sitepoint

# BUILD YOUR OWN REACT UNIVERSAL BLOG APP

COMPLETE PROJECT TUTORIAL

# Build Your Own React Universal Blog App



- **Authors:** Michael Wanyoike, Pavels Jelisejevs, and Tony Spiro
- **Cover Designer:** Alex Walker

## Notice of Rights



## Notice of Liability



## Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

## About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

# Table of Contents

# Preface

## Conventions Used

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

### Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.</p>
```

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed:

```
function animate() {
  ⋮
new_variable = "Hello";
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ↪ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-
↪design-real-user-testing/?responsive1");
```

## Tips, Notes, and Warnings

### Hey, You!

Tips provide helpful little pointers.

### Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

### Make Sure You Always ...

... pay attention to these important points.

### Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

### Live Code

This example has a Live Codepen.io Demo you can play with.

### Github

This example has a code repository available at Github.com.

Chapter

# 1

# Wait, what *is* React?

## React is:

... a Facebook creation which simply labels itself as being a JavaScript library for building user interfaces. It's an open-source project which, to date, has raked in over 74,000 stars on GitHub.

It is:

- **Declarative**: you only need to design simple views for each state in your application and React will efficiently update and render just the right components when your data changes.
- **Component-based**: you create your React-powered apps by assembling a number of encapsulated components, each managing its own state.
- **Learn Once, Write Anywhere**: React is not a full-blown framework; it's just a library for rendering views.

## React uses a "virtual DOM"

That's what makes React fast and responsive. The HTML Document Object Model or DOM is a representation of the document as a structured group of nodes and objects that have properties and methods. It connects web pages to scripts or programming languages.

Whenever you want to change any part of a web page programmatically, you need to modify the DOM. Depending on the complexity and size of the document, traversing the DOM and updating it could take longer than users might be prepared to accept. Every time the DOM gets updated, browsers need to recalculate the CSS and carry out layout and repaint operations on the web page.

The Virtual DOM is a lightweight, abstract model of the DOM. React uses the *render* method to create a node tree from React components and updates this tree in response to changes in the data model resulting from actions.

Each time there are changes to the underlying data in a React app, React creates a new Virtual DOM representation of the user interface.

## How React works

- Whenever something changes, React re-renders the entire UI in a Virtual DOM representation.
- React then calculates the difference between the previous Virtual DOM representation and the new one.
- Finally, React patches up the real DOM with what has actually changed. If nothing has changed, React won't be dealing with the HTML DOM at all.

Efficient diff algorithms, batching DOM read/write operations, and limiting DOM changes to the bare minimum necessary, make using React and its Virtual DOM a great choice for building performant apps.

## When should you use React?

React is great at making super reactive user interfaces — that is, UIs that are very quick at responding to events and consequent data changes.

React would be a great fit for web apps where you need to keep a complex, interactive UI in sync with frequent changes in the underlying data model.

React is designed to deal with stateful components in an efficient way (which doesn't mean devs don't need to optimize their code). So projects that would benefit from this capability could be considered good candidates for React.

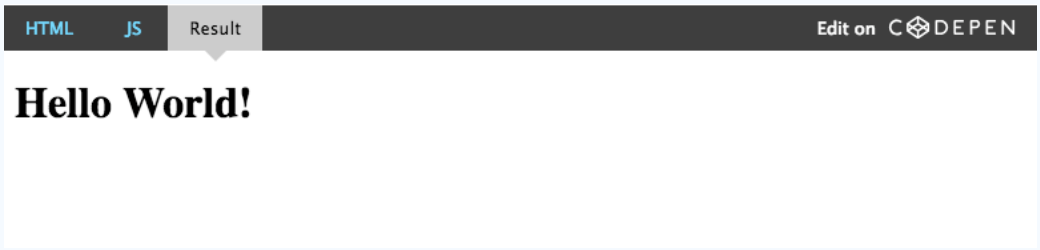
Chapter

# Hello, World

2

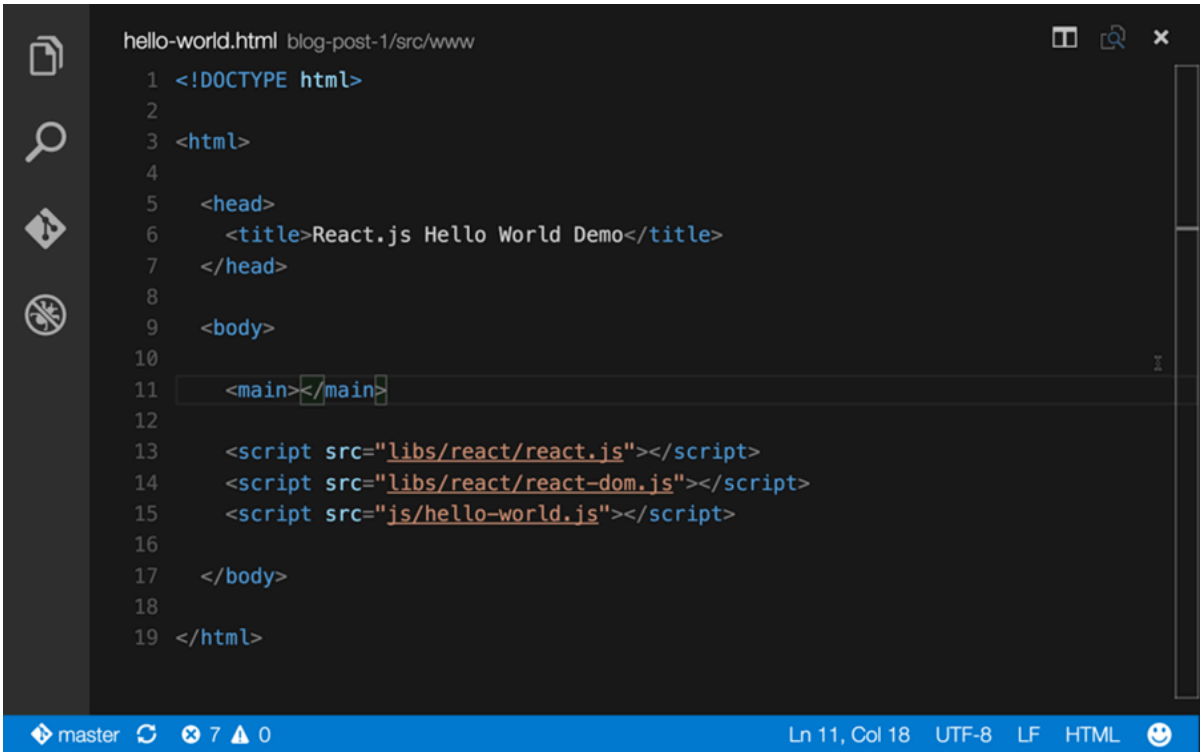
**Now that we know the theory and background behind React, let's get our hands dirty with the traditional Hello World example project. We'll use CodePen for this project, and you can click on any of the images below to be transported to the online demo. Or code along in your favorite text editor - some screenshots here use Visual Studio Code..**

### Live Code (click image for link to CodePen)




2-1. Edit on CodePen

To set up this CodePen, click on "Settings" in the header, then on "JavaScript", and you will see two external JavaScript files have been included: React and React-DOM. Both files are referenced from Facebook. The first JavaScript file contains the React library itself, while the second contains code to use React with the DOM of a web browser.




2-2.

To create React Components, use the *createClass* function of the React object. The *createClass* function expects an object configuring the component to be passed in. The *createClass* function is a helper function for creating new components which inherit from React.Component.

If you're using ES2015 natively in the browser or through a transpiler such as Babel, then it's possible to inherit directly from *React.Component* using the new *class* and *extends* keywords.

To use Babel in CodePen, click on "Settings", then "JavaScript", and select it from the "JavaScript Preprocessor" drop down list.

### Live Code (click image for link to CodePen)


2-3. Using Babel

Regardless of the approach to creating the class structure for the component, the result is the same.

The only required property for a component is the *render* property, which points to a function object which is used to actually render the DOM of the component. The implementation of the render function introduces a new function, *createElement*, provided by the React object. The *createElement* function is used to create new DOM elements with React. The function expects up to three parameters.

The first parameter is the name of the HTML element or React Component to create. HTML elements should be a lowercase string containing only the name of the element without the angle brackets and no attributes. Examples of acceptable HTML element arguments include *h1*, *p*, etc. In addition to HTML element names, React Component objects can be passed in. For React

Components, the object itself, not a string name of the object, is passed in.

The second parameter is an object of the properties to pass in. For HTML elements, these properties correspond to the attributes of the HTML element. For React Components, these properties correspond to stateless data for use when rendering the component. Learn more about state here.

Finally, the third parameter represents the child elements of the element being created. In the "Hello World" example, the child content of the **h1** element is the content "Hello World!" In addition to textual content, element objects can be passed in.

### Live Code (click image for link to CodePen)

JS

