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

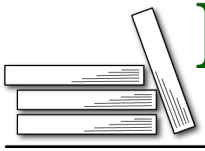
Edited by Katharine Dvorak

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

So You Want to Write Some Client-Side Code

"I need a website," the client said.

"Great," you think. Ruby on Rails is a solid way to go. Version 7.0 just came out. It's still the best way for a small team to be as productive as a big team. You are ready for this. You start thinking of estimates and modeling data structures…

"I want it to look cool, with lots of stuff moving around, and be extremely interactive," the client added.

"Ugh," you think. That brings in JavaScript. And with it, a whole lot of decisions. What language? There's not just JavaScript, but a host of languages that compile to JavaScript: TypeScript, Elm, ClojureScript. What framework? There are dozens: React, Vue, Ember, Hotwire, Svelte, Preact, and on and on. How to package the code and CSS? Should you use the existing Rails asset pipeline, or Propshaft, or jsbundling, or Webpacker? What about that new Hotwire thing the Rails team has been going on about?

Suddenly you are overwhelmed by the added complexity.

Although it's primarily a server-side tool, Ruby on Rails offers a lot of support for client-side code. Rails version 7.0 has tools that help you interact with the JavaScript ecosystem to build an exceptional front-end experience. In this book, you'll learn how you can enhance the user experience of a standard Rails application using front-end tools from the Rails ecosystem (Hotwire, Stimulus, Turbo, and jsbundling) and tools from the JavaScript ecosystem (esbuild, TypeScript, and React) to create a great Rails-based app.

So that interactive website your client wants? No problem.

Basic Assumptions

Rails is an opinionated framework, and this is an opinionated book. Being opinionated means that Rails makes certain tasks easier if you are willing to structure your program the way the Rails core team thinks you should. For this book, being opinionated means not trying to show you every possible way Rails and JavaScript can combine, but instead focusing on the tools I think will be most useful. Perhaps the most important opinion is that we're going to use JavaScript to enhance a mostly server-side Rails application rather than use JavaScript to build a completely separate single-page application (SPA) that only uses Rails as an application programming interface (API).

My basic argument for not writing an SPA is that between Rails and a standard browser, a tremendous amount of complexity is already handled for you. Moving to an SPA structure requires you to build much of that functionality yourself. Over time, the front-end frameworks have gotten better at handling the complexity for you, but to me, it often feels like taking three right turns rather than just taking one left turn. For now and for my money, Rails is less complicated than an SPA for many applications.

That said, there are legitimate places where an SPA might make sense. If your user experience is so different from the normal web structure that the existing behavior of Rails and the browser isn't much help, then an SPA begins to look attractive. If your back end is already an API supporting a mobile app or external services, then an SPA can also act as a consumer of that API, saving you from duplicating view-layer logic (but you can use Rails and web views to go surprisingly far in a mobile app). However, my experience is that most of the time, for most teams, starting by leveraging the Rails view and browser features is the best way to create a great application.

Within that assumption—Rails back end with some front-end interaction—there's still a wide range of tools, architectures, and techniques that might be appropriate for the application. We're going to navigate that space. And within that space, we are going to explore different ways of structuring a Rails/JavaScript collaboration.

The Tools We'll Use

Over the course of the book, we'll walk through the basics of getting Rails set up to serve JavaScript and CSS to the browser. Then we will write code to get the browser to do things. We're going to look at two different frameworks that have very different approaches—Hotwire and React:

- Hotwire is a framework that allows you to keep most of your logic on the server and communicate with the client by sending HTML.[1] Much of the Hotwire code uses Turbo, which is a library that allows you to do complex client-server interactions without writing custom JavaScript. Turbo itself consists of Turbo Drive, which is the successor to Turbolinks and allows you to speed up basic links through your site; Turbo Frames, which allows you to easily replace part of your page with new HTML from the server; and Turbo Streams, which allows you to do arbitrary Document Object Model (DOM) replacement without any custom JavaScript. Hotwire also includes Stimulus, a JavaScript library that manages client-side interactions more directly.

