Extracted from:

Modern Front-End Development for Rails, Second Edition

Hotwire, Stimulus, Turbo, and React

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Covers Rails 7

Modern Front-End Development for Rails

Second Edition

Hotwire, Stimulus, Turbo, and React

Noel Rappin

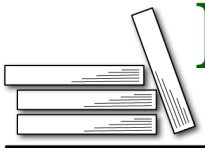
Edited by Katharine Dvorak

Modern Front-End Development for Rails, Second Edition

Hotwire, Stimulus, Turbo, and React

Noel Rappin

The Pragmatic Bookshelf
Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Katharine Dvorak
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-961-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2022

Every modern web application uses client-side features in an effort to improve the end-user experience. Even though Rails is a server-side framework, it has always offered tools that made it easier to deliver client-side code to the browser. From the very beginning of Rails, its ability to make client-side coding easier has been a selling point.

Over time the client-side tools have become more powerful, which is great, and more complicated, which is less great. One thing that hasn't changed, though, is that Rails still has opinions about how to deal with the client tools.

In this book, we're going to take a basic Rails application and add client-side interactivity. The JavaScript ecosystem includes a lot of tools that allow you to work with Rails; for our purposes, we're going to focus on two patterns that work well with Rails. In one pattern, the server communicates with the client by sending HTML, and the client's job is mostly to direct that HTML to the correct part of the page and add some client-only interactions. In the other pattern, the server communicates by sending data (usually in JSON format), and the client is responsible for the logic to convert that data into HTML and is responsible for client-only interactions.

To demonstrate the first pattern, we'll use the Hotwire family of tools, which comes from Basecamp and is a more-or-less official add-on to Rails.[1] Hotwire consists of Turbo,[2] a library that manages user navigation and communication with the server without needing custom JavaScript, and Stimulus,[3] which supports client-side interactions written in JavaScript. Stimulus is designed to be written as an extension to the HTML markup you are already writing and is well suited to small interactions that don't need to manage a lot of state, or where the majority of the state is managed by the server application. A third tool, Strada, which manages interactions with native mobile devices, has not yet been released as I write this.

Many frameworks support the JSON interaction pattern and create their own markup on the client. We'll use React to represent those tools.[4] React replaces your HTML markup with a custom hybrid of JavaScript and HTML called JSX. React automatically updates the DOM when data changes, and is suited for interactions where a lot of state is stored on the client.

The code we'll write will be in TypeScript, which is a language that extends JavaScript by allowing us to specify the type of any variable, class attribute,

1. http://hotwire.dev
2. https://turbo.hotwire.dev
3. http://stimulusjs.org
4. http://reactjs.org

function argument, or function return value.[5] I like using TypeScript because there are a whole bunch of mistakes I normally make in JavaScript that TypeScript catches for me. TypeScript's syntax is JavaScript with a few extra features. The first few chapters cover the TypeScript syntax as we need it, and then we'll take a look at TypeScript itself in more detail in Chapter 6, TypeScript, on page ? and Chapter 13, Validating Code with Advanced TypeScript, on page ?.

Finally, we'll use Rails bundling tools to put all these pieces together. These tools, jsbundling-rails, cssbundling-rails, and Propshaft, provide sensible defaults and useful conventions for where to put your JavaScript and other assets, allowing us to write it in a structure of our choosing and then package everything in a way that is easy for the browser to manage.

We'll start by creating a Rails 7.0 application with these tools, then show how much Hotwire and Turbo allow you to do without custom JavaScript. But first, before we dive in and write code, let's talk about web applications, Rails, and JavaScript for a second.

Managing State and Front-End Development

A lot of the decisions about program structure in web applications are about how to manage *state*, which is the data that controls the interactions between the user and the application. Managing state includes both the location of that data itself and the location of the logic that manipulates that state. Once you have the program structure set, you also have to worry about packaging, or how to convert your developer-friendly code to browser-friendly code.