```
"use strict";

var HelloWorld = React.createClass({

  render: function() {
    return React.createElement("header", null,
      React.createElement("h1", null, "Hello World!")
    );
  },

});

var mainElement = document.querySelector("main");

ReactDOM.render(React.createElement(HelloWorld), mainElement);
```

2-4. Passing elements

Or by using an array, multiple child elements can be passed in as well.

### Live Code (click image for link to CodePen)

JS

```
"use strict";

var ItemList = React.createClass({

  render: function() {
    return React.createElement("ul", null, [
      React.createElement("li", null, "Item 1"),
      React.createElement("li", null, "Item 2"),
      React.createElement("li", null, "Item 3"),
    ]);
  },

});

var mainElement = document.querySelector("main");

ReactDOM.render(React.createElement(ItemList), mainElement);
```

2-5. Passing child elements

To use the React Components in a web page, the ReactDOM object's *render* function is used. It expects an element object, and a root element to which the DOM of the element object will be appended. In the code demonstration, the *createElement function* is used to create an instance of the HelloWorld component, while *document.querySelector* is used to select the main element to which the instantiated and rendered HelloWorld component is appended. Once appended, the component appears in the web page, and the React demonstration is complete!

And now let's learn more about what you just did.

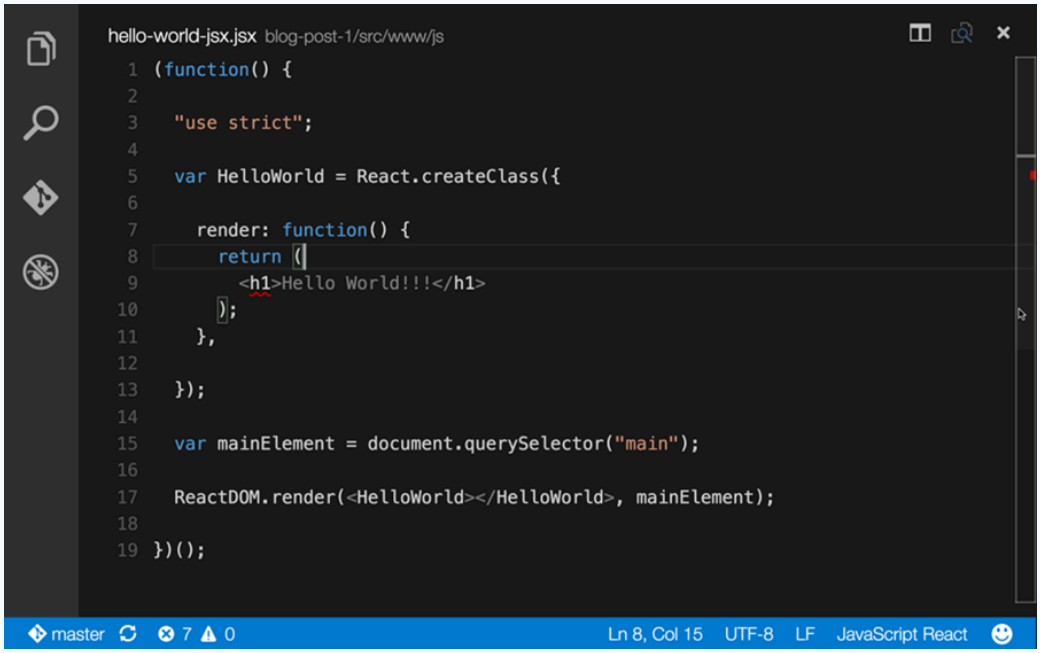
## JSX

React has many benefits over other UI solutions. Its overall API is comparatively small, and has a fast learning curve. In an hour or so, many developers are up and running using this new library to build reusable, extensible and maintainable UI components. For building small components, the pure JavaScript code API is great, but for larger components working with the *createElement* function quickly becomes tedious.

One of the goals of React is to eschew the template-driven UI solutions such as Angular.js or Handlebars. Many developers consider such template-driven UI approaches to be antithetical to proper UI design because they can allow XSS exploits, are difficult to update when data changes occur and can be difficult to extend/reuse.

Many of the complaints center around the usage of logic in the template itself. React offers an alternative of a composable (more on this later), code-driven approach for building UIs. The function *createElement* is used to create new elements in code, including the setting of element attributes and the definition of child content. Nevertheless, it is tedious to use especially when there is no logic per se, and some HTML elements (including attributes) or React Components simply need to be easily created. React's solution is to use a more familiar declarative syntax for such content, while wrapping that declarative content in JavaScript.

## Live Code (click image for link to CodePen)




2-6. Using JSX

The compromise between ease of coding and not introducing logic into the template was achieved through a technique called JSX (JavaScript syntax extension).

To view the demonstration and edit the code, please click on the image.

## Live Code (click image for link to CodePen)

JS (Babel)

