The Asset Pipeline

This guide covers the asset pipeline.

After reading this guide, you will know:

- What the asset pipeline is and what it does.
- How to properly organize your application assets.
- The benefits of the asset pipeline.
- How to add a pre-processor to the pipeline.
- How to package assets with a gem.

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 and pre-processors such as CoffeeScript, Sass, and ERB. It allows assets in your application to be automatically combined with assets from other gems.

The asset pipeline is implemented by the sprockets-rails gem, and is enabled by default. You can disable it while creating a new application by passing the `--skip-asset-pipeline` option.

```
$ rails new appname --skip-asset-pipeline
```

Rails can easily work with Sass by adding the `sassc-rails` gem to your `Gemfile`, which is used by Sprockets for Sass compilation:

```
gem 'sassc-rails'
```

To set asset compression methods, set the appropriate configuration options in `production.rb` - `config.assets.css_compressor` for your CSS and `config.assets.js_compressor` for your JavaScript:

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

NOTE: The `sassc-rails` gem is automatically used for CSS compression if included in the `Gemfile` and no `config.assets.css_compressor` option is set.

Main Features

The first feature of the pipeline is to concatenate assets, which can reduce the number of requests that a browser makes to render a web page. Web browsers are limited in the number of requests that they can make in parallel, so fewer requests can mean faster loading for your application.

Sprockets concatenates all JavaScript files into one master `.js` file and all CSS files into one master `.css` file. As you'll learn later in this guide, you can customize this strategy to group files any way you like. In production, Rails inserts an SHA256 fingerprint into each filename so that the file is cached by the web browser. You can invalidate the cache by altering this fingerprint, which happens automatically whenever you change the file contents.

The second feature of the asset pipeline is asset minification or compression. For CSS files, this is done by 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 of the asset pipeline is it allows coding assets via a higher-level language, with precompilation down to the actual assets. Supported languages include Sass for CSS, CoffeeScript for JavaScript, and ERB for both by default.

What is Fingerprinting and Why Should I Care?

Fingerprinting is a technique that makes the name of a file dependent on the contents of the file. When the file contents change, the filename is also changed. For content that is static or infrequently changed, this provides an easy way to tell whether two versions of a file are identical, even across different servers or deployment dates.

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

The technique Sprockets uses for fingerprinting is to insert a hash of the content into the name, usually at the end. For example a CSS file `global.css`

```
global-908e25f4bf641868d8683022a5b62f54.css
```

This is the strategy adopted by the Rails asset pipeline.

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

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

The query string strategy has several disadvantages:

1. **Not all caches will reliably cache content where the filename only differs by query parameters**

   Steve Souders recommends,

"...avoiding a querystring for cacheable resources". He found that in this case 5-20% of requests will not be cached. Query strings in particular do not work at all with some CDNs for cache invalidation.

2. **The file name can change between nodes in multi-server environments.**

   The default query string in Rails 2.x 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.

3. **Too much cache invalidation**

   When static assets are deployed with each new release of code, the mtime

(time of last modification) of *all* these files changes, forcing all remote clients to fetch them again, even when the content of those assets has not changed.

Fingerprinting fixes these problems by avoiding query strings, and by ensuring that filenames are consistent based on their content.

Fingerprinting is enabled by default for both the development and production 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

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.

Assets can still be placed in the `public` hierarchy. Any assets under `public` will be served as static files by the application or web server when `config.public_file_server.enabled` is set to true. You should use `app/assets` for files that must undergo some pre-processing before they are served.

In production, Rails precompiles these files to `public/assets` by default. The precompiled copies are then served as static assets by the web server. The files in `app/assets` are never served directly in production.

Controller Specific Assets

When you generate a scaffold or a controller, Rails also generates a Cascading Style Sheet file (or SCSS file if `sass-rails` is in the `Gemfile`) for that controller. Additionally, when generating a scaffold, Rails generates the file `scaffolds.css` (or `scaffolds.scss` if `sass-rails` is in the `Gemfile`.)

For example, if you generate a `ProjectsController`, Rails will also add a new file at `app/assets/stylesheets/projects.scss`. By default these files will be ready to use by your application immediately using the `require_tree` directive. See Manifest Files and Directives for more details on require_tree.