The Structure of Web Applications

One of the main questions we'll be dealing with is how to structure your web application so as to best manage your state. The goal is to avoid having multiple sources of truth, both by avoiding duplicating data as well as avoiding writing the same logic on both the client and the server side. We also want to make the program as easy as possible to understand and change.

A consistent problem in web development is that as far as the browser and HTTP server are concerned, the interaction for each page view is "stateless." Being stateless means that each interaction is completely self-contained. As far as the web server is concerned, each request has no relation to or memory of previous requests.

5. https://www.typescriptlang.org

This lack of state is quite useful if you are a web server, because it makes your life much easier not to have to keep track of any state. If you are a user of the web, however, the lack of state is annoying because the web server never remembers anything about you. Web applications depend on maintaining your state to remember who you are and what you are doing, so developers have created different solutions to manage the state of a user's interaction with a web server.

Almost since the beginning of the web, a technical solution to this problem has been *cookies*. A cookie is a small amount of data—which often consists of a random string of characters—generated by the server and managed by the browser. The cookie allows the browser to identify itself, and an application server can use that identification to remember the user's state for each request. Over time, interaction patterns were created where nearly all state would be managed on the server, and the browser's job was largely to ask for new pages or new parts of pages, receive the result of the state change, and display it to the user.

Designing Around Basic Web Actions

Ruby on Rails is a framework for managing the state of a web application on the server. Rails is, to a large extent, built around the idea that most web interactions involve a very small set of operations. This set of actions is often abbreviated CRUD: Create, Read, Update, and Delete. In Rails, these actions are represented by the seven operations provided for a standard resource (create, new, show, index, edit, update, and delete).

One of the great insights of Rails is that once you've settled on this set of basic actions, you can remove a lot of repetitive boilerplate and assume common behavior no matter what shape the data is in. In Rails, this default behavior is often defined by the scaffolding that Rails creates for new resources.

If you are dealing with these basic actions, it turns out web browsers can offer you a lot of help. Browsers can provide data input elements, manage the state of form elements, and maintain a list of historical actions. Working hand in hand with a browser makes the Rails' core actions more powerful.

In fact, the basic set of Rails interactions is so powerful that it starts to be worthwhile to take things that are not necessarily basic resource interactions and model them as if they were. Take, for example, Twitter. Twitter, which was originally built partially using Rails, can be modeled as a system where a tweet is a resource, and the user has actions to create one, show one or

more, and delete one.[6] (But not edit, which is an argument I'm not getting into here.) Is that the best way to model Twitter's user interaction? I don't know. Probably not. But it's at least a pretty good way to model Twitter, and doing so gives you a big head start because you can take advantage of the power of Rails and the browsers.

The server-side model has many advantages, but ten years ago, it was pretty limited in terms of user interaction. This created problems when users began to expect web applications to have the same rich and complex interactions as desktop interactions. Client-side logic was one response to this problem. Another was making the browser markup, particularly CSS, more powerful to allow browsers more access to complex interactivity.

Designing Around Client Logic

As a web application begins to act more like a desktop application, the application needs to maintain a lot of state and logic that only pertains to the user interface (UI). It doesn't always make sense to manage the client-only information and logic on the server, so JavaScript applications in the browser became more and more complex.

The interactions a user has that are managed by JavaScript may be harder to model as CRUD resources and actions. SPA JavaScript frameworks often have a different set of actions. As a result, these frameworks have structured themselves quite differently from server-side Rails applications. For example, a primary concern of JavaScript application frameworks is managing the state of the objects and data being interacted with on the client (for example, which items are active), and the frameworks often emphasize having a lot of relatively small constructs that manage the data and logic for a small part of the page. Therefore, a problem in a lot of client-side app frameworks is sharing common state information among otherwise unrelated small components across the page or the app.

On the server side, sharing common state is not a concern in the same way. A server-side Rails app stores global state in the database and generally doesn't worry about the mutability of individual instances because they don't last beyond a single request. How, then, would we structure an application that combines a lot of server-side logic that is Rails-shaped with a lot of client-side interaction that is JavaScript-shaped?