```
"use strict";

var HelloWorld = React.createClass({

  render: function() {
    return (
      <h1>Hello World!!!</h1>
    );
  },

});

var mainElement = document.querySelector("main");

ReactDOM.render(<HelloWorld></HelloWorld>, mainElement);
```

2-7. JSX demonstration

The result of the JSX transpilation is JavaScript with createElement calls. To see the resulting JavaScript, click on the “View Compiled” link at the upper right of the JavaScript frame in the CodePen. The JSX syntax is easier for developers, while the result is the same code-driven UI solution that React desired to achieve.

Babel is used to transpile the JSX to JavaScript. Originally, Babel was simply an ES6 JavaScript transpiler. With the completion of ES6 (ES2015), Babel has been extended to serve as a platform for creating JavaScript code including plugins such as the one for JSX. Within CodePen, Babel with the JSX plugin is automatically transpiling the JSX code each time the CodePen is executed.

The React web site recommends the use of JSX, and many developers use it.

Nevertheless, using JSX is not required, and React.js applications work the same regardless of its usage. The remaining code demonstrations in this post will feature JSX (unless otherwise noted).

## Properties

React Components have two kinds of data: state and properties. State data represents data that will be updated by a user or from the server. While state is important, it should be limited to as little as needed, with emphasis put on the properties which can be passed data (including state data from a component which manages state) to be rendered in the DOM. A future post will cover state data, as properties are the preferred way of configuring and managing data for a component. Also, as a side note for now, React does not support two-way data binding by default.

Earlier in this post, properties were discussed in conjunction with the createElementfunction. Within React Components, properties passed to the component can be accessed through the *props* property of the component. Click here to see a demonstration of this and edit the code.

In the above CodePen example using JSX, the property had a static value and was passed in using an attribute in the JSX syntax. In addition to passing a static value, a value from a JavaScript variable can passed in using the curly brace template variable syntax.

In addition to using the JSX syntax, a plain JavaScript object can be used with the non-JSX version.

When component properties (or state) change, React performs a process called **Reconciliation** to determine what DOM updates are needed to render the updated property and state information. React's mechanism for this is quite sophisticated and even involves the use of virtual DOM to compare changes to make selective and fast updates to the DOM without having to re-render everything. **Reconciliation** will be thoroughly discussed in a future post.

## Composable Components

React Components are designed to be composable, they can be combined together to build larger more sophisticated components. Consider the example of an HTML table. Typically, tables include a header and body sections. The header row is static while there are a variable number of body rows dependent upon the data available at any given moment. Such a table is a perfect example of composable component. The table itself can be a component with each row of data being a different component as shown in the demonstration below. Click here to view the demonstration and edit the code on CodePen.

Other JavaScript solutions such as Angular.js and Handlebars use template logic instead of composable components.

Chapter

# Create React App

3

**Today, let's take a look at Create React App, a toolkit that makes starting a new React project simple. We'll actually get more in-depth in our project and build it out ourselves, but it's good to know how the tool works and how it can help you.**

Create-React-App is a CLI tool from Facebook that allows you to generate a new React project and use a pre-configured Webpack build for development.

## How Does Create React App Work?

Create React App is a standalone tool that should be installed globally via npm, and called each time you need to create a new project:

```
npm install -g create-react-app
```

To create a new project, run:

```
create-react-app react-app
```

Create React App will set up the following project structure:

```
.
├── .gitignore
├── README.md
├── package.json
├── node_modules
├── public
│   ├── favicon.ico
│   └── index.html
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    └── logo.svg
```

3-1. Create React App default project structure

It will also add a *react-scripts* package to your project that will contain all of the configuration and build scripts. In other words, your project depends *react-scripts*, not on *create-react-app* itself. Once the installation is complete, you can start working on your project.

## Starting a Local Development Server

The first thing you'll need is a local development environment. Running *npm start* will fire up a Webpack development server with a watcher that will automatically reload the application once you change something. Hot reloading, however, is only supported for styles.

The application will be generated with a number of features built-in.

### ES6 and ES7

The application comes with its own Babel preset, babel-preset-react-app, to support a set of ES6 and ES7 features. It even supports some of the newer

features like async/await, and import/export statements. However, certain features, like decorators, have been intentionally left out.

## Asset import

You can also import CSS files from your JS modules that allow you to bundle styles that are only relevant for the modules that you ship. The same thing can be done for images and fonts.

## ESLint

During development, your code will also be run through ESLint, a static code analyzer that will help you spot errors during development.

## Environment variables

You can use Node environment variables to inject values into your code at built-time. *React-scripts* will automatically look for any environment variables starting with _REACT_APP__ and make them available under the global *process.env*. These variables can be in a *.env* file for convenience:

```
REACT_APP_BACKEND=http://my-api.com REACT_APP_BACKEND_USER=root
```

You can then reference them in your code:

```
fetch({process.env.REACT_APP_SECRET_CODE}/endpoint)
```

## Proxying to a backend

If your application will be working with a remote backend, you might need to be able to proxy requests during local development to bypass CORS. This can be set up by adding a proxy field to your *package.json* file:

```
"proxy": "http://localhost:4000",
```

This way, the server will forward any request that doesn't point to a static file the given address.

## Running Unit Tests

Executing *npm test* will run tests using Jest and start a watcher to re-run them whenever you change something:

```
PASS  src/App.test.js
  ✓ renders without crashing (7ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.123s, estimated 1s
Ran all test suites.

Watch Usage
 › Press p to filter by a filename regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

3-2. npm test results

Jest is a test runner also developed by Facebook as an alternative to Mocha or Karma. It runs the tests on a Node environment instead of a real browser, but provides some of the browser-specific globals using jsdom.

Jest also comes integrated with your VCS and by default will only run tests on files changed since your last commit. For more on this, refer to "How to Test React Components Using Jest".

# Creating a Production Bundle

When you finally have something to deploy, you can create a production bundle using npm run build. This will generate an optimized build of your application, ready to be deployed to your environment. The generated artifacts will be placed in the build folder.

```
.
├── asset-manifest.json
├── favicon.ico
├── index.html
└── static
    ├── css
    │   ├── main.9a0fe4f1.css
    │   └── main.9a0fe4f1.css.map
    ├── js
    │   ├── main.3b7bfee7.js
    │   └── main.3b7bfee7.js.map
    └── media
        └── logo.5d5d9eef.svg
```

3-3. Create React App bundle structure

The JavaScript and CSS code will be minified, and CSS will additionally be run through Autoprefixer to enable better cross-browser compatibility.