You can also opt to include controller specific stylesheets and JavaScript files only in their respective controllers using the following:

`<%= javascript_include_tag params[:controller] %>` or `<%= stylesheet_link_tag params[:controller] %>`

When doing this, ensure you are not using the `require_tree` directive, as that will result in your assets being included more than once.

WARNING: When using asset precompilation, you will need to ensure that your controller assets will be precompiled when loading them on a per page basis. By default `.coffee` and `.scss` files will not be precompiled on their own. See Precompiling Assets for more information on how precompiling works.

NOTE: You must have an ExecJS supported runtime in order to use CoffeeScript. If you are using macOS or Windows, you have a JavaScript runtime installed in your operating system. Check ExecJS documentation to know all supported JavaScript runtimes.

Asset Organization

Pipeline 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 and CSS frameworks. Keep in mind that third party code with references to other files also processed by the asset Pipeline (images, stylesheets, etc.), will need to be rewritten to use helpers like `asset_path`.

Search Paths

When a file is referenced from a manifest or a helper, Sprockets searches the three default asset locations for it.

The default locations are: the `images`, `javascripts` and `stylesheets` directories under the `app/assets` folder, but these subdirectories are not special - any path under `assets/*` will be searched.

For example, these files:

```
app/assets/javascripts/home.js
lib/assets/javascripts/moovinator.js
vendor/assets/javascripts/slider.js
vendor/assets/somepackage/phonebox.js
```

would be referenced in a manifest like this:

```
//= require home
//= require moovinator
//= require slider
//= require phonebox
```

Assets inside subdirectories can also be accessed.

```
app/assets/javascripts/sub/something.js
```

is referenced as:

```
//= require sub/something
```

You can view the search path by inspecting `Rails.application.config.assets.paths` in the Rails console.

Besides the standard `assets/*` paths, additional (fully qualified) paths can be added to the pipeline in `config/initializers/assets.rb`. For example:

```
Rails.application.config.assets.paths << Rails.root.join("lib", "videoplayer",
"flash")
```

Paths are traversed in the order they occur in the search path. By default, this means the files in `app/assets` take precedence, and will mask corresponding paths in `lib` and `vendor`.

It is important to note that files you want to reference outside a manifest must be added to the precompile array or they will not be available in the production environment.

Using Index Files

Sprockets uses files named `index` (with the relevant extensions) for a special purpose.

For example, if you have a jQuery library with many modules, which is stored in `lib/assets/javascripts/library_name`, the file `lib/assets/javascripts/library_name/index.js` serves as the manifest for all files in this library. This file could include a list of all the required files in order, or a simple `require_tree` directive.

The library as a whole can be accessed in the application manifest like so:

```
//= require library_name
```

This simplifies maintenance and keeps things clean by allowing related code to be grouped before inclusion elsewhere.

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", media: "all" %>
<%= javascript_include_tag "application" %>
```

If using the turbolinks gem, which is included by default in Rails, then include the 'data-turbo-track' option which causes Turbo to check if an asset has been updated and if so loads it into the page:

```
<%= stylesheet_link_tag "application", media: "all", "data-turbo-track" => "reload"
%>
<%= javascript_include_tag "application", "data-turbo-track" => "reload" %>
```

In regular views you can access images in the `app/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 web server.

Alternatively, a request for a file with an SHA256 hash such as `public/assets/rails-f90d8a84c707a8dc923fca1ca1895ae8ed0a09237f6992015fef1e11be77c023.png` is treated the same way. How these hashes are generated is covered in the In Production 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 paths added by Rails engines.

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

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

WARNING: If you're precompiling your assets (see In Production below), linking to an asset that does not exist will raise an exception in the calling page. This includes linking to a blank string. As such, be careful using `image_tag` and the other helpers with user-supplied data.

CSS and ERB

The asset pipeline automatically evaluates ERB. This means if you add an `erb` extension to a CSS asset (for example, `application.css.erb`), then helpers like `asset_path` are available 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 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 `-%>`.

CSS and Sass

When using the asset pipeline, paths to assets must be re-written and `sass-rails` provides `-url` and `-path` helpers (hyphenated in Sass, underscored in Ruby) for the following asset classes: image, font, video, audio, JavaScript and stylesheet.

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

The more generic form can also be used:

- `asset-url("rails.png")` returns `url(/assets/rails.png)`
- `asset-path("rails.png")` returns `"/assets/rails.png"`

JavaScript/CoffeeScript and ERB

If you add an `erb` extension to a JavaScript asset, making it something such as `application.js.erb`, you can then use the `asset_path` helper in your JavaScript code:

