MANNING

# Isomorphic Web Applications

Universal development with React

SAMPLE CHAPTER

Elyse Kolker Gordon

*Isomorphic Web Applications*
*Universal development with React*
by Elyse Kolker Gordon

**Sample Chapter 1**

# *brief contents*


**PART 1 FIRST STEPS ..... 1**

1 ■ Introduction to isomorphic web application architecture 3
2 ■ A sample isomorphic app 22

**PART 2 ISOMORPHIC APP BASICS ..... 51**

3 ■ React overview 53
4 ■ Applying React 78
5 ■ Tools: webpack and Babel 96
6 ■ Redux 113

**PART 3 ISOMORPHIC ARCHITECTURE ..... 133**

7 ■ Building the server 135
8 ■ Isomorphic view rendering 161
9 ■ Testing and debugging 184
10 ■ Handling server/browser differences 203
11 ■ Optimizing for production 222

**PART 4 APPLYING ISOMORPHIC ARCHITECTURE WITH OTHER TOOLS ..... 239**

12 ■ Other frameworks: implementing isomorphic without React 241
13 ■ Where to go from here 269

# Part 1

# First steps

Understanding what an isomorphic app is and why you'd want to build one is an important first step in learning about isomorphic architecture. The first part of this book explores the why and how of isomorphic apps with a bird's-eye view, giving you the context you need in order to comprehend the specific implementation details presented in later sections.

In chapter 1, you'll learn all the reasons to build an isomorphic app. This chapter also gives you an overview of the All Things Westies app you'll build later in the book. In chapter 2, you'll work through building an example app with the technologies used in the book: React, Node.js, webpack, and Babel. Instead of covering all the small details, this chapter allows you to see how the pieces fit together.

# 1 Introduction to isomorphic web application architecture

***This chapter covers***

- Differentiating between isomorphic, server-side rendered, and single-page apps
- Server rendering and the steps involved in transitioning from a server-rendered to a single-page app experience
- Understanding the advantages and challenges of isomorphic web apps
- Building isomorphic web apps with React's virtual DOM
- Using Redux to handle the business logic and data flow
- Bundling modules with dependencies via webpack

This book is intended for web developers looking to expand their architectural toolset and better understand the options available for building web apps. If you've ever built a single-page or server-rendered web app (say, with Ruby on Rails), you'll have an easier time following the content in this book. Ideally, you're comfortable with JavaScript, HTML, and CSS. If you're new to web development, this book isn't for you.

Historically, web apps and websites have come in two forms: server-rendered and single-page apps (SPAs). *Server-rendered apps* handle each action the user takes by making a new request to the server. In contrast *SPAs* handle loading the content and responding to user interactions entirely in the browser. *Isomorphic web apps* are a combination of these two approaches.

This book aspires to take a complex application architecture and break it into repeatable and understandable bits. By the end of this book, you'll be able to create a content site or an e-commerce web app with the following techniques:

- Render any page on the server by using React to achieve fast perceived performance and fully render pages for search engine optimization (SEO) crawlers (such as Googlebot).
- Choose not to render certain features on the server. Understand how to use the React lifecycle to achieve this.
- Handle user sessions on both the server and the browser.
- Implement single-direction data flow with Redux, making prefetching data on the server and rendering in the browser feasible.
- Use webpack and Babel to enable a modern JavaScript workflow.

## 1.1 Isomorphic web app overview

My team and I had a big problem: our SEO rendering system was brittle and eating up valuable time. Instead of building new features, we were troubleshooting why Googlebot was seeing a different version of our app from what our users were seeing. The system was complex, involved a third-party provider, and wasn't scaling well for our needs, so we moved forward with a new type of app—an isomorphic one.

An *isomorphic app* is a web app that blends a server-rendered web app with a single-page application. On the one hand, we want to take advantage of fast perceived performance and SEO-friendly rendering from the server. On the other hand, we want to handle complex user actions in the browser (for example, opening a modal). We also want to take advantage of the browser push history and XMLHttpRequest (XHR). These technologies prevent us from making a server request on every interaction.

To get started understanding all of this, you're going to use an example web app called All Things Westies (you'll build this app later in the book, starting in chapter 4). On this site, you can find all kinds of products to buy for your Westie (West Highland white terrier—a small, white dog). You can purchase dog supplies and buy products featuring Westies (socks, mugs, shirts, and so forth). If you're not a pet owner, you might find this example ridiculous. As a dog owner, even I thought it was over the top. But it turns out that dog products such as mugs are a huge thing. If you don't believe me, search Google for "pug mugs."

Because this is an e-commerce app, we care about having good SEO. We also want our customers to have a great experience with performance in the app. This makes it an ideal use case for isomorphic architecture.

### 1.1.1 Understanding how it works

Look at figure 1.1, which is a wireframe for the All Things Westies app. There's a standard header with some main site navigation on the right. Below the header, the main content areas promote products and the social media presence.

The first time you come to the site, the app content is rendered on the server using server-rendered techniques with Node.js. After being server-rendered, the content is sent to the browser and displayed to the user. As the user navigates around the pages, looking for a dog mug or supplies, each page is rendered by the JavaScript running in the browser and using SPA techniques.