## Deployment

React-scripts provides a way to deploy your application to GitHub pages by simply adding a homepage property to package.json. There's also a separate Heroku build pack.

# Opting Out

If at some point you feel that the features provided are no longer enough for your project, you can always opt out of using react-scripts by running *npm run*

*eject*. This will copy the Webpack configuration and build scripts from *react-scripts* into your project and remove the dependency. After that, you're free to modify the configuration however you see fit.

## You did it - lesson 3 out of 10 done and dusted!

That's Create React App. In T-minus three days we'll get started on building our main project for this course, a React Universal Blog. This won't use Create React App, but will show you how to do things "the hard way", and help you appreciate all the magic the toolkit performs.

Chapter

# 4

# Now, Let's Build Our App

**We've covered the easy way to set up a React app with Create React App, but now we'll delve into a more in-depth method to aid understanding of how React apps, and universal React apps, are structured and function.**

## First: what exactly is a *Universal* React App?

Single page apps like React are fantastic, but they do have one problem: invisibility to search engines. Because an SPA renders a page's content using JavaScript, and since web crawlers don't use a browser to view web pages, they can't view and index the content — or at least most of them can't.

This is a problem that some developers have tried to solve in various ways:

1. Adding an escaped fragment version of a website, which requires all pages to be available in static form and adds a lot of extra work (now deprecated).
2. Using a paid service to un-browserify an SPA into static markup for search engine spiders to crawl.
3. Trusting that search engines are now advanced enough to read our JavaScript-only content. (I wouldn't just yet.)

Or, using Node.js on the server and React on the client, we can build our JavaScript app to be **universal** (or **isomorphic**). This could offer several benefits from server-side and browser-side rendering, allowing both search engines and humans using browsers to view our SPA content.

In this step-by-step tutorial, I'll show you how to build a React Universal Blog App that will first render markup on the server side to make our content available to search engines. Then, it will let the browser take over in a single page application that is both fast and responsive.

## Getting Started

Our universal blog app will make use of the following technologies and tools:

1. Node.js for package management and server-side rendering
2. React for UI views
3. Express for an easy back-end JS server framework
4. React Router for routing
5. React Hot Loader for hot loading in development
6. Flux for data flow
7. Cosmic JS for content management

To start, run the following commands:

```
mkdir react-universal-blog

cd react-universal-blog
```

Now create a package.json file and add the following content (there's a lot!):

```
{
  "name": "react-universal-blog",
  "version": "1.0.0",
  "engines": {
    "node": "4.1.2",
    "npm": "3.5.2"
  },
  "description": "",
  "main": "app-server.js",
  "dependencies": {
    "babel-cli": "^6.26.0",
    "babel-loader": "^7.1.2",
```

```
    "babel-preset-es2015": "^6.24.1",
    "babel-preset-es2017": "^6.24.1",
    "babel-preset-react": "^6.24.1",
    "babel-register": "^6.26.0",
    "cosmicjs": "^2.4.0",
    "flux": "^3.1.3",
    "history": "1.13.0",
    "hogan-express": "^0.5.2",
    "html-webpack-plugin": "^2.30.1",
    "path": "^0.12.7",
    "react": "^15.6.1",
    "react-dom": "^15.6.1",
    "react-router": "1.0.1",
    "webpack": "^3.5.6",
    "webpack-dev-server": "^2.7.1"
  },
  "scripts": {
    "webpack-dev-server": "NODE_ENV=development PORT=8080 webpack-dev-server
    ↪--content-base public/ --hot --inline --devtool inline-source-map
    ↪--history-api-fallback", "development": "cp views/index.html
    ↪public/index.html && NODE_ENV=development webpack && npm run
    ↪webpack-dev-server"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "react-hot-loader": "^1.3.0"
  }
}
```

In this file, you'll notice that we've added the following:

1. Babel to package our CommonJS modules and convert our ES6 and React JSX into browser-compatible JavaScript
2. The Cosmic JS official Node.js client to easily serve our blog content from the Cosmic JS cloud-hosted content API

3. Flux for app data management (which is a very important element in our React application).
4. React for UI management on server and browser
5. React Router for routes on server and browser
6. webpack for bringing everything together into a bundle.js file.

We've also added a script in our `package.json` file. When we run `npm run development`, the script copies the `index.html` file from our views folder into our `public` folder. Then, it sets the content base for our *webpack-dev-server* to _public/_and enables hot reloading (on `.js` file save). Finally, it helps us debug our components at the source and gives us a fallback for pages it can't find (falls back to `index.html`).

Chapter

# Time to Meet Webpack

5

First, let's set up our webpack configuration file by editing the file *webpack.config.js*:

```
// webpack.config.js
var webpack = require('webpack')

module.exports = {
  devtool: 'eval',
  entry: './app-client.js',
  output: {
    path: __dirname + '/public/dist',
    filename: 'bundle.js',
    publicPath: '/dist/'
  },
  module: {
    loaders: [
      { test: /\.js$/, loaders: /\.js$/'babel-loader', exclude: /node_modules/ },
      { test: /\.jsx$/, loaders: 'babel-loader'de: /node_modules/ }
    ]
  },
  plugins: [
    new webpack/node_modules/webpack.DefinePlugin({
      'process.env.COSMIC_BUCKET': JSON.stringify(process.env.COSMIC_BUCKET),
      'process.env.COSMIC_READ_KEY': JSON.stringify(process.env.COSMIC_READ_KEY),
      'process.env.COSMIC_WRITE_KEY': JSON.stringify(process.env.COSMIC_WRITE_KEY)
    })
 ]
};
```

You'll notice that we've added an *entry* property with a value of *app-client.js*. This file serves as our app client entry point, meaning that from this point webpack will bundle our application and output it to */public/dist/bundle.js* (as specified in the *output* property). We also use loaders to let Babel work its magic on our ES6 and JSX code. React Hot Loader is used for hot-loading (no page refresh!) during development.