```
document.getElementById('logo').src = "<%= asset_path('logo.png') %>"
```

This writes the path to the particular asset being referenced.

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 (based on value of `Rails.application.config.assets.js_compressor`). By serving one file rather than many, the load time of pages can be greatly reduced because the browser makes fewer requests. Compression also reduces file size, enabling the browser to download them faster.

For example, with a `app/assets/javascripts/application.js` file containing the following lines:

```
// ...
//= require rails-ujs
//= require turbolinks
//= require_tree .
```

In JavaScript files, Sprockets directives begin with `//=`. In the above case, the file is using the `require` and the `require_tree` directives. The `require` directive is used to tell Sprockets the files you wish to require. Here, you are requiring the files `rails-ujs.js` and `turbolinks.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.

The `require_tree` directive tells Sprockets to recursively include *all* JavaScript files in the specified directory into the output. These paths must be specified relative to the manifest file. You can also use the `require_directory` directive which includes all JavaScript files only in the directory specified, without recursion.

Directives are processed top to bottom, but the order in which files are included by `require_tree` is unspecified. You should not rely on any particular order among those. If you need to ensure some particular JavaScript ends up above some other in the concatenated file, require the prerequisite file first in the manifest. Note that the family of `require` directives prevents files from being included twice in the output.

Rails also creates a default `app/assets/stylesheets/application.css` file which contains these lines:

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

Rails creates `app/assets/stylesheets/application.css` regardless of whether the `--skip-asset-pipeline` option is used when creating a new Rails application. This is so you can easily add asset pipelining later if you like.

The directives that work in JavaScript files also work in stylesheets (though obviously including stylesheets rather than JavaScript files). The `require_tree` directive in a CSS manifest 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 precise location of the `require_self` call.

NOTE. If you want to use multiple Sass files, you should generally use the Sass `@import` rule instead of these Sprockets directives. When using Sprockets directives, Sass files exist within their own scope, making variables or mixins only available within the document they were defined in.

You can do file globbing as well using `@import "*"`, and `@import "**/*"` to add the whole tree which is equivalent to how `require_tree` works. Check the sass-rails documentation for more info and important caveats.

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.

The same remarks about ordering made above apply. In particular, you can specify individual files and they are compiled in the order specified. For example, you might concatenate three CSS files together this way:

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

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, an SCSS file is generated in place of a regular CSS file. The example used before was a controller called "projects", which generated an `app/assets/stylesheets/projects.scss` file.

In development mode, or if the asset pipeline is disabled, when this file is requested it is processed by the processor provided by the `sass-rails` gem and then sent back to the browser as CSS. When asset pipelining is enabled, this file is preprocessed and placed in the `public/assets` directory for serving by either the Rails app or web server.

Additional layers of preprocessing 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.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.coffee.erb` is processed as ERB, then CoffeeScript, and served as JavaScript.

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

In Development

In development mode, assets are served as a concatenated file.

This manifest `app/assets/javascripts/application.js`:

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

would generate this HTML:

```
<script src="/assets/application-
728742f3b9daa182fe7c831f6a3b8fa87609b4007fdc2f87c134a07b19ad93fb.js"></script>
```

Raise an Error When an Asset is Not Found

If you are using sprockets-rails >= 3.2.0 you can configure what happens when an asset lookup is performed and nothing is found. If you turn off "asset fallback" then an error will be raised when an asset cannot be found.

