Table of Contents

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. (See section "Real World Applications" later in this book.)

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 installed (http://www.git-scm.com). You could use either rake or bundler. Follow these steps:

Rake

```
gem install rake
```

1. cd where/you/keep/your/projects
2. git clone git://github.com/sinatra/sinatra.git
3. cd sinatra
4. rake install

Bundler Alternatively you can use bundler (http://gembundler.com/).

```
gem install bundler
```

To use edge sinatra with bundler, you'll have to create a gemfile listing sinatra's dependencies; and other dependencies for your application. In your application's root create your 'Gemfile':

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

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 these 2 lines
require 'bundler'
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

Git Wiki

Git Wiki is minimal Wiki engine powered by Sinatra and Git. See also various forks with additional functionality.

- http://github.com/sr/git-wiki
- http://github.com/sr/git-wiki/network

Integrity

Integrity is small and clean *continuous integration* service using Sinatra, watching for failing builds of your codebase and notifying you by various channels.

- http://www.integrityapp.com/
- http://github.com/integrity/integrity

Seinfeld Calendar

Seinfeld Calendar is a fun application tracking your contributions to open-source projects, displaying your “streaks”, ie. continuous commits to Github repositories.

- http://www.calendaraboutnothing.com
- http://github.com/entp/seinfeld

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:

- http://www.ruby-lang.org
- http://www.ruby-lang.org/en/documentation/ruby-from-other-languages/
- http://www.ruby-doc.org
- http://www.ruby-doc.org/core-1.8.7/index.html
- http://www.ruby-doc.org/docs/ProgrammingRuby/

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.

Routes

HTTP methods

Sinatra’s routes are designed to respond to the HTTP request methods.

- GET
- POST
- PUT
- DELETE

Basic

The bare minimum route is a few lines. You must define a route with the HTTP method, then the path that you want to match. When that route gets matched, an attached code block will be run. Whatever that block returns will be sent back to the browser of the client.

```
get '/' do
  "Hello Sinatra!"
end
```

Sinatra will also automatically parse parameters from the URL:

```
# /name/Chris will return "You said your name was Chris" to the browser
# /name/Blake will return "You said your name was Blake" to the browser
get '/name/:name' do
```

```
  "You said your name was #{params[:name]}"
end
```

An alternate method of automatically parsing parameters is to use them as values that are passed into the block. This code is identical to the name example directly above:

```
get '/name/:name' do |name|
  "You said your name was #{name}"
end
```

Options

User agent

```
get '/foo', :agent => /Songbird (\d\.\d)[\d\/]*?/ do
  "You're using Songbird version #{params[:agent][0]}"
end

get '/foo' do
  # matches non-songbird browsers
end
```

Splats

Sometimes you want to match more than a single parameter, and instead want to match an entire set of URL components. You can use the splat operator to do this.

```
get '/say/*/to/*' do
  # matches /say/hello/to/world
  params["splat"] # => ["hello", "world"]
end

get '/download/*.*' do
  # matches /download/path/to/file.xml
  params["splat"] # => ["path/to/file", "xml"]
end
```

HTTP Methods

The other HTTP methods are requested exactly the same as “get” routes. You simply use the `post`, `put`, or `delete` functions to define the route, rather then the `get` one.

```
get '/foo' do
end

post '/foo' do
end

put '/foo' do
end

delete '/foo' do
end
```

The PUT and DELETE methods

Since browsers don’t natively support the PUT and DELETE methods, a hacky workaround has been adopted by the web community. There are two steps to using this workaround with Sinatra:

First, you must add a hidden element in your form with the name “_method” and the value equal to the HTTP method you want to use. The form itself is sent as a POST, but Sinatra will interpret it as the desired method. For example:

```
<form method="post" action="/destroy_it">
  <input type="hidden" name="_method" value="delete" />
  <div><button type="submit">Destroy it</button></div>
</form>
```

Then, include the Rack::MethodOverride middleware into your app:

```
require 'sinatra'

use Rack::MethodOverride

delete '/destroy_it' do
  # destroy it
end
```

Or, if you are subclassing Sinatra::Base, do it like this:

```
require 'sinatra/base'

class MyApp < Sinatra::Base
  use Rack::MethodOverride

  delete '/destroy_it' do
    # destroy it
  end
end
```

When you want to use PUT or DELETE from a client that does support them (like Curl, or ActiveResource), just go ahead and use them as you normally would, and ignore the `_method` advice above. That is only for hacking in support for browsers.

How routes are looked up

Each time you add a new route to your application, the URL definition is compiled down into a regular expression. The regex is stored in an array along with the Ruby block attached to that route.

When a new request comes in, each regex is checked in turn, until one matches. Then the code block attached to that route gets executed.

Handlers

Structure

Handler is the generic term that Sinatra uses for the “controllers”. A handler is the initial point of entry for new HTTP requests into your application.

To find more about the routes, head to the Routes section

Form parameters

In handlers you can access submitted form parameters directly via the params hash:

```
get '/' do
  params['post']
end
```

Parameters can be accessed by either a string, or symbol key:

```
params[:name]
params["name"]
```

Nested form parameters

The support for Rails-like nested parameters has been built-in since Sinatra version 0.9.0.

```
<form>
  <input ... name="post[title]" />
  <input ... name="post[body]" />
  <input ... name="post[author]" />
</form>
```

The parameters in this case became as a hash:

```
{"post"=>{ "title"=>"", "body"=>"", "author"=>"" }}
```

Therefore in handlers you can use nested parameters like a regular hash:

```
params['post']['title']
```

Redirect

The redirect helper is a shortcut to a common http response code (302).