Before we jump into React-related stuff, let's get the look-and-feel of our blog ready to go. Since I'd like you to focus more on functionality than style in this

tutorial, here we'll use a pre-built front-end theme. I've chosen one from Start Bootstrap called Clean Blog. In your terminal run the following commands:

Create a folder called `views` and inside it an `index.html` file. Open the HTML file and add the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <meta name="description" content="">
  <meta name="author" content="">
  <title>{{ site.title }}{{# page }} | {{ page.title }}{{/ page }}</title>
  <!-- Bootstrap Core CSS -->
  <link href="/css/bootstrap.min.css" rel="stylesheet">
  <!-- Custom CSS -->
  <link href="/css/clean-blog.min.css" rel="stylesheet">
  <link href="/css/cosmic-custom.css" rel="stylesheet">
  <!-- Custom Fonts -->
  <link href="//maxcdn.bootstrapcdn.com/font-awesome/4.1.0/css/font-awesome.min.css"
    rel="stylesheet" type="text/css">
  <link href="//fonts.googleapis.com/css?family=Lora:400,700,400italic,700italic"
    rel="stylesheet" type="text/css">
  <link href="//fonts.googleapis.com/css?family=Open+Sans:300italic,400italic,
  ↪600italic,700italic,800italic,400,300,600,700,800"
    rel="stylesheet" type="text/css">
  <!-- HTML5 Shim and Respond.js IE8 support of HTML5 elements and media queries -->
  <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
  <!--[if lt IE 9]>
    <script src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js">
    </script>
    <script src="https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js">
    </script>
  <![endif]-->
</head>
<body class="hidden">
  <div id="app">{{{ reactMarkup }}}</div>
```

```
  <script src="/js/jquery.min.js"></script>
  <script src="/js/bootstrap.min.js"></script>
  <script src="/js/clean-blog.min.js"></script>
  <script src="/dist/bundle.js"></script>
</body>
</html>
```

To get all of the JS and CSS files included in `public`, you can get them from the GitHub repository. Click here to download the files.

Generally I would use the fantastic React Bootstrap package and refrain from using jQuery. However, for the sake of brevity, we'll keep the theme's pre-built jQuery functionality.

In our `index.html` file, we'll have our React mount point set up at the `div` where `id="app"`. The template variable `{{{ reactMarkup }}}` will be converted into our server-rendered markup and then once the browser kicks in, our React application will take over and mount to the `div` with `id="app"`. To improve the user experience while our JavaScript loads everything, we add `class="hidden"` to our body. Then, we remove this class once React has mounted. It might sound a bit complicated, but I'll show you how we'll do this in a minute.

At this point, your app should have the following structure:

```
package.json
public
  |-css
    |-bootstrap.min.css
    |-cosmic-custom.css
  |-js
    |-jquery.min.js
    |-bootstrap.min.js
    |-clean-blog.min.js
views
  |-index.html
webpack.config.js
```

Now that we have our static pieces done, let's start building some React Components.

Chapter

# Components

6

**Today, it's all about components. There are some long stretches of code here, but it's not complicated, I promise!**

## Our Blog App Components (Basic Example)

Let's begin building the UI for our app by setting up the pages for our blog. Because this is going to be a portfolio blog for a creative professional, our blog will have the following pages:

1. Home
2. About
3. Work
4. Contact

Let's start by creating a file called `app-client.js` and add the following content to it:

```
// app-client.js
import React from 'react'
import { render } from 'react-dom'
import { Router } from 'react-router'
import createBrowserHistory from 'history/lib/createBrowserHistory'
const history = createBrowserHistory()

// Routes
import routes from './routes'

const Routes = (
  <Router history={history}>
    { routes }
  </Router>
)

const app = document.getElementById('app')
```

```
render(Routes, app)
```

To better understand how React Router works, you can visit their GitHub repo. The gist here is that we have in `app-client.js` our `Router` component that has a browser history for our client-side routing. Our server-rendered markup won't need browser history, so we'll create a separate `routes.js` file to be shared between our server and client entry points.

Add the following to the `routes.js` file:

```
// routes.js
import React, { Component } from 'react'
import { Route, IndexRoute, Link } from 'react-router'

// Main component
class App extends Component {
  componentDidMount(){
    document.body.className=''
  }
  render(){
    return (
      ''<div>
        <h1>React Universal Blog</h1>
        <nav>
          <ul>
            <li><Link to="/">Home</Link></li>
            <li><Link to="/about">About</Link></li>
            <li><Link to="/work">Work</Link></li>
            <li><Link to="/contact">Contact</Link></li>
          </ul>
        </nav>
        { this.props.children }
      </div>
    )
  }
}

// Pages
```

```
class Home extends Component {
  render(){
    return (
      <div>
        <h2>Home</h2>
        <div>Some home page content</div>
      </div>
    )
  }
}
class About extends Component {
  render(){
    return (
      <div>
        <h2>About</h2>
        <div>Some about page content</div>
      </div>
    )
  }
}
class Work extends Component {
  render(){
    return (
      <div>
        <h2>Work</h2>
        <div>Some work page content</div>
      </div>
    )
  }
}
class Contact extends Component {
  render(){
    return (
      <div>
        <h2>Contact</h2>
        <div>Some contact page content</div>
      </div>
    )
  }
}
class NoMatch extends Component {
```

```
  render(){
    return (
      <div>
        <h2>NoMatch</h2>
        <div>404 error</div>
      </div>
    )
  }
}

export default (
  <Route path="/" component={App}>
    <IndexRoute component={Home}/>
    <Route path="about" component={About}/>
    <Route path="work" component={Work}/>
    <Route path="contact" component={Contact}/>
    <Route path="*" component={NoMatch}/>
  </Route>
)
```

From here, we have a pretty basic working example of a blog app with a few different pages. Now, let's run our application and check it out! In your terminal, run the following commands:

```
mkdir public
npm install
npm run development
```

Then navigate to http://localhost:8080 in your browser to see your basic blog in action.

These things done, it's now time to get this to run on the server. Create a file called `app-server.js` and add this content:

```
// app-server.js
import React from 'react'
import { match, RoutingContext } from 'react-router'
```

```
import ReactDOMServer from 'react-dom/server'
import express from 'express'
import hogan from 'hogan-express'

// Routes
import routes from './routes'

// Express
const app = express()
app.engine('html', hogan)
app.set('views', __dirname + '/views')
app.use('/', express.static(__dirname + '/public/'))
app.set('port', (process.env.PORT || 3000))

app.get('*',(req, res) => {

  match({ routes, location: req.url }, (error, redirectLocation, renderProps) => {

    const reactMarkup = ReactDOMServer.renderToStaticMarkup(<RoutingContext
    ↪{...renderProps} />)

    res.locals.reactMarkup = reactMarkup

    if (error) {
      res.status(500).send(error.message)
    } else if (redirectLocation) {
      res.redirect(302, redirectLocation.pathname + redirectLocation.search)
    } else if (renderProps) {

      // Success!
      res.status(200).render('index.html')

    } else {
      res.status(404).render('index.html')
    }
  })
})

app.listen(app.get('port'))

console.info('==> Server is listening in ' + process.env.NODE_ENV + ' mode')
```

```
console.info('==> Go to http://localhost:%s', app.get('port'))
```

In `app-server.js`, we're loading the basic routes that we've set up. These are converting the rendered markup into a string and then passing it as a variable to our template.

We're ready to start our server and view our code on it, but first, let's create a script to do so.

Open your `package.json` file and edit the `script` section to look like the following:

