Episode 266

HTTP Streaming

In this episode we'll continue our series on some of the features of the first Rails 3.1 beta release. This time we'll take a look at HTTP streaming. This subject has been covered fairly well in a post on the Ruby on Rails Weblog[1] and it's worth taking a look at that article first. We'll take a look here at getting it set up in a Rails application and covering some of the potential problems you might have when using it.

To demonstrate HTTP streaming we'll use the simple ToDo application that we wrote in the last episode. We enable streaming in an application's controllers, either on a per-action or a per-controller basis; it isn't enabled by default.

To enable streaming for an action we use `render`, passing it the `:stream => true` option.

/app/controllers/projects_controller.rb

```
def index
  @projects = Project.all
  render :stream => true
end
```

To enable streaming across the whole controller we use the class method `stream`.

/app/controllers/projects_controller.rb

```
class ProjectsController < ApplicationController
  stream

  def index
    # rest of controller code
  end
end
```

The default webserver in Rails 3, WEBrick, doesn't support streaming so to see it in action we'll need to switch to another that does, such as Unicorn.

[1] http://weblog.rubyonrails.org/2011/4/18/why-http-streaming

The Unicorn gem is listed in the application's `Gemfile` but it's commented out by default. To use it we uncomment this line and then run `bundle` to install it.

/Gemfile

```
# Use unicorn as the web server
gem 'unicorn'
```

We need to configure Unicorn to get it to use streaming. Rails has some internal documentation on streaming[2] that includes information on how to do this. All we need to do is create a configuration file that we'll put in the `config` directory.

/config/unicorn.rb

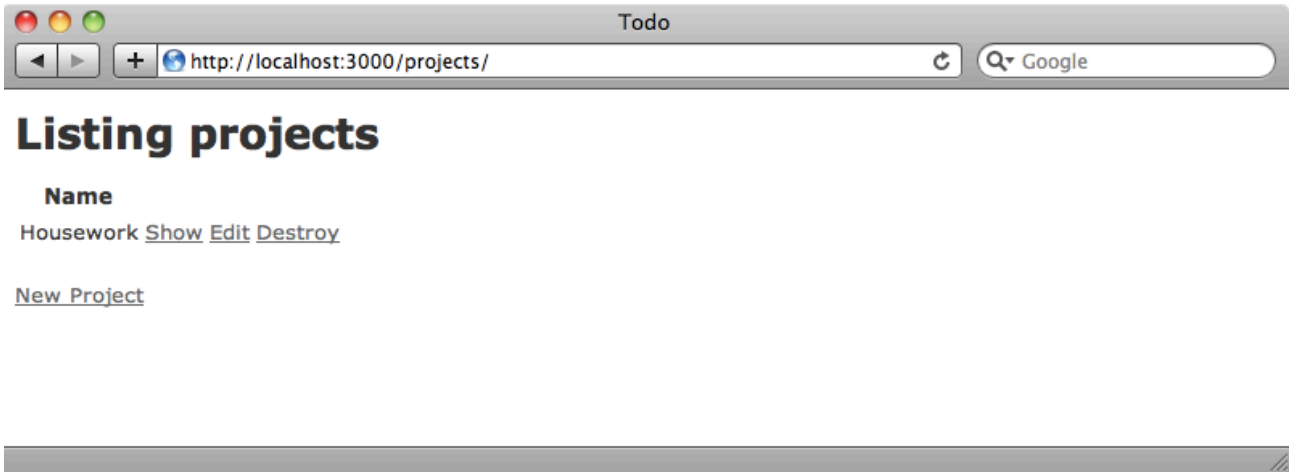
```
listen 3000, :tcp_nopush => false
```

We can then start up Unicorn and point it to that configuration file.

```
$ unicorn_rails --config-file config/unicorn.rb
```

If you have trouble running this command make sure that you ran `bundle` after uncommenting the `gem 'unicorn'` line in the `Gemfile`. If it still doesn't work try prefixing the command with `bundle exec`.

We can now open our application in a browser on port 3000, just as we would if we were running it with the default Rails server.



[2] https://github.com/rails/rails/blob/master/actionpack/lib/action_controller/metal/streaming.rb

Simulating Streaming

The page above now uses streaming but we can't really see its effects. We'll need to simulate something that takes a few seconds to run in the view layer so that we can see the response being returned in chunks. The simplest way to do this is to add a `sleep` call at the top of one of the views.

