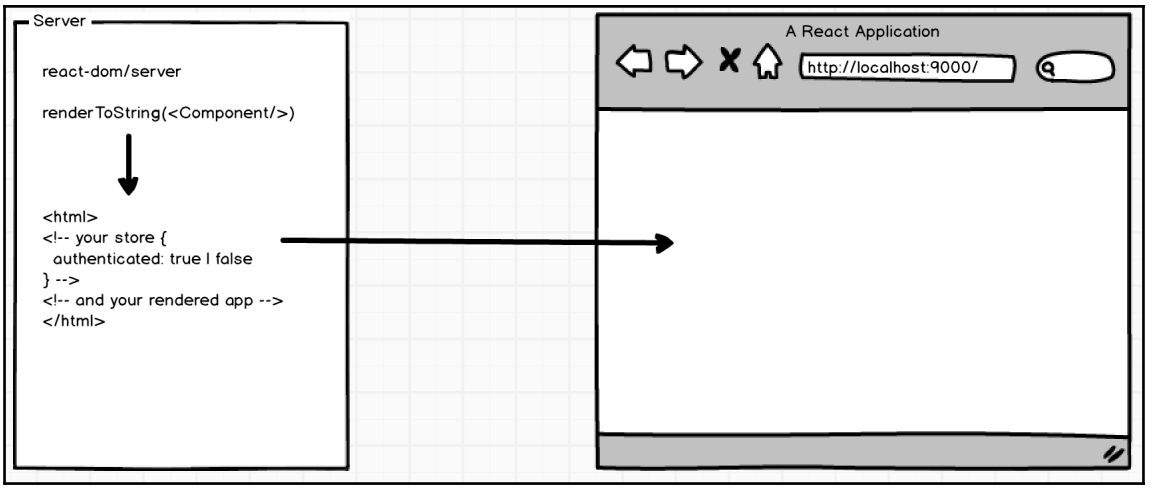
The `Redirect` element has more useful arguments:

```
// "push": when true, redirecting will push a new entry onto the history
// instead of replacing the current one
<Redirect push to="/login" />

// "from": A pathname to redirect from. This can only be used to match
// a location when rendering a <Redirect> inside of a <Switch>. Like this:
<Switch>
 <Route exact path="/" component={Home}/>

 <Route path="/users" component={Users}/>
 <Redirect from="/accounts" to="/users"/>

 <Route component={NoMatch}/>
</Switch>

// "to" can be an object too, a router location.
<Redirect to={{
 pathname: '/login',
 search: '?utm=your+search',
 state: { referrer: currentLocation }
}}/>
```

We covered how to protect routes and what happens when a user clicks on a link that takes him to a protected route. But, what if the user types the URL of a protected route? That's where the server-side check comes into picture.

Server-side authentication check

Now, let's check if the user is authenticated and set the right authentication flag in the initial store state created in the server:



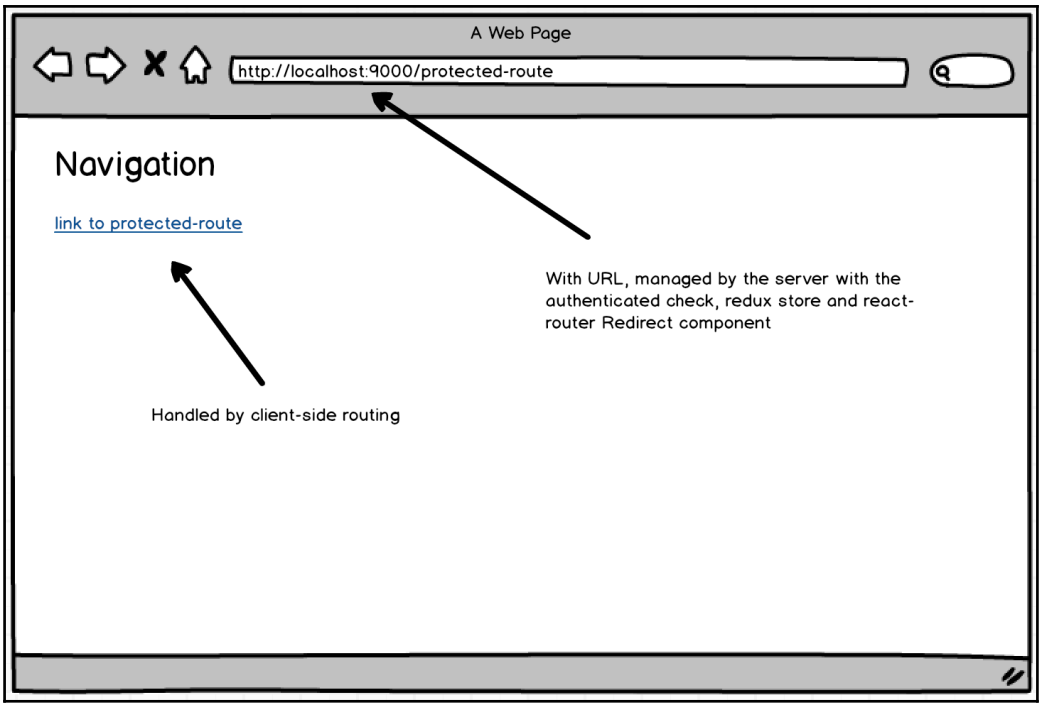
For this, we will use our `jwt-simple` package and the provided (or not) user token again:

```
import jwt from 'jwt-simple';
import { secret } from './secretFile';
import { createStore } from 'redux'
import { Provider } from 'react-redux'

// this function will check if the provided token is valid or not
function isLoggedIn(token) {
 try {
   const decodedToken = jwt.decode(token, secret);
   const username = decodedToken.userId;
   // check if userId exists in our database

   return /* does it exist? */;

 } catch(e) {
   // invalid token
   return false;
 }
}

app.get('*', (req, res) => {
 const context = {};
 // we make up a reducer that takes and returns the same state
 const store = createStore(state => state, {
 auth: {
   // the initial state contains if the user is authenticanted or not
   authenticated: isLoggedIn(req.headers['authorization'])
 }
```

```
  });
  const html = renderToString(
    <Provider store={store}>
      <StaticRouter
      location={req.url}
      context={context} >
        <Root />
      </StaticRouter>
    </Provider>
  );

  if (context.url) {
    // Somewhere a `<Redirect>` was rendered
    res.redirect(302, context.url);
  } else {
    res.set('content-type', 'text/html');
    // here you can use whatever view rendering technique that you like
    res.send(renderPage(html, initialData));
  }
});
//...
```

