Extracted from:

Agile Web Development with Rails 5.1

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Agile Web Development with Rails 5.1

Sam Ruby
David Bryant Copeland
with Dave Thomas

Foreword by
David Heinemeier Hansson

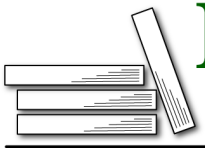
RAILS

Edited by Susannah Davidson Pfalzer

Agile Web Development with Rails 5.1

Sam Ruby
David Bryant Copeland
with Dave Thomas

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

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

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

Printed in the United States of America.
ISBN-13: 978-1-68050-251-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—May 10, 2017

In this chapter, you'll see:

- Using Webpacker to manage app-like Javascript
- Setting up a development environment that includes Webpack
- Using React to build a dynamic web form
- Using Capybara and ChromeDriver to test JavaScript-powered features

CHAPTER 13

Task H: Entering Additional Payment Details

Our customer is enthusiastic about our progress, but, after playing with the new checkout feature for a few minutes, has a question: how does a user enter payment details? It's a great question, since there isn't a way to do that. Making that possible is somewhat tricky, because each payment method requires different details. If a user wants to pay with a credit card, they need to enter a card number and expiration date. If they want to pay with a check, we'll need a routing number and account number. And, for purchase order, we need the purchase order number.

Although we could put all five fields on the screen at once, the customer immediately balks at the poor user experience that would result. Can we show the appropriate fields, depending on what payment type is chosen? Changing elements of a user interface dynamically is certainly possible with some JavaScript, but it's quite a bit more complex than the JavaScript we've used thus far. Rails calls JavaScript like this *app-like JavaScript*, and it includes a tool named *Webpacker* that will help us manage it. Webpacker will handle a lot of complex setup for us, so that we can focus most of our efforts on giving our customer—and our users—a great experience checking out. (Refer back to Chapter 1, *Installing Rails*, on page ? for installation instructions for the tools used in this chapter.)

Iteration H1: Adding Fields Dynamically to a Form

We need a dynamic form that changes what fields are shown based on what pay type the user has selected. While we could cobble something together with jQuery, it would be a bit cleaner if we could use a more modern Java-

Script library like React[1]. This will also form a solid base from which we can easily add additional features later.

Using JavaScript libraries or frameworks can often be difficult, as the configuration burden they bear is far greater than what we've seen with Rails. To help us manage this complexity, Rails includes *Webpacker*, which provides configuration for *Webpack*.[2] Webpack is a tool to manage JavaScript files that we write.

Managing JavaScript is surprisingly complex. By using Webpack we can easily put our JavaScript into several different files, bring in third-party libraries (like React), and use more advanced features of JavaScript not supported by a browser (such as the ability to define classes). Webpack then compiles all of our JavaScript, along with the third-party libraries we are using into a *pack*. Because this isn't merely sprinkling small bits of JavaScript in our view, Rails refers to this as *app-like* JavaScript.

While we could use Webpack directly with Rails, configuring Webpack is extremely difficult. It's highly customizable, and not very opinionated, meaning developers must make many decisions just to get something working. Webpacker essentially *is* the decisions, made by the Rails team, bundled up into a gem. Almost everything Webpacker does is to provide a working configuration for Webpack and React so that we can focus on writing JavaScript instead of configuring tools. But, Webpack is the tool that does the work of managing our JavaScript day-to-day.

React is a JavaScript view library that makes it easy to create dynamic user interfaces. We'll use it to create a dynamic payment method details form, and Webpacker will ensure that the configuration and setup for all this is as simple as possible.

Despite Webpacker's convenience, there's still a fair bit of one-time setup to get going, and we'll need to learn a bit of React as well. We'll take it slow so you can see exactly what is doing what in our application. First, we'll configure Webpacker and install React. After that, we'll replace our existing payment type drop-down with a React-rendered version, which will demonstrate how all the moving parts fit together. With that in place, we'll enhance our React-powered payment type selector to show the dynamic form elements we want.

