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

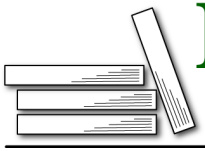
Modern Front-End Development for Rails

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

As Rails 6 developers, our primary interaction with webpack is going to be through the Webpacker tool provided by Rails. In this chapter, we'll take a look at using Webpacker in development and how to customize it when you want additional features.

As I write this, the current version of Webpacker is 6.0 beta 6. I expect there to be some changes before 6.0 goes final.

Installing Webpacker

The Webpacker gem is a default part of a new Rails 6 installation, which means Webpacker goes in the Gemfile. And by default, the webpack:install task is run when the app is created. You can skip Webpacker entirely with rails new . --skip-webpack-install. After the app is created, you can add Webpacker to an existing app with the rake task rails webpacker:install.

Webpacker Basics

Webpacker is a wrapper around webpack, designed to make webpack easier to manage within a Rails application.[1] The main thing Webpacker does is generate a webpack configuration using a set of inputs that is hopefully simpler to deal with than a full webpack configuration. It uses a YAML file, config/webpacker.yml, to specify a lot of webpack values, and specific environment overrides are in the config/webpack directory.

That basic configuration gives you the following features:

- Any file in app/packs/entrypoints with a known extension (usually a JavaScript or CSS extension) is the entry point of a new *pack*. The name of the pack is the base name of the file. If there are multiple files with the same base name but different extensions, they are combined into the same pack.
- Any file in app/packs can include any other file in app/packs relative to app/packs. Any file in node_modules can also be included.
- Resources in a pack can be added to a page by using the helper method javascript_pack_tag.
- CSS in a pack can be added to a page by using the helper method stylesheet_pack_tag.
- Static images can be used in a link or img tag with the asset_pack_path helper method.

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

- In development, Rails automatically calls webpack to compile code on a page load if the webpack files have changed. You can trigger this compilation manually by running a script named bin/webpack.
- A development server, bin/webpack-dev-server, can be run. This compiles webpack on page save, and live reloads the page if possible. When using webpack-dev-server, webpack assets aren't saved to disk; they are served by the dev server.
- When deploying, the webpack compiler can be invoked to put your static files in the public directory where they can be read.

That's a lot of things! Let's, well, unpack them.

Writing Code Using Webpacker

In development, we mostly worry about three things: writing our code, getting our code on to the page, and being able to recompile the code quickly and easily.

Somewhat uncharacteristically for Rails, Webpacker does not suggest any structure for your code beyond having the entry point be in app/packs/entrypoints. The important feature is that you can import files relative to either app/packs (for your own code) or node_modules (for third-party code).

That said, some suggestions:

- Keep as little code as possible in your actual entry point; it should mostly just import things.
- Where possible, having multiple modular small pack files is probably better than having a single one. (There's a webpack optimization that makes this optimal from a download standpoint.)
- If you import a directory, rather than a file, the module system will automatically import the index.js (or index.ts) file in that directory. We've already seen this in our boilerplate code: the pack imports controllers, and controllers/index.js handles the autoload of controller modules. You can use this to modularize your imports somewhat and make it easy to share common imports across pack files.
- Your framework of choice may have some community standards for how code is structured. If so, you should follow them.
- I wouldn't put anything other than entry point files in the entrypoints directory, and I wouldn't create any subdirectories there either. But I wouldn't use those subdirectories for regular source code.

- It's tempting, but avoid creating a top-level app/packs/src directory on the ground that anything in the app/packs directory is source of some kind or other. Try to be more specific about top level names.
- I would, though, try to separate out CSS into app/packs/stylesheets.

To use a pack in your Rails code, you use the helpers javascript_pack_tag or stylesheet_pack_tag. These use webpack "chunks" by default. In webpack, a chunk is a way to extract common dependencies so that if you are importing multiple packs, shared dependencies are only downloaded to the browser once.

Both of the helpers work the same way. The arguments are a list of pack names and an optional set of options. The helper creates a script tag (for JavaScript) or a link tag (for CSS) for each pack name in the list. Both methods just defer to the existing Rails helpers javascript_include_tag and stylesheet_include_tag, and any options are just passed right through; although in practice, most of the options to the existing Rails helpers have to do with modifying the eventual URL and aren't really relevant to packs.

There's a little bit of tension between the classic Rails structure of just putting the javascript_include_tag in the header for all pages and what is probably a more webpack-idiomatic structure of having a lot of small packs and only loading the ones you need for each page. Therefore, if your setup is at all complicated, I recommend you use the Rails content_for feature to customize the header on a per-page basis.

To do this, in the HTML header where you might otherwise have the call to javascript_pack_tag, try this instead:

```
<%= yield(:packs) %>
```

Then in any page that uses packs, do something like this:

```
<% content_for :packs do %>
  <%= javascript_pack_tag(:application) %>
<% end %>
```

The yield/content_for construct allows you to customize the webpack output on a page-by-page basis and has the side benefit of making the available JavaScript visible on the individual page itself, which can make it a little easier to figure out what's going on.

Integrating Webpacker with Frameworks

Once upon a time, when this book was young and Hotwire was just a gleam in DHH's eye, Webpacker included installation scripts for many different

frameworks and tools. The Webpacker team has abandoned that approach, presumably on the grounds that keeping up to date with nearly a dozen different JavaScript tools can be exhausting. (I can relate.) Also, most tools provide webpack instructions now, and those can be used directly.