/app/views/projects/index.html.erb

```
<% sleep 5 %>
<h1>Listing projects</h1>

<table>
  <tr>
    <th>Name</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @projects.each do |project| %>
  <tr>
    <td><%= project.name %></td>
    <td><%= link_to 'Show', project %></td>
    <td><%= link_to 'Edit', edit_project_path(project) %></td>
    <td><%= link_to 'Destroy', project, confirm: 'Are you sure?',
method: :delete %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New Project', new_project_path %>
```

We can see the chunked response by fetching this page with `curl`. If we use it to get the application's home page we'll get part of the response returned immediately.

```
$ curl -i localhost:3000
HTTP/1.1 200 OK
Date: Wed, 18 May 2011 08:18:56 GMT
Status: 200 OK
Connection: close
Cache-control: no-cache
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
X-UA-Compatible: IE=Edge
X-Runtime: 0.023745

<!DOCTYPE html>
<html>
<head>
  <title>Todo</title>
  <link href="/assets/application.css" media="screen"
rel="stylesheet" type="text/css" />
  <script src="/assets/application.js" type="text/javascript"></
script>
  <meta content="authenticity_token" name="csrf-param" />
<meta content="0eBxvhbMH6HA8ocRLw06uNnmh7zqWo5dGSeFIA8sfj8="
name="csrf-token" />
</head>
<body>
```

There will then be a few seconds' delay before the rest is returned. Note in the header information above that `Transfer-Encoding` is set to `chunked` and that there is no `Content-Length` header as the server can't know what size the full page will be when the first part of the response is sent.

```
<h1>Listing projects</h1>

<table>
  <tr>
    <th>Name</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

  <tr>
    <td>Housework</td>
    <td><a href="/projects/1">Show</a></td>
    <td><a href="/projects/1/edit">Edit</a></td>
    <td><a href="/projects/1" data-confirm="Are you sure?" data-
method="delete" rel="nofollow">Destroy</a></td>
  </tr>
</table>

<br />

<a href="/projects/new">New Project</a>


</body>
</html>
```

Without streaming enabled there would be a five-second delay before we saw any response from the server. Having the `head` section of the page returned immediately allows the web browser to start processing this part of the page and to load any JavaScript and CSS files that are referenced there while it waits for the rest of the main page to be returned from the server.

If we view this page in a browser it will still take at least five seconds to before we see anything but in the background it will be fetching the application's JavaScript and CSS files while it waits for the Rails app to send the rest of the main page.

To get the most from streaming we need to move as much processing as we can into the view layer so that the server can start streaming the page as soon as possible. In our `index` action we're using `Project.all` to fetch the projects to display on the

page. The server cannot begin to stream this page until this command has run and so we should replace it with something like `scoped`, which uses lazy loading, so that the database call is not made until the view layer starts iterating over the collection of projects.

/app/controllers/projects_controller.rb

```
def index
  @projects = Project.scoped
end
```

Potential Issues With Streaming

So far streaming sounds great but there are a few drawbacks to it that we'll need to be aware of before deciding to use it. The first is that the layout and template rendering is reversed. Normally in a Rails request the action's template is rendered first, followed by the layout. In a streaming request we need to render the layout's content as soon as possible as this usually contains the `head` section. The action's template won't be rendered until we reach the `yield` command in the layout.

This means that if we try to do something like setting an instance variable in the template we won't be access that variable in the layout as the template won't have yet been rendered. For example let's try to try set a `@title` instance variable in the `index` action.

/app/views/projects/index.html.erb

```
<% @title = "Projects "%>
<% sleep 5 %>
<h1>Listing projects</h1>

<table>
<!-- Rest of file omitted. -->
```

We'll then try to use that variable in the layout:

/app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
<head>
  <title><%= @title %></title>
  <%= stylesheet_link_tag    "application" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body>

<%= yield %>

</body>
</html>
```

If we fetch the page with `curl` and look at the head section the `title` element is empty. The layout is rendered before the template and so the `@title` variable won't have been set when it's read in the layout file. This isn't the case for non-streaming requests as the template is read first.

```
<head>
  <title></title>
  <link href="/assets/application.css" media="screen" ↵
    rel="stylesheet" type="text/css" />
  <script src="/assets/application.js" type="text/javascript"> ↵
  </script>
  <meta content="authenticity_token" name="csrf-param" />
  <meta content="x0CtIY+0vEbfkh6gohZp/WdOd0ZanobQHZT8+HUC/OE=" ↵
   name="csrf-token" />
</head>
```

Of course the correct way to pass information from a template to the layout is to use `content_for` but this won't work either. If we try it and then use `curl` to view the page the output will stop just before the `title` element.

This fails because of how `content_for` works. If we have multiple calls to `content_for` for the same item Rails concatenates them together. Rails cannot know, therefore, that when it sees `content_for :title` at the top that there aren't more calls to it further down the page.

Rails 3.1 provides a new method called `provide` that behaves exactly like `content_for` except that it doesn't concatenate values. We can use this to set the page's title.

/app/views/projects/index.html.erb

```
<% provide :title, "Projects" %>
```

In the layout we use `yield`, just as we would with `content_for`.

/app/views/layouts/application.html.erb

```
<title><%= yield :title %></title>
```

If we view the page now the `title` element contains the title that we set.