Basic usage is easy:

```
redirect '/'

redirect '/posts/1'

redirect 'http://www.google.com'
```

The redirect actually sends back a Location header to the browser, and the browser makes a followup request to the location indicated. Since the browser makes that followup request, you can redirect to any page, in your application, or another site entirely.

The flow of requests during a redirect is: Browser → Server (redirect to '/') → Browser (request '/') → Server (result for '/')

To force Sinatra to send a different response code, it's very simple:

```
redirect '/', 303 # forces the 303 return code

redirect '/', 307 # forces the 307 return code
```

Sessions

Default Cookie Based Sessions

Sinatra ships with basic support for cookie-based sessions. To enable it in a configure block, or at the top of your application, you just need to enable the option.

```
enable :sessions

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

The downside to this session approach is that all the data is stored in the cookie. Since cookies have a fairly hard limit of 4 kilobytes, you can't store much data. The other issue is that cookies are not tamper proof – the user can change any data in their session. But… it is easy, and it doesn't have the scaling problems that memory or database backed sessions run into.

Memory Based Sessions

Memcached Based Sessions

File Based Sessions

Database Based Sessions

Cookies

Cookies are a fairly simple thing to use in Sinatra, but they have a few quirks.

Lets first look at the simple use case:

```
require 'rubygems'
require 'sinatra'

get '/' do
    # Get the string representation
    cookie = request.cookies["thing"]

    # Set a default
    cookie ||= 0

    # Convert to an integer
    cookie = cookie.to_i

    # Do something with the value
    cookie += 1

    # Reset the cookie
    set_cookie("thing", cookie)

    # Render something
    "Thing is now: #{cookie}"
end
```

Setting a path, expiration date, or domain gets a little more complicated – see the source code for set_cookie if you want to dig deeper.

```
set_cookie("thing", :domain => myDomain,
                    :path => myPath,
                    :expires => Date.new)
```

That's the easy stuff with cookies – It can also serialize Array objects, separating them with ampersands (&), but when they come back, it doesn't deserialize or split them in any way, it hands you the raw, encoded string for your parsing pleasure.

Status

If you want to set your own status response instead of the normal 200 (Success), you can use the `status` helper to set the code, and then still render normally:

```
get '/' do
  status 404
  "Not found"
end
```

Because this is common, there's a `not_found` helper to do this. In this example, no response body will be sent, and the browser will display it's default content.

```
get '/' do
  not_found
end
```

The `not_found` helper takes an optional argument of the body to send. Use a template to have a complicated 404 page.

```
get '/' do
  not_found(haml :404) # renders views/404.haml
end
```

And another way, a bit more flexible because you can easily setup other statuses than 404 is to raise a special exception subclass.

```
get '/' do
  raise NotFound
end
```

Sinatra defines the `NotFound` exception. To define your own, subclass exception and define a code method returning the HTTP status code you want. For example, to return a 401 simply in your app you can use this code:

```
class Unauthorized < Exception
  def code
    401
  end
end

get '/' do
  raise Unauthorized
end
```

Filters

Overview

Before filters are run before or after the route has been processed:

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

get '/' do
  @list = MyStore.load_list
  haml :index
end
```

Similar to before filters, after filters are run after the request has been processed:

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

It is also possible to have multiple filters:

```
before { do_something }
before { something_else }

get '/' do
  # ...
end

after do
  do_something
end

after do
  something_else
end
```

Filter context

Filters are run in the same context as routes, and can therefore access all instance variables and helper methods:

```
helpers do
  attr_reader :current_user
  def user(id) User.find(id) end
end

before { @current_user = user session['user_id'] }

get '/admin' do
  pass unless current_user.admin?
  erb :"admin/index"
end
```

Changing the response

The return value of a filter is ignored, therefore a request to '/' will return 'yes' for the following Sinatra application:

```
get('/') { 'yes' }
after { 'no' }
```

However, it is possible to change the response using `halt`:

```
get('/') { return @buisness_secrets }
after { halt 403, 'you almost got me' }
```

Pattern matching filters

Filters optionally taking a pattern, causing them to be evaluated only if the request path matches that pattern:

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

after '/create/:slug' do |slug|
  session[:last_slug] = slug
end
```

Views

All file-based view files should be located in the directory `./views/`:

```
root
  | - views/
```

You access these views by calling various view helpers. These are methods that lookup the template, render it, and return a string containing the rendered output. These view methods do not return anything to the browser by themselves. The only output to the browser will be the return value of the handler block.

To use a different view directory:

```
set :views, File.dirname(__FILE__) + '/templates'
```

One important thing to remember is that you always have to reference templates with symbols, even if they're in a subdirectory, in this case use `:subdir/template`. You must use a symbol because otherwise rendering methods will render any strings passed to them directly.

Template Languages

Erb

```
## You'll need to require erb in your app
require 'erb'

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

This will render ./views/index.erb

Haml

The haml gem/library is required to render HAML templates:

```
set :haml, :format => :html5 # default Haml format is :xhtml

get '/' do
  haml :index, :format => :html4 # overridden
end
```

This will render ./views/index.haml

Haml's options can be set globally through Sinatra's configurations, see Options and Configurations, and overridden on an individual basis.