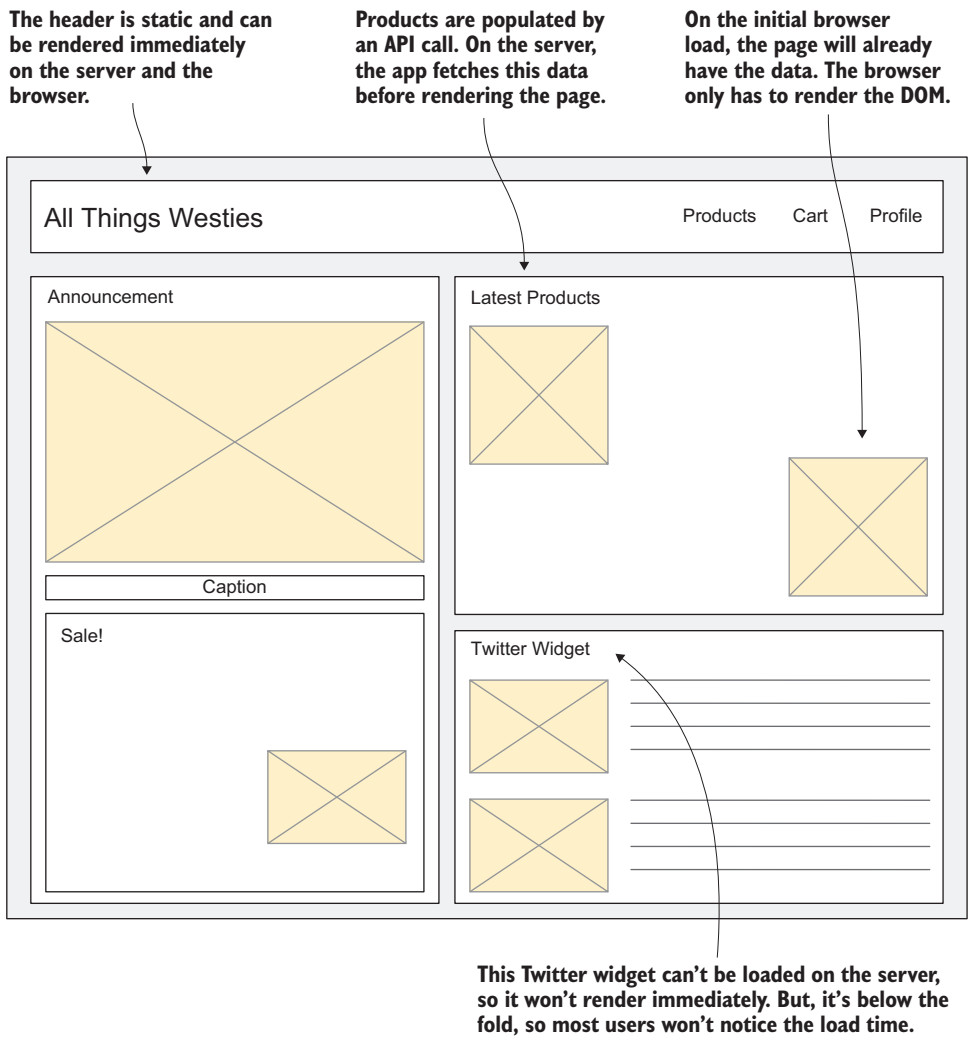


Figure 1.1 A wireframe showing the homepage for All Things Westies, an isomorphic web app

The All Things Westies app relies on reusing as much code as possible between the server and the browser. The app relies on JavaScript's ability to run in multiple environments: JavaScript runs in browsers and on the server via Node.js. Although JavaScript can run in other environments as well (for example, on Internet of Things devices and on mobile devices via React Native), the focus here is on web apps that run in the browser.

Many of the concepts in this book could be applied without writing all the code in JavaScript. Historically, the complexity of running an isomorphic app without being able to reuse code has been prohibitive. Although it's possible to server-render your site with Java or Ruby and then transition to a single-page app, it isn't commonly done because it requires duplicating large portions of code in two languages. That requires more maintenance.

To see this flow in action, look at figure 1.2. It shows how the code for All Things Westies gets deployed to the server and the browser. The server code is packaged and run on a Node.js web server, and the browser code is bundled into a file that's later downloaded in the browser. Because we take advantage of JavaScript running in both

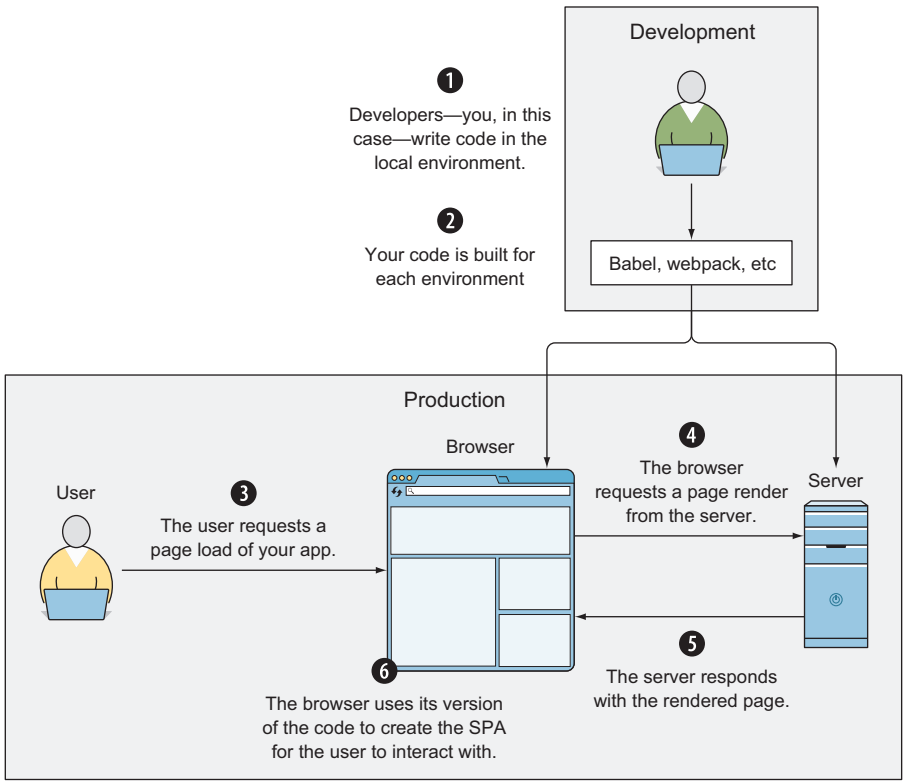


**Figure 1.2 Isomorphic apps build and deploy the same JavaScript code to both environments.**

environments, the same code that runs in the browser and talks to our API or data source also runs on the server to talk to our back end.

### 1.1.2 Building our stack

Building an app such as All Things Westies requires putting together several well-known technologies. Many of the concepts in this book are executed with open source libraries. Although you could build an isomorphic app using few or no libraries, I highly recommend taking advantage of the JavaScript communities' efforts in this area.

> **TIP** Make sure any libraries you include in an isomorphic app support running in both the server and browser environments. Check out chapter 10 for what to watch for and how to handle differences in environments. If you intend to use a library only on the server, you don't need to check for browser compatibility.

The HTML components that display the products (the view) will be built with React (in chapter 12, you'll explore how to use other popular frameworks, including Angular 2 and Ember, to implement isomorphic architecture). You'll use a single-direction data flow via Redux, the current community standard data management in React apps. You'll use webpack to compile the code that runs in the browser and to enable running Node.js packages in the browser.

On the server side, you'll build a Node.js server using Express to handle routing. You'll take advantage of React's ability to render on the server and use it to build up a complete HTML response that can be served to the browser. Table 1.1 shows how all these pieces fit together.

**Table 1.1 The technologies used in an isomorphic app and the environments they run in**

| Library (version) | Server | Browser | Build tool |
|---|---|---|---|
| Node.js (6.9.2) | ✓ | | |
| Express (4.15.3) | ✓ | | |
| React (15.6.1) | ✓ | ✓ | |
| React Router (3.0.5) | ✓ | ✓ | |
| Redux (3.7.2) | ✓ | ✓ | |
| Babel (6.25.0) | ✓ | ✓ | ✓ |
| webpack (3.4.1) | | ✓ | ✓ |

To make our application work everywhere, you'll build in data prefetching for your routes using React Router. You'll also handle differences in environments by building separate code entry points for the server and browser. If code can be run only in the browser, you'll gate the code or take advantage of the React lifecycle to ensure that the code won't run on the server. I introduce React in chapter 3 and the specifics of the server logic in chapter 7.