```
config.assets.unknown_asset_fallback = false
```

If "asset fallback" is enabled then when an asset cannot be found the path will be output instead and no error raised. The asset fallback behavior is disabled by default.

Turning Digests Off

You can turn off digests by updating `config/environments/development.rb` to include:

```
config.assets.digest = false
```

When this option is true, digests will be generated for asset URLs.

Turning Source Maps On

You can turn on source maps by updating `config/environments/development.rb` to include:

```
config.assets.debug = true
```

When debug mode is on, Sprockets will generate a Source Map for each asset. This allows you to debug each file individually in your browser's developer tools.

Assets are compiled and cached on the first request after the server is started. Sprockets sets a `must-revalidate` 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 change between requests, the server responds with a new compiled file.

In Production

In the production environment Sprockets uses the fingerprinting scheme outlined above. By default Rails assumes assets have been precompiled and will be served as static assets by your web server.

During the precompilation phase an SHA256 is generated from the contents of the compiled files, and inserted into the filenames as they are written to disk. 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"></script>
<link href="/assets/application-4dd5b109ee3439da54f5bdfd78a80473.css"
rel="stylesheet" />
```

NOTE: with the Asset Pipeline the `:cache` and `:concat` options aren't used anymore, delete these options from the `javascript_include_tag` and `stylesheet_link_tag`.

The fingerprinting behavior is controlled by the `config.assets.digest` initialization option (which defaults to `true`).

NOTE: Under normal circumstances the default `config.assets.digest` 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.

Precompiling Assets

Rails comes bundled with a command to compile the asset manifests and other files in the pipeline.

Compiled assets are written to the location specified in `config.assets.prefix`. By default, this is the `/assets` directory.

You can call this command on the server during deployment to create compiled versions of your assets directly on the server. See the next section for information on compiling locally.

The command is:

```
$ RAILS_ENV=production rails assets:precompile
```

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 command.

It is important that this folder is shared between deployments so that remotely cached pages referencing 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 non-JS/CSS files (this will include all image assets automatically) from `app/assets` folders including your gems:

```
[ Proc.new { |filename, path| path =~ /app\/assets/ && !%w(.js .css).include?
(File.extname(filename)) },
/application.(css|js)$/ ]
```

NOTE: The matcher (and other members of the precompile array; see below) is applied to final compiled file names. This means anything that compiles to JS/CSS is excluded, as well as raw JS/CSS files; for example, `.coffee` and `.scss` files are **not** automatically included as they compile to JS/CSS.

If you have other manifests or individual stylesheets and JavaScript files to include, you can add them to the `precompile` array in `config/initializers/assets.rb`:

```
Rails.application.config.assets.precompile += %w( admin.js admin.css )
```

NOTE. Always specify an expected compiled filename that ends with `.js` or `.css`, even if you want to add Sass or CoffeeScript files to the precompile array.

The command also generates a `.sprockets-manifest-randomhex.json` (where `randomhex` is a 16-byte random hex string) that contains a list with all your assets and their respective fingerprints. This is used by the Rails helper methods to avoid handing the mapping requests back to Sprockets. A typical manifest file looks like:

```
{"files":{"application-
aee4be71f1288037ae78b997df388332edfd246471b533dcedaa8f9fe156442b.js":
{"logical_path":"application.js","mtime":"2016-12-23T20:12:03-05:00","size":412383,
"digest":"aee4be71f1288037ae78b997df388332edfd246471b533dcedaa8f9fe156442b","integrity'
ruS+cfEogDeueLmX3ziDMu39JGRxtTPc7aqPn+FWRCs="},
"application-86a292b5070793c37e2c0e5f39f73bb387644eaeada7f96e6fc040a028b16c18.css":
{"logical_path":"application.css","mtime":"2016-12-23T19:12:20-05:00","size":2994,
"digest":"86a292b5070793c37e2c0e5f39f73bb387644eaeada7f96e6fc040a028b16c18","integrity'
hqKStQcHk8N+LA5fOfc7s4dkTq6tp/lub8BAoCixbBg="},
"favicon-8d2387b8d4d32cecd93fa3900df0e9ff89d01aacd84f50e780c17c9f6b3d0eda.ico":
{"logical_path":"favicon.ico","mtime":"2016-12-23T20:11:00-05:00","size":8629,
"digest":"8d2387b8d4d32cecd93fa3900df0e9ff89d01aacd84f50e780c17c9f6b3d0eda","integrity'
jSOHuNTTLOzZP6OQDfDp/4nQGqzYT1DngMF8n2s9Dto="},
"my_image-f4028156fd7eca03584d5f2fc0470df1e0dbc7369eaae638b2ff033f988ec493.png":
{"logical_path":"my_image.png","mtime":"2016-12-23T20:10:54-05:00","size":23414,
"digest":"f4028156fd7eca03584d5f2fc0470df1e0dbc7369eaae638b2ff033f988ec493","integrity'
9AKBVv1+ygNYTV8vwEcN8eDbxzaequY4sv8DP5iOxJM="}},
"assets":{"application.js":"application-
aee4be71f1288037ae78b997df388332edfd246471b533dcedaa8f9fe156442b.js",
"application.css":"application-
86a292b5070793c37e2c0e5f39f73bb387644eaeada7f96e6fc040a028b16c18.css",
"favicon.ico":"favicon-
8d2387b8d4d32cecd93fa3900df0e9ff89d01aacd84f50e780c17c9f6b3d0eda.ico",
"my_image.png":"my_image-
f4028156fd7eca03584d5f2fc0470df1e0dbc7369eaae638b2ff033f988ec493.png"}}
```

