The Asset Pipeline

This guide covers the asset pipeline.

After reading this guide, you will know:

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

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.

Making the asset pipeline a core feature of Rails means that 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 keynote at RailsConf 2011.

The asset pipeline is enabled by default. It can be disabled in `config/application.rb` by putting this line inside the application class definition:

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

You can also disable the asset pipeline while creating a new application by passing the
`--skip-sprockets`
option.

```
rails new appname --skip-sprockets
```

You should use the defaults for all new applications unless you have a specific reason to avoid the asset pipeline.

1.1 Main Features

The first feature of the pipeline is to concatenate assets. This is important in a production environment, because it 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.

Rails 2.x introduced the ability to concatenate JavaScript and CSS assets by placing
`cache: true`
at the end of the
`javascript_include_tag`
and
`stylesheet_link_tag`
methods. But this technique has some limitations. For example, it cannot generate the caches in advance, and it is not able to transparently include assets provided by third-party libraries.

Starting with version 3.1, Rails defaults to concatenating 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 MD5 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 that 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.

1.2 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 that Rails 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`
could be renamed with an MD5 digest of its contents:

```
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 production and disabled for all other 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.

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. 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.

2.1 Controller Specific Assets

When you generate a scaffold or a controller, 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 you generate a

`ProjectsController`
, Rails will also add a new file at
`app/assets/javascripts/projects.js.coffee`
and another at
`app/assets/stylesheets/projects.css.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] %>`
. Ensure that you are not using the
`require_tree`
directive though, as this will result in your assets being included more than once.

> When using asset precompilation (the production default), 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. This will result in false positives during development as these files will work just fine since assets will be compiled on the fly. When running in production however, you will see 500 errors since live compilation is turned off by default. See **Precompiling Assets** for more information on how precompiling works.

You must have an 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.

You can also disable the generation of asset files when generating a controller by adding the following to your
`config/application.rb`
configuration:

```
config.generators
do

|g|

g.assets
false
end
```

2.2 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.

2.2.1 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:
`app/assets/images`
and the subdirectories
`javascripts`
and
`stylesheets`

in all three asset locations, 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/application.rb`
. For example:

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

Paths are traversed in the order that 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.

2.2.2 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/library_name`
, the file
`lib/assets/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 site's application manifest like so:

```
//= require library_name
```

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

2.3 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 web server.

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 **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 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"

%>
```

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.

2.3.1 CSS and ERB

The asset pipeline automatically evaluates ERB. This means that 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 `-%>`.

2.3.2 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")`
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"`

2.3.3 JavaScript/CoffeeScript and ERB

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

```
$('#logo').attr({

src: "<%= asset_path('logo.png') %>"
});
```

This writes the path to the particular asset being referenced.

Similarly, you can use the
`asset_path`
helper in CoffeeScript files with
`erb`
extension (e.g.,
`application.js.coffee.erb`

):

```
$('#logo').attr src: "<%= asset_path('logo.png') %>"
```

2.4 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 true). 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 the file size enabling the browser to download it faster.

For example, a new Rails application includes a default `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.

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 .
*/
```

The directives that work in the 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. If
`require_self`
is called more than once, only the last call is respected.

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

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
*/
```

2.5 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 an
`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`
gems and then sent back to the browser as JavaScript and CSS respectively.

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.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, then CoffeeScript, and served as JavaScript.

Keep in mind that the order of these preprocessors is important. For example, if you called your JavaScript file
`app/assets/javascripts/projects.js.erb.coffee`
then it would be 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
`app/assets/javascripts/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
`config/environments/development.rb`
to include:

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

```
false
```

When debug mode is off, Sprockets concatenates and runs the necessary preprocessors on all files. With debug mode turned off the manifest above would generate instead:

```
<
script

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

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 have changed between requests, the server responds with a new compiled file.

Debug mode can also be enabled in the Rails helper methods:

```
<%=

stylesheet_link_tag
"application"
, debug:
true

%>
<%=

javascript_include_tag
```

```
"application"
, debug:
true

%>
```

The
`:debug`
option is redundant if debug mode is on.

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 Rails assumes 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"
></
script
>
<
link

href
=
"/assets/application-
4dd5b109ee3439da54f5bdfd78a80473.css"

media
=
"screen"

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 setting of `config.assets.digest` setting in Rails (which defaults to `true` for production and

`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 the disk.

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

You can call this task 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 rake task is:

```
$ RAILS_ENV=production bundle exec rake
assets:precompile
```

For faster asset precompiles, you can partially load your application by setting
`config.assets.initialize_on_precompile`
to false in
`config/application.rb`
, though in that case templates cannot see application objects or methods. **Heroku requires this to be false.**

> If you set
> `config.assets.initialize_on_precompile`
> to false, be sure to test
> `rake assets:precompile`

locally before deploying. It may expose bugs where your assets reference application objects or methods, since those are still in scope in development mode regardless of the value of this flag. Changing this flag also affects engines. Engines can define assets for precompilation as well. Since the complete environment is not loaded, engines (or other gems) will not be loaded, which can cause missing assets.

Capistrano (v2.8.0 and above) includes 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.

If you are precompiling your assets locally, you can use `bundle install --without assets` on the server to avoid installing the assets gems (the gems in the assets group in the Gemfile).

The default matcher for compiling files includes `application.js`, `application.css` and all non-JS/CSS files (this will include all image assets automatically):

```
[
Proc
.
new

{ |path| !%w(.js .css).include?(
File
.extname(path)) }, /application.(css|js)$/ ]
```

The matcher (and other members of the precompile array; see below) is applied to final compiled file names. This means that 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/application.rb`
:

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

Or you can opt to precompile all assets with something like this:

```
# config/application.rb
config.assets.precompile <<
Proc
.
new
```

```
do

|path|

if

path =~ /\.(css|js)\z/

full_path =
Rails.application.assets.resolve(path).to_path

app_assets_path = Rails.root.join(
'app'
,
'assets'
).to_path

if

full_path.starts_with? app_assets_path

puts
"including asset: "

+ full_path

true

else

puts
"excluding asset: "

+ full_path

false

end

else

false

end
end
```

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 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 to avoid handing the mapping requests back to Sprockets. A typical manifest file looks like:

```
---
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).

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

4.1.1 Far-future Expires Header

Precompiled assets exist on the filesystem 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 them.

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 "";

break;
}
```

4.1.2 GZip Compression

When files are precompiled, Sprockets also creates a **gzipped** (.gz) version of your assets. Web servers are typically configured to use a moderate compression ratio as a compromise, but since precompilation happens once, Sprockets uses the maximum compression ratio, thus reducing the size of the data transfer to the minimum. On the other hand, web servers can be configured to serve compressed content directly from disk, rather than deflating non-compressed files themselves.

Nginx is able to do this automatically enabling `gzip_static`:

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

This directive is available if the core module that provides this feature was compiled with the web server. Ubuntu packages, even `nginx-light` have the module compiled. Otherwise, you may need to perform a manual compilation:

```
./configure --with-http_gzip_static_module
```

If you're compiling nginx with Phusion Passenger you'll need to pass that option when prompted.

A robust configuration for Apache is possible but tricky; please Google around. (Or help update this Guide if you have a good example configuration for Apache.)

4.2 Local Precompilation

There are several reasons why you might want to precompile your assets locally. Among them are:

You may not have write access to your production file system.
You may be deploying to more than one server, and want to avoid the duplication of work.

You may be doing frequent deploys that do not include asset changes.

Local compilation allows you to commit the compiled files into source control, and deploy as normal.

There are two caveats:

You must not run the Capistrano deployment task that precompiles assets.
You must change the following two application configuration settings.

In `config/environments/development.rb`, place the following line:

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

You will also need this in application.rb:

```
config.assets.initialize_on_precompile = false
```

The `prefix` change makes Rails use a different URL for serving assets in development mode, and pass all requests to Sprockets. The prefix is still set to `/assets` in the production environment. Without this change, the application would serve the precompiled assets from `public/assets` in development, and you would not see any local changes until you compile assets again.

The `initialize_on_precompile` change tells the precompile task to run without invoking Rails. This is because the

precompile task runs in production mode by default, and will attempt to connect to your specified production database. Please note that you cannot have code in pipeline files that relies on Rails resources (such as the database) when compiling locally with this option.

You will also need to ensure that any compressors or minifiers are available on your development system.

In practice, this will allow you to precompile locally, have those files in your working tree, and commit those files to source control when needed. Development mode will work as expected.

4.3 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, 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
'therubyracer'
end
```

4.4 CDNs

If your assets are being served by a CDN, ensure they don't stick around in your cache forever. This can cause problems. If you use `config.action_controller.perform_caching = true`, Rack::Cache will use `Rails.cache` to store assets. This can cause your cache to fill up quickly.

Every cache is different, so evaluate how your CDN handles caching and make sure that it plays nicely with the pipeline. You may find quirks related to your specific set up, you may not. The defaults nginx uses, for example, should give you no problems when used as an HTTP cache.

5 Customizing the Pipeline

5.1 CSS Compression

There is currently one option for compressing CSS, 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
`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. It also includes other optimizations such as changing your
`if`
and
`else`
statements to ternary operators where possible.

The following line invokes
`uglifier`
for JavaScript compression.

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

Note that
`config.assets.compress`
must be set to
`true`
to enable JavaScript compression

You will need an **ExecJS** supported runtime in order to use
`uglifier`
. If you are using Mac OS X or Windows you have a JavaScript runtime installed in your operating system. Check the **ExecJS** documentation for information on all of the 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 are updating an older project that didn't use the asset pipeline and 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 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.

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
```

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`

).

6 Assets Cache Store

The default Rails cache store will be used by Sprockets to cache assets in development and production. This can be changed by setting `config.assets.cache_store`.

```
config.assets.cache_store =
:memory_store
```

The options accepted by the assets cache store are the same as the application's cache store.

```
config.assets.cache_store =
:memory_store
, { size:
32
.megabytes }
```

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

As Sprockets uses **Tilt** as a generic interface to different templating engines, your gem should just implement the Tilt template protocol. Normally, you would subclass `Tilt::Template`
and reimplement
`evaluate`
method to return final output. Template source is stored at
`@code`
. Have a look at
`Tilt::Template`
sources to learn more.

```
module

BangBang

class

Template < ::Tilt::Template

# Adds a "!" to original template.

def

evaluate(scope, locals, &block)

"#{@code}!"

end

end
end
```

Now that you have a
`Template`
class, it's time to associate it with an extension for template files:

```
Sprockets.register_engine
'.bang'
, BangBang::Template
```

9 Upgrading from Old Versions of Rails

There are a few issues when upgrading. The first is moving the files from `public/` to the new locations. See **Asset Organization** above for guidance on the correct locations for different file types.

Next will be avoiding duplicate JavaScript files. Since jQuery is the default JavaScript library from Rails 3.1 onwards, you don't need to copy `jquery.js` into `app/assets` and it will be included automatically.

The third 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

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

You should not need to change
`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', "~> 3.2.3"
  gem 'coffee-rails', "~> 3.2.1"
  gem 'uglifier'
end
```

If you use the `assets` group with Bundler, please make sure that your `config/application.rb` has the following Bundler require statement:

```
# If you precompile assets before deploying to
```

```
production, use this line
Bundler.require *Rails.groups(
:assets

=> %w(development test))
# If you want your assets lazily compiled in
production, use this line
# Bundler.require(:default, :assets, Rails.env)
```

Instead of the generated version:

```
# Require the gems listed in Gemfile, including any
gems
# you've limited to :test, :development, or
:production.
Bundler.require(
:default
, Rails.env)
```

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 the **rails** repository and open a new pull request. You can also ask for commit rights on **docrails** if you plan to submit several patches. Commits are reviewed, but that happens after you've submitted your contribution. This repository 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.

If for whatever reason you spot something to fix but cannot patch it yourself, please **open an issue**.

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