1. https://facebook.github.io/react/
2. https://webpack.js.org

Configuring Webpacker and installing React

Currently, Webpacker is a separate gem that you must install in addition to Rails. Since Rails 5.1 is still in beta, we can't use the more traditional Gemfile syntax of gem "webpacker". When Rails 5.1 is released , that will work fine, but for now, we need to get the latest and greatest bleeding edge version, which is on Github. Bundler supports this exact use-case via the github: option. This syntax tells Bundler to get the most recent commit from Github, instead of the most recent *release* from RubyGems. Add it to your Gemfile like so:

```
gem "webpacker", github: "rails/webpacker"
```

Install this with bundle install.

Next, set up Webpack by running bin/rails webpacker:install

```
$ bin/rails webpacker:install
Creating javascript app source directory
      create  app/javascript
      create  app/javascript/packs/application.js
Copying binstubs
       exist  bin
      create  bin/webpack-dev-server
      create  bin/webpack-watcher
      create  bin/webpack
   identical  bin/yarn
Copying webpack core config and loaders
      create  config/webpack
      create  config/webpack/configuration.js
      create  config/webpack/development.js
      create  config/webpack/development.server.js
      create  config/webpack/development.server.yml
      create  config/webpack/paths.yml
      create  config/webpack/production.js
      create  config/webpack/shared.js
      create  config/webpack/test.js
      create  config/webpack/loaders
      create  config/webpack/loaders/assets.js
      create  config/webpack/loaders/babel.js
      create  config/webpack/loaders/coffee.js
      create  config/webpack/loaders/erb.js
      create  config/webpack/loaders/sass.js
      create  .postcssrc.yml
      append  .gitignore
Installing all JavaScript dependencies
         run  ./bin/yarn add webpack webpack-merge js-yaml…
yarn add v0.20.3
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
```

```
[4/4] Building fresh packages...

«lots of output»

Done in 24.95s.
Installing dev server for live reloading
         run  ./bin/yarn add --dev webpack-dev-server from "."
yarn add v0.20.3
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...
success Saved lockfile.
success Saved 82 new dependencies.

«lots more output»

Done in 5.11s.
Webpacker successfully installed
```

As you can see from the output, this created several configuration files in config/webpack and installed various JavaScript libraries. The libraries that were installed are listed in package.json. package.json is the JavaScript equivalent to our Gemfile—it lists all the necessary JavaScript libraries for our app to run. The equivalent of Bundler is *Yarn*.

Just like bundle install downloads all the gems our app needs, yarn install downloads all the JavaScript libraries we need. As a convenience, the webpacker:install task ran yarn install for us.

Webpacker can also install and configure some common JavaScript frameworks such as Angular, Vue, or React. We chose React becuase it's the simplest overall and is the best fit for solving our problem. To have Webpacker set it all up for us, run the task webpacker:install:react:

```
$ bin/rails webpacker:install:react
Copying react loader to …config/webpack/loaders
      create  config/webpack/loaders/react.js
Copying .babelrc to app root directory
      create  .babelrc
Copying react example entry file to …app/javascript/packs
      create  app/javascript/packs/hello_react.jsx
Installing all react dependencies
     run  ./bin/yarn add react react-dom babel-preset-react from "."
yarn add v0.20.3
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
warning "react-dom@15.4.2" has unmet peer dependency "react@^15.4.2".
[4/4] Building fresh packages...
success Saved lockfile.
```

```
success Saved 26 new dependencies.

«lots of output»
Done in 7.17s.
Webpacker now supports react.js
```

If you've ever tried to set up Webpack and a JavaScript framework like React before, you'll appreciate how much work Webpacker has just done for us. If you've never had the privilege, trust me, this saved as a ton of time and aggravation.

Webpacker also created a simple React component in app/javascript/packs/hello_react.jsx. Don't worry about what that means for now. We're going to use this generated code to validate the installation and set up our development environment. This generated code will append the string "Hello React!" to the end of our page, but it's not activated by default. Let's find out why, configure it to be included in our views, and set up our development environment to work smoothly with Webpacker.

