COM1001 Introduction to Software Engineering, Spring Semester 2017

Practical 1: Sinatra Part 1

In this lab session we're going to be getting grips with Sinatra. As covered in the lecture, Sinatra is a simple web framework for Ruby, which will allow you to build your web application for the group project. Work through the following examples, then attempt the exercises at the end.

1 Preferred Ruby Environment

In this semester, you can use Ruby wherever you like in whatever environment you like. That could be on your own machine, using the Ruby installation on the CICS machines, or through Codio.

Our preference, however, is that you use Codio. We won't be able to spend lab time helping you get your personal machine working with Ruby, or figuring out any problems you're experiencing installing gems on it — sorry.

Our use of Codio will be limited this semester in that we'll just be using the virtual boxes — we won't be doing any formal units or any of that stuff. To get a box going, go to `https://codio.com/p/login/` and login with your university email address and password. Click "My Projects" in the "BUILD" menu to the left of the page. Then, click the blue "New Project" button in the top right. Click "Ruby + rails" as your starting point, name your project whatever you like, but ensure the visibility remains "Private". Then click "Create", and you're ready to go. When logging back into Codio later, you should always see your project listed under "My Projects".

2 Installing the Gems Needed

In order to get started, we need some gems.

First install the `sinatra` gem by going to the terminal ("Tools" → "Terminal" in Codio). On Codio, you'll need to prefix the usual command with "`sudo`" to get the necessary permissions (you may not need this on your own machine):

```
sudo gem install sinatra
```

We're going to serve up web pages using a different web server to that which Ruby/Sinatra uses by default — it's called Thin. To use Thin, we'll need to install the "`thin`" gem:

```
sudo gem install thin
```

We're ready to go!

3 Hello World!

The first thing any venerable programmer does when using a new programming language or framework is create a "Hello World" program so let's do this. Open a new Ruby file, call it "`hello_world.rb`" or some such, and copy and paste the following code:

```
require 'sinatra'
set :bind, '0.0.0.0' # Only needed if you're running from Codio

get '/' do
  'Hello World!'
end
```

Now run the code from the command line (i.e., "`ruby hello_world.rb`", or whatever you called your file.

Notice it does not terminate, instead we get a message that looks something like this:

```
== Sinatra (v1.4.8) has taken the stage on 4567 for development with backup from Thin
Thin web server (v1.7.0 codename Dunder Mifflin)
Maximum connections set to 1024
Listening on 0:0:0:0:4567, CTRL+C to stop
```

Ruby has recognised that we're developing a Sinatra web app and has instead directed activity to a locally-running web server (called "Thin").

This means we can open a web browser and see the results by loading up the output of our code. The URL you need to visit is of the form "`http://[DOMAIN]:[PORT]`". `[PORT]` is given by the first line of the Sinatra output — it's the number following "Sinatra ... has taken the stage on ...", so "4567" in the above.

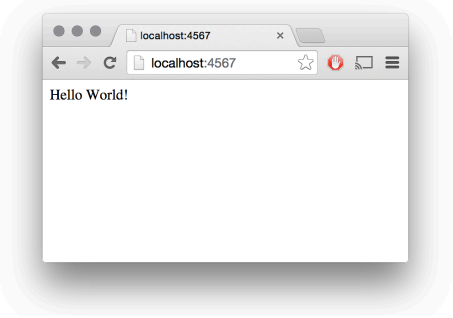
If you're running things on your own machine, you can replace `[DOMAIN]` with "`localhost`", meaning you can go to the URL "`http://localhost:4567`", assuming 4567 is your port number.

Like most of the class, however, you'll be using Codio, and you'll need to find your Codio box domain. This is given in the terminal output that Codio spits out when you open it for the first time:

```
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 4.4.0-62-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
 * Welcome to the Codio Terminal!
 * README: https://codio.com/docs/ide/boxes/
 * Your Codio Box domain is: gemini-special.codio.io[:1024-9999]

Last login: Sun Feb  5 21:56:08 2017 from 192.168.10.79

codio@gemini-special:~/workspace$_
```

