Getting to know Sinatra

It's Witchcraft

You saw in the introduction how to install Sinatra, its dependencies, and write a small "hello world" application. In this chapter you will get a whirlwind tour of the framework and familiarize yourself with its features.

Routing

Sinatra is super flexible when it comes to routing, which is essentially an HTTP method and a regular expression to match the requested URL. The four basic HTTP request methods will get you a long ways:

- GET
- POST
- PATCH
- PUT
- DELETE

Routes are the backbone of your application, they're like a guide-map to how users will navigate the actions you define for your application.

They also enable to you create RESTful web services, in a very obvious manner. Here's an example of how one-such service might look:

```
get '/dogs' do
  # get a listing of all the dogs
end

get '/dog/:id' do
  # just get one dog, you might find him like this:
  @dog = Dog.find(params[:id])
  # using the params convention, you specified in your route
end

post '/dog' do
  # create a new dog listing
end

put '/dog/:id' do
  # HTTP PUT request method to update an existing dog
end

patch '/dog/:id' do
  # HTTP PATCH request method to update an existing dog
  # See RFC 5789 for more information
```

```
end

delete '/dog/:id' do
  # HTTP DELETE request method to remove a dog who's been sold!
end
```

As you can see from this contrived example, Sinatra's routing is very easy to get along with. Don't be fooled, though, Sinatra can do some pretty amazing things with Routes.

Take a more in-depth look at Sinatra's routes, and see for yourself.

Filters

Sinatra offers a way for you to hook into the request chain of your application via Filters.

Filters define two methods available, before and after which both accept a block to yield corresponding the request and optionally take a URL pattern to match to the request.

before

The before method will let you pass a block to be evaluated **before** *each* and *every* route gets processed.

```
before do
  MyStore.connect unless MyStore.connected?
end

get '/' do
  @list = MyStore.find(:all)
  erb :index
end
```

In this example, we've set up a before filter to connect using a contrived MyStore module.

after

The after method lets you pass a block to be evaluated **after** *each* and *every* route gets processed.

```
after do
  MyStore.disconnect
end
```

As you can see from this example, we're asking the MyStore module to disconnect after the request has been processed.

Pattern Matching

Filters optionally take a pattern to be matched against the requested URI during processing. Here's a quick

example you could use to run a contrived authenticate! method before accessing any "admin" type requests.

```
before '/admin/*' do
  authenticate!
end
```

Handlers

Handlers are top-level methods available in Sinatra to take care of common HTTP routines. For instance there are handlers for halting and passing.

There are also handlers for redirection:

```
  get '/' do
  redirect '/someplace/else'
end
```

This will return a 302 HTTP Response to /someplace/else.

You can even use the Sinatra handler for sessions, just add this to your application or to a configure block:

```
enable :sessions
```

Then you will be able to use the default cookie based session handler in your application:

```
get '/' do
  session['counter'] ||= 0
  session['counter'] += 1
  "You've hit this page #{session['counter']} times!"
end
```

Handlers can be extremely useful when used properly, probably the most common use is the params convention, which gives you access to any parameters passed in via the request object, or generated in your route pattern.

Templates

Sinatra is built upon an incredibly powerful templating engine, Tilt. Which, is designed to be a "thin interface" for frameworks that want to support multiple template engines.

Some of Tilt's other all-star features include:

- Custom template evaluation scopes / bindings
- Ability to pass locals to template evaluation
- Support for passing a block to template evaluation for "yield"
- Backtraces with correct filenames and line numbers
- Template file caching and reloading

Best of all, Tilt includes support for some of the best templating engines available, including HAML, Less CSS, and CoffeeScript. Also a ton of other lesser known, but equally awesome templating languages that would take too much space to list.

All you need to get started is erb, which is included in Ruby. Views by default look in the views directory in your application root.

So in your route you would have:

```
get '/' do
  erb :index # renders views/index.erb
end
```

or to specify a template located in a subdirectory

```
get '/' do
  erb :"dogs/index"
  # would instead render views/dogs/index.erb
end
```

Another default convention of Sinatra, is the layout, which automatically looks for a views/layout template file to render before loading any other views. In the case of using erb, your views/layout.erb would look something like this:

```
<html>
  <head>..</head>
  <body>
    <%= yield %>
  </body>
</html>
```

The possibilities are pretty much endless, here's a quick list of some of the most common use-cases covered in the README:

- Inline Templates
- Embedded Templates
- Named Templates

For more specific details on how Sinatra handles templates, check the README.

Helpers

Helpers are a great way to provide reusable code snippets in your application.

```
helpers do
  def bar(name)
    "#{name}bar"
```

```
  end
end

get '/:name' do
  bar(params[:name])
end
```