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

NOTE: If there are missing precompiled files in production you will get a `Sprockets::Helpers::RailsHelper::AssetPaths::AssetNotPrecompiledError` exception indicating the name of the missing file(s).

Far-future Expires Header

Precompiled assets exist on the file system and are served directly by your web server. 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 those headers.

For Apache:

```
# The Expires* directives requires the Apache module
# `mod_expires` to be enabled.
<Location /assets/>
  # Use of ETag is discouraged when Last-Modified is present
  Header unset ETag
  FileETag None
  # RFC says only cache for 1 year
  ExpiresActive On
  ExpiresDefault "access plus 1 year"
</Location>
```

For NGINX:

```
location ~ ^/assets/ {
  expires 1y;
  add_header Cache-Control public;

  add_header ETag "";
}
```

Local Precompilation

Sometimes, you may not want or be able to compile assets on the production server. For instance, you may have limited write access to your production filesystem, or you may plan to deploy frequently without making any changes to your assets.

In such cases, you can precompile assets *locally* — that is, add a finalized set of compiled, production-ready assets to your source code repository before pushing to production. This way, they do not need to be precompiled separately on the production server upon each deployment.

As above, you can perform this step using

```
$ RAILS_ENV=production rails assets:precompile
```

Note the following caveats:

- If precompiled assets are available, they will be served — even if they no longer match the original (uncompiled) assets, *even on the development server.*

  To ensure that the development server always compiles assets on-the-fly (and thus always reflects the most recent state of the code), the development environment *must be configured to keep precompiled assets in a different location than production does.* Otherwise, any assets precompiled for use in production will clobber requests for them in development (*i.e.,* subsequent changes you make to assets will not be reflected in the browser).

  You can do this by adding the following line to `config/environments/development.rb`:

  ```
  config.assets.prefix = "/dev-assets"
  ```

- The asset precompile task in your deployment tool (*e.g.,* Capistrano) should be disabled.

- Any necessary compressors or minifiers must be available on your development system.

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 Assets Cache Store, and the manifest names used in the helpers are altered to include the SHA256 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, performs more poorly than the default, and is not recommended.

If you are deploying a production application to a system without any pre-existing JavaScript runtimes, you may want to add one to your `Gemfile`:

```
group :production do
  gem 'mini_racer'
end
```

CDNs

CDN stands for Content Delivery Network, they are primarily designed to cache assets all over the world so that when a browser requests the asset, a cached copy will be geographically close to that browser. If you are serving assets directly from your Rails server in production, the best practice is to use a CDN in front of your application.

A common pattern for using a CDN is to set your production application as the "origin" server. This means when a browser requests an asset from the CDN and there is a cache miss, it will grab the file from your server on the fly and then cache it. For example if you are running a Rails application on `example.com` and have a CDN configured at `mycdnsubdomain.fictional-cdn.com`, then when a request is made to `mycdnsubdomain.fictional-cdn.com/assets/smile.png`, the CDN will query your server once at `example.com/assets/smile.png` and cache the request. The next request to the CDN that comes in to the same URL will hit the cached copy. When the CDN can serve an asset directly the request never touches your Rails server. Since the assets from a CDN are geographically closer to the browser, the request is faster, and since your server doesn't need to spend time serving assets, it can focus on serving application code as fast as possible.