## 1.2 Architecture overview

Earlier in this chapter, I told you that an isomorphic application is the result of combining a server-rendered application and a single-page application. To get a better understanding of how to connect the concepts of a server-rendered application and a single-page application, see figure 1.3. This figure shows all the steps involved in getting an isomorphic app rendered and responding to user input, like a single-page application, starting when the user enters the web address.

### 1.2.1 Understanding the application flow

Every web app session is initiated when a user navigates to the web app or types the URL into the browser window. For allthingswesties.com, when a user clicks a link to the app from an email or from searching on Google, the flow on the server goes through the following steps (the numbers match those in figure 1.3):

1. The browser initiates the request.
2. The server receives the request.
3. The server determines what needs to be rendered.
4. The server gathers the data required for the part of our application being requested. If the request is for allthingswesties.com/product/mugs, the app requests the list of gift items for sale through the site. This list of mugs, along with all the information to be displayed (names, descriptions, price, images), is collected before moving on to the render step.
5. The server generates the HTML for our web page using the data collected for the mugs page.
6. The server responds to the request for allthingswesties.com/product/mugs with the fully built HTML.

The next part of the application cycle is the initial load in the browser. We differentiate the first time the user loads the app from subsequent requests because several things that will happen only once per session happen during this first load.

> **DEFINITION** *Initial load* is the first time the user interacts with your website. This means the first time the user clicks a link to your site in a Google search or from social media, or types it directly into the web address bar.

The first load on the browser begins as soon as the HTML response from the server is received and the DOM is able to be processed. At this point, single-page application flow takes over, and the app responds to user input, browser events, and timers. The user can add products to their cart, navigate around the site, and interact with forms.

7. The browser renders the markup received from the server.
8. The application is now able to respond to user input.
9. When the user adds an item to their cart, the code responds and runs any business logic necessary.
10. If required, the browser talks to the back end to fetch data.

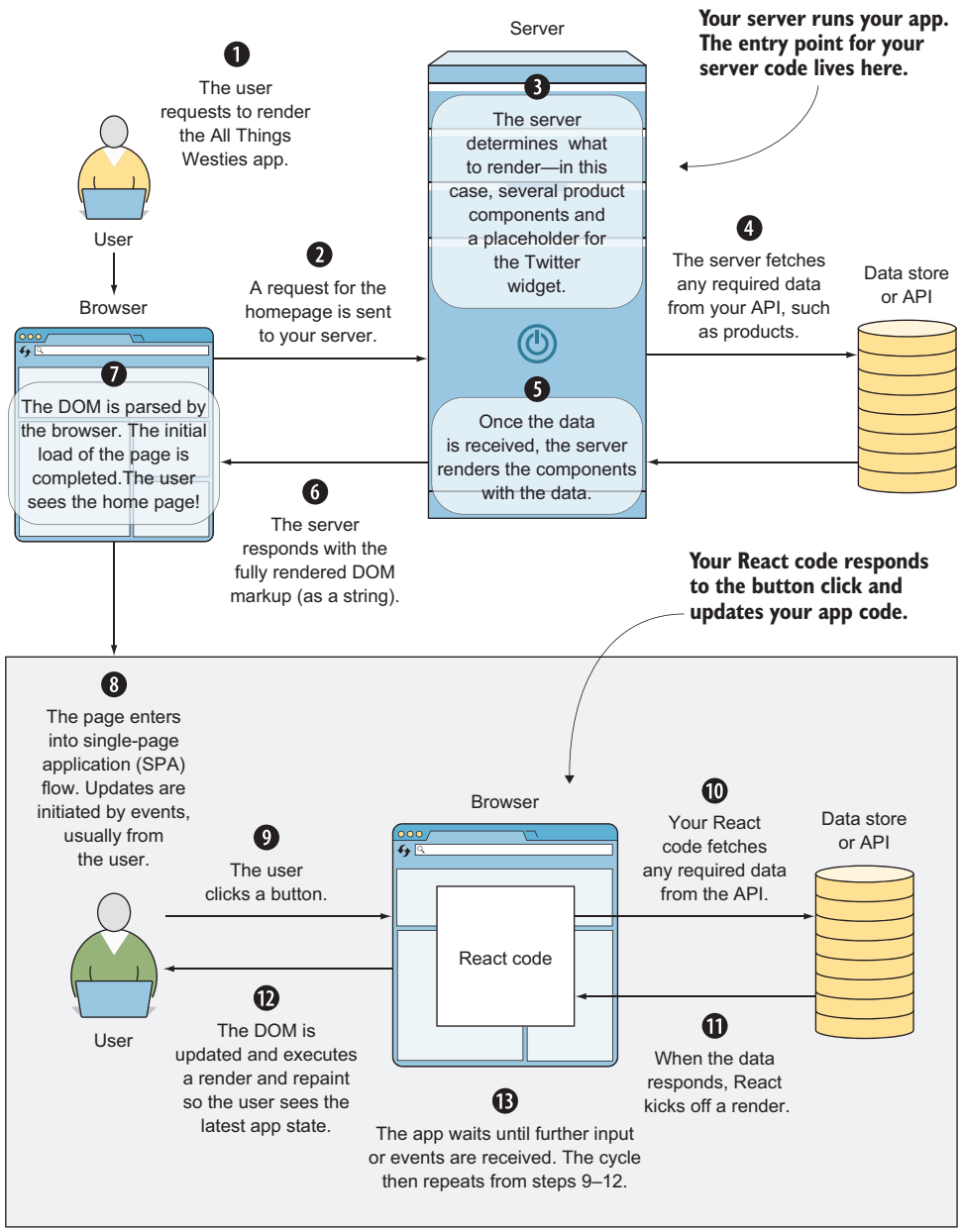


**Figure 1.3 The isomorphic app flow from initial browser request to SPA cycle**

11 React renders the components.

12 Updates are made, and any repaints are executed. For instance, the user's cart icon updates to show that an item has been added.

13 Each time the user interacts with the app, steps 9–12 repeat.

### 1.2.2 Handling the server-side request

Now let's take a closer look at what happens when the server receives the initial request to render the page. Look at what part of the site renders on the server. Figure 1.4 is similar to figure 1.1, except that it doesn't render the Twitter widget. The Twitter widget is designed to be loaded in the browser, so it doesn't render on the server.