```
// …
"scripts": {
    "start": "npm run production",
    "production": "rm -rf public/index.html && NODE_ENV=production webpack -p &&
    ↪NODE_ENV=production babel-node app-server.js --presets es2015",
    "webpack-dev-server": "NODE_ENV=development PORT=8080 webpack-dev-server
    ↪--content-base public/ --hot --inline --devtool inline-source-map
    ↪--history-api-fallback",
    "development": "cp views/index.html public/index.html && NODE_ENV=development
    ↪webpack && npm run webpack-dev-server"
  },
// …
```

Now that we have our `production` script set up, we can run our code on both the server side and the client side. In your terminal execute:

```
npm start
```

Navigate in your browser to http://localhost:3000. You should see your simple blog content and be able to quickly and easily navigate through the pages in SPA mode.

Go ahead and hit `view source`. Notice our SPA code is there for all robots to find

as well. We get the best of both worlds!

Chapter

7

# Preparing to Add Content to Our React Universal Blog App

As we add more pages and content to our blog, our *routes.js* file will quickly become big. Since it's one of React's guiding principles to break things up into smaller, manageable pieces, let's separate our routes into different files.

Open your `routes.js` file and edit it so that it'll have the following code:

```
// routes.js
import React from 'react'
import { Route, IndexRoute } from 'react-router'

// Store
import AppStore from './stores/AppStore'

// Main component
import App from './components/App'

// Pages
import Blog from './components/Pages/Blog'
import Default from './components/Pages/Default'
import Work from './components/Pages/Work'
import NoMatch from './components/Pages/NoMatch'

export default (
  <Route path="/" data={AppStore.data} component={App}>
    <IndexRoute component={Blog}/>
    <Route path="about" component={Default}/>
    <Route path="contact" component={Default}/>
    <Route path="work" component={Work}/>
    <Route path="/work/:slug" component={Work}/>
    <Route path="/blog/:slug" component={Blog}/>
    <Route path="*" component={NoMatch}/>
  </Route>
)
```

We've added a few different pages to our blog and significantly reduced the size of our `routes.js` file by breaking the pages up into separate components. Moreover, note that we've added a Store by including `AppStore`, which is very important for the next steps in scaling out our React application.

## The Store: the Single Source of Truth

In the Flux pattern, the Store is a very important piece, because it acts as the *single source of truth* for data management. This is a crucial concept in understanding how React development works, and one of the most touted benefits of React. The beauty of this discipline is that, at any given state of our app we can access the `AppStore` 's data and know exactly what's going on within it. There are a few key things to keep in mind if you want to build a data-driven React application:

1. We never manipulate the DOM directly.
2. Our UI answers to data and data live in the store
3. If we need to change out our UI, we can go to the store and the store will create the new data state of our app.
4. New data is fed to higher-level components, then passed down to the lower-level components through `props` composing the new UI, based on the new data received.

With those four points, we basically have the foundation for a **one-way data flow**application. This also means that, at any state in our application, we can `console.log(AppStore.data)` , and if we build our app correctly, we'll know exactly what we can expect to see. You'll experience how powerful this is for debugging as well.

Now let's create a store folder called `stores` . Inside it, create a file called `AppStore.js` with the following content:

```
// AppStore.js
import { EventEmitter } from 'events'
import _ from 'lodash'
```

```
export default _.extend({}, EventEmitter.prototype, {

  // Initial data
  data: {
    ready: false,
    globals: {},
    pages: [],
    item_num: 5
  },

  // Emit change event
  emitChange: function(){
    this.emit('change')
  },

  // Add change listener
  addChangeListener: function(callback){
    this.on('change', callback)
  },

  // Remove change listener
  removeChangeListener: function(callback) {
    this.removeListener('change', callback)
  }

})
```

You can see that we've attached an event emitter. This allows us to edit data in our store, then re-render our application using `AppStore.emitChange()`. This is a powerful tool that should only be used in certain places in our application. Otherwise, it can be hard to understand where `AppStore` data is being altered, which brings us to the next point...

## React Components: Higher and Lower Level

Dan Abramov wrote a great post on the concept of smart and dumb components. The idea is to keep data-altering actions just in the higher-level (smart) components, while the lower-level (dumb) components take the data

they're given through props and render UI based on that data. Any time there's an action performed on a lower-level component, that event is passed up through props to the higher-level components in order to be processed into an action. Then it redistributes the data (one-way data flow) back through the application.

Said that, let's start building some components. To do that, create a folder called `components`. Inside it, create a file called `App.js` with this content:

```
// App.js
import React, { Component } from 'react'

// Dispatcher
import AppDispatcher from '../dispatcher/AppDispatcher'

// Store
import AppStore from '../stores/AppStore'

// Components
import Nav from './Partials/Nav'
import Footer from './Partials/Footer'
import Loading from './Partials/Loading'

export default class App extends Component {

  // Add change listeners to stores
  componentDidMount(){
    AppStore.addChangeListener(this._onChange.bind(this))
  }

  // Remove change listeners from stores
  componentWillUnmount(){
    AppStore.removeChangeListener(this._onChange.bind(this))
  }

  _onChange(){
    this.setState(AppStore)
  }
```

```
  getStore(){
    AppDispatcher.dispatch({
      action: 'get-app-store'
    })
  }

  render(){

    const data = AppStore.data

    // Show loading for browser
    if(!data.ready){

      document.body.className = ''
      this.getStore()

      let style = {
        marginTop: 120
      }
      return (
        <div className="container text-center" style={ style }>
          <Loading />
        </div>
      )
    }

    // Server first
    const Routes = React.cloneElement(this.props.children, { data: data })

    return (
      <div>
        <Nav data={ data }/>
        { Routes }
        <Footer data={ data }/>
      </div>
    )
  }
}
```

In our `App.js` component, we've attached an event listener to our `AppStore` that will re-render the state when `AppStore` emits an `onChange` event. This re-

rendered data will then be passed down as props to the child components. Also note that we've added a `getStore` method that will dispatch the `get-app-store` action to render our data on the client side. Once the data has been fetched from the Cosmic JS API, it will trigger an `AppStore` change that will include `AppStore.data.ready` set to `true`, remove the loading sign and render our content.

## Page Components

To build the first page of our blog, create a `Pages` folder. Inside it, we'll create a file called `Blog.js` with the following code:

```
// Blog.js
import React, { Component } from 'react'
import _ from 'lodash'
import config from '../../config'

// Components
import Header from '../Partials/Header'
import BlogList from '../Partials/BlogList'
import BlogSingle from '../Partials/BlogSingle'