- React is a framework where most of the rendering logic is on the client.[2] In React, you describe your output using JSX, a language for specifying HTML in JavaScript. You also describe what variables make up the state of the system, and when that state changes, React automatically redraws the parts of the screen that reflect the new state. React typically communicates with the server as an API and frequently expects to receive JSON data in return, which is used to update the state.

We will use other more foundational tools—TypeScript, esbuild, jsbundling, and cssbundling—to build the infrastructure of our application, no matter what JavaScript frameworks we use on top:

- TypeScript is an extension of JavaScript that provides type checking and type inference, which means TypeScript ensures that values in your code have the types you expect.[3] It's a superset of JavaScript, which means that any JavaScript program is valid TypeScript, but TypeScript also allows you to add some basic type checking to your code. More advanced usage of TypeScript allows you to use the type system to prevent invalid states at compile time, which can make runtime errors less likely.

- esbuild calls itself "an extremely fast JavaScript bundler."[4] The purpose of esbuild is to convert developer-friendly front-end code into a browser-friendly package. The inputs are the code you write—which for our purposes are mostly JavaScript and TypeScript—all arranged in a hopefully logical structure. esbuild packages all the files into a bundle that the browser can use, which involves translating code and resolving references

1. https://www.hotwire.dev
2. https://reactjs.org
3. https://www.typescriptlang.org
4. https://esbuild.github.io

to code in different files. The converted JavaScript files can then be sent to a browser. esbuild is very fast and simpler to configure for basic tasks than other tools.

- JavaScript Bundling for Rails[5] is a new Rails 7.0 tool that allows you to use the JavaScript bundling tool of your choice—we're using esbuild—to wrap your JavaScript code and prepare it for a Rails asset pipeline.
- CSS Bundling for Rails[6] is a new Rails 7.0 tool that allows you to use one of a few different CSS processing tools—we're using Tailwind CSS—to convert your developer CSS to prepare it for asset download.
- Propshaft[7] is a new Rails 7.0 tool that takes the output of the bundling tools and delivers it to the browser, providing for digest hash creation, a developer platform, and the ability to reference assets in code. It's a simpler replacement for Sprockets, the older asset pipeline tool, for a toolchain where the asset handler needs to do much less work.
- Import Maps for Rails[8] is a standard tool that allows a browser to import all of your separate JavaScript modules individually, but still allows them to reference each other. This allows JavaScript code to be used without a bundling step. Rails 7 supports Import Maps (in fact, it's the default), but we won't use it throughout most of the book because TypeScript and React support is tricky. I'll discuss this more in Chapter 7, Bundling, on page ?.

How This Book Is Organized

This book is divided into four parts.

In the first part, we'll install and start using the tools we need to get Rails working with the JavaScript ecosystem. We'll start with a basic introduction to installing the front-end Rails tools. Then we'll add Turbo to the mix for richer interactions, sprinkle that with Stimulus, and then see how React can interact with Rails. Then we'll augment both tools by looking at some great ways to use CSS tools in our applications. Finally, we'll take a closer look at our foundation, including the basics of TypeScript and the Rails 7.0 bundling tools.

The second part has a deeper look at TypeScript and the bundling tools. In the third part, we take a look at one important concern for front-end code:

5. https://github.com/rails/jsbundling-rails
6. https://github.com/rails/cssbundling-rails
7. https://github.com/rails/propshaft
8. https://github.com/rails/importmap-rails

Why Doesn't This Book Use Import Maps?

Since the first version of this book was released, the default set of tools that Rails offers for a new application has changed. More than once. The new default is to use import maps and a standalone command line for Tailwind CSS, allowing for a Rails app that does not require Node.js or the Node Package Manager (NPM).

However, for most of this book, we do not use import maps (you can see samples of how they work in Appendix 1, Framework Swap, on page ?. There's a philosophical reason and a practical reason. The philosophical reason is that it's not clear to me what is the upper bound of how complex an app can get using import maps, and I've decided to err on the side of caution. (That said, the Hotwire flagship app Hey.com uses import maps.)

