# Webpacker

This guide will show you how to install and use Webpacker to package JavaScript, CSS, and other assets for the client-side of your Rails application but please note Webpacker has been retired.

After reading this guide, you will know:

- What Webpacker does and why it is different from Sprockets.
- How to install Webpacker and integrate it with your framework of choice.
- How to use Webpacker for JavaScript assets.
- How to use Webpacker for CSS assets.
- How to use Webpacker for static assets.
- How to deploy a site that uses Webpacker.
- How to use Webpacker in alternate Rails contexts, such as engines or Docker containers.

---

## What Is Webpacker?

Webpacker is a Rails wrapper around the webpack build system that provides a standard webpack configuration and reasonable defaults.

### What is webpack?

The goal of webpack, or any front-end build system, is to allow you to write your front-end code in a way that is convenient for developers and then package that code in a way that is convenient for browsers. With webpack, you can manage JavaScript, CSS, and static assets like images or fonts. Webpack will allow you to write your code, reference other code in your application, transform your code, and combine your code into easily downloadable packs.

See the webpack documentation for information.

### How is Webpacker Different from Sprockets?

Rails also ships with Sprockets, an asset-packaging tool whose features overlap with Webpacker. Both tools will compile your JavaScript into browser-friendly files and also minify and fingerprint them in production. In a development environment, Sprockets and Webpacker allow you to incrementally change files.

Sprockets, which was designed to be used with Rails, is somewhat simpler to integrate. In particular, code can be added to Sprockets via a Ruby gem. However, webpack is better at integrating with more current JavaScript tools and NPM packages and allows for a wider range of integration. New Rails apps are configured to use webpack for JavaScript and Sprockets for CSS, although you can do CSS in webpack.

You should choose Webpacker over Sprockets on a new project if you want to use NPM packages and/or want access to the most current JavaScript features and tools. You should choose Sprockets over Webpacker for legacy applications where migration might be costly, if you want to integrate using Gems, or if you have a very small amount of code to package.

If you are familiar with Sprockets, the following guide might give you some idea of how to translate. Please note that each tool has a slightly different structure, and the concepts don't directly map onto each other.

| Task | Sprockets | Webpacker |
| --- | --- | --- |
| Attach JavaScript | javascript_include_tag | javascript_pack_tag |
| Attach CSS | stylesheet_link_tag | stylesheet_pack_tag |

| | | |
|---|---|---|
| Link to an image | image_url | image_pack_tag |
| Link to an asset | asset_url | asset_pack_tag |
| Require a script | //= require | import or require |

## Installing Webpacker

To use Webpacker, you must install the Yarn package manager, version 1.x or up, and you must have Node.js installed, version 10.13.0 and up.

NOTE: Webpacker depends on NPM and Yarn. NPM, the Node package manager registry, is the primary repository for publishing and downloading open-source JavaScript projects, both for Node.js and browser runtimes. It is analogous to rubygems.org for Ruby gems. Yarn is a command-line utility that enables the installation and management of JavaScript dependencies, much like Bundler does for Ruby.

To include Webpacker in a new project, add `--webpack` to the `rails new` command. To add Webpacker to an existing project, add the `webpacker` gem to the project's `Gemfile`, run `bundle install`, and then run `bin/rails webpacker:install`.

Installing Webpacker creates the following local files:

| File | Location | Explanation |
|---|---|---|
| JavaScript Folder | `app/javascript` | A place for your front-end source |
| Webpacker Configuration | `config/webpacker.yml` | Configure the Webpacker gem |
| Babel Configuration | `babel.config.js` | Configuration for the Babel JavaScript Compiler |
| PostCSS Configuration | `postcss.config.js` | Configuration for the PostCSS CSS Post-Processor |
| Browserlist | `.browserslistrc` | Browserlist manages target browsers configuration |

The installation also calls the `yarn` package manager, creates a `package.json` file with a basic set of packages listed, and uses Yarn to install these dependencies.

## Usage

### Using Webpacker for JavaScript

With Webpacker installed, any JavaScript file in the `app/javascript/packs` directory will get compiled to its own pack file by default.

So if you have a file called `app/javascript/packs/application.js`, Webpacker will create a pack called `application`, and you can add it to your Rails application with the code `<%= javascript_pack_tag "application" %>`. With that in place, in development, Rails will recompile the `application.js` file every time it changes, and you load a page that uses that pack. Typically, the file in the actual `packs` directory will be a manifest that mostly loads other files, but it can also have arbitrary JavaScript code.

The default pack created for you by Webpacker will link to Rails' default JavaScript packages if they have been included in the project:

```
import Rails from "@rails/ujs"
import Turbolinks from "turbolinks"
import * as ActiveStorage from "@rails/activestorage"
import "channels"

Rails.start()
Turbolinks.start()
ActiveStorage.start()
```

You'll need to include a pack that requires these packages to use them in your Rails application.

It is important to note that only webpack entry files should be placed in the `app/javascript/packs` directory; Webpack will create a separate dependency graph for each entry point, so a large number of packs will increase compilation overhead. The rest of your asset source code should live outside this directory though Webpacker does not place any restrictions or make any suggestions on how to structure your source code. Here is an example:

```
app/javascript:
  ├── packs:
  │   # only webpack entry files here
  │   └── application.js
  │   └── application.css
  └── src:
  │   └── my_component.js
  └── stylesheets:
  │   └── my_styles.css
  └── images:
      └── logo.svg
```

Typically, the pack file itself is largely a manifest that uses `import` or `require` to load the necessary files and may also do some initialization.