With these additions, we take the user token, validate it against our secret and database, and if the token is valid, we create the initial redux store state with the authentication flag in true.

If that flag is true, then the right component should be rendered. If the flag is false and the request requests to navigate to a protected route, then the React Router (through our authentication high order component) should render a Redirect component. This creates the context object and makes the server redirect the user to the safe route with an `HTTP 302` message:



Log out

Now, the last big use case missing is logging out.

When logging out, we need to set the authentication flag in our store to false and delete the token from the client-side persistence, wherever that is, in a cookie or in local storage. As a token is by default a stateless authentication mechanism, the only way to invalidate it is by not using it anymore or by the expiration claim. Deleting it makes it disappear, and in the next log in, we would get a new one.

The other approach would be to have a set of valid tokens in some DB (this would make it stateful) and invalidate it there so that if it is used in the future, then we can reject it. But, this brings another set of complications, mainly because of its statefulness. We won't cover this topic in this chapter, but it is important for you to think about these things in case your application needs it.

With this, the main lifecycle is closed.

Summary

In this chapter, we have seen how you can manage, in an isomorphic way, the authentication of a user, covering the most popular cases like sign in, login, and authenticated and not authenticated requests. Almost all complex systems need to cover these use cases; that's why I think it's useful to cover them in this chapter and book.

In the next chapter, we will talk about testing and deployment. Testing is one of the most important topics of any mature application, no matter if it's web-based, terminal-based, or a console game. You must have a set of tests covering your application use cases, letting you develop with confidence and at a faster pace. We will then explore some concepts that you need to have in mind when preparing your application for production time and some online platforms where you can deploy your application in a simple and elastic way.

Enough, let's go!

9
Testing and Deploying Your App

In this chapter, we will cover two topics: tests and deployment. Although they are covered in the last part of the book, they should not be considered to be the last things to think about when developing an application.

For example, **TDD** is a software development process that relies on the repetition of a very short development cycle: requirements are turned into test cases, then the software is improved to pass those tests. This means that, in this case, tests are done first while developing a feature.

Deployment and production practices are key to the successful life of the application. It is preferable that your application is not tied to a particular deployment environment and it is externally configured to run on it. Also, it is very important to think about how your application will behave in production; you need to think about the stability, availability, and recoverability of your application. These things, and more are managed from within your code, and we will see some good practices around them.

Tests and deployment

We will see how to test your application in three types of tests and how to deploy your application in the cloud, along with some necessary practices for your application to behave in a robust way when it's live.

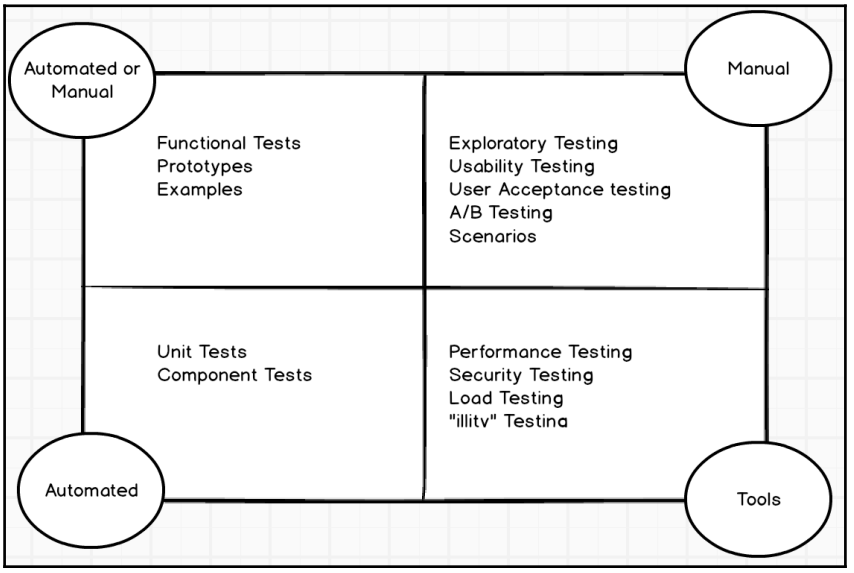
Tests

Testing is a vital part of the development of any serious application. Isomorphic applications do not differ from this idea.

Testing comes in different forms, some of them are:

- Unit testing
- Integration testing
- Functional testing
- System testing
- Stress testing
- Performance testing
- Usability testing
- Acceptance testing

We can also represent them in a diagram as follows:



Although we won't cover all of these in this chapter, we will cover Unit testing, functional testing, and integration testing. Integration testing could be included in the same quadrant as functional testing, as it is also functional but between components or systems.

