Guides.rubyonrails.org

Asset Pipeline

This guide covers the **asset pipeline** introduced in **Rails** 3.1. By referring to this guide you will be able to:

- Understand what the **asset pipeline** is and what it does
- Properly **organize** your application assets
- Understand the benefits of the **asset pipeline**
- Adding a pre-processor to the **pipeline**
- Package assets with a gem

Chapters

1 What is the Asset Pipeline?

The **asset pipeline** provides a framework to concatenate and minify or compress **JavaScript** and CSS assets. It also adds the ability to write these assets in other languages such as CoffeeScript, Sass and ERB.

Prior to **Rails** 3.1 these features were added through third-party Ruby libraries such as Jammit and Sprockets. **Rails** 3.1 is integrated with Sprockets through ActionPack which depends on the `sprockets` gem, by default.

By having this as a core feature of **Rails**, all developers can benefit from the power of having their assets pre-processed, compressed and minified by one central library, Sprockets. This is part of **Rails**' "Fast by default" strategy as outlined by DHH in his 2011 keynote at Railsconf.

In new **Rails** 3.1 application the **asset** **pipeline** is enabled by default. It can be disabled in `application.rb` by putting this line inside the `Application` class definition:

```
config.assets.enabled = false
```

It is recommended that you use the defaults for all new apps.

1.1 Main Features

The first feature of the **pipeline** is to concatenate assets. This is important in a production environment, as it reduces the number of requests that a browser must make to render a web page. While **Rails** already has a feature to concatenate these types of assets — by placing `:cache => true` at the end of tags such as `javascript_include_tag` and `stylesheet_link_tag` — many people do not use it.

The default behavior in **Rails** 3.1 and onward is to concatenate all files into one master file each for JS and CSS. However, you can separate files or groups of files if required (see below). In production, an MD5 fingerprint is inserted into each filename so that the file is cached by the web browser but can be invalidated if the fingerprint is altered.

The second feature is to minify or compress assets. For CSS, this usually involves removing whitespace and comments. For **JavaScript**, more complex processes can be applied. You can choose from a set of built in options or specify your own.

The third feature is the ability to code these assets using another language, or language extension. These include Sass for CSS, CoffeeScript for **JavaScript**, and ERB for both.

1.2 What is Fingerprinting and Why Should I Care?

Fingerprinting is a technique whereby the filenames of content that is static or infrequently updated is altered to be unique to the content contained in the file.

When a filename is unique and based on its content, HTTP headers can be set to encourage caches everywhere (at ISPs, in browsers) to keep their own copy of the content. When the content is updated, the fingerprint will change and the remote clients will request the new file. This is generally known as *cachebusting*.

The most effective technique is to insert a hash of the content into the name, usually at the end. For example a CSS file `global.css` is hashed and the filename is updated to incorporate the hash.

```
global.css => global-908e25f4bf641868d8683022a5b62f54.css
```

This is the strategy adopted by the **Rails** **asset** **pipeline**.

Rails' old strategy was to append a query string to every **asset** linked with a built-in helper. In the source the generated code looked like this:

```
/stylesheets/global.css?1309495796
```

This has several disadvantages:

1. **Not all caches will cache content with a query string**
   Steve Souders recommends, "…avoiding a querystring for cacheable resources". He found that in these case 5-20% of requests will not be cached.
2. **The file name can change between nodes in multi-server environments.**
   The query string in **Rails** is based on the modification time of the files. When assets are deployed to a cluster, there is no guarantee that the timestamps will be the same, resulting in different values being used depending on which server handles the request.

The other problem is that when static assets are deployed with each new release of code, the mtime of **all** these files changes, forcing all remote clients to fetch them again, even when the content of those assets has not changed.

Fingerprinting avoids all these problems by ensuring filenames are consistent based on their content.

Fingerprinting is enabled by default for production and disabled for all the others environments. You can enable or disable it in your configuration through the `config.assets.digest` option.