If you want to change these directories, you can adjust the `source_path` (default `app/javascript`) and `source_entry_path` (default `packs`) in the `config/webpacker.yml` file.

Within source files, `import` statements are resolved relative to the file doing the import, so `import Bar from "./foo"` finds a `foo.js` file in the same directory as the current file, while `import Bar from "../src/foo"` finds a file in a sibling directory named `src`.

### Using Webpacker for CSS

Out of the box, Webpacker supports CSS and SCSS using the PostCSS processor.

To include CSS code in your packs, first include your CSS files in your top-level pack file as though it was a JavaScript file. So if your CSS top-level manifest is in `app/javascript/styles/styles.scss`, you can import it with `import styles/styles`. This tells webpack to include your CSS file in the download. To actually load it in the page, include `<%= stylesheet_pack_tag "application" %>` in the view, where the `application` is the same pack name that you were using.

If you are using a CSS framework, you can add it to Webpacker by following the instructions to load the framework as an NPM module using `yarn`, typically `yarn add <framework>`. The framework should have instructions on importing it into a CSS or SCSS file.

### Using Webpacker for Static Assets

The default Webpacker configuration should work out of the box for static assets. The configuration includes several image and font file format extensions, allowing webpack to include them in the generated `manifest.json` file.

With webpack, static assets can be imported directly in JavaScript files. The imported value represents the URL to the asset. For example:

```
import myImageUrl from '../images/my-image.jpg'

// ...
let myImage = new Image();
myImage.src = myImageUrl;
myImage.alt = "I'm a Webpacker-bundled image";
document.body.appendChild(myImage);
```

If you need to reference Webpacker static assets from a Rails view, the assets need to be explicitly required from Webpacker-bundled JavaScript files. Unlike Sprockets, Webpacker does not import your static assets by default. The default `app/javascript/packs/application.js` file has a template for importing files from a given directory, which you can uncomment for every directory you want to have static files in. The directories are relative to `app/javascript`. The template uses the directory `images`, but you can use anything in `app/javascript`:

```
const images = require.context("../images", true)
const imagePath = name => images(name, true)
```

Static assets will be output into a directory under `public/packs/media`. For example, an image located and imported at `app/javascript/images/my-image.jpg` will be output at `public/packs/media/images/my-image-abcd1234.jpg`. To render an image tag for this image in a Rails view, use `image_pack_tag 'media/images/my-image.jpg`.

The Webpacker ActionView helpers for static assets correspond to asset pipeline helpers according to the following table:

| ActionView helper | Webpacker helper |
| --- | --- |
| favicon_link_tag | favicon_pack_tag |
| image_tag | image_pack_tag |

Also, the generic helper `asset_pack_path` takes the local location of a file and returns its Webpacker location for use in Rails views.

You can also access the image by directly referencing the file from a CSS file in `app/javascript`.

### Webpacker in Rails Engines

As of Webpacker version 6, Webpacker is not "engine-aware," which means Webpacker does not have feature-parity with Sprockets when it comes to using within Rails engines.

Gem authors of Rails engines who wish to support consumers using Webpacker are encouraged to distribute frontend assets as an NPM package in addition to the gem itself and provide instructions (or an installer) to demonstrate how host apps should integrate. A good example of this approach is Alchemy CMS.

### Hot Module Replacement (HMR)

Webpacker out-of-the-box supports HMR with webpack-dev-server, and you can toggle it by setting dev_server/hmr option inside `webpacker.yml`.

Check out webpack's documentation on DevServer for more information.

To support HMR with React, you would need to add react-hot-loader. Check out React Hot Loader's *Getting Started* guide.

Don't forget to disable HMR if you are not running webpack-dev-server; otherwise, you will get a "not found error" for stylesheets.

## Webpacker in Different Environments

Webpacker has three environments by default `development`, `test`, and `production`. You can add additional environment configurations in the `webpacker.yml` file and set different defaults for each environment. Webpacker will also load the file `config/webpack/<environment>.js` for additional environment setup.

## Running Webpacker in Development

Webpacker ships with two binstub files to run in development: `./bin/webpack` and `./bin/webpack-dev-server`. Both are thin wrappers around the standard `webpack.js` and `webpack-dev-server.js` executables and ensure that the right configuration files and environmental variables are loaded based on your environment.

By default, Webpacker compiles automatically on demand in development when a Rails page loads. This means that you don't have to run any separate processes, and compilation errors will be logged to the standard Rails log. You can change this by changing to `compile: false` in the `config/webpacker.yml` file. Running `bin/webpack` will force the compilation of your packs.

If you want to use live code reloading or have enough JavaScript that on-demand compilation is too slow, you'll need to run `./bin/webpack-dev-server` or `ruby ./bin/webpack-dev-server`. This process will watch for changes in the `app/javascript/packs/*.js` files and automatically recompile and reload the browser to match.

Windows users will need to run these commands in a terminal separate from `bundle exec rails server`.

Once you start this development server, Webpacker will automatically start proxying all webpack asset requests to this server. When you stop the server, it'll revert to on-demand compilation.

The Webpacker Documentation gives information on environment variables you can use to control `webpack-dev-server`. See additional notes in the rails/webpacker docs on the webpack-dev-server usage.

### Deploying Webpacker

Webpacker adds a `webpacker:compile` task to the `assets:precompile` rake task, so any existing deploy pipeline that was using `assets:precompile` should work. The compile task will compile the packs and place them in `public/packs`.

## Additional Documentation

For more information on advanced topics, such as using Webpacker with popular frameworks, consult the [Webpacker Documentation].