

---

## browserify-rails

This project is currently in maintenance mode. New contributors are more than welcome!

downloads 2.4M downloads 2.4M

This library adds CommonJS module support to Sprockets (via Browserify).

It lets you mix and match `//= require` directives and `require()` calls for including plain javascript files as well as modules. However, it is important to remember that once you are into code that is being browserified you can no longer use sprockets-style require (so no `//= require`). In many cases, it makes sense to put all your sprockets-required code in a separate file or at the very least at the top of your main JavaScript file. Then use `require()` to pull in the CommonJS code.

1. Manage JS modules with `npm`
2. Serve assets with Sprockets
3. Require modules with `require()` (without separate `//= require` directives)
4. Only build required modules
5. Require *npm modules* in your Rails assets
6. Require modules relative to asset paths (ie `app/assets/javascript`) with non-relative syntax (see below before using)
7. Configure browserify options for each JavaScript file so you can mark modules with `--require`, `--external`, etc

### Should you use this gem?

As the primary developer, I'm going to offer some opiniated advice. The sweet spot for this gem is for Rails projects with legacy JavaScript (not using CommonJS/modules). This gem is a great way to make it possible to rewrite that legacy JavaScript to CommonJS on a timeline that you dictate. Then consider stepping off the Rails asset pipeline or using another gem.

If you're starting a new Rails project today, I highly recommend looking at alternatives to this gem. The primary reason is that this gem, while it works well, is not as efficient as most would like for local development. Also a lot has changed over the last couple of years.

An example of that change is this project from Rails:

`rails/webpacker`

This is a huge step in the right direction for the Rails community. In the past, it has been extremely frustrating working with JavaScript on the asset pipeline. The good news is you have a lot of great choices. If I were starting a new Rails project today, I think the safest choice is one in which you have

---

a Procfile that kicks off a separate Webpack build and you use zero Rails magic. A slightly less safe but maybe more convenient choice would be trying rails/webpacker or another gem. The choice is yours.

For more discussion on this topic, see issues 203, 161, 43, etc.

## Getting Started

Add this line to your application's Gemfile:

```
1 gem "browserify-rails"
```

Create **package.json** in your Rails root:

```
1 {
2   "name": "something",
3   "dependencies": {
4     "browserify": "^14.0.0",
5     "browserify-incremental": "^3.1.0"
6   },
7   "license": "MIT",
8   "engines": {
9     "node": ">= 0.10"
10  }
11 }
```

Then run:

```
1 npm install
```

Then start writing CommonJS, and everything will magically work!:

```
1 // foo.js
2 module.exports = function (n) { return n * 11 }
3
4 // application.js
5 var foo = require('./foo');
6 console.log(foo(12));
```

**Gotchas with require and module.exports** Do not put `module.exports` or `require()` in JavaScript comments or strings. Doing so will certainly cause issues with compilation that are difficult to track down.

This happens because browserify-rails works by parsing your JavaScript files for these keywords that indicate whether it is a module, or is requiring a module. If a file meets one of these criteria, browserify will compile the modules as expected.

---

Because browserify-rails is working within the restraints of Ruby/Sprockets, the parsing is done by Ruby and therefore does not know whether it is a JavaScript string, comment, or function.

## CoffeeScript

For CoffeeScript support, make sure to follow the standard rails `.js.coffee` naming convention. You'll also need to add the npm package `coffeeify` as a dependency:

```
1 {
2   // ...
3   "dependencies" : {
4     // ...
5     "coffeeify": "~0.6"
6   }
7 }
```

and configure `browserify_rails` accordingly:

```
1 config.browserify_rails.commandline_options = "-t coffeeify --extension
  =\".js.coffee\""
```

