Sinatra The Book

Introduction

What is Sinatra?

Sinatra is a Domain Specific Language (DSL) for quickly creating web-applications in Ruby.

It keeps a minimal feature set, leaving the developer to use the tools that best suit them and their application.

It doesn't assume much about your application, apart from that:

- it will be written in Ruby programming language
- it will have URLs

In Sinatra, you can write short *ad hoc* applications or mature, larger application with the same easiness.

You can use the power of various Rubygems and other libraries available for Ruby.

Sinatra really shines when used for experiments and application mock-ups or for creating a quick interface for your code.

It isn't a *typical* Model-View-Controller framework, but ties specific URL directly to relevant Ruby code and returns its output in response. It does enable you, however, to write clean, properly organized applications: separating *views* from application code, for instance.

Installation

The simplest way to install Sinatra is through Rubygems:

```
$ gem install sinatra
```

Dependencies

Sinatra depends on the *Rack* gem (http://rack.rubyforge.org).

Sinatra supports many different template engines (it uses the Tilt library internally to support practically every template engine in Ruby) For optimal experience, you should install the template engines you want to work with. The Sinatra dev team suggests using either ERB, which is included with Ruby, or installing HAML as your first template language.

```
$ gem install haml
```

Living on the Edge

The *edge* version of Sinatra lives in its Git repository, available at **http://github.com/sinatra/sinatra/tree/master**.

You can use the *edge* version to try new functionality or to contribute to the framework. You need to have Git version control software and bundler.

```
$ gem install bundler
```

To use Sinatra *edge* with bundler, you'll have to create a `Gemfile` listing Sinatra's and any other dependencies you're going to need.

```
source :rubygems
gem 'sinatra', :git => 'git://github.com/sinatra/sinatra.git'
```

Here we use the gemcutter source to specify where to get Sinatra's dependencies; alternatively you can use the git version, but that is up to you. So now we can install our bundle:

```
$ bundle install
```

Hello World Application

Sinatra is installed, how about making your first application?

```
require 'rubygems'

# If you're using bundler, you will need to add this
require 'bundler/setup'

require 'sinatra'

get '/' do
  "Hello world, it's #{Time.now} at the server!"
end
```

Run this application by `$ ruby hello_world.rb` and load `http://localhost:4567` in your browser.

As you can see, Sinatra doesn't force you to setup much infrastructure: a request to a URL evaluates some Ruby code and returns some text in response. Whatever the block returns is sent back to the browser.

Real World Applications in Sinatra

Github Services

Git hosting provider Github uses Sinatra for post-receive hooks, calling user specified services/URLs, whenever someone pushes to their repository:

- http://github.com/blog/53-github-services-ipo
- http://github.com/guides/post-receive-hooks
- http://github.com/pjhyett/github-services

Check out a full list of Sinatra apps in the wild.

About this book

This book will assume you have a basic knowledge of the Ruby scripting language and a working

Ruby interpreter.

For more information about the Ruby language visit the following links:

- ruby-lang
- ruby-lang / documentation

Need Help?

The Sinatra club is small, but super-friendly. Join us on IRC at irc.freenode.org in #sinatra if you have any questions. It's a bit slow at times, so give us a bit to get back to your questions.

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
```

Organizing your application

```
"You don't know how paralyzing that is, that stare of a blank canvas is"
- Vincent van Gogh
```

Sinatra is a blank canvas. It can transform into a single-file application, an embedded administration page, an API server, or a full-fledged hundred page website. Each use case is unique. This chapter will provide some advice on specific situations, but you will need to experiment to determine the best structure of your code. Don't be afraid to experiment.

A Single File Application

Obviously the file system structure is easy for this one.

A single file can contain an entire multi-page application. Don't be afraid to use inline templates, multiple classes (this isn't Java, multiple classes will live happily next to each other in the same file)

A Large Site

This one is trickier. My advice is to look to Rails for advice. They have a well structured set of directories to hold many of the components that make up a larger application. Remember that this file structure is just a suggestion.