It's not that the ones we are covering are the most important ones or anything like that, but they are the most popular types of tests that developers write. Others, as the diagram represents, run by tools or manually by people.

Let's start testing!

Unit testing React with Mocha, Sinon, Chai, jsdom, and Enzyme

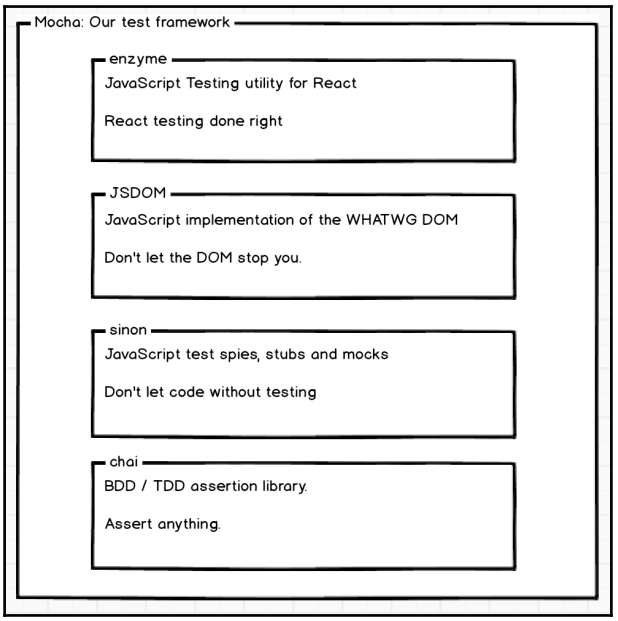
The first line of tests for an application is the unit tests. This kind of tests checks that your components, in isolation, and work as expected.

Unit testing is the testing of an individual unit or a group of related units. It falls under the class of white box testing. It is often done by the programmer to test that the unit he/she has implemented is producing the expected output against the given input.

Functional testing is the testing to ensure that the specified functionality required in the system requirements works. It falls under the class of black box testing.

The ecosystem for testing in JavaScript is very big (like everything in JavaScript at the moment). We will use one of the most popular combinations of tools for this. All these tools, except for Enzyme, are not React-specific, so they can be reused with other application frameworks.

This is an image to have in mind while reading about the tools we will use:



Use it if you suddenly forget what each tool does. Let's start going a little bit deeper into them.

Mocha

Mocha is a JavaScript test framework running on Node.js and in the browser. Mocha executes your tests, provides two styles to write them (BDD and TDD), offers async tests functionality, and much more:

- Browser support
- Test coverage reporting
- JavaScript API for running tests
- Maps uncaught exceptions to the correct test case
- Async test timeout support
- Test retry support
- Test-specific timeouts
- Highlights slow tests
- File watcher support

- Global variable leak detection
- Optionally run tests that match a regexp
- Node debugger support
- Extensible reporting, bundled with 9+ reporters
- Arbitrary transpiler support (coffee-script)

From `https://mochajs.org/`.

You install `mocha` like this:

```
npm install mocha --save-dev
```

The hello world test for Mocha would be a file with something like this:

```
var assert = require('assert');
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      assert.equal(-1, [1,2,3].indexOf(4));
    });
  });
});
```

Mocha allows you to use any assertion library you wish. In the preceding example, we're using Node.js' built-in `assert` module (`https://nodejs.org/api/assert.html`), but you can use other libraries such as Chai, expect, and others.

- **BDD:** The preceding example was written using the BDD syntax. It's one way to define your suites, tests, hooks, and more.

  The BDD interface provides `describe()`, `context()`, `it()`, `specify()`, `before()`, `after()`, `beforeEach()`, and `afterEach()`.
  It is meant to be more business friendly, but it does the same as the following style, the TDD style.

- **TDD:** TDD is similar to BDD, but the organization is set around suites and tests. The TDD interface provides `suite()`, `test()`, `suiteSetup()`, `suiteTeardown()`, `setup()`, and `teardown()`.

  This one is more similar to other tools that you might use for unit testing. Now, let's move to the `assertion` library that we are going to use, Chai.

Chai

Chai is a very popular BDD / TDD `assertion` library for Node.js and the browser that can be paired with any JavaScript testing framework. Its flexibility and pluggability are the main qualities why we choose it for our tests.

Here's how to install it:

```
npm install chai --save-dev
```

Chai has several interfaces that allow the developer to choose the most comfortable. Again, the chain-capable BDD styles provide an expressive language and readable style, while the TDD assert style provides a more classical feel.

- `should`

```
chai.should();
foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.lengthOf(3);
tea.should.have.property('flavors')
 .with.lengthOf(3);
```

- `expect`

```
var expect = chai.expect;
expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(tea).to.have.property('flavors')
 .with.lengthOf(3);
```

- `assert`

```
var assert = chai.assert;
assert.typeOf(foo, 'string');
assert.equal(foo, 'bar');
assert.lengthOf(foo, 3)
assert.property(tea, 'flavors');
assert.lengthOf(tea.flavors, 3);
```

Examples from `http://chaijs.com/`.

As you can see, it gives you three different styles (`expect` and `should` are BDD) and many ways of asserting your data with expected values. This is why we will use Chai. It also has a plugin ecosystem where you can use specific packages to assert other types of data or behavior, such as `chai-as-promised` or `chai-json` that help assert promises and JSON data, respectively. More exist for different use cases.