More reading:

- Optimize caching
- Revving Filenames: don't use querystring

2 How to Use the Asset Pipeline

In previous versions of **Rails**, all assets were located in subdirectories of `public` such as `images`, `javascripts` and `stylesheets`. With the **asset** **pipeline**, the preferred location for these assets is now the `app/assets` directory. Files in this directory are served by the Sprockets middleware included in the sprockets gem.

This is not to say that assets can (or should) no longer be placed in `public`; they still can be and will be served as static files by the application or web server. You would only use `app/assets` if you wish your files to undergo some pre-processing before they are served.

In production, the default is to precompile these files to `public/assets` so that they can be more efficiently delivered by the webserver.

When a scaffold or controller is generated for the application, **Rails** also generates a **JavaScript** file (or CoffeeScript file if the `coffee-rails` gem is in the `Gemfile`) and a Cascading Style Sheet file (or SCSS file if `sass-rails` is in the `Gemfile`) for that controller.

For example, if a `ProjectsController` is generated, there will be a new file at `app/assets/javascripts/projects.js.coffee` and another at `app/assets/stylesheets/projects.css.scss`. You should put any **JavaScript** or CSS unique to a controller inside their respective **asset** files, as these files can then be loaded just for these controllers with lines such as `<%= javascript_include_tag params[:controller] %>` or `<%= stylesheet_link_tag params[:controller] %>`.

You will need a ExecJS – supported runtime in order to use CoffeeScript. If you are using Mac OS X or Windows you have a **JavaScript** runtime installed in your operating system. Check ExecJS documentation to know all supported **JavaScript** runtimes.

2.1 Asset Organization

Assets can be placed inside an application in one of three locations: `app/assets`, `lib/assets` or `vendor/assets`.

`app/assets` is for assets that are owned by the application, such as custom images, **JavaScript** files or stylesheets.

`lib/assets` is for your own libraries' code that doesn't really fit into the scope of the application or those libraries which are shared across applications.

`vendor/assets` is for assets that are owned by outside entities, such as code for **JavaScript** plugins.

All subdirectories that exist within these three locations are added to the search path for Sprockets (visible by calling `Rails.application.config.assets.paths` in a console). When an **asset** is requested, these paths are traversed to see if they contain an **asset** matching the name specified. Once an **asset** has been found, it's processed by Sprockets and served.

You can add additional (fully qualified) paths to the **pipeline** in `application.rb`. For example:

```
config.assets.paths << File.join(Rails.root, 'app', 'assets', 'flash')
```

2.2 Coding Links to Assets

Sprockets does not add any new methods to access your assets – you still use the familiar `javascript_include_tag` and `stylesheet_link_tag`.

```
<%= stylesheet_link_tag "application" %>
<%= javascript_include_tag "application" %>
```

In regular views you can access images in the `assets/images` directory like this:

```
<%= image_tag "rails.png" %>
```

Provided that the **pipeline** is enabled within your application (and not disabled in the current environment context), this file is served by Sprockets. If a file exists at `public/assets/rails.png` it is served by the webserver.

Alternatively, a request for a file with an MD5 hash such as `public/assets/rails-af27b6a414e6da00003503148be9b409.png` is treated the same way. How these hashes are generated is covered in the Production Assets section later on in this guide.

Sprockets will also look through the paths specified in `config.assets.paths` which includes the standard application paths and any path added by **Rails** engines.

Images can also be organized into subdirectories if required, and they can be accessed by specifying the directory's name in the tag:

```
<%= image_tag "icons/rails.png" %>
```

2.2.1 CSS and ERB

If you add an `erb` extension to a CSS **asset**, making it something such as `application.css.erb`, then you can use the `asset_path` helper in your CSS rules:

```
.class { background-image: url(<%= asset_path 'image.png' %>) }
```

This writes the path to the particular asset being referenced. In this example, it would make sense to have an image in one of the asset load paths, such as `app/assets/images/image.png`, which would be referenced here. If this image is already available in `public/assets` as a fingerprinted file, then that path is referenced.

If you want to use a css data URI — a method of embedding the image data directly into the CSS file — you can use the `asset_data_uri` helper.

```
#logo { background: url(<%= asset_data_uri 'logo.png' %>) }
```

This inserts a correctly-formatted data URI into the CSS source.