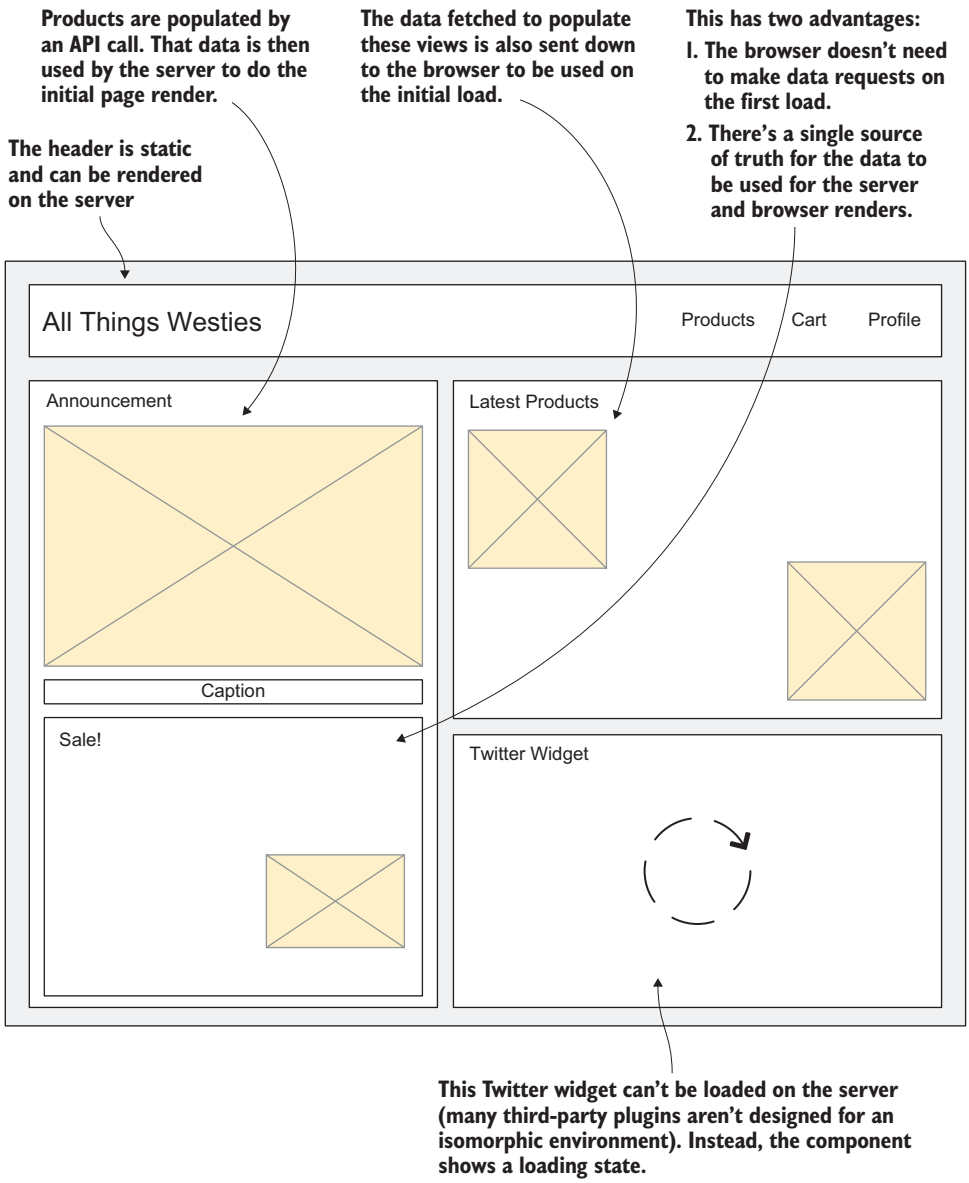


Figure 1.4 The server-rendered version of the All Things Westies homepage

The server does three important things. First, it fetches the data required for the view. Then it takes that data and uses it to render the DOM. Finally, it attaches that data to the DOM so the browser can read in the app state. Figure 1.5 shows the flow on the server.

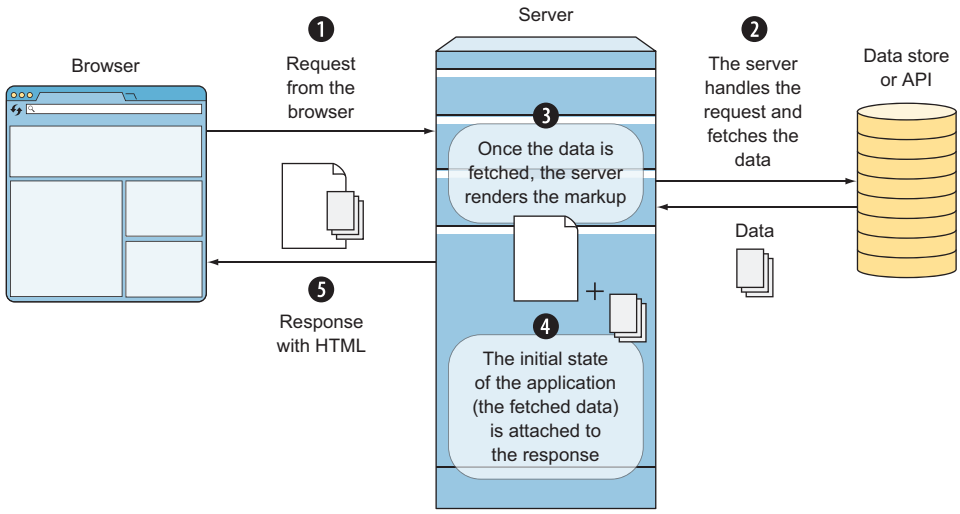


**Figure 1.5 App flow for the initial server render**

Let's step through the flow:

1. The server receives a request.
2. The server fetches the required data for that request. This can be from either a persistent data store such as a MySQL or NoSQL database or from an external API.
3. After the data is received, the server can build the HTML. It generates the markup with React's virtual DOM via React's `renderToString` method.
4. The server injects the data from step 2 into your HTML so the browser can access it later.
5. The server responds to the request with your fully built HTML.

### 1.2.3 Rendering in the browser

Now let's look more closely at what happens in the browser. Figure 1.6 shows the flow in the browser, from the point the browser receives the HTML to the point it bootstraps the app:

1. The browser starts to render the mugs page immediately because the HTML sent by the server is fully formed with all the content you generated on the

server. This includes the header and the footer of your app along with the list of mugs for purchase. *The app won't respond to user input yet. Things like adding a mug to the cart or viewing the detail page for a specific mug won't work.*

2. When the browser reaches the JavaScript entry for our application, the application bootstraps.
3. The virtual DOM is re-created in React. Because the server sent down the app state, this virtual DOM is identical to the current DOM.
4. Nothing happens! React finds no differences between the DOM and the virtual DOM it built (the virtual DOM is explained in depth in chapter 3). The user is already being shown the list of mugs in the browser. *The application can now respond to user input, such as adding a mug to the cart.*

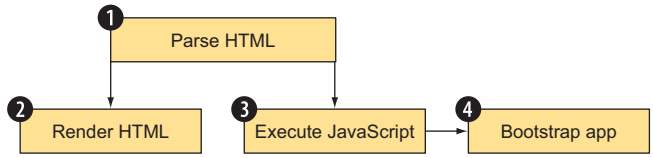


**Figure 1.6 Browser render and bootstrap—between steps 1 and 4, the app won't respond to user input.**

This is when the single-page application flow kicks in again. This is the most straightforward part. It handles user events, makes XHR calls, and updates the application as needed.