Erubis

The erubis gem/library is required to render erubis templates:

```
## You'll need to require erubis in your app
require 'erubis'

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

Render `./views/index.erubis`

Nokogiri

The nokogiri gem/library is required to render nokogiri templates:

```
## You'll need to require nokogiri in your app
require 'nokogiri'

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

Renders `./views/index.nokogiri`.

Builder

The builder gem/library is required to render builder templates:

```
## You'll need to require builder in your app
require 'builder'

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

This will render ./views/index.builder

```
get '/' do
  builder do |xml|
    xml.node do
      xml.subnode "Inner text"
    end
  end
end
```

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

Atom Feed

RSS Feed

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.

Sass

The sass gem/library is required to render Sass templates:

```
## You'll need to require sass in your app
require 'sass'

get '/' do
  sass :styles
end
```

This will render `./views/styles.sass`

Sass' options can be set globally through Sinatra's configuration, see Options and Configuartion, and overridden on an individual basis.

```
set :sass, :style => :compact # default Sass style is :nested

get '/stylesheet.css' do
    sass :stylesheet, :style => :expanded # overridden
end
```

Scss Templates

The sass gem/library is required to render Scss templates:

```
## You'll need to require haml or sass in your app
require 'sass'

get '/stylesheet.css' do
  scss :stylesheet
end
```

Renders `./views/stylesheet.scss`.

Scss' options can be set globally through Sinatra's configurations, see Options and Configurations, and overridden on an individual basis.

```
set :scss, :style => :compact # default Scss style is :nested

get '/stylesheet.css' do
  scss :stylesheet, :style => :expanded # overridden
end
```

Less Templates

The less gem/library is required to render Less templates:

```
## You'll need to require less in your app
require 'less'

get '/stylesheet.css' do
  less :stylesheet
end
```

Renders `./views/stylesheet.less`.

Liquid Templates

The liquid gem/library is required to render Liquid templates:

```
## You'll need to require liquid in your app
require 'liquid'

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

Renders `./views/index.liquid`.

Since you cannot call Ruby methods, except for `yield`, from a Liquid template, you almost always want to pass locals to it:

```
liquid :index, :locals => { :key => 'value' }
```

Markdown Templates

The rdiscount gem/library is required to render Markdown templates:

```
## You'll need to require rdiscount in your app
require "rdiscount"

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

Renders `./views/index.markdown` (+md+ and +mkd+ are also valid file extensions).

It is not possible to call methods from markdown, nor to pass locals to it. You therefore will usually use it in combination with another rendering engine:

```
erb :overview, :locals => { :text => markdown(:introduction) }
```

Note that you may also call the markdown method from within other templates:

```
%h1 Hello From Haml!
%p= markdown(:greetings)
```

Textile Templates

The RedCloth gem/library is required to render Textile templates:

```
## You'll need to require redcloth in your app
require "redcloth"

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

Renders `./views/index.textile`.

It is not possible to call methods from textile, nor to pass locals to it. You therefore will usually use it in combination with another rendering engine:

```
erb :overview, :locals => { :text => textile(:introduction) }
```

Note that you may also call the textile method from within other templates:

```
%h1 Hello From Haml!
%p= textile(:greetings)
```

RDoc Templates

The RDoc gem/library is required to render RDoc templates:

```
## You'll need to require rdoc in your app
require "rdoc"

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

Renders `./views/index.rdoc`.

It is not possible to call methods from rdoc, nor to pass locals to it. You therefore will usually use it in combination with another rendering engine:

```
erb :overview, :locals => { :text => rdoc(:introduction) }
```

Note that you may also call the rdoc method from within other templates:

```
%h1 Hello From Haml!
%p= rdoc(:greetings)
```

Radius Templates

The radius gem/library is required to render Radius templates:

```
## You'll need to require radius in your app
require 'radius'

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

Renders `./views/index.radius`.

Since you cannot call Ruby methods, except for `yield`, from a Radius template, you almost always want to pass locals to it:

```
radius :index, :locals => { :key => 'value' }
```

Markaby Templates

The markaby gem/library is required to render Markaby templates:

```
## You'll need to require markaby in your app
require 'markaby'

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

Renders `./views/index.mab`.

CoffeeScript Templates

The coffee-script gem/library and the `coffee` binary are required to render CoffeeScript templates:

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

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

Renders `./views/application.coffee`.

Inline Templates

```
get '/' do
  haml '%div.title Hello World'
```

```
end
```

Renders the inlined template string.

Subdirectories in views

In order to create subdirectories in `./views/`, first you need to just create the directory structure. As an example, it should look like:

```
root
  | - views/
    | - users/
      | - index.haml
      | - edit.haml
```

Then you can call the haml view helper with a symbol pointing to the path of the view. There’s a syntax trick for this in ruby, to convert a string to a symbol.