The practical reason is that the book's code already relies on React's JSX and TypeScript, both of which require the kind of compilation step that import maps are supposed to get rid of. Because Rails 7 provides a whole other set of great new tools for bundling projects that require compilation steps, I decided to present that, rather than limit the scope of the book only to tools supported by import maps.

communicating with the server. Then we'll look at managing the state of the data in your client-side application. We'll look at a JavaScript pattern called a reducer and then talk about Redux, a library that implements the reducer pattern and is commonly used with React.

The fourth part is about validating your code. We go further into TypeScript and take a look at how we can use the type system to prevent error conditions. We then talk about debugging and testing our applications.

Finally, in the appendix, we'll rewrite all of the book's code, first rewriting the React code using Hotwire, then flipping the script and rewriting the Hotwire code using React.

By the end of the book, you'll have options that will show you how to structure your code for different levels of client-side needs.

Let's Build an App

Before we start talking about front-end structure, we need to have an app to attach all that front-end structure to. I've created a sample website for a fictional music festival called North By, where multiple bands will perform at

various concerts during the event. This app contains a schedule of all the concerts and venues. There isn't much to this app. I used Rails scaffolding for a minimal amount of administration, but it's just a structure that lets us get at the two pages we'll be managing in this book: the schedule page and the concert display page.

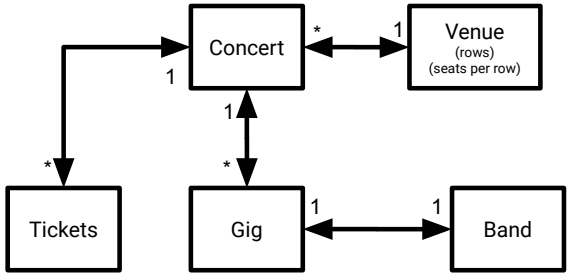
The schedule page shows all the concerts, acts, and times for the entire festival. We'll be adding features to this for inline editing, date filters, and search. We'll let users list favorite concerts, and eventually we'll show up-to-date information on how many tickets have been purchased.

The concert page shows you a simplified theater diagram for each concert and lets you select seats for a simulated ticket purchase. On this page, users can select seats and see their subtotal increase, or search for a block of seats and see which seats are available.

The data model for the app looks like this:

- The festival includes several concerts that take place at particular start times.
- Each concert has a venue, and each venue has a number of rows and a number of seats per row (which I realize is vastly simplified from real music venues, but we're just going to pretend for now, because that gets very complicated very quickly).
- Each concert has one or more gigs that make up the concert.
- Each gig matches a band to a concert, and has a start order and a duration.
- Each concert has a bunch of sold tickets, which link a concert to a particular row and seat in the venue.
- We've got users. A user can have tickets and a list of favorite concerts.

Here's a diagram of the data model:



The app uses the Tailwind CSS framework,[9] which is effectively a default choice in Rails 7, though there are many other options.

The Sample Code

If you'd like to follow along with the application throughout the course of the book, you can download the sample code files from the book page on the Pragmatic Bookshelf website.[10]

The version of the code in the `chapter_01/02` directory is the complete app setup with all the data structures, but none of the working JavaScript (the `chapter_01/01` directory has mostly just the startup code from creating a new Rails application). That's probably the best place to start if you are following along. After that, the directories are named after their chapter numbers and should progress in order.

To run the code, you need a few dependencies:

- The code uses Ruby version 3.1. I recommend installing a Ruby version manager such as RVM,[11] rbenv,[12] or chruby.[13]
- The code uses PostgreSQL,[14] so you'll need to have that set up on your machine. And to help set up the Node.js packages, you'll need Node.js (versions 12.x, 14.x, or 16.x should work)[15] and Yarn (version 1.22 is preferable; the 2.0 version doesn't currently work).[16]

A number of the tools used in this book are still in active development as I write this. Here's the combination of the most important versions of tools that back the code in this book—please note that there might be some slight variation as the book moves forward because some tools updated point releases even during late revisions:

- cssbundling-rails 1.1.0
- Cypress 9.5.4
- jsbundling-rails 1.0.2
- Propshaft 0.6.4

9. https://tailwind_url
10. https://pragprog.com/titles/nrclient2
11. https://rvm.io
12. https://github.com/rbenv/rbenv
13. https://github.com/postmodern/chruby
14. https://www.postgresql.org/download
15. https://nodejs.org/en/download
16. https://yarnpkg.com/getting-started/install

- Rails 7.0.2.3
- Ruby 3.1.2
- Stimulus 3.0.1
- Stimulus-Rails 1.0.4
- React 18.0.0
- Tailwind CSS 3.0.24
- Turbo 7.1.1
- Turbo-Rails 1.0.1
- TypeScript 4.6

To install this application, you need to be able to install Ruby and a Rails application on your machine. I'm assuming that you are broadly familiar with setting up Rails and its connection to the PostgreSQL database.