Updating our Development Environment for Webpack

Webpacker includes a helper method called javascript_pack_tag() that takes, as an argument, the name of the file in app/javascript/packs whose JavaScript should be included on the page.

The reason Rails doesn't simply include all JavaScript all the time is that you might not want that to happen for performance reasons. Although our payment details code won't be terribly complex, it'll still be a chunk of code our users will have to download. Since it won't be needed anywhere else in our app, we can make the user experience faster and better by only downloading the code when it's needed.

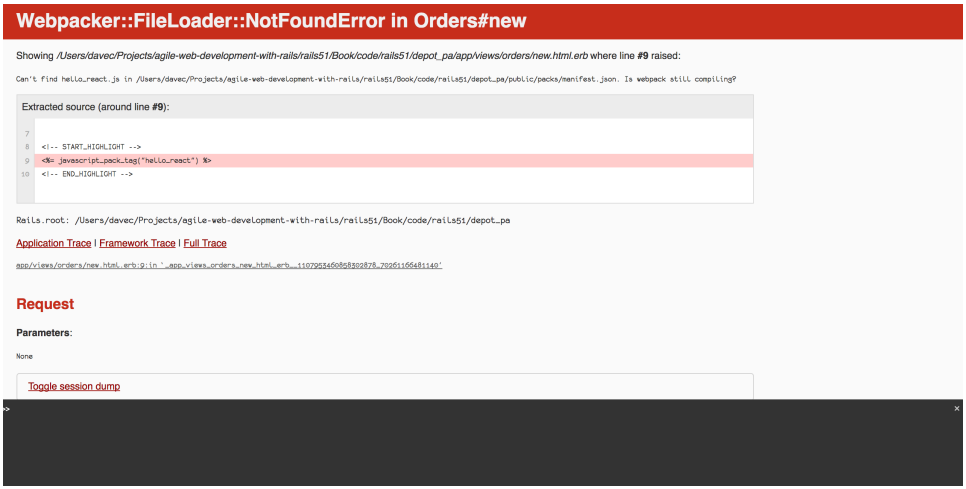
Webpacker allows us to have any number of these separately managed *packs*. We can include any that we like, wherever we like. To see how this works, let's add a call to javascript_pack_tag() to our app/views/orders/new.html.erb page to bring in the sample React component that Webpacker created for us.

rails51/depot_pa/app/views/orders/new.html.erb

```
<div class="depot_form">
  <fieldset>
    <legend>Please Enter Your Details</legend>
    <%= render 'form', order: @order %>
  </fieldset>
</div>

➤ <%= javascript_pack_tag("hello_react") %>
```

If you add some items to your cart, and navigate to the checkout page…you'll get an error, like the one shown below.



Rails is complaining that it can't find hello_react.js. This is the file we asked it to include by adding javascript_pack_tag("hello_react") to our view, so what are we missing?

If you were looking closely at the output of bin/rails webpacker:install:react, you may have noticed it created the file app/javascript/packs/hello_react.jsx. Is that the file Rails is trying to find?

We'll explain what a .jsx is in *Learning Just Enough React*, on page ?, but for now, it's a file that requires some sort of translation so that Rails can serve it and the browser can understand it. This is exactly what Webpack does and is the reason we are using it.

Webpack is, among other things, a JavaScript compiler. It takes one or more files as input and produces a single JavaScript file our browser can understand as output. Meaning, we need to run Webpack so that it can do that, and thus produce the file Rails is trying to load in our view.

In production, the Rake task assets:precompile would perform this compilation step, and produce a file in public/ that works. This is rather inconvenient for local development, so in development mode, javascript_pack_tag() is configured to ask Webpack's dev server for the compiled JavaScript, allowing that server to re-compile it on the fly as we make changes.