## 1.3 Advantages of isomorphic app architecture

At this point, you may be thinking this sounds complicated. You may be wondering why this approach to building a web app would ever be worth it. There are several compelling reasons to go down this path:

- Simplified and improved SEO—bots and crawlers can read all the data on page load.
- Performance gains in user-perceived performance.
- Maintenance gains.
- Improved accessibility because the user can view the app without JavaScript.

Isomorphic app architecture also has challenges and trade-offs. There's increased complexity in managing and deploying code running in multiple environments. Debugging and testing are more complicated. Server-rendered HTML via Node.js and React can be slow for views that have many components. For example, a page that displays many items for sale might quickly end up with hundreds of React components. As this number increases, the speed at which React can build these components on the server declines. First, I'll cover the benefits of building an isomorphic app. Let's start by discussing SEO.

### 1.3.1 SEO benefits

Our example app, All Things Westies, is an e-commerce site, so to be successful it needs shoppers! And it needs good SEO to maximize the number of people who come to the app from search engines. Single-page applications are difficult for search-engine bots to crawl because they don't load the data for the app until after the JavaScript has run in the browser. Isomorphic apps also need to bootstrap after JavaScript is run, but because their content is rendered by the server, neither users nor bots have to wait for the application to bootstrap in order to see the content of the site.

> **DEFINITION** *Bootstrapping* an application means running the code required to get everything set up. This code is run only once on the initial load of the application and is run from the entry point of the browser application.

On the All Things Westies app, you want to make sure all the SEO-relevant content is fetched on the server so you don't rely on the SEO crawlers to try to render your page. Crawlers (both searchbots such as Google or Bing and sharebots such as Facebook) either can't run all this code or don't want to wait long enough for this code to run. For example, Google will try to run JavaScript but penalizes sites that take too long for the content to load. That can be seen in the warning shown in figure 1.7. This warning shows up when you enter a URL for a single-page application into the Google PageSpeed Insights tool.

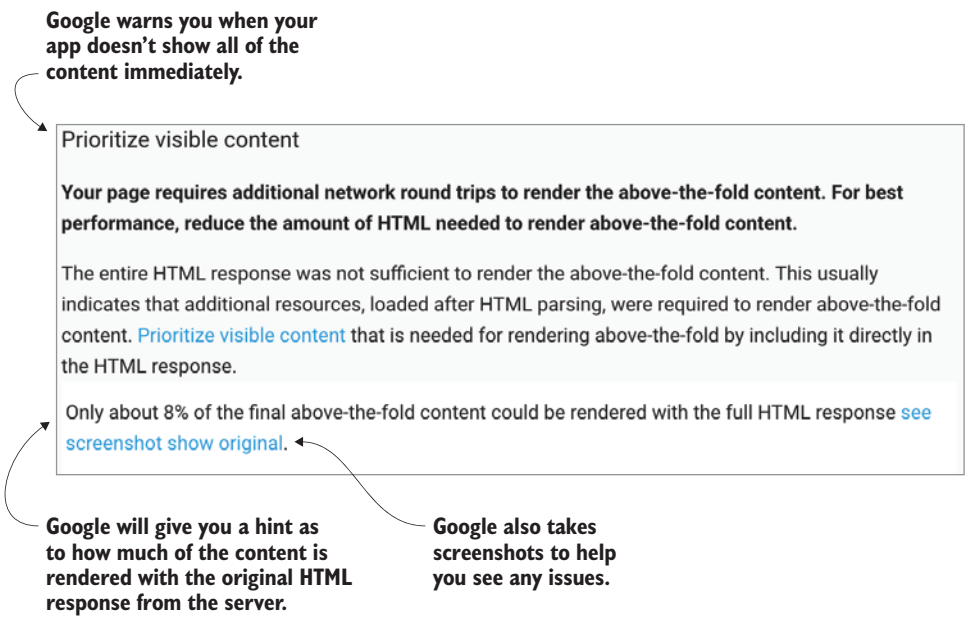


**Figure 1.7 Google PageSpeed Insights presents a warning for a single-page application. The application makes too many AJAX calls to fetch visible content after the initial load of the page.**

**Google PageSpeed Insights tool**

Google's PageSpeed Insights tool helps measure how your page is doing on a scale of 0 to 100. You get a score for both speed-related issues (size of images, size of JavaScript, magnification, round trips made, and so forth) and UI (size of click areas, for example). Test it on your web app at https://developers.google.com/speed/pagespeed/insights.

Google also has the Lighthouse tool (available as a Chrome extension or command-line tool), which will run an in-depth analysis of pages on your site. It makes recommendations on everything from performance, to using service workers to allow offline use, to improved accessibility for screen readers. You can learn more about Lighthouse at https://developers.google.com/web/tools/lighthouse/.

If you don't deal with this warning, you may end up with a lower ranking and fewer customers. Also, there's no guarantee that any page content that relies on API calls will be run by the crawler. Whole services have popped into existence to solve this problem for single-page apps. Dev teams pour time into developing systems to crawl and prerender their pages. They then redirect bots to these prerendered pages. These systems are complex and brittle to maintain.

Personally, I can't wait for the day when crawlers and bots will be able to get to all our content regardless of when the data is fetched (on the server or in the browser). Until that day, server-rendering the initial content gives a big advantage over single-page application rendering. This is especially true for above-the-fold content and any other content that has SEO benefits.

**DEFINITION** *Above-the-fold* is a term that comes from the newspaper business. It refers to all the content that shows on the front page when a newspaper is folded in half and sitting on a newsstand. For web apps, this term is used to refer to all the content that's in the viewable area of a user's screen when the app loads. To see below-the-fold content, the user must scroll.

In addition to SEO crawlers, many social sites and apps that allow inline website previews (for example, Facebook, Twitter, Slack, or WhatsApp), also use bots that don't run JavaScript. These sites assume that all content that's available to build a social card or inline preview will be available on the server-rendered page. Isomorphic apps are ideal for handling the social bot use case.

At the beginning of this section, I mentioned that both bots and users don't need to wait for the isomorphic application to bootstrap to see the dynamic content. Another way to say that is that the perceived performance of isomorphic web apps is fast. The next section describes this in detail.