Note that the closing tag cannot be of the style `-%>`.

2.2.2 CSS and Sass

When using the asset pipeline, paths to assets must be re-written and `sass-rails` provides `_url` and `_path` helpers for the following asset classes: image, font, video, audio, javascript, stylesheet.

- `image-url("rails.png")` becomes `url(/assets/rails.png)`
- `image-path("rails.png")` becomes `"/assets/rails.png"`.

The more generic form can also be used but the asset path and class must both be specified:

- `asset-url("rails.png", image)` becomes `url(/assets/rails.png)`
- `asset-path("rails.png", image)` becomes `"/assets/rails.png"`

2.3 Manifest Files and Directives

Sprockets uses manifest files to determine which assets to include and serve. These manifest files contain *directives* — instructions that tell Sprockets which files to require in order to build a single CSS or JavaScript file. With these directives, Sprockets loads the files specified, processes them if necessary, concatenates them into one single file and then compresses them (if `Rails.application.config.assets.compress` is set to `true`). By serving one file rather than many, the load time of pages are greatly reduced as there are fewer requests to make.

For example, in the default Rails application there's a `app/assets/javascripts/application.js` file which contains the following lines:

```
//= require jquery
//= require jquery_ujs
//= require_tree .
```

In JavaScript files, the directives begin with `//=`. In this case, the file is using the `require` and the `require_tree` directives. The `require` directive is used to tell Sprockets the files that you wish to require. Here, you are requiring the files `jquery.js` and `jquery_ujs.js` that are available somewhere in the search path for Sprockets. You need not supply the extensions explicitly. Sprockets assumes you are requiring a `.js` file when done from within a `.js` file.

In Rails 3.1, the `jquery.js` and `jquery_ujs.js` files are located inside the `vendor/assets/javascripts` directory contained within the `jquery-rails` gem.

The `require_tree .` directive tells Sprockets to include *all* JavaScript files in this directory into the output. Only a path relative to the file can be specified. There is also a `require_directory` directive which includes all JavaScript files only in the directory specified (no nesting).

There's also a default `app/assets/stylesheets/application.css` file which contains these lines:

```
/* ...
*= require_self
*= require_tree .
*/
```

The directives that work in the **JavaScript** files also work in stylesheets, obviously including stylesheets rather than **JavaScript** files. The `require_tree` directive here works the same way as the **JavaScript** one, requiring all stylesheets from the current directory.

In this example `require_self` is used. This puts the CSS contained within the file (if any) at the top of any other CSS in this file unless `require_self` is specified after another `require` directive.

You can have as many manifest files as you need. For example the `admin.css` and `admin.js` manifest could contain the JS and CSS files that are used for the admin section of an application.

For some assets (like CSS) the compiled order is important. You can specify individual files and they are compiled in the order specified:

```
/* ...
*= require reset
*= require layout
*= require chrome
*/
```

2.4 Preprocessing

The file extensions used on an **asset** determine what preprocessing is applied. When a controller or a scaffold is generated with the default **Rails** gemset, a CoffeeScript file and a SCSS file are generated in place of a regular **JavaScript** and CSS file. The example used before was a controller called "projects", which generated an `app/assets/javascripts/projects.js.coffee` and a `app/assets/stylesheets/projects.css.scss` file.

When these files are requested, they are processed by the processors provided by the `coffee-script` and `sass-rails` gems and then sent back to the browser as **JavaScript** and CSS respectively.

Additional layers of pre-processing can be requested by adding other extensions, where each extension is processed in a right-to-left manner. These should be used in the order the processing should be applied. For example, a stylesheet called `app/assets/stylesheets/projects.css.scss.erb` is first processed as ERB, then SCSS and finally served as CSS. The same applies to a **JavaScript** file — `app/assets/javascripts/projects.js.coffee.erb` is processed as ERB, CoffeeScript and served as **JavaScript**.

Keep in mind that the order of these pre-processors is important. For example, if you called your **JavaScript** file `app/assets/javascripts/projects.js.erb.coffee` then it is processed with the CoffeeScript interpreter first, which wouldn't understand ERB and therefore you would run into problems.