```
|- config.ru            # A rackup file. Load server.rb, and
|- server.rb            # Loads all files, is the base class
|- app/
\--- routes/
\------ users.rb
\--- models/
\------ user.rb         # Model. Database or not
\--- views/
\------ users/
\--------- index.erb
\--------- new.erb
\--------- show.erb
```

In `server.rb` it should be a barebones application.

```
class Server < Sinatra::Base
  configure do
    # Load up database and such
  end
end

# Load all route files
Dir[File.dirname(__FILE___) + "/app/routes/**"].each do |route|
  require route
end
```

And the route files look something like:

```
# users.rb
class Server < Sinatra::Base
  get '/users/:id' do
    erb :"users/show"
  end

  # more routes...
end
```

Views

RSS Feed

The builder gem/library for creating XML is required in this recipe.

Assume that your site url is `http://liftoff.msfc.nasa.gov/`.

```
get '/rss.xml' do
  builder do |xml|
    xml.instruct! :xml, :version => '1.0'
    xml.rss :version => "2.0" do
      xml.channel do
        xml.title "Liftoff News"
        xml.description "Liftoff to Space Exploration."
        xml.link "http://liftoff.msfc.nasa.gov/"

        @posts.each do |post|
          xml.item do
            xml.title post.title
            xml.link "http://liftoff.msfc.nasa.gov/posts/#{post.id}"
            xml.description post.body
            xml.pubDate Time.parse(post.created_at.to_s).rfc822()
            xml.guid "http://liftoff.msfc.nasa.gov/posts/#{post.id}"
          end
        end
      end
    end
  end
end
```

This will render the RSS inline, directly from the handler.

CoffeeScript

To render CoffeeScript templates you first need the `coffee-script` gem and `therubyracer`, or access to the `coffee` binary.

Here's an example of using CoffeeScript with Sinatra's template rendering engine Tilt:

```
## You'll need to require coffee-script in your app
require 'coffee-script'

get '/application.js' do
  coffee :application
end
```

Renders `./views/application.coffee`.

This works great if you have access to nodejs or `therubyracer` gem on your platform of choice and hosting environment. If that's not the case, but you'd still like to use CoffeeScript, you can precompile your scripts using the `coffee` binary:

```
$ coffee -c -o public/javascripts/ src/
```

Or you can use this example rake task to compile them for you with the `coffee-script` gem, which can use either `therubyracer` gem or the `coffee` binary:

```
require 'coffee-script'

namespace :js do
  desc "compile coffee-scripts from ./src to ./public/javascripts"
  task :compile do
    source = "#{File.dirname(__FILE__)}/src/"
    javascripts = "#{File.dirname(__FILE__)}/public/javascripts/"
```

```
    Dir.foreach(source) do |cf|
      unless cf == '.' || cf == '..'
        js = CoffeeScript.compile File.read("#{source}#{cf}")
        open "#{javascripts}#{cf.gsub('.coffee', '.js')}", 'w' do |f|
          f.puts js
        end
      end
    end
  end
end
```

Now, with this rake task you can compile your coffee-scripts to `public/javascripts` by using the `rake js:compile` command.

Resources

If you get stuck or want to look into other ways of implementing CoffeeScript in your application, these are a great place to start:

- coffee-script
- therubyracer
- ruby-coffee-script

Models

DataMapper

Start out by getting the DataMapper gem if you don't already have it, and then making sure it's in your application. A call to `setup` as usual will get the show started, and this example will include a 'Post' model.

```
require 'rubygems'
require 'sinatra'
require 'data_mapper' # metagem, requires common plugins too.

# need install dm-sqlite-adapter
DataMapper::setup(:default, "sqlite3://#{Dir.pwd}/blog.db")

class Post
  include DataMapper::Resource
  property :id, Serial
  property :title, String
  property :body, Text
  property :created_at, DateTime
end

# Perform basic sanity checks and initialize all relationships
# Call this when you've defined all your models
DataMapper.finalize

# automatically create the post table
Post.auto_upgrade!
```

Once that is all well and good, you can actually start developing your application!