### 1.3.2 Performance benefits

Users want to see the content of All Things Westies right away. Otherwise, they'll get impatient and leave before seeing all the products and information being offered.

Loading an SPA can be a slow experience for a user (especially on mobile phones). Even though the browser may connect quickly to your application, it takes time to run the startup code and fetch the content, which leaves the user waiting. In the best-case scenario, SPAs display loading indicators and messaging for the user. In the worst-case scenario, there's no visual feedback, and the user is left wondering whether anything is happening.

Figure 1.8 shows what All Things Westies would look like during the initial rendering if it were a single-page application. Instead of seeing all content immediately, you'd see loading spinners in all of the content areas.

A server-rendered page displays its content (all the HTML, images, CSS, and data for your site) to the user as soon as the browser receives and renders the HTML. This leads to content being seen by the user several seconds faster than in an SPA.

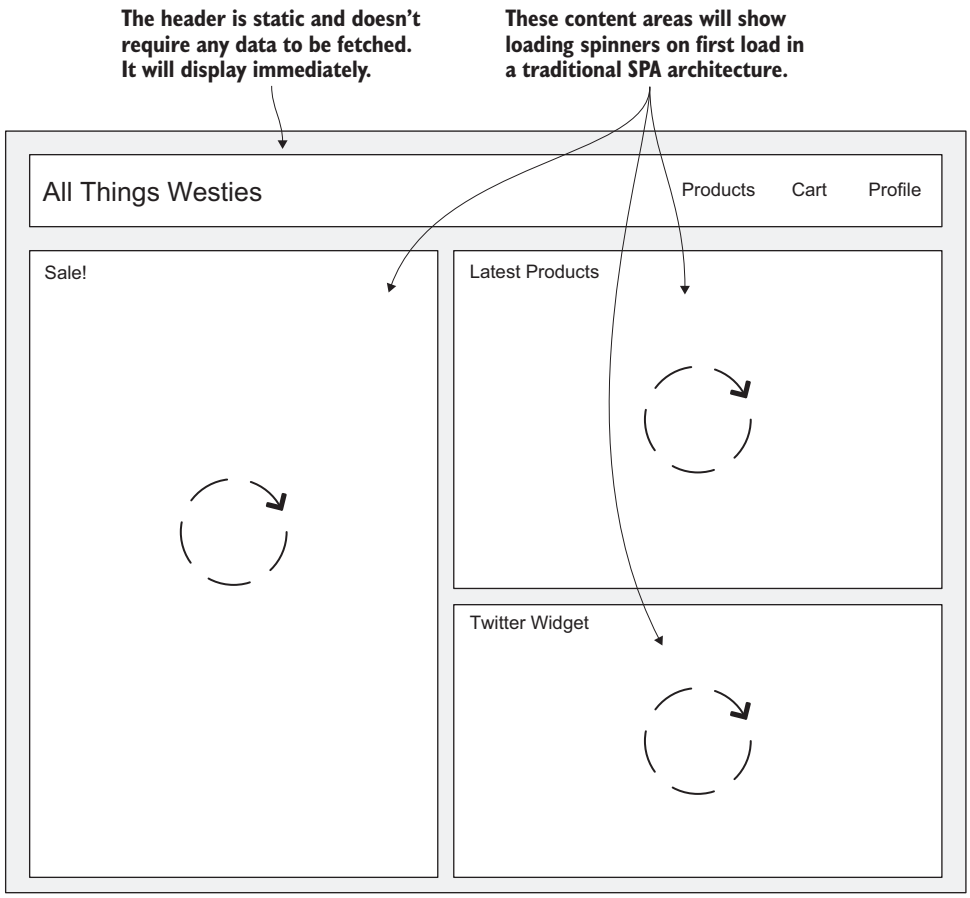


**Figure 1.8 In a single-page app version of All Things Westies, spinners would be shown during the first load instead of the real content.**

Although the site still requires JavaScript to be loaded and executed before user interactions can take place, this fast load allows the user to start visually processing your content quickly. This is called *perceived performance*. The app content is presented to the user quickly. The user isn't aware that JavaScript is being run in the background.

When this process is executed well, the user will never know that the JavaScript loaded after the view rendered. For all practical purposes, your user has a great experience because they believe the app loaded fast. This greatly reduces the need for loading spinners or other waiting states on the first load of the app. This leads to happier users. Figure 1.9 demonstrates the differences between SPA and isomorphic apps.

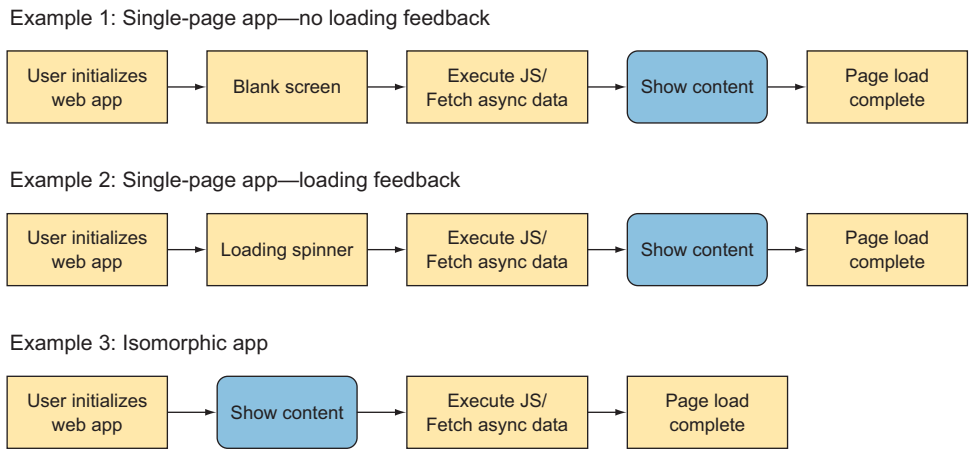


**Figure 1.9 Comparison of when the user sees the content of a web app. An isomorphic app displays its content much sooner than a single-page app.**

Now I'll walk you through the single-page app and isomorphic scenarios in detail. You can see these flows in figure 1.9 as well.

First, look at example 1. Imagine going to our example web app and being shown a blank screen for six seconds. What would you do? How likely are you to get frustrated and give up on using that web app? If you were looking to buy a pair of Westie socks, you'd be inclined to give up on All Things Westies and take your business elsewhere.

Now imagine that the web app still took six seconds to load (as in example 2), but this time it showed you a basic structure (a loading spinner) to let you know that the web app is doing something but you can't interact with it yet, just as in figure 1.8 previously. Are you willing to wait for this site to load?