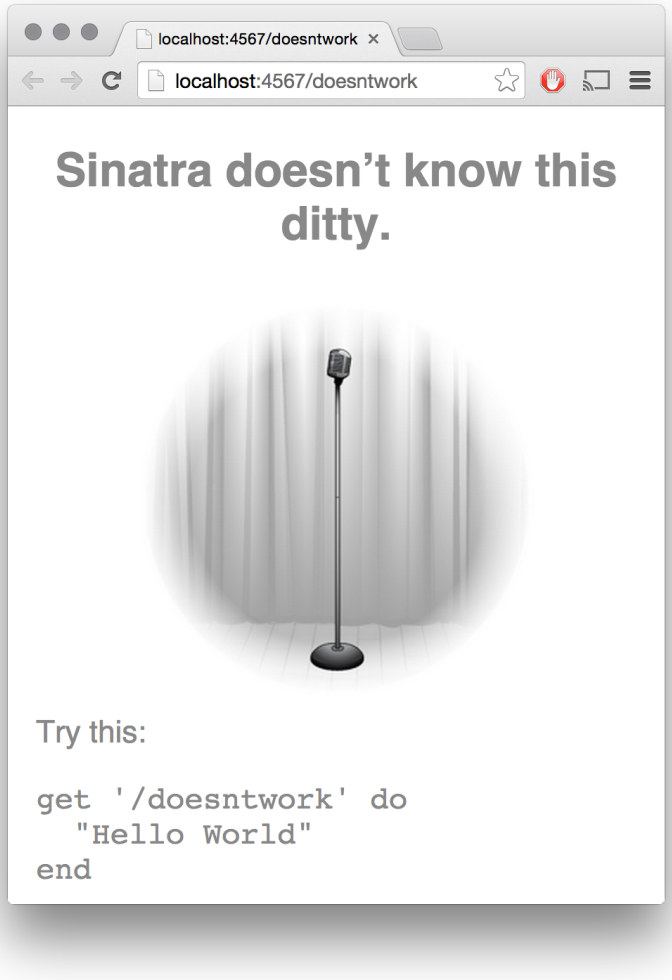
See the fifth line of output, which reveals your box domain as "`[BOX].codio.io`", where `[BOX]` is the name of your Codio box. The URL you need to visit is "`http://[BOX].codio.io:PORT`". So for me, in the above, my box is "`gemini-special`" and the URL I'd need to visit is `http://gemini-special.codio.io:4567`.

Phew! Assuming you got everything in the right place, loading your URL in your browser should give you an empty page but for your simple "Hello World!" message:



Let's dissect the code for that Controller a second. Basically what we have defined is a so-called *route*. A route maps an incoming path (everything after the port number (e.g., "4567"), which is simply '/' in this case) to some code that needs to be run when a URL is requested.

Note what happens when you request a URL for which no route exists. Try the following route in your browser — `http://[DOMAIN]:[PORT]/doesntwork`. You should see something like this, if you're using a browser like Chrome:



Note that Internet Explorer likes to use its own custom error pages, so you might not see it if you're using that browser. This is Sinatra's "404" message. It's telling us that there's no route defined for the URL that we're requested, and being the helpful chap that he is, is telling us the code we need to add in order to get the route working. Let's do just that. Stop the Ruby app (CTRL + C in the terminal), copy in the code to your Ruby file and run it. If you request your browser window, the error message should disappear and the words "Hello World" should appear again.

Don't Forget to Terminate the Current Running Session

Before running any further examples, **always ensure you've terminated any previous Sinatra session that you have running**, like we just did. This is because you cannot have more than one web server process listening to the same port at one time — you'll either get an error, or your browser will still be interacting with the previous version of your application.

One way to avoid having to restart the web server every time you make a change to a controller is to use the reloading capability of the `sinatra-contrib` gem. Install the gem and include the line `require 'sinatra/reloader'` after the `require 'sinatra'` line.

4 Multiple Routes

Let's do something a bit cooler with multiple routes:

```
require 'sinatra'
set :bind, '0.0.0.0' # Only needed if you're running from Codio

get '/' do
  '<a href="page2">Click me</a>'
end

get '/page2' do
  '<a href=".">Go back</a>'
end
```

Run the above example and go and view `http://[DOMAIN]:[PORT]/`. You should see a link — the "/" route now outputs some HTML. The link takes us to our other defined route ("`/page2`", at the URL `http://[DOMAIN]:[PORT]/page2`).

5 Room with a View

As covered in the lecture mixing HTML into our Controllers is not a good idea. Most web frameworks separate the underlying logic of an application from the *presentation* components of the app, more commonly referred to as *Views*. Views in Sinatra are essentially HTML templates — skeleton HTML with "gaps" to be filled by the values of variables.