```
get '/' do
  # get the latest 20 posts
```

```
  @posts = Post.all(:order => [ :id.desc ], :limit => 20)
  erb :index
end
```

Finally, the view at `./view/index.erb`:

```
<% @posts.each do |post| %>
  <h3><%= post.title %></h3>
  <p><%= post.body %></p>
<% end %>
```

For more information on DataMapper, check out the project documentation.

Helpers

Implementation of rails style partials

Using partials in your views is a great way to keep them clean. Since Sinatra takes the hands off approach to framework design, you'll have to implement a partial handler yourself.

Here is a really basic version:

```
# Usage: partial :foo
helpers do
  def partial(page, options={})
    haml page, options.merge!(:layout => false)
  end
end
```

A more advanced version that would handle passing local options, and looping over a hash would look like:

```
# Render the page once:
# Usage: partial :foo
#
# foo will be rendered once for each element in the array, passing in a local
variable named "foo"
# Usage: partial :foo, :collection => @my_foos

helpers do
  def partial(template, *args)
    options = args.extract_options!
    options.merge!(:layout => false)
    if collection = options.delete(:collection) then
      collection.inject([]) do |buffer, member|
        buffer << haml(template, options.merge(
                                  :layout => false,
                                  :locals => {template.to_sym => member}
                                )
                      )
      end.join("\n")
    else
      haml(template, options)
    end
  end
end
```

Middleware

Sinatra rides on Rack, a minimal standard interface for Ruby web frameworks. One of Rack's most interesting capabilities for application developers is support for "middleware" -- components that sit between the server and your application monitoring and/or manipulating the HTTP request/response to provide various types of common functionality.

Sinatra makes building Rack middleware pipelines a cinch via a top-level `use` method:

```
require 'sinatra'
require 'my_custom_middleware'

use Rack::Lint
use MyCustomMiddleware

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

Rack HTTP Basic Authentication

The semantics of "use" are identical to those defined for the Rack::Builder DSL (most frequently used from rackup files). For example, the use method accepts multiple/variable args as well as blocks:

```
use Rack::Auth::Basic do |username, password|
  username == 'admin' && password == 'secret'
end
```

Rack is distributed with a variety of standard middleware for logging, debugging, URL routing, authentication, and session handling. Sinatra uses many of of these components automatically based on configuration so you typically don't have to use them explicitly.

Testing

Using Rack::Test

Testing is an integral part of software development. In this section we will look into testing the Sinatra application itself. For unit testing your models or other classes, please consult the documentation of frameworks used (including your test framework itself). Sinatra itself uses Contest for testing, but feel free to use any framework you like.

Bryan Helmkamp's Rack::Test offers tools for mocking Rack request, sending those to your application and inspecting the response all wrapped in a small DSL.

Firing Requests

You import the DSL by including `Rack::Test::Methods` into your test framework. It is even usable without a framework and for other tasks besides testing.

Imagine you have an application like this:

```
# myapp.rb
```

```
require 'sinatra'

get '/' do
  "Welcome to my page!"
end

post '/' do
  "Hello #{params[:name]}!"
end
```

You have to define an `app` method pointing to your application class (which is `Sinatra::Application` per default):

```
begin
  # try to use require_relative first
  # this only works for 1.9
  require_relative 'my-app.rb'
rescue NameError
  # oops, must be using 1.8
  # no problem, this will load it then
  require File.expand_path('my-app.rb', __FILE__)
end

require 'test/unit'
require 'rack/test'

class MyAppTest < Test::Unit::TestCase
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_my_default
    get '/'
    assert last_response.ok?
    assert_equal 'Welcome to my page!', last_response.body
  end

  def test_with_params
    post '/', :name => 'Frank'
    assert_equal 'Hello Frank!', last_response.body
  end
end
```

Modifying env

While parameters can be send via the second argument of a get/post/put/delete call (see the post example above), the env hash (and thereby the HTTP headers) can be modified with a third argument:

```
get '/foo', {}, 'HTTP_USER_AGENT' => 'Songbird 1.0'
```

This also allows passing internal `env` settings:

```
get '/foo', {}, 'rack.session' => { 'user_id' => 20 }
```

Cookies

For example, add the following to your app to test against:

```
"Hello #{request.cookies['foo']}!"
```

Use `set_cookie` for setting and removing cookies, and the access them in your response:

```
response.set_cookie 'foo=bar'
get '/'
assert_equal 'Hello bar!', last_response.body
```

Asserting Expectations About The Response

Once a request method has been invoked, the following attributes are available for making assertions:

- `app` - The Sinatra application class that handled the mock request.
- `last_request` - The `Rack::MockRequest` used to generate the request.
- `last_response` - A `Rack::MockResponse` instance with information on the response generated by the application.