// Dispatcher
import AppDispatcher from '../../dispatcher/AppDispatcher'

export default class Blog extends Component {

  componentWillMount(){
    this.getPageData()
  }

  componentDidMount(){
    const data = this.props.data
    document.title = config.site.title + ' | ' + data.page.title
  }

  getPageData(){
    AppDispatcher.dispatch({
```

```
    action: 'get-page-data',
    page_slug: 'blog',
    post_slug: this.props.params.slug
  })
}

getMoreArticles(){
  AppDispatcher.dispatch({
    action: 'get-more-items'
  })
}

render(){

  const data = this.props.data
  const globals = data.globals
  const pages = data.pages
  let main_content

  if(!this.props.params.slug){

    main_content = &lt;BlogList getMoreArticles={ this.getMoreArticles }
    ↪data={ data }/&gt;

  } else {

    const articles = data.articles

    // Get current page slug
    const slug = this.props.params.slug
    const articles_object = _.keyBy(articles, 'slug')
    const article = articles_object[slug]
    main_content = &lt;BlogSingle article={ article } /&gt;

  }

  return (
    <div>
      <Header data={ data }/>
      <div id="main-content" className="container">
        <div className="row">
```

```
            <div className="col-lg-8 col-lg-offset-2 col-md-10 col-md-offset-1">
            { main_content }
            </div>
          </div>
        </div>
      </div>
    )
  }
}
```

This page is going to serve as a template for our blog list page (home) and our single blog pages. Here we've added a method to our component that will get the page data prior to the component mounting using the React lifecycle `componentWillMount` method. Then, once the component has mounted at `componentDidMount()`, we'll add the page title to the `<title>` tag of the document.

Along with some of the rendering logic in this higher-level component, we've included the `getMoreArticles` method. This is a good example of a call to action that's stored in a higher-level component and made available to lower-level components through props.

Let's now get into our `BlogList` component to see how this works.

Create a new folder called `Partials`. Then, inside it, create a file called `BlogList.js` with the following content:

```
// BlogList.js
import React, { Component } from 'react'
import _ from 'lodash'
import { Link } from 'react-router'

export default class BlogList extends Component {

  scrollTop(){
    $('html, body').animate({
```

```
    scrollTop: $("#main-content").offset().top
  }, 500)
}

render(){

  let data = this.props.data
  let item_num = data.item_num
  let articles = data.articles

  let load_more
  let show_more_text = 'Show More Articles'

  if(data.loading){
    show_more_text = 'Loading...'
  }

  if(articles && item_num <= articles.length){
    load_more = (
      <div>
        <button className="btn btn-default center-block" onClick={ this.props.
        ↪getMoreArticles.bind(this) }>
        { show_more_text }
        </button>
      </div>
    )
  }

  articles = _.take(articles, item_num)

  let articles_html = articles.map(( article ) => {
    let date_obj = new Date(article.created)
    let created = (date_obj.getMonth()+1) + '/' + date_obj.getDate() + '/' +
    ↪date_obj.getFullYear()
    return (
      <div key={ 'key-' + article.slug }>
        <div className="post-preview">
          <h2 className="post-title pointer">
            <Link to={ '/blog/' + article.slug } onClick={ this.scrollTop }>
            ↪{ article.title }</Link>
          </h2>
```

```
            <p className="post-meta">Posted by <a href="https://cosmicjs.com"
            ↪target="_blank">Cosmic JS</a> on { created }</p>
          </div>
          <hr/>
        </div>
      )
    })

    return (
      <div>
        <div>{ articles_html }</div>
        { load_more }
      </div>
    )
  }
}
```

In our `BlogList` component, we've added an `onClick` event to our `Show More Articles` button. The latter executes the `getMoreArticles` method that was passed down as props from the higher-level page component. When that button is clicked, the event bubbles up to the `Blog` component and then triggers an action on the `AppDispatcher`. `AppDispatcher` acts as the middleman between our higher-level components and our `AppStore`.

For the sake of brevity, we're not going to build out all of the `Page` and `Partial` components in this tutorial, so please download the GitHub repo and add them from the `components` folder.

Chapter 8

# AppDispatcher

The *AppDispatcher* is the operator in our application that accepts information from the higher-level components and distributes actions to the store, which then re-renders our application data.

To continue this tutorial, create a folder named *dispatcher*. Inside it, create a file called *AppDispatcher.js*, containing the following code:

```
// AppDispatcher.js
import { Dispatcher } from 'flux'
import { getStore, getPageData, getMoreItems } from '../actions/actions'

const AppDispatcher = new Dispatcher()

// Register callback with AppDispatcher
AppDispatcher.register((payload) => {

  let action = payload.action

  switch(action) {

    case 'get-app-store':
      getStore()
      break

    case 'get-page-data':
      getPageData(payload.page_slug, payload.post_slug)
      break

    case 'get-more-items':
      getMoreItems()
      break

    default:
      return true

  }

  return true
```

```
})

export default AppDispatcher
```

We've introduced the `Flux` module into this file to build our dispatcher. Let's add our actions now.

## Actions: Last Stop Before the Store

To start, let's create an `actions.js` file inside a newly created folder called `actions`. This file will feature the following content:

```
// actions.js
import config from '../config'
import Cosmic from 'cosmicjs'
import _ from 'lodash'

// AppStore
import AppStore from '../stores/AppStore'

export function getStore(callback){

  let pages = {}

  Cosmic.getObjects(config, function(err, response){

    let objects = response.objects

    /* Globals
    ======================== */
    let globals = AppStore.data.globals
    globals.text = response.object['text']
    let metafields = globals.text.metafields
    let menu_title = _.find(metafields, { key: 'menu-title' })
    globals.text.menu_title = menu_title.value

    let footer_text = _.find(metafields, { key: 'footer-text' })
    globals.text.footer_text = footer_text.value
```

```
let site_title = _.find(metafields, { key: 'site-title' })
globals.text.site_title = site_title.value

// Social
globals.social = response.object['social']
metafields = globals.social.metafields
let twitter = _.find(metafields, { key: 'twitter' })
globals.social.twitter = twitter.value
let facebook = _.find(metafields, { key: 'facebook' })
globals.social.facebook = facebook.value
let github = _.find(metafields, { key: 'github' })
globals.social.github = github.value

// Nav
const nav_items = response.object['nav'].metafields
globals.nav_items = nav_items

AppStore.data.globals = globals

/* Pages
======================== */
let pages = objects.type.page
AppStore.data.pages = pages

/* Articles
======================== */
let articles = objects.type['post']
articles = _.sortBy(articles, 'order')
AppStore.data.articles = articles

/* Work Items
======================== */
let work_items = objects.type['work']
work_items = _.sortBy(work_items, 'order')
AppStore.data.work_items = work_items