Create a new Ruby file called "`using_a_view.rb`". To use views, we need to create a subfolder in our application's directory called "`views`". So in the same directory that created your Ruby file, make this subfolder. In the subfolder, add a file called "`index.erb`" and set its contents to the following:

```
<!DOCTYPE html>
<html>
<head>
  <TITLE>Using views!</TITLE>
</head>
<body>
  <h1>Using views!</h1>
  <p>Hey, this worked!</p>
</body>
</html>
```

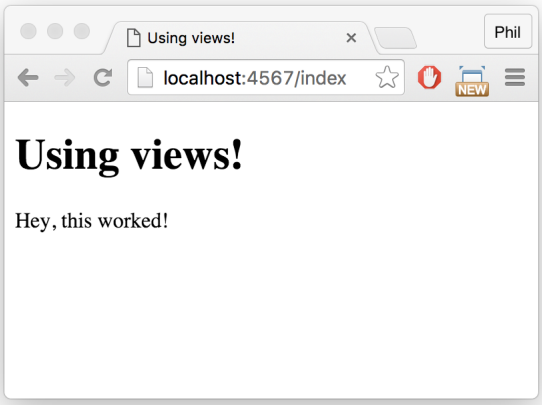
views/index.erb

In `using_a_view.rb`, add the following code.

```
require 'sinatra'
set :bind, '0.0.0.0' # Only needed if you're running from Codio

get '/index' do
  erb :index
end
```

using_a_view.erb

Run your app, and go to `http://[DOMAIN]:[PORT]/index` You should get something that looks like the following:



What happened there? Essentially the line of code "`erb :index`" told Sinatra to render the `index.erb` template to the Browser. ("Erb" is name of the template engine we're using)

6 Putting Ruby Code into Views

We can put Ruby code in the template too (something which anyone has had experience with PHP will be familiar with). This code must appear between <% and %> tags. To insert the value of an expression or variable we need to use `<%= x %>`.

Replace `index.erb` with the following and refresh the index page of your app (you don't need to restart your app if you're merely changing a view). You'll probably recognise this example — note how the code is easier to understand for embedding instructions within the HTML rather than the other way round...

```
<% TIMES_TABLE = 48 %>
<% TITLE = "Do you know your #{TIMES_TABLE} times table?" %>
<!DOCTYPE html>
<html>
<head>
  <TITLE><%= TITLE %></TITLE>
</head>
<body>
<h1><%= TITLE %></h1>
<ul>
<% for i in 1..12 %>
    <li><%= i %> times <%= TIMES_TABLE %> = <%= i * TIMES_TABLE %></li>
<% end %>
</ul>
</body>
</html>
```

6.1 Passing Values Between Sinatra Code and the View

Code should only appear in the template if it's to do with the presentation of the page. Any other code should reside in the main Ruby file. We can pass information from one to the other using special variables prefixed with "`@`". Copy and paste the following example into suitable files and run it:

```
require 'sinatra'
set :bind, '0.0.0.0' # Only needed if you're running from Codio

get '/say_hello' do
  @name = 'Phil'
  erb :say_hello
end
```

say_hello.rb

```
<!DOCTYPE html>
<html>
<head>
  <TITLE>Hi!</TITLE>
</head>
<body>
<h1>Hi <%= @name %>!</h1>
</body>
</html>
```

views/say_hello.erb

7 Using Sub-Templates

Suppose we want our site to have the same header and footer, or, a chunk of template code/HTML frequently occurs in our site? We can make these into standalone templates that we then include in the main template. See the following. Here's the Sinatra code, which just outputs the template for the page:

```
require 'sinatra'
set :bind, '0.0.0.0' # Only needed if you're running from Codio

get '/using_a_header_and_a_footer' do
  erb :using_a_header_and_a_footer
end
```

using_a_header_and_a_footer.rb

This main template looks as follows:

```
<%= erb :header %>

<p>Page specific content goes here.</p>

<%= erb :footer %>
```

views/using_a_header_and_a_footer.erb

Note how we've embedded two subtemplates for a common site header and site footer using the `erb` command in the form `<%= erb :subtemplate %>`. Note we need the equals in the opening tag, i.e. "`<%=`", since the `erb` is printing HTML. The two subtemplates are included in the `views` directory and have the name `header.erb` and `footer.erb` respectively. The template code for `header` and `footer` is as follows:

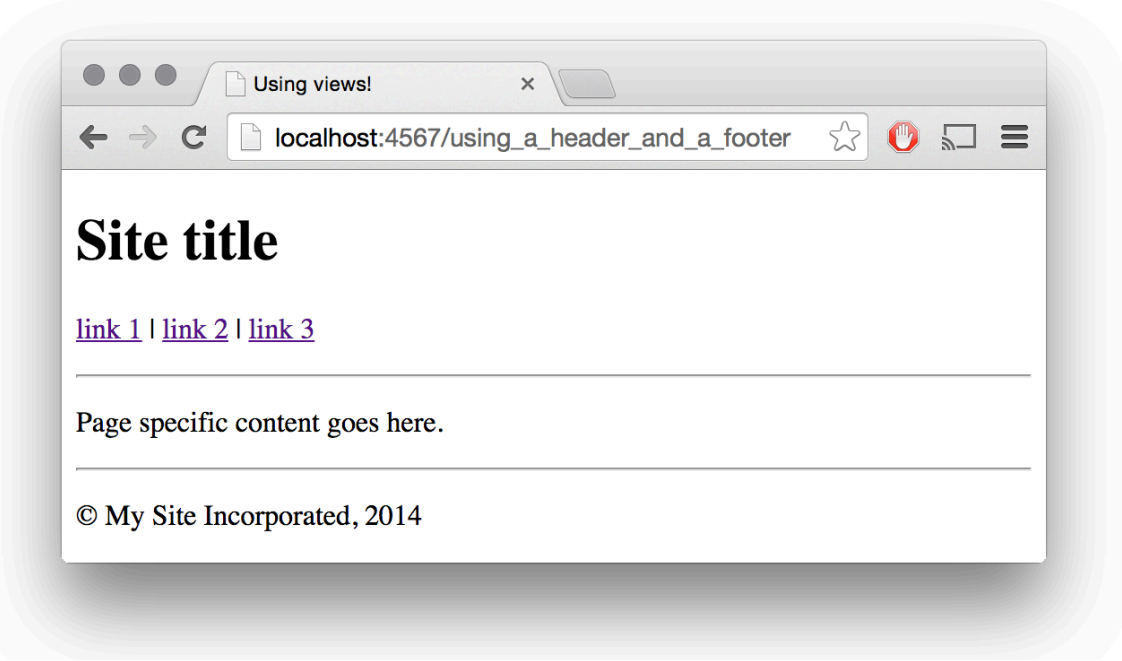
```
<!DOCTYPE html>
<html>
<head>
  <TITLE>Using views!</TITLE>
</head>
<body>

<h1>Site TITLE</h1>

<p><a href="">link 1</a> | <a href="">link 2</a> | <a href="">link 3</a> </p>

<hr />
```

views/header.erb

```
<hr />

<p>&copy; My Site Incorporated, 2014</p>

</body>
</html>
```

views/footer.erb

... and the final page looks like this:



And we can now re-use header and footer on other pages of our site using this scheme.

8 Incorporating Static Files

For our web pages to look nice, we going to want to include images, CSS files etc. These should all live in a subdirectory of your app, called `public`. In the simple example below, my app is made up of four files. The first is the app file, called `using_static_files.rb` that lives in the root directory:

```
require 'sinatra'
set :bind, '0.0.0.0' # Only needed if you're running from Codio

get '/' do
  erb :using_static_files
end
```

using_static_files.rb

This uses the template "`using_static_files`", which lives in the views directory:

```
<!DOCTYPE html>
<html>
<head>
  <link href="style/style.css" rel="stylesheet" type="text/css" />
  <TITLE>Testing referenced files</TITLE>
</head>
<body>
    <p><img src="images/me.jpg" /></p>
</body>
</html>
```

views/using_static_files.erb

As can be seen, the template uses an image and a CSS file. The image lives in the `public/images` directory, and the CSS file lives in the `public/style` directory:

```
img {
    border: 5px solid black;
}
```

public/style/style.css

9 Exercises

1. Think about your team project. Assuming you have an array of tweets (just make up an array and put some random strings in it!), design a template that puts each tweet into the row of a HTML table (i.e., using the `<table>`, `<tr>` and `<td>` tags — Google them if you're unfamiliar!). Now "run" your template by calling it from a controller.
2. Design a template that involves an image that resides in the `public` directory, and perhaps some CSS too.