```
:"users/index"
```

You can also use the more verbose version of the same thing:

```
"users/index".to_sym
```

Layouts

Layouts are simple in Sinatra. Put a file in your views directory named “layout.erb”, “layout.haml”, or “layout.builder”. When you render a page, the appropriate layout will be grabbed, of the same filetype, and used.

The layout itself should call `yield` at the point you want the content to be included.

An example haml layout file could look something like this:

```
%html
  %head
    %title SINATRA BOOK
  %body
    #container
      = yield
```

Avoiding a layout

Sometimes you don’t want the layout rendered. In your render method just pass `:layout => false`, and you’re good.

```
get '/' do
  haml :index, :layout => false
end
```

Specifiying a custom layout

If you want to use a layout not named “layout”, you can override the name that’s used by passing `:layout => :custom_layout`

```
get '/' do
  haml :index, :layout => :custom_layout
end
```

In File Views

For your micro-apps, sometimes you don't even want a separate views file. Ruby has a way of embedding data at the end of a file, which Sinatra makes use of to embed templates directly into its file.

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

enable :inline_templates

__END__

@@ layout
%html
= yield

@@ index
%div.title Hello world!!!!!
```

NOTE: Inline templates defined in the source file that requires sinatra are automatically loaded. Call `enable :inline_templates` explicitly if you have inline templates in other source files.

Named Templates

Templates may also be defined using the top-level `template` method:

```
template :layout do
  "%html\n  =yield\n"
end

template :index do
  '%div.title Hello World!'
end

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

If a template named "layout" exists, it will be used each time a template is rendered. You can disable layouts by passing `:layout => false`.

```
get '/' do
  haml :index, :layout => !request.xhr?
end
```

Partials

Partials are not built into the default installation of Sinatra.

The minimalist implementation of partials takes zero helper code. Just call your view method from your view code.

```
<%= erb :_my_partial_file, :layout => false %>
```

You can even pass local variables via this approach.

```
<%= erb :_my_partial_file, :layout => false, :locals => {:a => 1} %>
```

If you find that you need a more advanced partials implementation that handles collections and other features, you will need to implement a helper that does that work.

```
helpers do
  def partial(template, options={})
    erb template, options.merge(:layout => false)
    #TODO: Implementation
  end
end
```

Accessing Variables in Templates

Templates are evaluated within the same context as route handlers. Instance variables set in route handlers are direcly accessible by templates:

```
get '/:id' do
  @foo = Foo.find(params[:id])
  haml '%h1= @foo.name'
end
```

Or, specify an explicit Hash of local variables:

```
get '/:id' do
  foo = Foo.find(params[:id])
  haml '%h1= foo.name', :locals => { :foo => foo }
end
```

This is typically used when rendering templates as partials from within other templates.

Models

Datamapper

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

```
require 'rubygems'
require 'sinatra'
require 'datamapper'

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

# automatically create the post table
Post.auto_migrate! unless Post.table_exists?
```

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

```
get '/' do
    # get the latest 20 posts
    @posts = Post.get(:order => [ :id.desc ], :limit => 20)
    erb :index
end
```

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

```
<% for post in @posts %>
    <h3><%= post.title %></h3>
    <p><%= post.body %></p>
<% end %>
```

Sequel

Require the Sequel gem in your app:

```
require 'rubygems'
require 'sinatra'
require 'sequel'
```

Use a simple in-memory DB:

```
DB = Sequel.sqlite
```

Create a table:

```
DB.create_table :links do
 primary_key :id
 varchar :title
 varchar :link
end
```

Create the Model class:

```
class Link < Sequel::Model
end
```

Create the route:

```
get '/' do
 @links = Link.all
 haml :links
end
```

ActiveRecord

First require ActiveRecord gem in your application, then give your database connection settings:

```
require 'rubygems'
require 'sinatra'
require 'active_record'

ActiveRecord::Base.establish_connection(
  :adapter => 'sqlite3',
  :database =>  'sinatra_application.sqlite3.db'
)
```

Now you can create and use ActiveRecord models just like in Rails (the example assumes you already have a ‘posts’ table in your database):

```
class Post < ActiveRecord::Base
end

get '/' do
  @posts = Post.all()
  erb :index
end
```

This will render ./views/index.erb:

```
<% for post in @posts %>
  <h1><%= post.title %></h1>
<% end %>
```

Helpers

The basics

It is ill-advised to create helpers on the root level of your application. They muddy the global namespace, and don’t have easy access to the request, response, session or cookie variables.

Instead, use the handy helpers method to install methods on `Sinatra::EventContext` for use inside events and templates.

Example:

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

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

Implemention 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
```

Rack 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
```

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.

Error Handling

Overview

These are run inside the Sinatra::EventContext which means you get all the helpers is has to offer, including template rendering via `haml`, `erb`, calling halt, and using send_file.

not_found

Whenever NotFound is raised this will be called. Using the `not_found` helper in a route *does not* trigger this block.