Now, we may need something else if we want mock objects or services, and spy functions executions to make sure that they are called and they are called the right amount of times.

That's where Sinon comes into the picture.

Sinon

Standalone and test framework agnostic JavaScript test spies, stubs, and mocks.

Some of its goals:

- Easy to embed seamlessly with any testing framework
- Easily fake any interface
- Ship with ready-to-use fakes for `XMLHttpRequest`, timers, and more

Description from `http://sinonjs.org/`.

Install it with:

```
npm install sinon --save-dev
```

As described earlier, Sinon provides several functionalities; just to show a couple, we will write a couple of examples for mocks and spies.

- **Mocks:** Mocks (and mock expectations) are fake methods (such as spies) with pre-programmed behavior (such as stubs) as well as pre-programmed expectations.

  A `mock` will fail your test if it is not used as expected:

```
"test should call all subscribers when exceptions": function ()
{
    var myAPI = { method: function () {} };

    var spy = sinon.spy();
    var mock = sinon.mock(myAPI);
    mock.expects("method").once().throws();
```

```
        PubSub.subscribe("message", myAPI.method);
        PubSub.subscribe("message", spy);
        PubSub.publishSync("message", undefined);

        mock.verify();
        assert(spy.calledOnce);
    }
```

- **Spies:** A test spy is a function that records arguments, returns values, the value of this and exception were thrown (if any) for all its calls. There are two types of spies: some are anonymous functions, while others wrap methods that already exist in the system under test.

```
    "test should call subscribers on publish": function () {
        var callback = sinon.spy();
        PubSub.subscribe("message", callback);

        PubSub.publishSync("message");

        assertTrue(callback.called);
    }
```

Enough of tools! (We will cover Enzyme and jsdom while testing).

Let's start testing!

Testing

To unit test our React application, we have a pre-requisite: we need to mock the window and document. As Mocha does not run our tests in the browser and React/JQuery assumes we are in the browser environment, we need to mock that environment. To do this, we need a library called **jsdom**, a JavaScript implementation of the **WHATWG DOM** and HTML standards, for use with Node.js:

```
import {JSDOM} from 'jsdom';

// Setup testing environment to run like a browser in the command line
const dom = new JSDOM('<!doctype html><html><body></body></html>');
global.window = dom.window;
global.document = dom.window.document;
```

As you can see, `JSDOM` creates a `dom` object that will serve as your real DOM. We set the global `window`/`document` variables to the ones created by jsdom.

This can be integrated into your tests as follows:

```
import {JSDOM} from 'jsdom';
import jquery from 'jquery';

// This function can be taken out to another file for better modularity
function setupTests() {
  const dom = new JSDOM('<!doctype html><html><body></body></html>');
  global.window = dom.window;
  global.document = dom.window.document;
}

suite('test suite', function () {
  setupTests();
  test('simple test', function () {
    TestUtils.renderIntoDocument(
      <About />
    );
    // nothing yet
  })
});
```

Let's test something real:

```
import {JSDOM} from 'jsdom';
import jquery from 'jquery';

// This function can be taken out to another file for better modularity
function setupTests() {
  // ...
}

suite('test suite', function () {
  setupTests();
  test('simple test', function () {
    const component = TestUtils.renderIntoDocument(
      <About />
    );
    const renderedComponent = ReactDOM.findDOMNode(component);
    expect(renderedComponent.textContent).to.equal('About');
  })
});
```

In this case, we are rendering the `About` component into the page, then using the `ReactDOM` package to get the `dom` node from the component, and finally checking whether the text of that element is what we expected. Obviously, you can test whatever you want from your `dom` node.

If you feel more comfortable with JQuery, you can use `chai-jquery` (remember Chai plugins?) to assert things in that `node`:

```
import TestUtils from 'react-dom/test-utils';
import About from '../components/About'
import React from 'react';

import ReactDOM from 'react-dom';
import chai, {expect} from 'chai';
import chaiJquery from 'chai-jquery';
import {JSDOM} from 'jsdom';
import jquery from 'jquery';

function setupTests() {
  const dom = new JSDOM('<!doctype html><html><body></body></html>');
  global.window = dom.window;
  global.document = dom.window.document;
  //we need to tell jquery to use our jsdom window
  const $ = jquery(global.window);
  // Set up chai-jquery to have other kind of expect/assert
  chaiJquery(chai, chai.util, $);
  return $; //to use it in our tests
}

suite('test suite', function () {
  const $ = setupTests();
  test('simple test', function () {
   const component = TestUtils.renderIntoDocument(
      <About />
    );
    const $node = $(ReactDOM.findDOMNode(component));
    expect($node).to.have.text('About');
  });
});
```

This way is clearer as to what to expect in the test, and you can continue using the JQuery API that you already know.

Now let's go into Enzyme, which gives us more tools on top of React components, to make our tests even more complete.

Enzyme

Enzyme is a JavaScript testing utility for React that makes it easier to assert, manipulate, and traverse your React components' output.

The enzyme's API is meant to be intuitive and flexible by mimicking jQuery's API for DOM manipulation and traversal.

The enzyme is un-opinionated regarding which test runner or assertion library you use and should be compatible with all major test runners and assertion libraries out there.

From `https://github.com/airbnb/enzyme`.