3 In Development

In development mode assets are served as separate files in the order they are specified in the manifest file.

This manifest `application.js`:

```
//= require core
//= require projects
//= require tickets
```

would generate this HTML:

```
<script src='/assets/core.js?body=1'></script>
<script src='/assets/projects.js?body=1'></script>
<script src='/assets/tickets.js?body=1'></script>
```

The `body` param is required by Sprockets.

3.1 Turning Debugging off

You can turn off debug mode by updating `development.rb` to include:

```
config.assets.debug = false
```

When debug mode is off Sprockets will concatenate and run the necessary preprocessors on all files, generating the following HTML:

```
<script src='/assets/application.js'></script>
```

Assets are compiled and cached on the first request after the server is started. Sprockets sets a `must-validate` Cache-Control HTTP header to reduce request overhead on subsequent requests — on these the browser gets a 304 (not-modified) response.

If any of the files in the manifest have changed between requests, the server responds with a new compiled file.

You can put `?debug_assets=true` or `?debug_assets=1` at the end of a URL to enable debug mode on-demand, and this will render individual tags for each file. This is useful for tracking down exact line numbers when debugging.

Debug can also be set in the **Rails** helper methods:

```
<%= stylesheet_link_tag "application", :debug => true %>
<%= javascript_include_tag "application", :debug => true %>
```

The `:debug` option is ignored if the debug mode is off.

You could potentially also enable compression in development mode as a sanity check, and disable it on-demand as required for debugging.

4 In Production

In the production environment **Rails** uses the fingerprinting scheme outlined above. By default it is assumed that assets have been precompiled and will be served as static assets by your web server.

During the precompilation phase an MD5 is generated from the contents of the compiled files, and inserted into the filenames as they are written to disc. These fingerprinted names are used by the **Rails** helpers in place of the manifest name.

For example this:

```
<%= javascript_include_tag "application" %>
<%= stylesheet_link_tag "application" %>
```

generates something like this:

```
<script src="/assets/application-908e25f4bf641868d8683022a5b62f54.js"
type="text/javascript"></script>
<link href="/assets/application-4dd5b109ee3439da54f5bdfd78a80473.css" media="screen"
rel="stylesheet" type="text/css" />
```

The fingerprinting behavior is controlled by the setting of `config.assets.digest` setting in **Rails** (which is `true` for production, `false` for everything else).

Under normal circumstances the default option should not be changed. If there are no digests in the filenames, and far-future headers are set, remote clients will never know to refetch the files when their content changes.

4.1 Precompiling Assets

Rails comes bundled with a rake task to compile the **asset** manifests and other files in the **pipeline** to disc.

Compiled assets are written to the location specified in `config.assets.prefix`. The default setting will use the `public/assets` directory.

You must use this task either during deployment or locally if you do not have write access to your production filesystem.

The rake task is:

```
bundle exec rake assets:precompile
```

Capistrano (v2.8.0`) has a recipe to handle this in deployment. Add the following line to` Capfile+:

```
load 'deploy/assets'
```

This links the folder specified in `config.assets.prefix` to `shared/assets`. If you already use this shared folder you'll need to write your own deployment task.

It is important that this folder is shared between deployments so that remotely cached pages that reference the old compiled assets still work for the life of the cached page.

The default matcher for compiling files includes `application.js`, `application.css` and all files that do not end in `js` or `css`:

```
[ /\w<notextile><tt>\.(?!js|css).</tt></notextile>/, /application.(css|js)$/ ]
```

If you have other manifests or individual stylesheets and **JavaScript** files to include, you can add them to the `precompile` array:

```
config.assets.precompile += ['admin.js', 'admin.css', 'swfObject.js']
```

The rake task also generates a `manifest.yml` that contains a list with all your assets and their respective fingerprints. This is used by the **Rails** helper methods and avoids handing the mapping requests back to Sprockets. Manifest file typically look like this:

```
---
rails.png: rails-bd9ad5a560b5a3a7be0808c5cd76a798.png
jquery-ui.min.js: jquery-ui-7e33882a28fc84ad0e0e47e46cbf901c.min.js
jquery.min.js: jquery-8a50feed8d29566738ad005e19fe1c2d.min.js
application.js: application-3fdab497b8fb70d20cfc5495239dfc29.js
application.css: application-8af74128f904600e41a6e39241464e03.css
```

The default location for the manifest is the root of the location specified in `config.assets.prefix` ('/assets' by default).