Set up a CDN to Serve Static Assets

To set up your CDN you have to have your application running in production on the internet at a publicly available URL, for example `example.com`. Next you'll need to sign up for a CDN service from a cloud hosting provider. When you do this you need to configure the "origin" of the CDN to point back at your website `example.com`, check your provider for documentation on configuring the origin server.

The CDN you provisioned should give you a custom subdomain for your application such as `mycdnsubdomain.fictional-cdn.com` (note fictional-cdn.com is not a valid CDN provider at the time of this writing). Now that you have configured your CDN server, you need to tell browsers to use your CDN to grab assets instead of your Rails server directly. You can do this by configuring Rails to set your CDN as the asset host instead of using a relative path. To set your asset host in Rails, you need to set `config.asset_host` in `config/environments/production.rb`:

```
config.asset_host = 'mycdnsubdomain.fictional-cdn.com'
```

NOTE: You only need to provide the "host", this is the subdomain and root domain, you do not need to specify a protocol or "scheme" such as `http://` or `https://`. When a web page is requested, the protocol in the link to your asset that is generated will match how the webpage is accessed by default.

You can also set this value through an environment variable to make running a staging copy of your site easier:

```
config.asset_host = ENV['CDN_HOST']
```

NOTE: You would need to set `CDN_HOST` on your server to `mycdnsubdomain .fictional-cdn.com` for this to work.

Once you have configured your server and your CDN, asset paths from helpers such as:

```
<%= asset_path('smile.png') %>
```

Will be rendered as full CDN URLs like `http://mycdnsubdomain.fictional-cdn.com/assets/smile.png` (digest omitted for readability).

If the CDN has a copy of `smile.png`, it will serve it to the browser, and your server doesn't even know it was requested. If the CDN does not have a copy, it will try to find it at the "origin" `example.com/assets/smile.png`, and then store it for future use.

If you want to serve only some assets from your CDN, you can use custom `:host` option your asset helper, which overwrites value set in `config.action_controller.asset_host`.

```
<%= asset_path 'image.png', host: 'mycdnsubdomain.fictional-cdn.com' %>
```

Customize CDN Caching Behavior

A CDN works by caching content. If the CDN has stale or bad content, then it is hurting rather than helping your application. The purpose of this section is to describe general caching behavior of most CDNs, your specific provider may behave slightly differently.

CDN Request Caching

While a CDN is described as being good for caching assets, in reality caches the entire request. This includes the body of the asset as well as any headers. The most important one being `Cache-Control` which tells the CDN (and web browsers) how to cache contents. This means that if someone requests an asset that does not exist `/assets/i-dont-exist.png` and your Rails application returns a 404, then your CDN will likely cache the 404 page if a valid `Cache-Control` header is present.

CDN Header Debugging

One way to check the headers are cached properly in your CDN is by using curl. You can request the headers from both your server and your CDN to verify they are the same:

```
$ curl -I http://www.example/assets/application-
d0e099e021c95eb0de3615fd1d8c4d83.css
HTTP/1.1 200 OK
Server: Cowboy
Date: Sun, 24 Aug 2014 20:27:50 GMT
Connection: keep-alive
Last-Modified: Thu, 08 May 2014 01:24:14 GMT
Content-Type: text/css
Cache-Control: public, max-age=2592000
Content-Length: 126560
Via: 1.1 vegur
```

Versus the CDN copy.

```
$ curl -I http://mycdnsubdomain.fictional-cdn.com/application-
d0e099e021c95eb0de3615fd1d8c4d83.css
HTTP/1.1 200 OK Server: Cowboy Last-
Modified: Thu, 08 May 2014 01:24:14 GMT Content-Type: text/css
Cache-Control:
public, max-age=2592000
Via: 1.1 vegur
Content-Length: 126560
Accept-Ranges:
bytes
Date: Sun, 24 Aug 2014 20:28:45 GMT
Via: 1.1 varnish
Age: 885814
Connection: keep-alive
X-Served-By: cache-dfw1828-DFW
X-Cache: HIT
X-Cache-Hits:
68
X-Timer: S1408912125.211638212,VS0,VE0
```

Check your CDN documentation for any additional information they may provide such as `X-Cache` or for any additional headers they may add.

