The Asset Pipeline

Introduction

You've learned about Models, Views, and Controllers. That's the nuts and bolts, but we've got plenty of neat stuff to cover which makes Rails much more useful to you. In this lesson, we'll talk about the Asset Pipeline and a few other topics that don't necessarily fit well in other lessons but are important to cover nonetheless.

Points to Ponder

Look through these now and then use them to test yourself after doing the assignment

- What is the "Asset Pipeline"?
- What are "Manifest Files"?
- Why would you namespace your stylesheets?
- What does it mean to "Escape" HTML?

The Asset Pipeline

Assets in your application are additional files that get called by the browser after your initial gob of HTML is received. They include things like CSS stylesheets, Javascript files, images, videos etc... basically anything that requires an additional request to grab it.

Often times, it's easiest to organize your code for development purposes into many different files so you can keep track of them better. But if the browser has to grab a dozen different CSS files, each one of those requests is going to slow things down. Too many requests and you've harpooned your user's experience with your application.

A similar organizational issue has to do with storing things like images. It's easier to keep them separate and separated in your directory but you want them to be really simple to link to so your image tags are robust.

Rails' solution to these problems is to flatten everything out and mash all your asset files together into one big one for each filetype (called "concatenation"). The process used to do this is the Asset Pipeline. For your CSS files, this means that Rails will take all the individual `.css` files and just stack them on top of each other in one giant asset file. It will then run an "uglifier" or "minifier" program on the file to remove extraneous spaces and make everything nice and small for shipping to the browser.

Javascript files are the same -- all of them get smooshed together and then uglified before being shipped to the browser as one single file. It's better to have one slightly larger file than to make several full HTTP requests.

Manifest Files

Rails needs to know which files to include in that giant blob, so it uses so-called "manifest" files to determine this. Your javascript manifest file will be `app/assets/javascripts/application.js`. It looks commented out, but the lines starting with `//=` tell Rails which files to go find and include. The comments in the file are pretty useful -- they say:

```
// This is a manifest file that'll be compiled into application.js, which will include all
the files
// listed below.
//
// Any JavaScript/Coffee file within this directory, lib/assets/javascripts,
vendor/assets/javascripts,
// or vendor/assets/javascripts of plugins, if any, can be referenced here using a relative
path.
//
// It's not advisable to add code directly here, but if you do, it'll appear at the bottom
of the
// compiled file.
//
```

```
// Read Sprockets README (https://github.com/sstephenson/sprockets#sprockets-directives)
for details
// about supported directives.
//
//= require jquery
//= require jquery_ujs
//= require turbolinks
//= require_tree .
```

The `require_tree` helper method just grabs everything in the current directory.

Your stylesheet manifest file operates on the same principle -- it's available at `app/assets/stylesheets/application.css.scss`:

```
/*
 * This is a manifest file that'll be compiled into application.css, which will include all
the files
 * listed below.
 *
 * Any CSS and SCSS file within this directory, lib/assets/stylesheets,
vendor/assets/stylesheets,
 * or vendor/assets/stylesheets of plugins, if any, can be referenced here using a relative
path.
 *
 * You're free to add application-wide styles to this file and they'll appear at the top of
the
 * compiled file, but it's generally better to create a new file per style scope.
 *
 *= require_self
 *= require_tree .
 */
```

Again, you see the `require_tree` helper method which brings in all CSS files in the current directory.

Reading the comments, you can also see that a couple other directories are assumed to be a "local directory" and can be easily referenced as well, like the `lib/assets` and `vendor/assets` files. Sometimes, if you start using a new gem (like some of the Twitter-Bootstrap gems) you manually need

to add the new bootstrap stylesheets and javascripts to the manifest files to make sure your application actually includes them in the final output.

The Output

Speaking of final output, what is it? Well, Rails mashes all the specified files together and creates a new one called something like: `application-1fc71ddbb281c144b2ee4af31cf0e308.js`. That nonsensical string of characters is meant to differentiate between files if you end up making any changes. If they were just called something like `application.js`, then your browser would cache it and never know to ask you for the latest version because it's always named the same thing.

But wait, how does the browser know to go looking for `application-1fc71ddbb281c144b2ee4af31cf0e308.js`? That's what the asset tags we talked about in the previous lesson are useful for. When you write in your application layout `<%= javascript_include_tag "application" %>`, Rails automatically knows which filename to request to get all your javascripts properly imported.

