Sinatra

Frequently Asked Questions

How do I make my Sinatra app reload on changes?

First off, in-process code reloading in Ruby is hard and having a solution that works for every scenario is technically impossible.

Which is why we recommend you to do out-of-process reloading.

First you need to install rerun if you haven’t already:

```
$ gem install rerun
```

Now if you start your Sinatra app like this:

```
$ ruby app.rb
```

All you have to do for reloading is instead do this:

```
$ rerun 'ruby app.rb'
```

If you are for instance using `rackup`, instead do the following:

```
$ rerun 'rackup'
```

You get the idea.

If you still want in-process reloading, check out Sinatra::Reloader.

What are my deployment options?

See the book.

How do I use sessions?

Sessions are disabled by default. You need to enable them and then use the `session` hash from routes and views:

```
enable :sessions

get '/foo' do
  session[:message] = 'Hello World!'
  redirect to('/bar')
end

get '/bar' do
  session[:message]   # => 'Hello World!'
end
```

If you need to set additional parameters for sessions, like expiration date, use `Rack::Session::Cookie` directly instead of `enable :sessions` (example from *Rack* documentation):

```
use Rack::Session::Cookie, :key => 'rack.session',
                           :domain => 'foo.com',
                           :path => '/',
                           :expire_after => 2592000, # In seconds
                           :secret => 'change_me'
```

How do I use session-based flash?

Use Rack::Flash.

Can I run Sinatra under Ruby 1.9?

Yes. As of Sinatra 0.9.2, Sinatra is fully Ruby 1.9 and Rack 1.0 compatible. Since 1.1 you do not have

to deal with encodings on your own, unless you want to.

How do I get the “route” for the current page?

The `request` object probably has what you’re looking for:

```
get '/hello-world' do
  request.path_info   # => '/hello-world'
  request.fullpath    # => '/hello-world?foo=bar'
  request.url         # => 'http://example.com/hello-world?foo=bar'
end
```

See Rack::Request for a detailed list of methods supported by the `request` object.

How do I access helpers from within my views?

Call them! Views automatically have access to all helper methods. In fact, Sinatra evaluates routes, views, and helpers within the same exact object context so they all have access to the same methods and instance variables.

In `hello.rb`:

```
helpers do
  def em(text)
    "<em>#{text}</em>"
  end
end

get '/hello' do
  @subject = 'World'
  haml :hello
end
```

In `views/hello.haml`:

```
%p= "Hello " + em(@subject)
```

How do I render partials?

Sinatra’s template system is simple enough that it can be used for page and fragment level rendering tasks. The `erb` and `haml` methods simply return a string.

Since Sinatra 1.1, you can use the same calls for partials you use in the routes:

```
<%= erb :mypartial %>
```

In versions prior to 1.1, you need to make sure you disable layout rendering as follows:

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

See Sam Elliott and Iain Barnett's Sinatra-Partial extension for a more robust partials implementation. It also supports rendering collections and partials in subdirectories.

The code used to live in a gist, but we have put it in a gem so we can maintain it properly and provide an easier way for developers to include its behaviour. It was adapted from Chris Schneider's original partials.rb implementation.

Use it as follows to render the `mypartial.haml`(1) or the `admin/mypartial.haml`(2) partials, or with a collection (3) & (4):

```
<%= partial(:mypartial) %> <!--(1)-->
<%= partial(:'admin/mypartial') %> <!--(2)-->
<%= partial(:object, :collection => @objects) %> <!--(3)-->
<%= partial(:'admin/object', :collection => @objects) %> <!--(4)-->
```

In (1) & (2), the partial will be rendered plain from their files, with no local variables (specify them with a hash passed into `:locals`). In (3) & (4), the partials will be rendered, populating the local variable `object` with each of the objects from the collection.

It is also possible to enable using underscores, a la Rails, and to choose a different rendering engine from haml.

Can I have multiple URLs trigger the same route/handler?

Sure:

```
["/foo", "/bar", "/baz"].each do |path|
  get path do
    "You've reached me at #{request.path_info}"
  end
end
```

Seriously.

How do I make the trailing slash optional?

Put a question mark after it:

```
get '/foo/bar/?' do
  "Hello World"
end
```

The route matches `"/foo/bar"` and `"/foo/bar/"`.

How do I render templates nested in subdirectories?