This utility becomes very helpful in large projects where you have different types of needs for testing. We will use it with `chai-enzyme` which is a library that extends `chai` with enzyme related functions.

```
import { MemoryRouter } from 'react-router-dom'
import About from '../components/About'
import chai, {expect} from 'chai';

import {Provider} from 'react-redux'
import {createStore} from 'redux'
import reducers from './../reducers';

import React from 'react';

import { configure, mount, shallow, render } from 'enzyme';
import chaiEnzyme from 'chai-enzyme'
import Adapter from 'enzyme-adapter-react-16';

configure({ adapter: new Adapter() });

chai.use(chaiEnzyme());
suite('Enzyme test suite', function () {
test('simple dom test', function () {
//Enzyme's render function is used to render react components to static
HTML and analyze the resulting HTML structure.
    const component = render(
      <About />
    );
    expect(component.find('.about-msg')).to.have.text('Hello');
 });

 test('simple test', function () {
 //Shallow rendering is useful to constrain yourself to testing a component
as a unit, and to ensure that your tests aren't indirectly asserting on
```

```
behavior of child components.
 const $node = shallow(
 <About />
 );
 expect($node.find('.about-msg')).to.have.text('Hello');
 });

 test('test router', function () {
 //Full DOM rendering is ideal for use cases where you have components that
may interact with DOM APIs, or may require the full lifecycle in order to
fully test the component (i.e.componentDidMount, etc.).Full DOM rendering
requires that a full DOM API be available at the global scope. Hence our
JSDOM usage
 const $node = mount(
 <Provider store={createStore(reducers)}>
 <MemoryRouter >
 <About />
 </MemoryRouter>
 </Provider>
 );

 $node.find('.about-btn').simulate('click');

 expect($node.find('.about-msg').text()).to.equal('World');
 });

});
```

Without going into much detail about the Enzyme testing utilities, we can see that there are different flavors to render a React component.

We started from a simple render to markup, to test the structure and content of the `node` behind the component, shallow rendering, to isolate and test the component, and full DOM rendering, where we tested our router!

The first two kinds of tests can be considered unit tests, as they are testing a single component behavior or structure. The last one, testing the router, can be considered as an integration test, integrating different components.

But, we will see another way to test the application, which is better known as integration testing.

Integration tests with Nightwatch

Integration testing is testing in which a group of components is combined to produce an output. It may fall under both white box testing and black box testing. In our case, we will test it as a black box, as we don't know how things work behind the scenes but we use them to test them.

We can test our application in an integrated way with a tool like **Selenium**, where we interact with the page already loaded, clicking in places, filling inputs, and expecting a different types of results.

We chose **Nightwatch** because it runs on top of Selenium and it provides several features that are useful when testing large applications in large environments. We will test our components in a very basic way, but the framework will be ready to scale when you need to.

Chrome headless is another very important choice; at the moment, it is not supported on Windows, so we won't go over it but you can see how to set it up if you are running in a Unix system here (`https://medium.com/@kenfehling/ui-testing-with-nightwatch-js-headless-chrome-and-docker-part-1-f0ce2e8a23a1`).

We will be following Nightwatch v0.9.14 official installation instructions (`http://nightwatchjs.org/gettingstarted`); check them out to see whether you want to use a newer version (released after the publication of this book).

To install `nightwatch`, you need to run:

```
npm install --save-dev nightwatch
```

Also, download the latest version of the `selenium-server-standalone-{VERSION}.jar` file from the Selenium downloads page (`http://selenium-release.storage.googleapis.com/index.html`) and place it on the computer with the browser you want to test. In most cases, this will be on your local machine and typically inside your project's source folder.

With these two things, you are ready to start coding tests. But, we will see a couple of useful configurations first.

Create a `nightwatch.json` configuration first, where you will configure the run of your tests:

```
{
  "src_folders" : ["src/tests/it/"],

  "selenium" : {
    "start_process" : true,
```

```
      "server_path" : "<path/to/your/selenium/server>"
    },

    "test_settings" : {
      "default" : {
        "launch_url" : "http://localhost",
        "selenium_port"  : 4444,
        "selenium_host"  : "localhost",
        "silent": true,
        "screenshots" : {
          "enabled" : false,
          "path" : ""
        },
        "desiredCapabilities": {
          "browserName": "chrome"
        }
      }
    }
}
```

In my opinion, this is the minimal configuration you can get, a chrome browser will come up and run your tests inside the defined `src_folders`.

The `selenium/server_path` property is useful to start the Selenium server when triggering Nightwatch, if not, the server needs to be started first manually by you.

We can use the default syntax provided by Nightwatch to create the tests, but there's an option to use Mocha. We need to add this line to your configuration:

```
...
  "test_runner" : "mocha",
...
```

With that, you are ready to code the first integration test using `mocha`. More options are available at `http://nightwatchjs.org/guide#using-mocha`, but this is the simplest it can get.

Okay, enough with configuration; let's test some stuff.

```
describe('Testing our app', function() {
  describe('with Nightwatch', function() {
    before(function(client, done) {
      done();
    });
    after(function(client, done) {
      client.end(function() {
        done();
```

```
      });
    });
    afterEach(function(client, done) {
      done();
    });
    beforeEach(function(client, done) {
      done();
    });
    it('testing navigation to repos', function(client) {
      client
        .url('http://localhost:8080')
        .expect.element('body').to.be.present.before(1000);
    });
  });
});
```

With this test, we are just checking whether the page is loaded, and waiting for the document body to appear within a second (that equals to 1000 milliseconds). Obviously, this is the hello world of our tests; it's just the first step. Though, you can make this test a performance test, if the page is not loaded in less than *X* seconds, then fail. In this way, define a threshold for the initial load of your page.