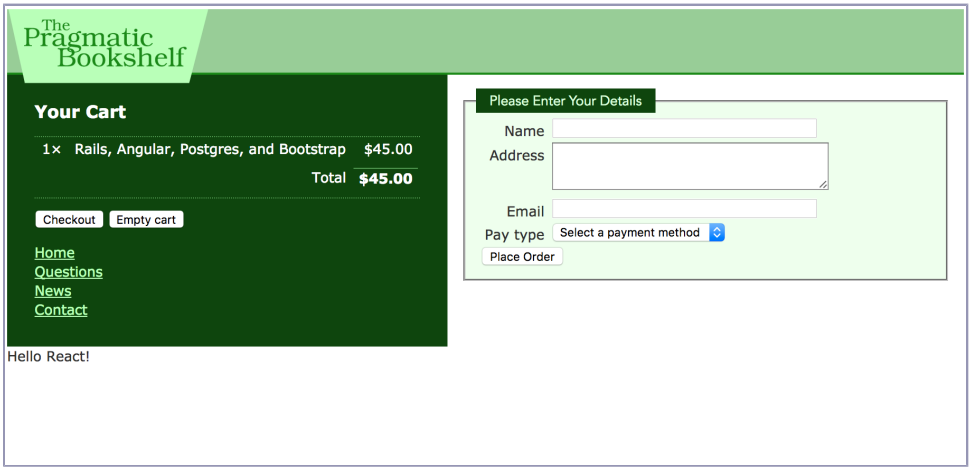
This makes our workflow much faster, but we have to actually run that server. Fortunately, Webpacker installed bin/webpack-dev-server that runs Webpack's dev server—exacty what we need (note, if you are using Cloud9, what

we're about to describe won't work. See *Working Around Environment Issues with webpack-dev-server*, on page 10 for how to work around it).

Open a new terminal window and run it (make sure to keep your Rails server running, too):

```
$ bin/webpack-dev-server
```

Now if you reload the page, you should see "Hello React!" at the bottom of the checkout form!



It's a bit of a pain to have to run two commands in two different windows, so let's fix this problem by installing *Foreman*, which is a tool that can run multiple commands from one terminal window. Foreman is a Ruby gem that includes a command-line app named foreman that reads in a file named Procfile, which lists the commands to run. First, we'll add Foreman to our Gemfile

rails51/depot_pa/Gemfile

```
gem 'foreman'
```

After installing it with bundle install, create Procfile like so:

rails51/depot_pa/Procfile

```
web:     bin/rails server
webpack: bin/webpack-dev-server
```

And now, when we run foreman start -p 3000, both Rails and Webpack will be running in one terminal (by default, Foreman runs our app on port 5000, but to keep things consistent, we're using the -p flag to set the port to the Rails default):

```
$ forman start -p 3000
13:14:57 web.1     | started with pid 83361
13:14:57 webpack.1 | started with pid 83362
```

```
«Lots more output»
```

Wow, that was a lot of setup! This is, sadly, the state of the art when using modern JavaScript tools. They aren't as opinionated as Rails and require developers to make a lot more decisions about how to configure everything, but it doesn't mean these tools aren't powerful! Webpacker has saved us from as much of the complexity and configuration as it can, and everything we just did is something we only have to do once.

Now that we've validated all the configuration gotten our development environment working, we can start building our feature. The next thing to do is replace the existing drop-down with one powered by React and our Webpacker-managed JavaScript. Doing *that* requires a slight diversion to learn about React.

Working Around Environment Issues with webpack-dev-server

bin/webpack-dev-server runs a server on your machine alongside Rails. When your browser requests JavaScript files, it requests them from this server. Some environments, notably Cloud9, prevent this from working.

To get around this limitation, you'll need to tell Webpacker not to run the dev server in the development environment. You can do this by editing config/webpack/development.server.yml and changing enabled to false:

```
# Restart webpack-dev-server if you make changes here
default: &default
➤  enabled: false
  host: localhost
  port: 8080

development:
  <<: *default

test:
  <<: *default
  enabled: false

production:
  <<: *default
  enabled: false
```

Once you do this, you'll either need to run bin/wepack anytime you change your JavaScript, or you will need to run bin/webpack-watcher alongside your rails server (which automates re-running bin/webpack).