Sinatra apps do not typically have a very complex file hierarchy under `views`. First, consider whether you really need subdirectories at all. If so, you can use the `views/foo/bar.haml` file as a template

with:

```
get '/' do
  haml :'foo/bar'
end
```

This is basically the same as sending `#to_sym` to the filename and can also be written as:

```
get '/' do
  haml 'foo/bar'.to_sym
end
```

I'm running Thin and an error occurs but there's no output

Try starting Thin with the `--debug` argument:

```
thin --debug --rackup config.ru start
```

That should give you an exception and backtrace on `stderr`.

How do I send email from Sinatra?

How about a Pony (`sudo gem install pony`):

```
require 'pony'
post '/signup' do
  Pony.mail :to => 'you@example.com',
            :from => 'me@example.com',
            :subject => 'Howdy, Partna!'
end
```

You can even use templates to render the body. In `email.erb`:

```
Good day <%= params[:name] %>,

Thanks for my signing my guestbook. You're a doll.

Frank
```

And in `mailerapp.rb`:

```
post '/guestbook/sign' do
  Pony.mail :to => params[:email],
            :from => "me@example.com",
            :subject => "Thanks for signing my guestbook, #{params[:name]}!",
            :body => erb(:email)
end
```

How do I escape HTML?

Use Rack::Utils in your helpers as follows:

```
helpers do
  def h(text)
    Rack::Utils.escape_html(text)
  end
end
```

Now you can escape HTML in your templates like this:

```
<%= h scary_output %>
```

Thanks to Chris Schneider for the tip!

How do I automatically escape HTML?

Require Erubis and set `escape_html` to `true`:

```
require 'erubis'
set :erb, :escape_html => true
```

Then, any templates rendered with Erubis will be automatically escaped:

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

Read more on the Tilt Google Group and see this example app for details.

How do I use ActiveRecord migrations?

From Adam Wiggins's blog:

> To use ActiveRecord's migrations with Sinatra (or other non-Rails project), add the following to your Rakefile:
>
> ```
> namespace :db do
>   desc "Migrate the database"
>   task(:migrate => :environment) do
>     ActiveRecord::Base.logger = Logger.new(STDOUT)
>     ActiveRecord::Migration.verbose = true
>     ActiveRecord::Migrator.migrate("db/migrate")
>   end
> end
> ```
>
> This assumes you have a task called `:environment` which loads your app's environment (requires the right files, sets up the database connection, etc).

Now you can create a directory called `db/migrate` and fill in your migrations. I usually call the first one `001_init.rb`. (I prefer the old sequential method for numbering migrations vs. the datetime method used since Rails 2.1, but either will work.)

How do I use HTTP authentication?

You have at least two options for implementing basic access authentication (Basic HTTP Auth) in your application.

I. When you want to protect all requests in the application, simply put Rack::Auth::Basic middleware in the request processing chain by the `use` directive:

```
require 'sinatra'

use Rack::Auth::Basic, "Restricted Area" do |username, password|
  username == 'admin' and password == 'admin'
end

get '/' do
  "You're welcome"
end

get '/foo' do
  "You're also welcome"
end
```

II. When you want to protect only certain URLs in the application, or want the authorization to be more complex, you may use something like this:

```
require 'sinatra'

helpers do
  def protected!
    return if authorized?
    headers['WWW-Authenticate'] = 'Basic realm="Restricted Area"'
    halt 401, "Not authorized\n"
  end

  def authorized?
    @auth ||=  Rack::Auth::Basic::Request.new(request.env)
    @auth.provided? and @auth.basic? and @auth.credentials and @auth.credentials ==
['admin', 'admin']
  end
end

get '/' do
  "Everybody can see this page"
end

get '/protected' do
  protected!
  "Welcome, authenticated client"
end
```

How do I test HTTP authentication?

Assuming you have this simple implementation of HTTP authentication in your `application.rb`:

```
require 'sinatra'

use Rack::Auth::Basic do |username, password|
  username == 'admin' and password == 'admin'
end

get '/protected' do
  "You're welcome"
end
```

You can test it like this with *Rack::Test*:

```
ENV['RACK_ENV'] = 'test'
require 'test/unit'
require 'rack/test'

require 'application'

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

  def app
    Sinatra::Application
  end

  def test_without_authentication
    get '/protected'
    assert_equal 401, last_response.status
  end

  def test_with_bad_credentials
    authorize 'bad', 'boy'
    get '/protected'
    assert_equal 401, last_response.status
  end

  def test_with_proper_credentials
    authorize 'admin', 'admin'
    get '/protected'
    assert_equal 200, last_response.status
    assert_equal "You're welcome", last_response.body
  end
end
```