But, let's continue with another simple example of something that can be tested using Nightwatch:

```
it('testing navigation to repos', function(client) {
  client
    .url('http://localhost:8080')
    .expect.element('body').to.be.present.before(1000);
  // let's click on the <Link className="repos-link" to="repos">
  client.click('.repos-link', function () {
    //let's assert that we are in the right page
    client.expect.element('.repos-title').text.to.equal('Repos');
    client.assert.urlEquals('http://localhost:8080/repos');
  })
});

it('testing url navigation to repos', function(client) {
  client
    .url('http://localhost:8080/repos');
    // now we navigate to /repos directly
    client.expect.element('.repos-title').text.to.equal('Repos');
});
```

These are a couple of examples that involve navigation; click on a `link` component or navigate through the URL. This way, you can check that the server-side rendering worked and the isomorphic routing is working properly.

Deployment

The last step on the development iteration is to deploy our application. Ours is basically a Node.js application. We are using a Node.js server to handle requests, render our application, and more, and on the client, we are using a React stack.

We are going to look at both practices for production-ready apps and some popular options to deploy our application.

Production best practices

In this section, we will see the best practices for performance and reliability for production-ready applications. These are practices that will help our application to behave correctly and in a reliable way; many things can be work-around or mocked during development, but in production, it is the real deal. If your application crashes, takes too long to load or carry out a critical operation, runs slow, is hacked, and so on, then users suffer; if your users suffer, they leave you.

Things to do in your code

We will use Express in this examples the same that we used for our React examples.

Here are some things you can do in your code to improve your application's performance:

- Use gzip compression
- Don't use synchronous functions
- Lock dependencies
- Carry out logging correctly
- Be stateless
- Error handling

Using gzip compression

Gzip compression is a must in all web applications; it applies the gzip compression algorithm to decrease the size of the resources sent over the wire. This can save a tons of space and hence time for the user, making your app more useable. For example, you can use the express compression middleware to gzip your resources:

```
var compression = require('compression'); // npm install compression --save
var express = require('express');
```

```
var app = express();
app.use(compression());
```

For a high-traffic website in production, the best way to put compression in place is to implement it at a reverse proxy level. In that case, you do not need to use compression middleware.

Nginx is a very well-known reverse proxy (among other things). For details on enabling gzip compression in Nginx, see the `ngx_http_gzip_module` module in the Nginx documentation.

Don’t use synchronous functions

As you may know, Node.js runs in a single thread, so if you use synchronous methods for doing things like reading from the filesystem or just carry out some CPU-intensive work, you block that thread until your function returns. A call to a synchronous function can return in a matter of milliseconds, but if you add a lot of those calls or you multiply them for each request you receive, it can affect your application performance. That's why you need to avoid their use in production.

Node.js and many modules provide asynchronous versions of their functions, and you should always use them instead of their synchronous counterpart. I would say that the only moment where synchronous functions can be called is when starting up the application, although you don't want to stress it, as your app will take longer to start or recover from a crash.

In Node.js 4.0+, while developing your application, you can use the command-line flag `trace-sync-io` to print a warning whenever your application uses a synchronous API.

Lock dependencies

Your code must be identical across all environments. NPM 4- has `npm-shrinkwrap.json`, NPM5 has `package-lock.json`, and the yarn has `yarn.lock`. Choose one of the latest two and don't let dependencies change without you knowing. Without this, your life can become very difficult; different code running in development versus production or even different code running in the same cluster.

Carry out logging correctly

Logs can be a dumb collection of debug statements or what enables you to see a dashboard that tells the story of your app. Plan your logging strategy from day one: how logs are collected, stored, and analyzed. Think about when you need to see these logs in the future, what you will need to make sense of those logs, and do something about it (you will probably be tracing an error). Also, think about tracing logs through different services and servers. If you don't think about this from the beginning, you will end up either doing it while in production, having to redeploy your application or guessing what happened in your black box application.

- Debugging: If you are using logging for debugging, don't use `console.log()`, use a special debugging module such as `debug`. This module enables you to use the `DEBUG` environment variable to control what debug messages are sent to the console. It's pretty simple and effective on what it does.
- App activity: If you're logging app activity, do not think of using `console.log()`, use a real logging library like `Winston` or `Bunyan`. They provide much more functionality like different kinds of streams, types of logging depending on what is logged, and they are more used in production environments, so you can trust they are tested. Provide all the contextual properties that will make your log useful for the one analyzing it, such as user ID, operation type, timestamp, and so on. Then, aggregate all the log sources into one place and visualize it. It needs to be searchable, and you shouldn't need to go to the code so easily; the most important information should be there. Also, show important operational metrics such as error rate and average CPU throughout the day. They will help you to monitor and improve your app. Assign the same identifier, transaction ID: {some value}, to each log entry within a single request. So then, when inspecting errors in logs, you can see what happened in it.

Be stateless

Store any type of data (such as users sessions or cache) in external data stores such as Redis. Think about the server as something that can be killed and replaced at any time. This means that it's probably better not to store anything there; hence, using external structures. This helps to make your application scalable and it's easier to understand and maintain. There are frameworks or practices today that even kill production servers on purpose to test this; regardless of this, prepare your application and your server to be killed.

Error handling

Error handling in Node.js can be a book apart, but these are a couple of things that you need to know. First, remember, the apps run in one process. Not handling exceptions and taking appropriate actions will make your app crash and go offline. Node.js apps crash when they encounter an uncaught exception. You want to avoid crashing and to do that, you need to handle exceptions properly. In addition to this, you want to restart the application as soon as it crashes.