## Requirements

- node-browserify 4.x
- browserify-incremental

## Configuration

### Global configuration

You can configure different options of browserify-rails by adding one of the lines mentioned below into your `config/application.rb` or your environment file (`config/environments/*.rb`):

```
1 class My::Application < Rails::Application
2   # Specify the file paths that should be browserified. We browserify
3   # everything that
4   # matches (==) one of the paths. So you will most likely put lambdas
5   # regexes in here.
6   #
7   # By default only files in /app and /node_modules are browserified,
8   # vendor stuff is normally not made for browserification and may stop
9   # working.
10  config.browserify_rails.paths << /vendor/assets/javascripts/module
    \.js/
```

---

```
10
11 # Environments in which to generate source maps
12 #
13 # The default is none
14 config.browserify_rails.source_map_environments << "development"
15
16 # Should the node_modules directory be evaluated for changes on page
  load
17 #
18 # The default is `false`
19 config.browserify_rails.evaluate_node_modules = true
20
21 # Force browserify on every found JavaScript asset if true.
22 # Can be a proc.
23 #
24 # The default is `false`
25 config.browserify_rails.force = ->(file) { File.extname(file) == ".ts
  " }
26
27 # Command line options used when running browserify
28 #
29 # can be provided as an array:
30 config.browserify_rails.commandline_options = ["-t browserify-shim",
  "--fast"]
31
32 # or as a string:
33 config.browserify_rails.commandline_options = "-t browserify-shim --
  fast"
34
35 # Define NODE_ENV to be used with envify
36 #
37 # defaults to Rails.env
38 config.browserify_rails.node_env = "production"
```

## browserify-incremental

browserify-incremental is used to cache browserification of CommonJS modules. One of the side effects is that the absolute module path is included in the emitted JavaScript. Most people do not want this for production code so browserify-incremental is currently disabled for the `production` and `staging` environments. Note that counter-intuitively, browserify-incremental helps even with a single build pass of your code because typically the same modules are used multiple times. So it helps even for say asset compilation on a push to Heroku.

**Enabling browserify-incremental in production** To enable browserify-incremental in production, add the following line to `config/environments/production.rb`:

---

```
1 config.browserify_rails.use_browserifyinc = true
```

## Multiple bundles

node-browserify supports multiple bundles and so does rails-browserify. It does this using `config/browserify.yml`. Below is an example.

Say you have three JavaScript files and one is a huge library you would like to use in both. Browserify lets you mark that huge library with `-require` in one file (to both bundle it and mark it with a special internal ID) and then require it in the other file and mark it with `-external` (so it is not bundled into the file but instead accessed via browserify internals using that special ID). Note that this only works when the file that has the library bundled is loaded before the file that uses the library with `-external`.

```
1 javascript:
2   main:
3     require:
4       - a_huge_library
5   secondary:
6     external:
7       - a_huge_library
```

Note that any valid browserify option is allowed in the YAML file but not all use cases have been considered. If your use case does not work, please open an issue with a runnable example of the problem including your browserify.yml file.

## Inside Isolated Engines

To make browserify-rails work inside an isolated engine, add the engine app directory to the browserify-rails paths (inside engine.rb):

```
1 config.browserify_rails.paths << -> (p) { p.start_with?(Engine.root.
  join("app").to_s) }
```

If you wish to put the `node_modules` directory within the engine, you have some control over it with:

```
1 config.browserify_rails.node_bin = "some/directory"
```

## Example setup

Refer to this repo for setting up this gem with ES6 and all front-end goodies like react and all - [github.com/gauravtiwari/browserify-rails](https://github.com/gauravtiwari/browserify-rails)

---

## Support for rails asset directories as non-relative module sources

In the Rails asset pipeline, it is common to have files in `app/assets/javascripts` and being able to do `//= require some_file` which exists in one of the asset/javascript directories. In some cases, it is useful to have similar functionality with browserify. This has been added by putting the Rails asset paths into `NODE_PATH` environment variable when running browserify.

But this comes at a large cost: right now, it appears to break source maps. This might be a bug or a fixable breakage but it hasn't been solved yet. The use of `NODE_PATH` is also contentious in the NodeJS community.

Why leave it in? Because some typical Rails components break without it. For example, jasmine-rails expects to be able to move JavaScript to different depths. So if you do a relative require from `spec/javascript` to `app/assets/javascripts`, your tests will fail to run when `RAILS_ENV=test`.