This can be changed with the `config.assets.manifest` option. A fully specified path is required:

```
config.assets.manifest = '/path/to/some/other/location'
```

If there are missing precompiled files in production you will get an "AssetNoPrecompiledError" exception indicating the name of the missing file.

4.1.1 Server Configuration

Precompiled assets exist on the filesystem and are served directly by your webserver. They do not have far-future headers by default, so to get the benefit of fingerprinting you'll have to update your server configuration to add them.

For Apache:

```
<LocationMatch "^/assets/.*$">
  # Some browsers still send conditional-GET requests if there's a
  # Last-Modified header or an ETag header even if they haven't
  # reached the expiry date sent in the Expires header.
  Header unset Last-Modified
  Header unset ETag
  FileETag None
  # RFC says only cache for 1 year
  ExpiresActive On
  ExpiresDefault "access plus 1 year"
</LocationMatch>
```

TODO: nginx instructions

When files are precompiled, Sprockets also creates a Gzip (.gz) version of your assets. This avoids the server having to do this for any requests; it can simply read the compressed files from disc. You must configure your server to use gzip compression and serve the compressed assets that will be stored in the `public/assets` folder. The following configuration options can be used:

For Apache:

```
<LocationMatch "^/assets/.*$">
  # 2 lines to serve pre-gzipped version
  RewriteCond %{REQUEST_FILENAME}.gz -s
  RewriteRule ^(.+) $1.gz [L]

  # without it, Content-Type will be "application/x-gzip"
  <FilesMatch .*\.css.gz>
      ForceType text/css
  </FilesMatch>

  <FilesMatch .*\.js.gz>
```

```
    ForceType text/javascript
  </FilesMatch>
</LocationMatch>
```

For nginx:

```
location ~ ^/(assets)/  {
  root /path/to/public;
  gzip_static on; # to serve pre-gzipped version
  expires max;
  add_header  Cache-Control public;
}
```

4.2 Live Compilation

In some circumstances you may wish to use live compilation. In this mode all requests for assets in the **Pipeline** are handled by Sprockets directly.

To enable this option set:

```
config.assets.compile = true
```

On the first request the assets are compiled and cached as outlined in development above, and the manifest names used in the helpers are altered to include the MD5 hash.

Sprockets also sets the `Cache-Control` HTTP header to `max-age=31536000`. This signals all caches between your server and the client browser that this content (the file served) can be cached for 1 year. The effect of this is to reduce the number of requests for this **asset** from your server; the **asset** has a good chance of being in the local browser cache or some intermediate cache.

This mode uses more memory and is lower performance than the default. It is not recommended.

5 Customizing the Pipeline

5.1 CSS Compression

There is currently one option for compressing CSS, YUI. This Gem extends the CSS syntax and offers minification.

The following line enables YUI compression, and requires the `yui-compressor` gem.

```
config.assets.css_compressor = :yui
```

The `config.assets.compress` must be set to `true` to enable CSS compression

5.2 JavaScript Compression

Possible options for **JavaScript** compression are `:closure`, `:uglifier` and `:yui`. These require the use of the `closure-compiler`, `uglifier` or `yui-compressor` gems respectively.

The default Gemfile includes uglifier. This gem wraps UglifierJS (written for NodeJS) in Ruby. It compresses your code by removing white space and other magical things like changing your `if` and `else` statements to ternary operators where possible.

The following line invokes `uglifier` for **JavaScript** compression.

```
config.assets.js_compressor = :uglifier
```

The `config.assets.compress` must be set to `true` to enable **JavaScript** compression

You will need a ExecJS — supported runtime in order to use `uglifier`. If you are using Mac OS X or Windows you have installed a **JavaScript** runtime in your operating system. Check ExecJS documentation to know all supported **JavaScript** runtimes.