Operational errors: this is the kind of errors that represent runtime problems but by correctly written programs. These are not bugs in the program. In fact, these are usually problems with something else: the network (for example, socket hang-up), the system itself (for example, out of memory or too many open files), or a remote service (for example, a 500 error, failure to connect, or the like).

In these cases, you can either carry out the following:

- Deal with the failure directly, doing what you think is right in that case
- Propagate the error up and let your client handle it
- Retry it
- Crash (for errors that should never happen), the app should boot itself up again after crashing
- Log and go on if there's no need to do anything else you can just log it and analyze this kind of errors later

Programmer errors: programmer errors are basically bugs in the program. These need you changing the code and they can never be handled properly (since the code in question is broken). Although it seems a bit drastic, crashing the application is the fastest way to reliably restore your service. If not, many things can happen that can make your system weak or faulty, as follows:

- Memory leaks
- Opened connections to DB
- Wrong state
- Opened network sockets

Like these, the worst thing about this is that it is unpredictable, as you sadly can't predict your mistakes. Configure Node.js to dump core on an uncaught exception to diagnose the crashing error and read the next part of this chapter to know how to make sure your app restarts after crashing.

Things to do in your environment/setup

The following are things that are more outside of your application; these are some things you can do in your system environment to improve your app's life:

- Set `NODE_ENV` to `production`
- Ensure your app automatically restarts
- Make use of all your CPU cores
- Use tools that automatically detect vulnerabilities
- Use a load balancer
- Use a reverse proxy

Set NODE_ENV to production

A very common practice is to use the Node.js `NODE_ENV` environment variable which specifies the environment in which the application is running (usually it's development or production). One of the easiest and simplest things you can do to improve your app's performance is to set `NODE_ENV` to `production`.

Setting `NODE_ENV` to `production`, in the case of Express, makes it:

- Cache view templates
- Cache CSS files generated from CSS extensions
- Generate less verbose error messages

Depending on the framework used, this can mean other things, many npm packages determining the current environment and optimize their code for production.

Ensure your app automatically restarts

In production, you never want your application to be offline. As we saw in the previous section, your app can crash on an uncaught exception. And you need to make sure that your application restarts both if the app crashes and if the server itself crashes. Although we don't want either of these events to happen, realistically you should account for both eventualities by:

- Using a process manager to restart the app (and Node.js) when it crashes, like forever.
- Using the OS's init system provided to restart the process manager when it crashes. Although it's also possible to use the init system without a process manager.

As we saw in the previous section, all you need to do is to ensure your app is well-tested and handles all exceptions. But as a fail safe, put a mechanism in place to ensure that if and when your app crashes, it will automatically restart.

Use an init system

One very important layer of reliability is to ensure that your app restarts when the server restarts. Systems can go down for many reasons. To ensure that your app restarts if your server crashes, use the init system built into your OS. This way when the server restarts, the application restarts too.

Run your app in a cluster

In a multi-core system, you can increase a lot the performance of a Node.js app by launching a cluster of processes. A cluster runs multiple instances of the app, ideally one instance on each CPU core, thereby distributing the load and tasks among the instances.

Balancing between application instances using the cluster API

Your app instances run in different processes, not sharing their memory space. So do not assume that you can reuse in-memory information between requests because each process has its own context. This is very related to the section, *Be stateless*. It's important to you to maintain data and state externally, it will let you horizontally scale later too.

In clustered apps, worker processes can crash individually without affecting the rest of the processes. Apart from performance advantages, failure isolation is another reason to run a cluster of app processes. Whenever a worker process crashes, always make sure to log the event and spawn a new process using `cluster.fork()`.

Use tools that automatically detect vulnerabilities

Code dependencies, in fact, tend to have vulnerabilities often, including the most famous and battle-tested packages. Tools, such as nsp and snyk, can keep an automatic eye on these threats, warn the team, and the latter can even patch these vulnerabilities automatically.

Use a load balancer

If you make your single application instance handle all your requests, you will soon have a problem, no matter how well your app is optimized. To make your application manage a big amount of traffic, you need a load balancer.

A load balancer takes on the task of deciding how the traffic should be channeled and protects each server from running the risk of being overloaded. Setting up a load balancer can improve both your application's speed and performance. It also enables it to scale more than is possible with a single instance or machine.

One note though, if you are using load balancing and you are storing sessions in the instance, you might have to ensure that requests that are associated with a particular session ID are connected to the process that originated them. If not, the next request will be handled by a server that doesn't have the required state and the application might behave incorrectly. This is known as session affinity or sticky sessions. You can use a data store, such as Redis, for externalizing session data. If your application is stateless, then there's no need for sticky sessions; if you have state in a process, then you need it. Remember the session/jwt token/cookies section? This is related.

Use a reverse proxy

Remember that Node.js's execution model is optimized for short tasks or async IO related tasks. The best approach to this kind of tasks is to use a tool that actually has expertise in networking tasks – the most popular ones are nginx and HAproxy, which is also used by the biggest cloud vendors.

A reverse proxy sits in front of a web app and performs directing requests to the app and does supporting operations on the incoming requests. In addition to this, it can also handle error pages, compression, cache, and serving files.

Handing over tasks that do not require knowledge of application state, so a reverse proxy frees up your application to perform specialized application tasks.

Deploying on a cloud platform

Today, a lot of cloud platforms exist where you deploy your application and forget about many things that were a pain to manage before. Scaling vertically, horizontally, network management, monitoring, and more.

Some of these are:

- Heroku
- Microsoft Azure
- Nodejitsu
- Google Cloud Platform

We will pick Heroku for our example, and the reason that we will explain only Heroku is that the following instructions are going to be outdated pretty fast, as these platforms evolve as fast as the technologies they provide and support.