Finally, let's imagine that when you come to All Things Westies, you see the content in under two seconds, as shown in example 3. This flow matches that of figure 1.1 at the beginning of the chapter. This time, your brain starts processing the information as

soon as it's displayed. You don't feel like you had to wait. In the background, the app is still loading and working to get everything set up, but you don't have to wait for this to finish before being able to see the content.

Notice that the app is able to show content much earlier in the page-load flow. Although the page-load time as measured by performance metrics will be the same in all three approaches, the user *perceives* the performance of an isomorphic app to be much faster.

### 1.3.3 No JavaScript? No problem!

Another user-facing benefit of isomorphic app architecture is that you can serve portions of your site without requiring JavaScript. Users who can't or don't want to run JavaScript can still benefit from using your site when it's built isomorphically. Because you serve a complete page to the browser, users can at least see your content despite not being able to interact with the app.

This allows you to use progressive enhancement to better provide for users across a spectrum of browsers and devices. Although it may be unlikely to encounter a user with no JavaScript running, there are other good reasons for loading a full page from the server. For example, if you support older browsers or devices, isomorphic apps are good tools for providing the best experience possible across a multitude of browser/device/OS combinations.

We've covered the user-facing benefit of isomorphic apps. Next we'll look at the developer benefits that come with this architecture.

### 1.3.4 Maintenance and developer benefits

When building an isomorphic app, most of the code can be run on both the server and the browser. If you want to render a view, you need to write your code only once. If you want to have helper functions for a common task in the app, you need to write this logic only once, and it'll run in both places.

This is an advantage over apps that have server-side code written in one language and browser code written in JavaScript. Developers can keep their focus without having to switch between languages. Builds, environment management, and dependencies are all simplified, which makes your overall workflow cleaner.

This isn't to say that building isomorphic apps is easy. Writing everything in one language comes with its own set of problems.

### 1.3.5 Challenges and trade-offs

Choosing to build an app with isomorphic web architecture isn't without trade-offs. For one, it requires a new way of thinking, which takes time to adjust to. The good news is that's what you'll learn in this book. Some of the challenges include the following:

- Handling the differences between Node.js and the browser
- Debugging and testing complexity
- Managing performance on the server

#### HANDLING THE DIFFERENCES BETWEEN THE SERVER AND THE BROWSER

Node.js has no concept of a window or document. The browser doesn't know about Node.js environment variables and has no idea what a request or response object is. Both environments know about cookies, but they handle them in different ways. In chapter 10, you'll look at strategies for dealing with these environment tensions.

#### DEBUGGING AND TESTING COMPLEXITY

All your code needs to be tested twice: loaded directly off the server and as part of the single-page flow. Debugging requires mastery of both browser and server debugging tools and knowing whether a bug is happening on the server, on the browser, or in both environments. Additionally, a thorough unit-testing strategy is needed, where tests are written and run in the appropriate environments. Server-only code should be tested in Node.js, but shared code should be tested in all the environments where it'll eventually be run.

#### MANAGING PERFORMANCE ON THE SERVER

Performance on the server also presents a challenge as the React-provided `renderToString` method is slow to execute on complex pages with many components. In chapter 11, I'll show you how to optimize your code as much as possible without breaking React best practices. We'll also discuss caching as a tool to minimize issues with server performance.

At this point, you understand the benefits and trade-offs that come with isomorphic app architecture. Next let's take an in-depth look at how to execute an isomorphic app.

## 1.4 Building the view with React

React is one of the pieces that makes building an isomorphic web app possible. *React* is a library, open sourced by Facebook, for creating user interfaces (the view layer in your app). React makes it easy to express your views via HTML and JavaScript. It provides a simple API that's easy to get up and running but that's designed to be composable in order to facilitate building user interfaces quickly and efficiently. Like many other view libraries and implementations, React provides a template language (JSX) and hooks into commonly used parts of the DOM and JavaScript.

React also takes advantage of functional concepts by adhering to single-direction data flows from the top-level component down to its children. What makes it appealing for isomorphic apps is how it uses a virtual DOM to manage changes and updates to the application.

React isn't a framework like Angular or Ember. It only provides the code you use to write your view components. It can fit easily into a Model-View-Controller (MVC) style architecture as the view. But there's a recommended way to build complex React apps, which is covered throughout the book.

The *virtual DOM* is a representation of the browser DOM written with JavaScript. At its core, React is composed of React elements. Since React introduced the virtual DOM to the web community, this idea has started to show up in many major libraries and frameworks. Some people are even writing their own virtual DOM implementations.

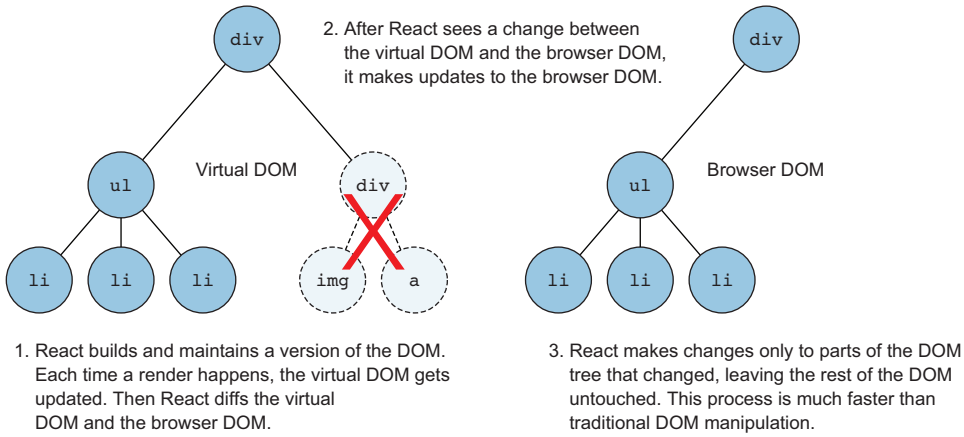


**Figure 1.10 Comparing the DOM trees: the virtual DOM changes are compared to the browser DOM. Then React intelligently updates the browser DOM tree based on the calculated diff.**

Like the browser DOM, the virtual DOM is a tree comprising a root node and its child nodes. After the virtual DOM is created, React compares the virtual tree to the current tree and calculates the updates it needs to make to the browser DOM. If nothing has changed, no update is made. If changes have occurred, React updates only the parts of the browser's DOM that have changed. Figure 1.10 shows what happens at this point. On the left, the virtual DOM has been updated to remove the right subtree with the `<div>` tag whose children are an `<img>` tag and an `<a>` tag. This results in these same children being removed from the browser DOM.

React uses JavaScript to represent DOM nodes. In JavaScript, this is written as follows:

```
let myDiv = React.createElement('div');
```

When a React render occurs, each component returns a series of React elements. Together they form the virtual DOM, a JavaScript representation of the DOM tree.