```
not_found do
  'This is nowhere to be found'
end
```

error

By default error will catch Sinatra::ServerError, but you can customize your error conditions to match your specific application. For example, this can be useful to implement rate limiting in an API.

Sinatra will pass you the specific exception that was raised via the 'sinatra.error' in request.env

```
error do
  'Sorry there was a nasty error - ' + request.env['sinatra.error'].name
end
```

A quick example of a custom error class:

```
# Define an error class
class RateLimitError < Exception
end

# Define a handler for the error class
error RateLimitError do
  'You are over your limit: ' + request.env['sinatra.error'].message
end

get '/' do
  raise RateLimitError, '100 API calls allowed per hour'
end
```

You will see this as the output:

```
You are over your limit: 100 API calls allowed per hour
```

Additional Information

Sinatra gives you default not_found and error handlers in the production environment that are secure (hides application specific error information). If you want to customize the error handlers for the production environment, but leave the friendly Sinatra error pages in Development, then put your error handlers in a configure block.

```
configure :production do
  not_found do
    haml :'404'
  end

  error do
    haml :'500'
  end
end
```

Configuration

Use Sinatra's "set" option

Configure blocks are not executed in the event context, and don't have access to the same instance variables. To store a piece of information that you want to access in your routes, use `set`.

```
configure :development do
```

```
  set :dbname, 'devdb'
end

configure :production do
  set :dbname, 'productiondb'
end
```

…

```
get '/whatdb' do
  'We are using the database named ' + options.dbname
end
```

External config file via the configure block

Application module / config area

Development Techniques

Bundler

Whether you need a specific gem and version, or a set of gems for a certain environment; Bundler is a fantastic tool for managing your applications dependencies.

Gemfiles

Gemfiles are the source of your bundle, used by bundler to determine what gems to install and require in different situations. It's important to understand the difference between the Gemfile and Gemfile.lock; Gemfiles are where you specify the actual gems required by your application, and the Gemfile.lock is a definition of all the required gems and the exact versions used by your application. As it's necessary for other developers to know exactly what versions of third party libraries you're using, the Gemfile.lock is recommended to be checked into source control.

Gemcutter

In most cases you're going to require gems from the official rubygems repository. Here's an example Gemfile for an application that uses Sinatra as a main dependency and RSpec for testing:

```
# define our source to loook for gems
source "http://rubygems.org/"

# declare the sinatra dependency
gem "sinatra"

# setup our test group and require rspec
group :test do
  gem "rspec"
end

# require a relative gem version
gem "i18n", "~> 0.4.1"
```

Git

Bundler also supports the installation of gems through git, so long as the repository contains a valid gemspec for the gem you're trying to install.

```
# lets use sinatra edge
gem "sinatra", :git => "http://github.com/sinatra/sinatra.git"
```

```
# and lets we use the rspec 2.0 release candidate from git
group :test do
  gem "rspec", :git => "http://github.com/rspec/rspec.git",
    :tag => "v2.0.0.rc"
end

# as well as i18n from git
gem "i18n", :git => "http://github.com/svenfuchs/i18n.git"
```

Commands (CLI)

Bundle is the command line utility provided with Bundler to install, update and manage your bundle. Here’s a quick overview of some of the most common commands.

Installing

```
# Install specified gems from your Gemfile and Gemfile.lock
bundle install

# Inspect your bundle to see if you've met your applications requirements
bundle check

# List all gems in your bundle
bundle list

# Show source location of a specific gem in your bundle
bundle show [gemname]

# Generate a skeleton Gemfile to start your path to using Bundler
bundle init
```

Updating

Updating your bundle will look in the given repositories for the latest versions available. This will bypass your Gemfile.lock and check for completely new versions. Alternatively you can specify an individual gem to update, and bundler will only update that gem to the latest version available in the specified repository.

```
# Update all gems specified to the latest versions available
bundle update

# Update just i18n to the latest gem version available
bundle update i18n
```

Requiring

Bundler provides two main ways to use your bundle in your application, `Bundler.setup` and `Bundler.require`. `Setup` basically tells Ruby all of your gems loadpaths, and `require` will load all of your specified gems.

```
# If you're using Ruby 1.9 you'll need to specifially load rubygems
require 'rubygems'

# and now load bundler with your dependencies load paths
require 'bundler'
Bundler.setup

# next you'll have to do the gem requiring yourself
require 'sinatra'
require 'i18n'
```

Now if say you skip the last step, and just auto require gems from your groups

```
require 'rubygems'
require 'bundler'

# this will require all the gems not specified to a given group (default)
# and gems specified in your test group
Bundler.require(:default, :test)
```

Resources

- Bundler's Purpose and Rationale – For a longer explanation of what Bundler does and how it works
- Gemfile Manual
- CLI Manual – Basic command line utilities provided with Bundler
- Gems from Git repositories – Using git repositories with your Gemfile
- Using Groups – Using groups with bundler
- bundle install manual
- bundle update manual
- bundle package manual
- bundle exec manual
- bundle config manual

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.

Sinatra::Reloader

The `sinatra-reloader` gem offers a faster alternative, that also works on Windows and JRuby. It only reloads files that actually have been changed and automatically detects orphaned routes that have to be removed. Most other implementations delete all routes and reload all code if one file changed, which takes way more time than reloading only one file, especially in larger projects.

Install it by running

```
gem install sinatra-reloader
```

If you use the top level DSL, you just have to require it in development mode:

```
require "sinatra"
require "sinatra/reloader" if development?
```

```
get('/') { 'change me!' }
```

When using a modular style application, you have to register the `Sinatra::Reloader` extension:

```
require "sinatra/base"
require "sinatra/reloader"

class MyApp < Sinatra::Base
  configure :development do
    register Sinatra::Reloader
  end
end
```

For safety and performance reason, Sinatra::Reloader will per default only reload files defining routes. You can, however, add files to the list of reloadable files by using `also_reload`:

```
require "sinatra"

configure :development do |config|
  require "sinatra/reloader"
  config.also_reload "models/*.rb"
end
```

Other Tools and Resources

- Magical Reloading Sparkles – similar to Shotgun, but relies on Unicorn and only restarts on file changes.
- Reloading Ruby Code – a blog post explaining how different code reloading techniques work and which one to use.

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
- `gem install heroku`
- Make a config.ru in the root-directory
- Create the app on heroku
- Push to it

1. 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
   ```

2. Create the app and push to it

   ```
   From the root-directory of the application
   ```

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

For more details see this

Nginx Proxied to Unicorn

Nginx and Unicorn combine to provide a very powerful setup for deploying your Sinatra applications. This guide will show you how to effectively setup this combination for deployment.

Installation

First thing you will need to do is get nginx installed on your system. This should be handled by your operating systems package manager.

For more information on installing nginx, check the official docs.

Once you have nginx installed, you can install unicorn with rubygems:

```
gem install unicorn
```

Now that's done we can setup a basic Rack applicaiton in Sinatra.

The Example Application

To start our example application, let's first create a `config.ru` in our application root.

```
require "rubygems"
require "sinatra"

require 'myapp.rb'

run MyApp
```

Let's now use the `myapp.rb` that we specified in our Rack config file as our Sinatra application.

```
require "rubygems"
require "sinatra/base"

class MyApp < Sinatra::Base

  get '/' do
     'Hello, nginx and unicorn!'
  end

end
```

Now that we have our application in place, let's get on to configuring our proxy.

Configuration

So if you've made it this far you should have a Sinatra application ready with nginx and unicorn installed. You're ready to move on to configuring the web server.

Unicorn

Configuring unicorn is really easy and provides an easy to use Ruby DSL for doing so. In our application's root we'll first need to make a couple directories, if you haven't already:

```
mkdir tmp
mkdir tmp/sockets
mkdir tmp/pids
mkdir log
```

Once those are in place, we're ready to setup our `unicorn.rb` configuration.

```
# set path to app that will be used to configure unicorn,
# note the trailing slash in this example
@dir = "/path/to/app/"

worker_processes 2
working_directory @dir

preload_app true

timeout 30

# Specify path to socket unicorn listens to,
# we will use this in our nginx.conf later
listen "#{@dir}tmp/sockets/unicorn.sock", :backlog => 64

# Set process id path
pid "#{@dir}tmp/pids/unicorn.pid"

# Set log file paths
stderr_path "#{@dir}log/unicorn.stderr.log"
stdout_path "#{@dir}log/unicorn.stdout.log"
```

As you can see, unicorn is extremely simple to setup. Let's move onto nginx now, soon enough you'll be well on your way to serving up all kinds of great Rack applications!

nginx

Nginx is a little more difficult to configure than unicorn, but still a fairly straightforward process.

In this example we'll be putting all of our configuration in the `nginx.conf` file of our nginx installation. You could alternatively separate some of the configuration out into `sites-enabled` and other nginx conventions. However, for most common and simple implementations this guide should do the trick.

```
# this sets the user nginx will run as,
#and the number of worker processes
user nobody nogroup;
worker_processes  1;

# setup where nginx will log errors to
# and where the nginx process id resides
error_log  /var/log/nginx/error.log;
pid        /var/run/nginx.pid;

events {
  worker_connections  1024;
  # set to on if you have more than 1 worker_processes
  accept_mutex off;
}

http {
  include       /etc/nginx/mime.types;

  default_type application/octet-stream;
  access_log /tmp/nginx.access.log combined;

  # use the kernel sendfile
  sendfile        on;
  # prepend http headers before sendfile()
  tcp_nopush     on;

  keepalive_timeout  65;
  tcp_nodelay        on;

  gzip  on;
  gzip_disable "MSIE [1-6]\.(?!.*SV1)";
  gzip_types text/plain text/html text/xml text/css
     text/comma-separated-values
     text/javascript application/x-javascript
     application/atom+xml;

  # use the socket we configured in our unicorn.rb
  upstream unicorn_server {
    server unix:/path/to/app/tmp/sockets/unicorn.sock
    fail_timeout=0;
  }

  # configure the virtual host
  server {
    # replace with your domain name
    server_name my-sinatra-app.com;
    # replace this with your static Sinatra app files, root + public
    root /path/to/app/public;
    # port to listen for requests on
    listen 80;
    # maximum accepted body size of client request
    client_max_body_size 4G;
    # the server will close connections after this time
    keepalive_timeout 5;

    location / {
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header Host $http_host;
      proxy_redirect off;
      if (!-f $request_filename) {
        # pass to the upstream unicorn server mentioned above
        proxy_pass http://unicorn_server;
        break;
      }
    }
  }
}
```

Once you replace `path/to/app` with the location to your Sinatra application, you should be able now start your application and web server.