So if you really need this, use it. But if you really need it for files that are not tests, you should definitely figure out an alternative. Support for this may go away if we cannot fix the issue(s) with source maps being invalid.

## Deploying to Heroku

Heroku is a very common target for deploying. You'll have to add custom buildpacks that run `bundle` and `npm install` on the target machine.

```
1 $ heroku buildpacks:add https://github.com/heroku/heroku-buildpack-nodejs.git
2 $ heroku buildpacks:add https://github.com/heroku/heroku-buildpack-ruby.git
```

## Using Browserify Transforms

You can easily use a browserify transform by making some additions to your `package.json` and creating a `.babelrc`. For example, here is how you can add ES6 support in your app:

1. Add `babelify` and `babel-preset-es2015` to your `package.json` in your app's root directory either by editing the file directly and running `npm install` or using `npm install babelify --save` and `npm install babel-preset-es2015 --save`
2. Update your `package.json` to contain the babelify transform by adding the following lines

```
1 "browserify": {
2   "transform": [
3     [
```

---

```
4     "babelify"
5   ]
6 ]
7 }
```

3. Create a `.babelrc` file in the project root with the following contents

```
1 {
2   "plugins": [],
3   "presets": ["es2015"]
4 }
```

4. Create some `.es6` files and require them with `var m = require('./m.es6')` or `import m from './m.es6'`
5. Restart your server, and you now have ES6 support!

## Troubleshooting

### Clear the asset pipeline cache

The Rails asset pipeline caches some files in the `tmp` directory inside Rails root. It can happen that sometimes the cache does not get invalidated correctly. You can manually clear the cache in at least two ways:

1. `rake tmp:cache:clear`
2. `rm -rf ./tmp` (when in the root directory of the Rails project)

The second method is definitely brute force but if you experience issues, it is definitely worth trying before spending too much time debugging why something that is browserified appears to not match the sources files.

### Javascript Tests

If you want to use `browserify` to process test files as well, you will need to configure `browserify-rails` to process files in your `spec` or `test` directories.

```
1 config.browserify_rails.paths << -> (p) { p.start_with?(Rails.root.join
  ("spec/javascripts").to_s) }
```

---

## Acceptance Test Failures

If you have Sprockets precompile multiple JS files, each of which include certain browserified files, your acceptance tests may timeout before some of the assets have finished compiling.

To avoid this problem, run `rake assets:precompile` before running your acceptance tests.

## Contributing

Pull requests appreciated. Pull requests will not be rejected based on ideological neurosis of either the NodeJS or the Ruby on Rails communities. In other words, technical needs are respected.

## Running the dummy Rails app

There is a dummy rails app in `test/dummy`. You can change to that directory and run `bundle install` and then `bundle exec rails server`. You can see the test JavaScript files in `app/assets/javascripts` so try loading one – for example `http://localhost:3000/assets/application.js`.

You can use this dummy app to try out your coding/refactoring/hacking ideas and also see how the tests are written. To run the tests, run `bundle exec rake test` in the root directory of the browserify-rails code (not in the dummy app).

## Potential areas of change (contributions welcome)

### Multiple modules

Often one has one main module (say a library module) and other modules that consume the main module. It would be nice to be able to establish this relationship in the YAML file to avoid having to manually manage the require and external entries for the involved modules.

## Alternatives

### Use webpack or browserify directly instead of the asset pipeline

Use a tool like ProcMan to kick off a webpack or browserify process to rebuild your JavaScript on change. Reference the bundle in your Rails template and away you go. With webpack, you can even use the dev server and point to the dev server port in your Rails template to load JavaScript directly



---

from webpack (it'll block on build so you'll always get your latest code). This does require configuring webpack hot middleware to have a port (see `__webpack_hmr` goes to the wrong port and fails).

## Contributors

- Henry Hsu
- Cássio Souza
- Marten Lienen
- Lukasz Sagol
- Cymen Vig