However, there are some special cases:

CoffeeScript

If you install the correct loader, `yarn add coffeescript coffee-loader`, Webpacker will compile `.coffee` files using the CoffeeScript compiler.[2]

CSS

You need to install several packages for CSS support: `yarn add css-loader mini-css-extract-plugin css-minimizer-webpack-plugin`. There's an optional change to the `config/webpack/base.js` file for easier file resolving. Also, you can optionally add support for PostCSS (`yarn add postcss-loader`), Sass (`yarn add sass sass-loader`), Less (`yarn add less less-loader`), or Stylus (`yarn add stylus stylus-loader`). Files with the correct extension will be processed by that tool:

```
const { webpackConfig, merge } = require("@rails/webpacker")
const customConfig = {
  resolve: {
    extensions: [".css"],
  },
}

module.exports = merge(webpackConfig, customConfig)
```

ERB

With the `rails-erb-loader` installed, any webpack file can have an `.erb` extension, which causes the file to be parsed by ERB before any other processing.[3]

React

Installing `yarn add react react-dom @babel/preset-react` causes the Babel preset to adjust to compile `.jsx` files (and `.tsx` files if you are using TypeScript).[4]

TypeScript

Installing `yarn add typescript @babel/preset-typescript` makes Babel aware of `.ts` files, and `yarn add fork-ts-checker-webpack-plugin` adds the actual type checking to the compiler. You also need a `tsconfig.json` file, which we'll talk about in Chapter 14, Validating Code with Advanced TypeScript, on page ?.[5]

2. https://coffeescript.org
3. https://github.com/usabilityhub/rails-erb-loader
4. https://reactjs.org
5. https://www.typescriptlang.org

Stylesheets and Assets

With Webpacker, you can include CSS files and external modules into your JavaScript in the same way you would a JavaScript module. In fact, many of the CSS frameworks package themselves as npm modules for easy installation (though in some cases you need your own SCSS file to import them). But even if you have written a CSS or SCSS file, you would still include it using the same `import` syntax you would use for a JavaScript file.

Once included, Webpacker uses the `mini-css-extract-plugin` to create a separate pack for the CSS information that you then load. Alternately, you can use a `css` or `scss` file with the same name as the `js` pack to create a Stylesheet pack.

Often, third-party Node modules will have CSS imports as well as JavaScript imports (for example, enhanced form tools like Chosen[6]). The CSS file is just included in the entry point along with the JavaScript file.

If you have included PostCSS,[7] you also need to create a `postcss.config.js` file, which you can use if you want PostCSS behavior and which involves many different kinds of processing on CSS and SCSS files.

If you want to access static files served by webpack, Webpacker provides a few helper files. Images require a little bit of special treatment. The `image_pack_path` and `image_pack_tag` helpers mimic the default Rails `image_path` and `image_tag` helpers and assume that image paths are relative to `app/packs/images`. So our previous example uses `image_pack_tag("chevron-right.svg")` to find a file that is at `app/packs/images/chevron-right.svg`. For images that are not in the `app/packs/images` directory, prepend your file with `media`, as in `image_pack_tag("media/static/file.gif")`. Webpacker also provides a `favicon_pack_tag` and a generic `asset_pack_path` helper.

Running webpack

Rails offers three ways to compile your webpack packs:

- Running `bin/webpack` from the command line
- Compiling automatically from the Rails development server
- Running `bin/webpack-dev-server` to automatically compile when files are changed

Rails provides `bin/webpack`, which is just a command-line interface to running the webpack compiler. You can run this at any time, and webpack will output files to (by default) `/public/packs`.

6. https://harvesthq.github.io/chosen
7. https://postcss.org

Rerunning webpack manually all the time is something of a pain, so Rails will do it for you in development. By default, if Rails encounters a pack_tag helper and there are changed files in the pack, Rails will automatically rerun webpack before rendering the page. That works, but it can be slow, especially if you are compiling a lot of files.

The alternative is webpack-dev-server, which is a server that manages compilation and delivery of your webpack files in development. To run it, you need to start a new terminal session, go to your application directory, and invoke this command:

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

You'll see some output from the compilation of your webpack assets and then the server will wait. When you save a file that is part of a pack, the dev server will recompile, and will also live reload a browser page if you have one open. If the compilation fails, the reloaded browser page will get an error message. This is usually more convenient in development than just allowing Rails to compile on page hit.

A Note or Two from Experience

webpack-dev-server recompiles and reloads when the JavaScript changes. It does not recompile and reload when you change your Rails view file. So staring at the browser waiting for your changes to show up does not work if you've only touched the view file.

Also, if you are working on multiple Rails apps at once, make sure you close webpack-dev-server on the apps you aren't working on or you'll get weird results if your running webpack-dev-server doesn't match your running application.

You can configure the dev server—the default configuration is stored in the config/webpack.xml file, and those values are passed right through to the webpack config. I've never needed to touch these values. The configuration does let you change the port and host that the server listens on, and I can see where that would be useful if you had multiple apps running at once or if you had a non-standard development environment (for example, you might need to change the host if you are using Docker).