```
<!DOCTYPE html>
<html>
<head>
  <title>Projects</title>
  <link href="/assets/application.css" media="screen" ↵
    rel="stylesheet" type="text/css" />
  <script src="/assets/application.js" type="text/javascript"> ↵
  </script>
  <meta content="authenticity_token" name="csrf-param" />
  <meta content="NzdFt92dDBSXRRgFR0pRZRizirN87Qb5CVdqgGEAvTU=" ↵
    name="csrf-token" />
</head>
<!-- rest of page -->
```

Another problem we need to be wary of when using streaming is what happens when as exception is raised. We can show this by adding a call to a non-existent method in the `index` template.

/app/views/projects/index.html.erb

```
<% provide :title, "Projects" %>

<% fall_over %>

<% sleep 5 %>
<h1>Listing projects</h1>
<!-- rest of page -->
```

If we look at the page now we'll see some interesting output.

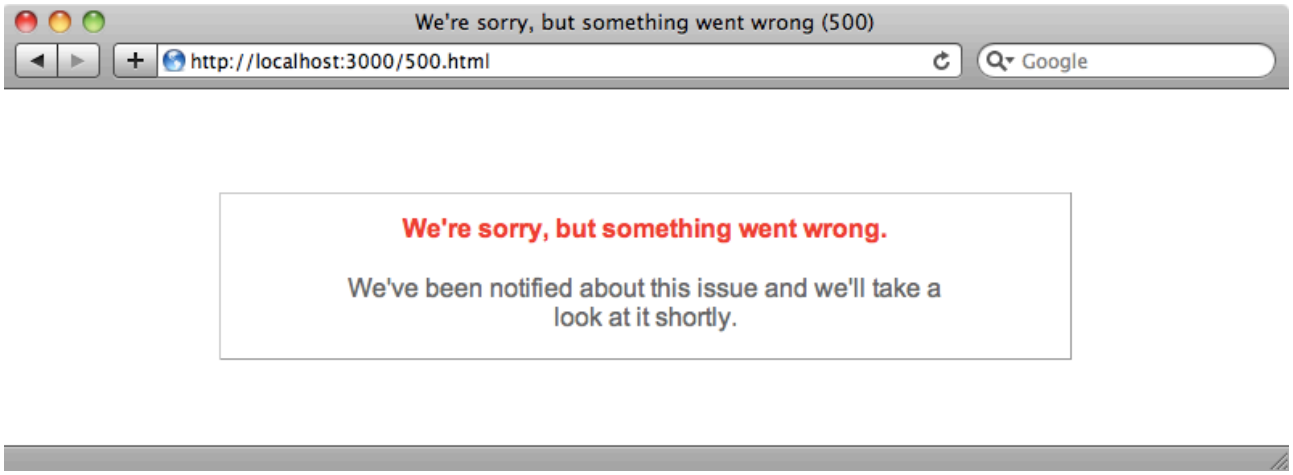
```
$ curl -i http://localhost:3000/

(header information omitted)

<!DOCTYPE html>
<html>
<head>
  <title>Projects</title>
  <link href="/assets/application.css" media="screen"
rel="stylesheet" type="text/css" />
  <script src="/assets/application.js" type="text/javascript"></
script>
  <meta content="authenticity_token" name="csrf-param" />
<meta content="rhFAQuK2s5Rxi6jjC3jA12k07GjD75VeWlbsyf47bLc="
name="csrf-token" />
</head>
<body>

"><script type="text/javascript">window.location = "/500.html"</
script></html>
```

When an exception is raised Rails sends a script element back to the browser containing a line of JavaScript that redirects the browser to the `500.html` page. If we view this page in the browser we'll be redirected to the standard error page that Rails applications show when running in production mode.



This means that we don't get the debugging information that we usually see when an error is thrown in development mode and we have to look for the error in the development log.

Setting session and cookie information inside the template also won't work if we're using streaming. If we try setting a session variable in the template Rails will have already sent the header information to the browser so there's no way we can send further header information in the template. This also applies to cookies and to `flash` messages, which use the session. Session and cookie information needs to be set in the controller when we're using streaming.

Two final potential issues are that Rails' HTTP streaming feature makes use of Ruby fibres so we'll need to be running Ruby 1.9 to use it. Also, streaming is incompatible with some middleware. If the middleware modifies the response then it will not work with streaming. You can probably understand now why streaming isn't enabled by default. There are a number of potential problems that we need to be aware of when using streaming and so it's best to restrict its use to only those pages in which you need as much performance as possible.

Even with all of these problems streaming is well worth considering as it can improve the end-user experience, especially on pages that include a number of JavaScript and CSS files as the browser can get on with loading and processing these files as quickly as possible.