Taking This Into Account in Your Code: Namespacing

This sounds great and wonderful and faster for your application, but does it change anything you do? Oftentimes you can just forget about the manifest files and keep coding along your way. For your initial applications, you might keep all the styles and javascripts in one file anyway, so it's not going to change anything on your end.

It becomes important when, for instance, you have a ton of different pages that likely want to use different stylesheets. What if you want the `.container` class to do slightly different things in your user login pages versus the checkout pages? With the asset pipeline, Rails will jam all those files together and you can't be sure which `.container` styles are going to override which others.

In theory, you could override styles from your stylesheets stored at `app/assets/stylesheets` with either inline styles or `<style>` tags, but that gets really messy and totally defeats the purpose of having external stylesheets for keeping code clean.

Let's also assume that you really like using `.container` classes to keep your `<div>` elements neatly organized. The solution is to use "Namespacing", which means that you basically nest your classes beneath some sort of variable or function name. This is actually a principle that gets used a LOT, so it's important to understand it. You'll see it with stylesheets, javascripts, modules of code and more.

The basic idea is to be able to say "all this code/css/whatever inside here only belongs to XYZ". You sort of fence it off. It's best explained with an example:

```
# app/views/users/show.html.erb
<div class="user">
  <div class="container">
    <!-- a bunch of code for displaying the user -->
  </div>
</div>
```

Now this container and all the code inside of it is also within the `.user` class. So we can set up our stylesheet to specifically address the `.container` class that's inside a `.user` class:

```
# app/assets/stylesheets/user.css.scss
# Note: I'm not going to use SCSS code because we haven't covered it yet
.user .container{
  // style stuff
}
```

This is good because we're now specifically targeting containers used by User pages.

The same principle applies to javascript, though I won't cover it here because that's material for a later course.

So any time you want to make only a portion of your stylesheets or javascript code available to a specific set of views, try namespacing it.

Rails in Development

The asset pipeline functions a bit differently in development mode. If you look at your Rails server output when you're working with a webpage in the local environment, it actually sends out a whole bunch of stylesheets and the like. This is just to give you the ability to debug easier.

Images

For images, the asset pipeline keeps them in the `/assets` directory unless you've made your own subdirectories. Use `image_tag`'s to avoid confusion, e.g. `<%= image_tag "fuzzy_slippers.jpg" %>`.

Preprocessors

Remember the preprocessors we talked about in the previous lesson on Views? Filetypes like ERB and SASS and HAML and Coffeescript all get preprocessed as part of the pipeline.

Un-Escaping HTML

Let's say you're building a blog and you want to be able to write posts that include HTML code. If you just write something like `this is the <strong>BODY</strong> of my post` and then try to display it in a view later, the `<strong>` tags will just be regular text... they will literally say ''. That's called "escaping" the characters.

To get your views to actually render HTML as HTML, you need to let Rails know that the code is safe to run. Otherwise, it's easy for a malicious attacker to inject code like `<script>` tags that cause major issues when you try to render them.

To tell Rails a string is safe, just use the method `raw` in your view template, for example:

```
<%= raw "<p>hello world!</p>" %>   <!-- this will create real <p> tags -->
```

If you don't want to rely on Rails' native behavior and would like to make absolutely sure the HTML does not get run, use the `CGI` class's `escapeHTML` method, e.g.

```
    CGI::escapeHTML('usage: foo "bar" <baz>')
    # => "Usage: foo &quot;bar&quot; &lt;baz&gt;"
```

Your Assignment

Some necessary and straightforward reading on the Asset Pipeline:

1. Read Rails Guides on the Asset Pipeline sections 1 to 3.

Conclusion

The Asset Pipeline isn't something that you often think about, especially when just building little toy apps, but it becomes important to understand as soon as you want to deploy your application (because you'll need to take it into account, which we'll talk about in that lesson later) or work with anything but the vanilla asset structure.

Additional Resources

This section contains helpful links to other content. It isn't required, so consider it supplemental for if you need to dive deeper into something

- Ryan Bates' asset pipeline Railscast
- Stack Overflow on Escaping HTML in Rails