The sample code is split into a number of different directories, each corresponding to a different stage of the app in the book. Examples in the book will specify which directory is being used at any time.

From the downloaded code, you can run `bin/setup` within any of the individual application directories. (You need to be on a system that runs a Unix-style shell, like Bash or Zsh. You may also need to make `bin/setup` executable with `chmod +x bin/setup`. If you need to do this for one file, you'll likely need to do it for all the files in the `bin` directory.)

I've slightly tweaked the setup script to make it a little more useful (Yarn isn't in the default file anymore).

The setup script will do the following:

- Install Bundler.
- Run `bundler install`.
- Run `yarn install`
- Run `rails db:prepare`—this creates the database
- Run `rails restart`.

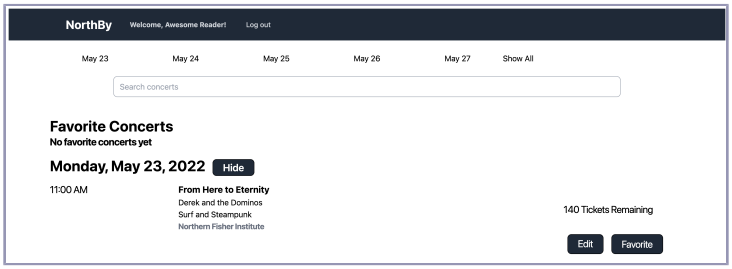
The `db:prepare` command should also trigger `rails db:seed` to get sample data in the database, however if for some reason the database already exists when you run the script, then the seed command won't be run and you'll need to run it separately.

With the app set up and the main branch running, run it using the command `bin/dev`—this command will start the Rails server, but also bundle the JavaScript and CSS, such as it is at this point. You should hit http://localhost:3000 where you'll see the schedule page with a bunch of dates at the top, a search field in the middle, and a lot of schedule information

at the bottom, with each scheduled day having a kind of ugly button labeled "Hide." If you click any of the concert names, you'll be taken to a concert page that shows basic data as well as a grid of sets for the show. Neither of these pages has any interactivity at the moment.

From the login link, you can log in with username "areader@example.com" and password "awesome." Doing so will take you back to the schedule page, with an additional option to make each concert a favorite.

The schedule page should look something like this (your randomized data will be different):



If you want to keep following along, each separate step in the application is a different directory in the sample code download, and you can move from one to another to see the entire application at different steps.

Sample Code Troubleshooting

Every effort has been made to make sure this code can be set up and run. However, this book covers a lot of tools, and there are a lot of developer setups out there. Here are a few things to keep in mind as you work with the code:

- You are likely better off starting from the `chapter_01/02` directory than from scratch. The code in that directory is pre-seeded with some boilerplate data files that aren't completely described in the book, and the `Gemfile.lock` and `yarn.lock` are already pinned to working versions—more recent versions of the libraries may have breaking changes. The `chapter_01/01` directory doesn't have any of the data models, it's much closer to a bare new Rails 7 app.
- There may be cases where background files are changed and not mentioned in the text. I hope not, but it happens.
- Sometimes the node package manager can get into weird states, especially if you have a lot of incremental changes to libraries. In particular, if you get an error that suggests that React and Redux have been duplicated,

you may need to delete the entire node_modules directory (and maybe also the yarn.lock file) and re-run yarn to refresh the modules.

- If you can't get things started, reach out on Devtalk.com[17] and post your issue there.

What's Next

There are a lot of ways to do client-side coding, but Rails is here to help. Let's start by taking a look at the tools it provides.

17. https://devtalk.com/books/modern-front-end-development-for-rails-second-edition/errata