So we mention other alternatives, but let you pick the one that fits you and your application best. Let's go.

Heroku

Heroku is a cloud platform that lets people build, deliver, monitor, and scale apps in many languages. It makes you forget about the hardware behind your application.

When it comes to running apps in the cloud or otherwise, containerization abstracts away the burden of managing hardware or virtual machines. With Heroku, instead of hardware management, you deploy something which packages both the app's code and its dependencies into containers—which are lightweight, isolated environments that provide compute, memory, an OS, and an ephemeral filesystem. Although containers are typically run on a shared host, they are completely isolated from each other.

The Heroku platform (`https://www.heroku.com/platform`) uses the container model to run and scale all Heroku apps. The containers used at Heroku are called *dynos*. Dynos are isolated, virtualized Linux containers that are designed to execute code based on a user-specified command. Your app can scale to any specified number of dynos based on its resource demands. Heroku's container management capabilities provide you with an easy way to scale and manage the number, size, and type of dynos (`https://devcenter.heroku.com/articles/dyno-types`) your app may need at any given time.

Let's talk about deploying a Node.js app:

First of all, go to `heroku.com` and sign up for an account if you do not have one. Accounts are free and quick to get.

Then, you'll next need to download the Heroku CLI (command-line interface) tools that are appropriate for your platform. You can find the installer for these at `https://devcenter.heroku.com/articles/heroku-command-line`.

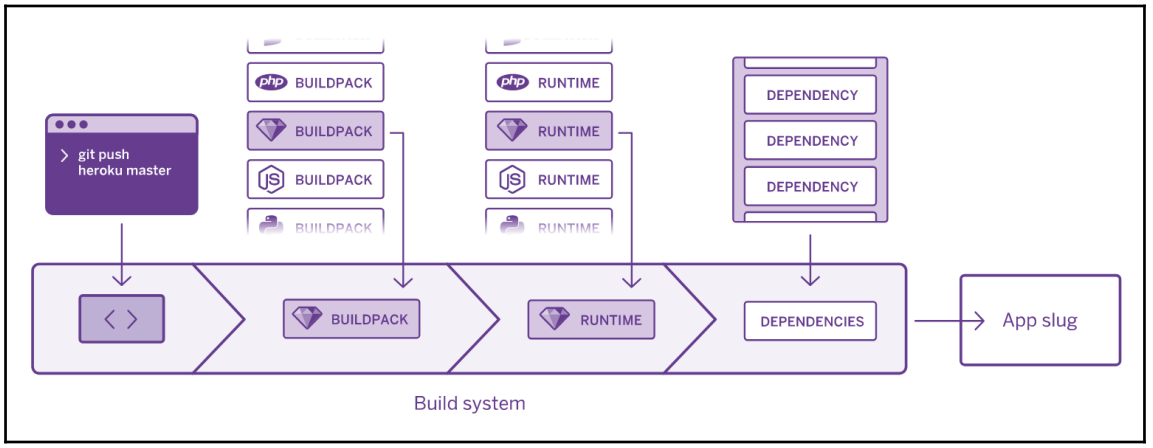
After this, what you need to do to deploy a Node.js application is:

First, the `package.json` should have all its dependencies, and a `node` engine version defined, as follows:

```
"engines": {
    "node": "your node version"
  }
```

To check which version you're running locally, at the command line, type `node --version`.

Heroku needs only three things from the developer: source code, a list of dependencies, and a Procfile (a text file that indicates which command should be used to start the code running). The build system takes the application, its dependencies, and the language runtime and produces a slug. It contains everything needed to run the app, except for the operating system `https://www.heroku.com/dynos/build`:



As we said, Heroku first looks for a Procfile; if no Procfile exists for a Node.js app, it will attempt to start a default web process via the `start` script in your `package.json`:

```
"scripts": {
    "start": "<your command>"
  }
```

Before deploying your app, you can try it locally, running:

```
heroku local web
```

You should be able to see your application running on `localhost:5000`.

If everything succeeds, now you can deploy your application to Heroku:

```
$ heroku login
Enter your Heroku credentials.
...
$ heroku create
Creating hot-fire-2348... done, stack is cedar
http://hot-fire-2348.herokuapp.com/ | git@heroku.com:hot-fire-2348.git
Git remote heroku added
$ git push heroku master
...
-----> Node.js app detected
...
-----> Launching... done
       http://hot-fire-2348.herokuapp.com deployed to Heroku
```

Heroku starts the app by deploying the slug to a dyno (or set of dynos) and invoking a command specified in the Procfile. After that is done, your app is ready to be used.

To open the app in your browser, type `heroku open`.

```
heroku open
```

For more information about deployment in heroku, you can visit `https://devcenter.heroku.com/articles/deploying-nodejs`.

As your app traffic and infrastructure needs grow, you can scale your application with more processing power and dedicated server resources, which are available on a pay-as-you-go model.

I encourage you to deploy your application, play with Heroku's services (all the free ones), and read their documentation to fully understand its capabilities and use them the right way.

Summary

We have finished the deployment and testing chapter. We explored different types of testing your application, important practices to have in mind for your production application, and finally how to deploy it in a cloud service. Both crucial topics for any mature product development and life.

And by finishing this chapter, we finish the book. We really hope that the book helped you to understand and develop great isomorphic applications and taught you something new in the programming world. New ideas and frameworks will continue to appear, but we hope this book will help you to understand and evaluate them.

Index

N

P

R

S

T

U

V

W