Because the virtual DOM is a JavaScript representation of the browser DOM and isn't dependent on browser-provided objects such as the window and document (although certain code paths may depend on these items), it can be rendered on the server. But rendering a DOM on the server wouldn't work. Instead, React provides a way to output the rendered DOM as a string (`ReactDOM.renderToString`). This string can be used to build a complete HTML page that's served from your server to the user.

## 1.5 Business logic and model: Redux

In real-world web apps, you need a way to manage the data flow. Redux provides an application state implementation that works nicely with React. It's important to note that you don't have to use Redux with React, or vice versa, but their concepts mesh well because they both use functional programming ideas. Using Redux and React together is also a community best practice.

Like React, Redux follows a single-direction flow of data. Redux holds the state of your app in its store, providing a single source of truth for your application. To update this store, *actions* (JavaScript objects that represent a discrete change of app state) are dispatched from the views. These actions, in turn, trigger reducers. *Reducers* are pure functions (a function with no side effects) that take in a change and return a new store after responding to the change. Figure 1.11 shows this flow.

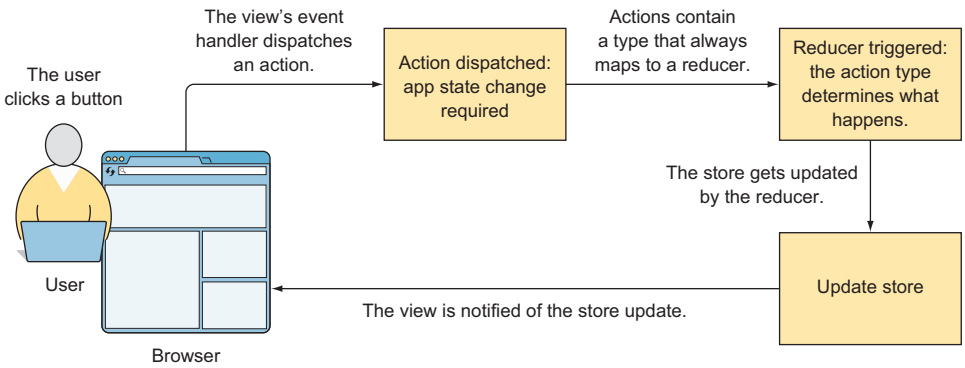


**Figure 1.11 The view (React) uses Redux to update the app state when the user takes an action. Redux then lets the view know when it should update based on the new app state.**

The key thing to remember about Redux is that only reducers can update the store. All other components can only read from the store. Additionally, the store is immutable. This is enforced via the reducers. I cover this again in chapter 2 and do a full Redux explanation in chapter 6.

The ability to transfer state between server and browser is important in an isomorphic app. Redux's store provides top-level state. By relying on a single root object to hold your application state, you can easily serialize your state on the server and send it down to the browser to be deserialized. Chapter 7 covers this topic in more detail. The final piece of the app is the build tool. The next section gives an overview of webpack.

## 1.6 Building the app: webpack

Webpack is a powerful build tool that makes packaging code into a single bundle easy. It has a plugin system in the form of loaders, allowing simple access to tools such as Babel for ES6 compiling or Less/Sass/PostCSS compiling. It also lets you package Node.js module code (npm packages) into the bundle that will be run in the browser.

**DEFINITION** There are many names for current and future JavaScript versions (ES6, ES2015, ES2016, ES7, ES Next). To keep things consistent, I refer to modern JavaScript that's not yet 100% adopted in browsers as *ES6*.

This is key for our isomorphic app. By using webpack, you can bundle all your dependencies together and take advantage of the ecosystem of libraries available via npm, the Node package manager. This allows you to share nearly all the code in your app with both environments—the browser and the server.

> **NOTE** You won't use webpack for our Node.js code. That's unnecessary, as you can write most ES6 code on Node.js, and Node.js can already take advantage of environment variables and npm packages.

Webpack also lets you use environment variables inside your bundled code. This is important for our isomorphic app. Although you want to share as much code between environments as possible, some code from the browser can't run on the server, and vice versa. On a Node.js server, you can take advantage of an environment variable like this:

```
if (NODE_ENV.IS_BROWSER) { // execute code }
```

But this code won't run in the browser because it has no concept of Node.js environment variables. You can use webpack to inject a `NODE_ENV` object into your webpacked code, so this code can run in both environments. Chapter 5 covers this concept in depth.

## Summary

In this chapter, you learned that isomorphic web apps are the result of combining server-rendered HTML pages with single-page application architecture. Doing so has several advantages but does require learning a new way of thinking about web app architecture. The next chapter presents a high-level overview of an isomorphic application.

- Isomorphic web apps blend server-side architecture and single-page app architecture to provide a better overall experience for users. This leads to improved perceived performance, simplified SEO, and developer benefits.
- Being able to run JavaScript on the server (Node.js) and in the browser allows you to write code once and deploy it to both environments. React's virtual DOM lets you render HTML on the server.
- Redux helps you manage application state and easily serialize this state to be sent from the server to the browser.
- By building your app with webpack, you can use Node.js code in the browser and flag code to run only in the browser.

JAVASCRIPT/WEB DEVELOPMENT

# Isomorphic Web Applications

Elyse Kolker Gordon


Build secure web apps that perform beautifully with high, low, or no bandwidth. Isomorphic web apps employ a pattern that exploits the full stack, storing data locally and minimizing server hits. They render flawlessly, maximize SEO, and offer opportunities to share code and libraries between client and server.

**Isomorphic Web Applications** teaches you to build production-quality web apps using isomorphic architecture. You'll learn to create and render views for both server and browser, optimize local storage, streamline server interactions, and handle data serialization. Designed for working developers, this book offers examples in relevant frameworks like React, Redux, Angular, Ember, and webpack. You'll also explore unique debugging and testing techniques and master specific SEO skills.

## What's Inside

- Controlling browser and server user sessions
- Combining server-rendered and SPA architectures
- Building best-practice React applications
- Debugging and testing

To benefit from this book, readers need to know JavaScript, HTML5, and a framework of their choice, including React and Angular.

**Elyse Kolker Gordon** runs the growth engineering team at Strava. Previously, she was director of web engineering at Vevo, where she regularly solved challenges with isomorphic apps.

"A practical guide to performant and modern JavaScript applications."
—Bojan Djurkovic, Cvent

"Clear and powerful. If you need just one resource, this is it."
—Peter Perlepes, Growth

"Thorough and methodical coverage for novice users, with handy insights and many 'aha' moments for advanced users. Highly recommended."
—Devang Paliwal, Synapse

"An essential guide for anyone developing modern JavaScript applications."
—Mike Jensen, UrbanStems

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/books/isomorphic-web-applications

ISBN-13: 978-1-61729-439-6
ISBN-10: 1-61729-439-X
53999
9 781617 294396

MANNING $39.99 / Can $56.99 [INCLUDING eBOOK]