Extracted from:

Modern Front-End Development for Rails

Hotwire, Stimulus, Turbo, and React

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Modern Front-End Development for Rails

Hotwire, Stimulus, Turbo, and React

Noel Rappin

edited by Katharine Dvorak

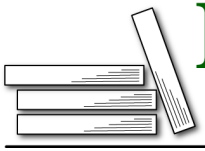
Modern Front-End Development for Rails

Hotwire, Stimulus, Turbo, and React

Noel Rappin

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

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: Adaobi Obi Tulton
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: 978-1-68050-721-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2021

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. 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 Rails asset pipeline or Webpacker? What about that new Turbo 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 6.1 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 Webpacker) and tools from the JavaScript ecosystem (webpack, 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 use Webpacker to serve JavaScript and CSS to the browser. Then we will write code to get the browser to do things. We're going to talk about two different frameworks that have very different approaches.

- 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 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 three more foundational tools—TypeScript, webpack, and Webpacker—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.
- webpack (lowercase) calls itself a "static module bundler,"[4] which I think is pretty jargony, not that anybody asked me. webpack's purpose in life is to convert developer-friendly inputs to browser-friendly outputs. The inputs are the code you write—JavaScript, TypeScript, CSS, what have you—all arranged in a hopefully logical structure. webpack converts all the files to JavaScript and CSS that the browser can understand, which involves translating code and also resolving references to code in different

1. https://www.hotwire.dev
2. https://reactjs.org
3. https://www.typescriptlang.org
4. https://webpack.js.org

files. The converted HTML and CSS and JavaScript files can then be sent to a browser.

- Webpacker (uppercase) is a Rails-specific front-end wrapper around webpack.[5] The similarity in names might be confusing, so I'll try to make it as clear as possible when referring to one tool or the other. The most important thing Webpacker gives is some Rails-style convention over configuration structure to your webpack builds. It also builds in support for common tools and adds default setups for common frameworks like the ones we'll be using in this book.

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 Hotwire and Turbo to the mix for richer interactions, sprinkle that with Stimulus, and then show how React can interact with Rails. Then we'll augment both tools by showing 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, webpack, and Webpacker.

The second part starts with a deeper look at TypeScript, webpack, and Webpacker, and takes a look at one important concern for front-end code: communicating with the server.

In the third part, 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.

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.

5. https://github.com/rails/webpacker

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.

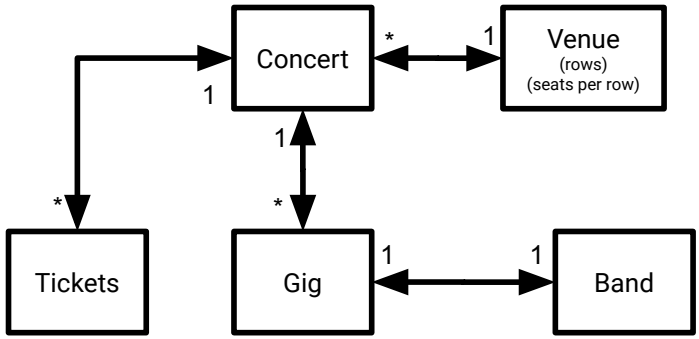
The schedule page shows all the concerts, acts, and times for the entire festival. We'll be adding features to this for inline editing, date filters, and search. We'll let users list favorite concerts, and eventually we'll show up-to-date information on how many tickets have been purchased.

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,[6] though when we talk about cascading style sheets (CSS) later in the book, we will generally be writing our own CSS.

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.[7]

The version of the code in the main directory is the complete app setup with all the data structures, but none of the working JavaScript. That's probably the best place to start if you are following along. After that, the directories are named after their chapter numbers.

To run the code, you need a few dependencies:

The code uses Ruby version 3.0, although use of 3.0 features are minimal, and I think anything from 2.6.0 and up should work if you update the .ruby_version file and change the version number in the Gemfile. I recommend installing a Ruby version manager such as RVM,[8] rbenv,[9] or chruby.[10]

6. https://tailwind_url
7. https://pragprog.com/book/nrclient
8. https://rvm.io
9. https://github.com/rbenv/rbenv
10. https://github.com/postmodern/chruby

The code uses PostgreSQL,[11] so you'll need to have that set up on your machine. And you'll need Node (versions 10.x, 12.x, or 14.x should work)[12] and Yarn (version 1.22 is preferable; the 2.0 version doesn't work as I write this)[13] to help set up the Node packages.

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 that back the code in this book:

- Hotwire-Rails 0.1.3
- Rails 6.1.3.1
- Ruby 3.0
- Stimulus 2.0.0
- React 17.0.2
- Turbo 7.0.0-beta4
- Turbo-Rails 0.5.9
- TypeScript 4.2.3
- Webpacker 6.0.0-beta.6 (Be particularly careful with this one; the Webpacker team has been changing the API)

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`.)

The setup script will do the following:

- Install bundler.
- Run `bundler install`.
- Run Yarn install.

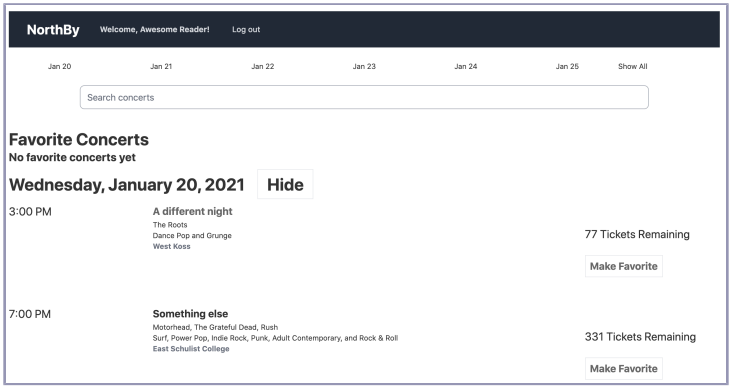
11. https://www.postgresql.org/download
12. https://nodejs.org/en/download
13. https://yarnpkg.com/getting-started/install

- Run `rails db:prepare`—this creates the database and generates a bunch of random data for the application.
- Run `rails restart`.

With the app set up and the main branch running, run it using `rails s`. 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.

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.