5.3 Using Your Own Compressor

The compressor config settings for CSS and **JavaScript** also take any Object. This object must have a `compress` method that takes a string as the sole argument and it must return a string.

```
class Transformer
  def compress(string)
    do_something_returning_a_string(string)
  end
end
```

To enable this, pass a `new` Object to the config option in `application.rb`:

```
config.assets.css_compressor = Transformer.new
```

5.4 Changing the *assets* Path

The public path that Sprockets uses by default is `/assets`.

This can be changed to something else:

```
config.assets.prefix = "/some_other_path"
```

This is a handy option if you have any existing project (pre **Rails** 3.1) that already uses this path or you wish to use this path for a new resource.

5.5 X-Sendfile Headers

The X-Sendfile header is a directive to the server to ignore the response from the application, and instead serve the file specified in the headers. This option is off by default, but can be enabled if your server supports it. When enabled, this passes responsibility for serving the file to the web server, which is faster.

Apache and nginx support this option which is enabled in `config/environments/production.rb`.

```
# config.action_dispatch.x_sendfile_header = "X-Sendfile" # for apache
# config.action_dispatch.x_sendfile_header = 'X-Accel-Redirect' # for nginx
```

If you are upgrading an existing application and intend to use this option, take care to paste this configuration option only into `production.rb` (and not `application.rb`) and any other environment you define with production behavior.

6 How Caching Works

Sprockets uses the default **rails** cache store to cache assets in development and production.

TODO: Add more about changing the default store.

7 Adding Assets to Your Gems

Assets can also come from external sources in the form of gems.

A good example of this is the `jquery-rails` gem which comes with Rails as the standard JavaScript library gem. This gem contains an engine class which inherits from `Rails::Engine`. By doing this, Rails is informed that the directory for this gem may contain assets and the `app/assets`, `lib/assets` and `vendor/assets` directories of this engine are added to the search path of Sprockets.

8 Making Your Library or Gem a Pre-Processor

TODO: Registering gems on Tilt enabling Sprockets to find them.

9 Upgrading from Old Versions of Rails

There are two issues when upgrading. The first is moving the files to the new locations. See the section above for guidance on the correct locations for different file types.

The second is updating the various environment files with the correct default options. The following changes reflect the defaults in version 3.1.0.

In `application.rb`:

```
# Enable the asset pipeline
config.assets.enabled = true

# Version of your assets, change this if you want to expire all your assets
config.assets.version = '1.0'

# Change the path that assets are served from
# config.assets.prefix     = "/assets"
```

In `development.rb`:

```
# Do not compress assets
config.assets.compress = false

# Expands the lines which load the assets
config.assets.debug = true
```

And in `production.rb`:

```
# Compress JavaScripts and CSS
config.assets.compress = true

# Choose the compressors to use
# config.assets.js_compressor  = :uglifier
# config.assets.css_compressor = :yui

# Don't fallback to assets pipeline if a precompiled asset is missed
config.assets.compile = false

# Generate digests for assets URLs.
```

```
config.assets.digest = true

# Defaults to Rails.root.join("public/assets")
# config.assets.manifest = YOUR_PATH

# Precompile additional assets (application.js, application.css, and all non-JS/CSS are
already added)
# config.assets.precompile += %w( search.js )
```

There are no changes to `test.rb`. The defaults in the test environment are: `config.assets.compile` is true and `config.assets.compress`, `config.assets.debug` and `config.assets.digest` are false.

The following should also be added to `Gemfile`:

```
# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails', "  ~> 3.1.0"
  gem 'coffee-rails', "~> 3.1.0"
  gem 'uglifier'
end
```

Feedback

You're encouraged to help improve the quality of this guide.

If you see any typos or factual errors you are confident to patch, please clone docrails and push the change yourself. That branch of Rails has public write access. Commits are still reviewed, but that happens after you've submitted your contribution. docrails is cross-merged with master periodically.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Check the Ruby on Rails Guides Guidelines for style and conventions.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the rubyonrails-docs mailing list.