// Emit change
AppStore.data.ready = true
AppStore.emitChange()
```

```
    // Trigger callback (from server)
    if(callback){
      callback(false, AppStore)
    }

  })
}

export function getPageData(page_slug, post_slug){

  if(!page_slug || page_slug === 'blog')
    page_slug = 'home'

  // Get page info
  const data = AppStore.data
  const pages = data.pages
  const page = _.find(pages, { slug: page_slug })
  const metafields = page.metafields
  if(metafields){
    const hero = _.find(metafields, { key: 'hero' })
    page.hero = config.bucket.media_url + '/' + hero.value

    const headline = _.find(metafields, { key: 'headline' })
    page.headline = headline.value

    const subheadline = _.find(metafields, { key: 'subheadline' })
    page.subheadline = subheadline.value
  }

  if(post_slug){
    if(page_slug === 'home'){
      const articles = data.articles
      const article = _.find(articles, { slug: post_slug })
      page.title = article.title
    }
    if(page_slug === 'work'){
      const work_items = data.work_items
      const work_item = _.find(work_items, { slug: post_slug })
      page.title = work_item.title
    }
  }
```

```
  AppStore.data.page = page
  AppStore.emitChange()
}

export function getMoreItems(){

  AppStore.data.loading = true
  AppStore.emitChange()

  setTimeout(function(){
    let item_num = AppStore.data.item_num
    let more_item_num = item_num + 5
    AppStore.data.item_num = more_item_num
    AppStore.data.loading = false
    AppStore.emitChange()
  }, 300)
}
```

There are a few methods here that are exposed by this `actions.js` file. `getStore()` connects to the Cosmic JS API to serve our blog's content. `getPageData()` gets the page data from a provided `slug` (or page key). `getMoreItems()` controls how many items will be seen in our `BlogList` and `WorkList` components.

When `getMoreItems()` is triggered, it first sets `AppStore.data.loading` to `true`. Then, 300 milliseconds later (for effect), it allows five more items to be added to our list of blog posts or work items. Finally, it sets `AppStore.data.loading` to `false`.

## Configure Your Cosmic JS CMS

To begin receiving data from your cloud-hosted content API on Cosmic JS, let's create a `config.js` file. Open this file and paste the following content:

```
// config.js
export default {
```

```
  site: {
    title: 'React Universal Blog'
  },
  bucket: {
    slug: process.env.COSMIC_BUCKET || 'react-universal-blog',
    media_url: 'https://cosmicjs.com/uploads',
    read_key: process.env.COSMIC_READ_KEY || '',
    write_key: process.env.COSMIC_WRITE_KEY || ''
  },
}
```

This means content will be coming from the Cosmic JS bucket `react-universal-blog`. To create content for your own blog or app, sign up for a free account with Cosmic JS. When asked to "Add a New Bucket", click "Install Starter Bucket" and you'll be able to follow the steps to install the "React Universal Blog". Once this is done, you can add your unique bucket's slug to this config file.

Chapter

# Server-Side Rendering

9

Now that we have most of our React components and Flux architecture set up, let's finish up by editing our `app-server.js` file to render everything in server-side production. This file will have the following code:

```
// app-server.js
import React from 'react'
import { match, RoutingContext, Route, IndexRoute } from 'react-router'
import ReactDOMServer from 'react-dom/server'
import express from 'express'
import hogan from 'hogan-express'
import config from './config'

// Actions
import { getStore, getPageData } from './actions/actions'

// Routes
import routes from './routes'

// Express
const app = express()
app.engine('html', hogan)
app.set('views', __dirname + '/views')
app.use('/', express.static(__dirname + '/public/'))
app.set('port', (process.env.PORT || 3000))

app.get('*',(req, res) => {

  getStore(function(err, AppStore){

    if(err){
      return res.status(500).end('error')
    }

    match({ routes, location: req.url }, (error, redirectLocation, renderProps) => {

      // Get page data for template
      const slug_arr = req.url.split('/')
      let page_slug = slug_arr[1]
      let post_slug
      if(page_slug === 'blog' || page_slug === 'work')
```

```
        post_slug = slug_arr[2]
      getPageData(page_slug, post_slug)
      const page = AppStore.data.page
      res.locals.page = page
      res.locals.site = config.site

      // Get React markup
      const reactMarkup = ReactDOMServer.renderToStaticMarkup(<RoutingContext
      ↪{...renderProps} />)
      res.locals.reactMarkup = reactMarkup

      if (error) {

        res.status(500).send(error.message)

      } else if (redirectLocation) {

        res.redirect(302, redirectLocation.pathname + redirectLocation.search)

      } else if (renderProps) {

        // Success!
        res.status(200).render('index.html')

      } else {

        res.status(404).render('index.html')

      }
    })

  })
})

app.listen(app.get('port'))

console.info('==> Server is listening in ' + process.env.NODE_ENV + ' mode')
console.info('==> Go to http://localhost:%s', app.get('port'))
```

This file uses our `getStore` action method to get our content from the Cosmic JS API server-side, then goes through React Router to determine which

component will be mounted. Everything will then be rendered into static markup with `renderToStaticMarkup`. This output is then stored in a template variable to be used by our `views/index.html` file.

Once again, let's update the `scripts` section of our `package.json` file so that it looks like the one shown below:

```
"scripts": {
    "start": "npm run production",
    "production": "rm -rf public/index.html && NODE_ENV=production webpack -p &&
    ↪NODE_ENV=production babel-node app-server.js --presets es2015",
    "webpack-dev-server": "NODE_ENV=development PORT=8080 webpack-dev-server
    ↪--content-base public/ --hot --inline --devtool inline-source-map
    ↪--history-api-fallback",
    "development": "cp views/index.html public/index.html && NODE_ENV=development
    ↪webpack && npm run webpack-dev-server"
  },
```

We can now run in development mode with hot reloading and we can run in production mode with server-rendered markup. Run the following command to run the full React Universal Blog Application in production mode:

```
npm start
```

Our blog is now ready to view at http://localhost:3000. It can be viewed on the server side, the browser side, and our content can be managed through Cosmic JS, our cloud-hosted content platform.

## OK, so... now what?

This was a wide-ranging project that introduced you to many concepts, all of which will be relevant and useful in your future React adventures.

Which adventures, you ask? Here are some ideas for where you can go next:

- Build a new blog and introduce yourself to GraphQL, the new hotness in APIs. This tutorial also includes CosmicJS, so you're well on your way.
- Learn how to integrate user authentication in a React app.
- Integrate a React app with Rails, if you're Rails-inclined.
- Build a to-do app, a Reddit clone, a game–even some procedurally-generated terrain with the *5 Practical React Projects* book.
- And a guide to writing *better* React code, while we're at it.