Starting the server

First thing you will need to do is boot up the unicorn processes:

```
unicorn -c path/to/unicorn.rb -E development -D -l 0.0.0.0:3001
```

It's important to note the flags here, `-c` is path to your unicorn configuration. `-E` is the Rack environment for your application to run under. `-D` will daemonize the process, and `-l` is the address which unicorn will listen to.

Lastly, let's start up nginx. On most debian-based systems you can use the following command:

```
/etc/init.d/nginx start
```

However, you should check with your distribution as to where the nginx daemon resides.

Now you should have successfully deployed your Sinatra application on nginx and unicorn.

Stopping the server

So now that you're using nginx and unicorn, at some point you might end up asking yourself: How do I stop this thing?

Here's how:

```
$ ps -ax | grep unicorn
```

This will output the processes running unicorn, in the first column should be the process id (pid). In order to stop unicorn in it's tracks:

```
kill -9 <PID>
```

There should be a `master` process which once that is killed, the workers should follow. Feel free to search the processes again to make sure they've all stopped before restarting.

To stop nginx you can use a similar technique as above, or if you've got the nginx init scripts installed on any debian-based system use:

```
sudo /etc/init.d/nginx stop
```

That should wrap things up for deploying nginx and unicorn, for more information on stopping the server look into `man ps` and `man kill`.

Resources

- unicorn source on github
- original unicorn announcement
- official unicorn homepage
- unicorn rdoc
- nginx official homepage
- nginx wiki

Lighttpd Proxied to Thin

This will cover how to deploy Sinatra to a load balanced reverse proxy setup using Lighttpd and Thin.

1. Install Lighttpd and Thin

   ```
   # Figure out lighttpd yourself, it should be handled by your
   # linux distro's package manager

   # For thin:
   gem install thin
   ```

2. Create your rackup file – the `require 'app'` line should require the actual Sinatra app you have written.

   ```
   ## This is not needed for Thin > 1.0.0
   ENV['RACK_ENV'] = "production"

   require 'app'

   run Sinatra::Application
   ```

3. Setup a config.yml – change the /path/to/my/app path to reflect reality.

   ```
   ---
     environment: production
     chdir: /path/to/my/app
     address: 127.0.0.1
     user: root
     group: root
     port: 4567
     pid: /path/to/my/app/thin.pid
     rackup: /path/to/my/app/config.ru
     log: /path/to/my/app/thin.log
     max_conns: 1024
     timeout: 30
     max_persistent_conns: 512
   ```

```
  daemonize: true
```

4. Setup lighttpd.conf – change mydomain to reflect reality. Also make sure the first port here matches up with the port setting in config.yml.

   ```
   $HTTP["host"] =~ "(www\.)?mydomain\.com"  {
     proxy.balance = "fair"
     proxy.server =  ("/" =>
                        (
                          ( "host" => "127.0.0.1", "port" => 4567 ),
                          ( "host" => "127.0.0.1", "port" => 4568 )
                        )
                      )
   }
   ```

5. Start thin and your application. I have a rake script so I can just call "rake start" rather than typing this in.

   ```
   thin -s 2 -C config.yml -R config.ru start
   ```

You're done! Go to mydomain.com/ and see the result! Everything should be setup now, check it out at the domain you setup in your lighttpd.conf file.

Variation – nginx via proxy – The same approach to proxying can be applied to the nginx web server

```
upstream www_mydomain_com {
  server 127.0.0.1:5000;
  server 127.0.0.1:5001;
}

server {
  listen     www.mydomain.com:80
  server_name  www.mydomain.com live;
  access_log /path/to/logfile.log

  location / {
    proxy_pass http://www_mydomain_com;
  }

}
```

Variation – More Thin instances – To add more thin instances, change the `-s 2` parameter on the thin start command to be how ever many servers you want. Then be sure lighttpd proxies to all of them by adding more lines to the proxy statements. Then restart lighttpd and everything should come up as expected.

Apache and Passenger (mod rails)

Hate deployment via FastCGI? You're not alone. But guess what, Passenger supports Rack; and this book tells you how to get it all going.

You can find additional documentation at the official modrails website.

The easiest way to get started with Passenger is via the gem.

Installation

First you will need to have Apache installed, then you can move onto installing passenger and the apache passenger module.

You have a number of options when installing phusion passenger, however the gem is likely the easiest way to get started.

```
gem install passenger
```

Once you've got that installed you can build the passenger apache module.

```
passenger-install-apache2-module
```

Follow the instructions given by the installer.

Deploying your app

Passenger lets you easily deploy Sinatra apps through the Rack interface.

There are some assumptions made about your application, however, particularly the `tmp` and `public` sub-directories of your application.

In order to fit these prerequisites, simply make sure you have the following setup:

```
mkdir public
mkdir tmp
config.ru
```

The public directory is for serving static files and tmp directory is for the `restart.txt` application restart mechanism. `config.ru` is where you will place your rackup configuration.

Rackup

Once you have these directories in place, you can setup your applications rackup file, `config.ru`.

```
require 'rubygems'
require 'sinatra'
require 'app.rb'

run Sinatra::Application
```

Virtual Host

Next thing you'll have to do is setup the Apache Virtual Host for your app.

```
<VirtualHost *:80>
    ServerName www.yourapplication.com
    DocumentRoot /path/to/app/public
    <Directory /path/to/app/public>
        Allow from all
        Options -MultiViews
    </Directory>
</VirtualHost>
```

That should just about do it for your basic apache and passenger configuration. For more specific information please visit the official modrails documentation.