Assertions are typically made against the `last_response` object. Consider the following examples:

```
def test_it_says_hello_world
  get '/'
  assert last_response.ok?
  assert_equal 'Hello World'.length.to_s, last_response.headers['Content-
Length']
  assert_equal 'Hello World', last_response.body
end
```

Optional Test Setup

The `Rack::Test` mock request methods send requests to the return value of a method named `app`.

If you're testing a modular application that has multiple `Sinatra::Base` subclasses, simply set the `app` method to return your particular class.

```
def app
  MySinatraApp
end
```

If you're using a classic style Sinatra application, then you need to return an instance of `Sinatra::Application`.

```
def app
  Sinatra::Application
end
```

Making `Rack::Test` available to all test cases

If you'd like the `Rack::Test` methods to be available to all test cases without having to include it each time, you can include the `Rack::Test` module in the `Test::Unit::TestCase` class:

```
require 'test/unit'
require 'rack/test'

class Test::Unit::TestCase
  include Rack::Test::Methods
end
```

Now all `TestCase` subclasses will automatically have `Rack::Test` available to them.

Development Techniques

Automatic Code Reloading

Restarting an application manually after every code change is both slow and painful. It can easily be avoided by using a tool for automatic code reloading.

Shotgun

Shotgun will actually restart your application on every request. This has the advantage over other reloading techniques of always producing correct results. However, since it actually restarts your application, it is rather slow compared to the alternatives. Moreover, since it relies on `fork`, it is not available on Windows and JRuby.

Usage is rather simple:

```
gem install shotgun # run only once, to install shotgun
shotgun my_app.rb
```

If you want to run a modular application, create a file named `config.ru` with similar content:

```
require 'my_app'
run MyApp
```

And run it by calling `shotgun` without arguments.

The `shotgun` executable takes arguments similar to those of the `rackup` command, run `shotgun --help` for more information.

Deployment

Heroku

This is the easiest configuration + deployment option. Heroku has full support for Sinatra applications. Deploying to Heroku is simply a matter of pushing to a remote git repository.

Steps to deploy to Heroku:

- Create an account if you don't have one
- Download and install Heroku toolbelt
- Make sure you have a `Gemfile`. Otherwise, you can create one and install the `sinatra` gem using `bundler`.
- Make a config.ru in the root-directory
- Create the app on heroku
- Push to it

1. Install `bundler` if you haven't yet (`gem install bundler`). Create a `Gemfile` using `bundle init`. Modify your `Gemfile` to look like:

   ```
   source 'http://rubygems.org'

   gem 'sinatra'
   ```

   Run `bundle` to install the gem.

   It is possible to specify a specific Ruby version on your Gemfile. For more information, check this.

2. Here is an example config.ru file that does two things. First, it requires your main app file, whatever it's called. In the example, it will look for `myapp.rb`. Second, run your application. If you're subclassing, use the subclass's name, otherwise use Sinatra::Application.

   ```
   require "myapp"

   run Sinatra::Application
   ```

3. Create the app and push to it

From the root directory of the application, run these commands:

```
```bash
$ heroku create <app-name>  # This will add heroku as a remote
$ git push heroku master
```
```

For more details see this.

Contributing

There are plenty of ways to contribute to Sinatra.

Got a recipe or tutorial? Check out the Sinatra Recipes project for all of the recent additions from the community.

If you're looking for something to work on be sure to check the issue tracker. Once you have forked the project, feel free to send us a pull request.

Check the wiki for more information.

Join us on IRC (#sinatra at irc.freenode.org) if you need help with anything.