CDNs and the Cache-Control Header

The cache control header is a W3C specification that describes how a request can be cached. When no CDN is used, a browser will use this information to cache contents. This is very helpful for assets that are not modified so that a browser does not need to re-download a website's CSS or JavaScript on every request. Generally we want our Rails server to tell our CDN (and browser) that the asset is "public", that means any cache can store the request. Also we commonly want to set `max-age` which is how long the cache will store the object before invalidating the cache. The `max-age` value is set to seconds with a maximum possible value of `31536000` which is one year. You can do this in your Rails application by setting

```
config.public_file_server.headers = {
  'Cache-Control' => 'public, max-age=31536000'
}
```

Now when your application serves an asset in production, the CDN will store the asset for up to a year. Since most CDNs also cache headers of the request, this `Cache-Control` will be passed along to all future browsers seeking this asset, the browser then knows that it can store this asset for a very long time before needing to re-request it.

CDNs and URL-based Cache Invalidation

Most CDNs will cache contents of an asset based on the complete URL. This means that a request to

```
http://mycdnsubdomain.fictional-cdn.com/assets/smile-123.png
```

Will be a completely different cache from

```
http://mycdnsubdomain.fictional-cdn.com/assets/smile.png
```

If you want to set far future `max-age` in your `Cache-Control` (and you do), then make sure when you change your assets that your cache is invalidated. For example when changing the smiley face in an image from yellow to blue, you want all visitors of your site to get the new blue face. When using a CDN with the Rails asset pipeline `config.assets.digest` is set to true by default so that each asset will have a different file name when it is changed. This way you don't have to ever manually invalidate any items in your cache. By using a different unique asset name instead, your users get the latest asset.

Customizing the Pipeline

CSS Compression

One of the options for compressing CSS is YUI. The YUI CSS compressor provides minification.

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

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

The other option for compressing CSS if you have the sass-rails gem installed is

```
config.assets.css_compressor = :sass
```

JavaScript Compression

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

Take the `terser` gem, for example. This gem wraps Terser (written for Node.js) in Ruby. It compresses your code by removing white space and comments, shortening local variable names, and performing other micro-optimizations such as changing `if` and `else` statements to ternary operators where possible.

The following line invokes `terser` for JavaScript compression.

```
config.assets.js_compressor = :terser
```

NOTE: You will need an ExecJS supported runtime in order to use `terser`. If you are using macOS or Windows you have a JavaScript runtime installed in your operating system.

GZipping your assets

By default, gzipped version of compiled assets will be generated, along with the non-gzipped version of assets. Gzipped assets help reduce the transmission of data over the wire. You can configure this by setting the `gzip` flag.

```
config.assets.gzip = false # disable gzipped assets generation
```

Refer to your web server's documentation for instructions on how to serve gzipped assets.

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
```

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 are updating an older project that didn't use the asset pipeline and already uses this path or you wish to use this path for a new resource.

X-Sendfile Headers

The X-Sendfile header is a directive to the web server to ignore the response from the application, and instead serve a specified file from disk. 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. Have a look at send_file on how to use this feature.

Apache and NGINX support this option, which can be 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
```

WARNING: 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 any other environments you define with production behavior (not `application.rb`).

TIP: For further details have a look at the docs of your production web server:

- Apache
- NGINX

Assets Cache Store

By default, Sprockets caches assets in `tmp/cache/assets` in development and production environments. This can be changed as follows:

```
config.assets.configure do |env|
  env.cache = ActiveSupport::Cache.lookup_store(:memory_store,
                                                { size: 32.megabytes })
end
```

To disable the assets cache store:

```
config.assets.configure do |env|
  env.cache = ActiveSupport::Cache.lookup_store(:null_store)
end
```

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. 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.

Making Your Library or Gem a Pre-Processor

Sprockets uses Processors, Transformers, Compressors, and Exporters to extend Sprockets functionality. Have a look at Extending Sprockets to learn more. Here we registered a preprocessor to add a comment to the end of text/css (`.css`) files.

```
module AddComment
  def self.call(input)
    { data: input[:data] + "/* Hello From my sprockets extension */" }
  end
end
```

Now that you have a module that modifies the input data, it's time to register it as a preprocessor for your mime type.

```
Sprockets.register_preprocessor 'text/css', AddComment
```