6. http://twitter.com

One option is to do as little in JavaScript as possible. In this paradigm, the server generates rendered HTML and the client manipulates the existing DOM. When the client wants to update all or part of the page, it makes requests to the server, receives HTML, and inserts that HTML directly into the DOM. This was more or less the original Web 2.0 paradigm and was the interaction supported by early versions of Rails via helper methods that made remote calls and inserted the returned HTML into a DOM element with a given ID.

These days, Hotwire and Turbo allow a more powerful version of this paradigm to create very interactive client experiences while writing little to no JavaScript. Again, it also helps that many client-side flourishes can now be done in CSS.

On the other extreme, you have a single-page JavaScript app that does the maximum amount of work on the client. After sending the original JavaScript, the server is limited to sending data back and forth, probably using JSON, while the client converts that data to DOM elements using templates or JSX or something. The client also manages the state of the application, including the address the browser displays in the address bar, and the way the browser's Back button works. The client is responsible for making sure the server is informed of any data change that needs to be persisted.

Both of these options have their benefits and drawbacks. There's also a middle ground, where individual web pages might have their own rich interactions, but we let the server handle the transition between pages.

Patterns of Web Applications

To make this architecture discussion more concrete, let's look at how these decisions might play out in a specific web app. Slack is a real-time collaboration and chat application that runs in a browser.[7] Later in the book we'll look at how an application might handle real-time chat notification. For now, let's focus on two user interactions: (1) when users click in the sidebar to remain in chat but change the Slack channel they are looking at, and (2) when users click to view their profiles, which completely takes away the chat interface and replaces it with something more like a form.

When the user clicks on a different channel, the basic UI stays in place, but all the displayed data changes. Very broadly speaking, there are a few ways to handle this change. Clicking on a channel could trigger an entire page refresh, making a normal HTTP request to the server and redrawing the entire page. This would usually cause the page to flicker, leading to a poor user

7. http://www.slack.com

experience, although in theory, when using default Rails, the Turbo Drive part of the Turbo library prevents flickering. This is the solution with the least JavaScript.

Clicking on a channel could trigger sending a request to the server, where the server handles the rendering logic and returns the HTML of the part of the page that changes. The client side integrates that HTML into the page. There is a little bit of UI cleanup, such as changing the active status of channels in the sidebar. This would either be done in the client-side code, or, alternately, the server could return a JavaScript script that both updated the HTML and did the UI cleanup or return multiple chunks of HTML that manage the cleanup. This is the JavaScript interaction pattern that many early Rails applications used and which you see a new version of in the Hey[8] email app and other apps that use Turbo Frames and Turbo Streams.

Clicking on a channel could also trigger sending a request to the server that returns data in JSON format. The JavaScript code is responsible for using that data to update, which different frameworks handle in different ways. The data update would trigger changes to the DOM, and the specific updated DOM elements would be redrawn. This is the interaction pattern that React and most JavaScript applications use these days.

When we switch to the profile page, we have the same options, plus we now have one more kind of state to deal with—the name of the entire page that we are looking at. In an older Rails app, switching to a new profile page would be a request to a different URL, and the routing table in the Rails app would handle the decision of what view templates to render and send back to the browser.

In an SPA, the routing happens on the client, and the client's routing table intercepts the click, determines what components are displayed, and what data needs to be retrieved from the server to render that data. This routing is slightly more complicated because it splits one step (call the server) into two (navigate internally, then make calls to the server).

SPAs often come at a high complexity cost. They can be effective in cases where the interaction pattern is different from the typical CRUD set of actions. Over time, the single-page frameworks have been hard at work lowering the cost of duplicating browser functionality.

For the most part, in this book we'll deal with apps where the server side determines what page is drawn, but each page might have its own interactiv-

8. http://hey.com

ity that is managed by a smaller page-level set of components that run in the browser.

Now that we have our structure in mind, it's time to start adding client interaction to the pages of our North By application. Let's begin with the schedule page using the tools that will help us implement all these designs, starting with the Rails installation itself.