A note about restarting the server

Once you've got everything configured it's time to restart Apache.

On most debian-based systems you should be able to:

```
sudo apache2ctl stop
# then
sudo apache2ctl start
```

To restart Apache. Check the link above for more detailed information.

In order to restart the Passenger application, all you need to do is run this simple command for your application root:

```
touch tmp/restart.txt
```

You should be up and running now with Phusion Passenger and Apache, if you run into any problems please consult the official docs.

Dreamhost Deployment via Passenger

You can deploy your Sinatra apps to Dreamhost, a shared web hosting service, via Passenger with relative ease. Here's how.

1. Setting up the account in the Dreamhost interface

   ```
   Domains -> Manage Domains -> Edit (web hosting column)
   Enable 'Ruby on Rails Passenger (mod_rails)'
   Add the public directory to the web directory box. So if you were using
   'rails.com', it would change to 'rails.com/public'
   Save your changes
   ```

2. Creating the directory structure

   ```
   domain.com/
   domain.com/tmp
   domain.com/public
   # a vendored version of sinatra - not necessary if you use the gem
   domain.com/sinatra
   ```

3. 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
   ```

4. A very simple Sinatra application

   ```
   # this is myapp.rb referred to above
    require 'sinatra'
    get '/' do
      "Worked on dreamhost"
    end

    get '/foo/:bar' do
      "You asked for foo/#{params[:bar]}"
    end
   ```

And that's all there is to it! Once it's all setup, point your browser at your domain, and you should see a 'Worked on Dreamhost' page. To restart the application after making changes, you need to run `touch tmp/restart.txt`.

Please note that currently passenger 2.0.3 has a bug where it can cause Sinatra to not find the view directory. In that case, add `:views => '/path/to/views/'` to the Sinatra options in your Rackup file.

You may encounter the dreaded "Ruby (Rack) application could not be started" error with this message "can't activate rack (>= 0.9.1, < 1.0, runtime), already activated rack-0.4.0". This happens because DreamHost has version 0.4.0 installed, when recent versions of Sinatra require more recent versions of Rack. The solution is to explicitly require the rack and sinatra gems in your config.ru. Add the following two lines to the start of your config.ru file:

```
   require '/home/USERNAME/.gem/ruby/1.8/gems/rack-VERSION-OF-RACK-GEM-YOU-
HAVE-INSTALLELD/lib/rack.rb'
   require '/home/USERNAME/.gem/ruby/1.8/gems/sinatra-VERSION-OF-SINATRA-GEM-
YOU-HAVE-INSTALLELD/lib/sinatra.rb'
```

FastCGI

The standard method for deployment is to use Thin or Mongrel, and have a reverse proxy (lighttpd, nginx, or even Apache) point to your bundle of servers.

But that isn't always possible. Cheaper shared hosting (like Dreamhost) won't let you run Thin or Mongrel, or setup reverse proxies (at least on the default shared plan).

Luckily, Rack supports various connectors, including CGI and FastCGI. Unluckily for us, FastCGI doesn't quite work with the current Sinatra release without some tweaking.

Deployment with Sinatra version 0.9

From version 0.9.0 Sinatra requires Rack 0.9.1, however FastCGI wrapper from this version seems not working well with Sinatra unless you define your application as a subclass of Sinatra::Application class and run this application directly as a Rack application.

Steps to deploy via FastCGI:

- htaccess
- subclass your application as Sinatra::Application
- dispatch.fcgi

1. .htaccess

   ```
   RewriteEngine on

   AddHandler fastcgi-script .fcgi
   Options +FollowSymLinks +ExecCGI

   RewriteRule ^(.*)$ dispatch.fcgi [QSA,L]
   ```

2. Subclass your application as Sinatra::Application

   ```
   # my_sinatra_app.rb
   class MySinatraApp < Sinatra::Application
     # your sinatra application definitions
   end
   ```

3. dispatch.fcgi – Run this application directly as a Rack application

   ```
   #!/usr/local/bin/ruby

   require 'rubygems'
   require 'rack'

   fastcgi_log = File.open("fastcgi.log", "a")
   STDOUT.reopen fastcgi_log
   STDERR.reopen fastcgi_log
   STDOUT.sync = true

   module Rack
     class Request
       def path_info
         @env["REDIRECT_URL"].to_s
       end
       def path_info=(s)
         @env["REDIRECT_URL"] = s.to_s
       end
     end
   end
   ```

```
load 'my\_sinatra\_app.rb'

builder = Rack::Builder.new do
  map '/' do
    run MySinatraApp.new
  end
end

Rack::Handler::FastCGI.run(builder)
```

Contributing

How can I clone the Sinatra repository?

First of all, you'll need the Git version control system. Git is available for all major platforms:

- Windows
- Mac OS X
- Linux and BSD users can usually acquire Git through their Package Management System, e.g. `apt-get install git-core` on Debian systems.

After that, cloning the Sinatra repository is as easy as typing the following into your command line:

```
git clone git://github.com/sinatra/sinatra.git
```

How to create a patch?

How to get that patch into the official Sinatra?