CODE DOCS README CREW CONTRIBUTE ABOUT

HOME CODE DOCS README BLOG CREW ABOUT
CONTRIBUTE SLACK

This page is also available in Chinese, French, German, Hungarian, Korean, Portuguese (Brazilian), Portuguese (European), Russian, Spanish and Japanese.

Getting Started

build passing

Sinatra is a DSL for quickly creating web applications in Ruby with minimal effort:

```
# myapp.rb
require 'sinatra'

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

Install the gem:

```
gem install sinatra
```

And run with:

```
ruby myapp.rb
```

View at: http://localhost:4567

It is recommended to also run `gem install thin`, which Sinatra will pick up if available.

Routes

In Sinatra, a route is an HTTP method paired with a URL-matching pattern. Each route is associated with a block:

```
get '/' do
  .. show something ..
end
```

```

post '/' do
  .. create something ..
end

put '/' do
  .. replace something ..
end

patch '/' do
  .. modify something ..
end

delete '/' do
  .. annihilate something ..
end

options '/' do
  .. appease something ..
end

link '/' do
  .. affiliate something ..
end

unlink '/' do
  .. separate something ..
end
```

Routes are matched in the order they are defined. The first route that matches the request is invoked.

Routes with trailing slashes are different from the ones without:

```
get '/foo' do
  # Does not match "GET /foo/"
end
```

Route patterns may include named parameters, accessible via the `params` hash:

```
get '/hello/:name' do
  # matches "GET /hello/foo" and "GET /hello/bar"
  # params['name'] is 'foo' or 'bar'
  "Hello #{params['name']}!"
end
```

You can also access named parameters via block parameters:

```
get '/hello/:name' do |n|
  # matches "GET /hello/foo" and "GET /hello/bar"
  # params['name'] is 'foo' or 'bar'
  # n stores params['name']
  "Hello #{n}!"
end
```

Route patterns may also include splat (or wildcard) parameters, accessible via the `params['splat']` array:

```
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
```

Or with block parameters:

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

Route matching with Regular Expressions:

```
get /\/hello\/([\w]+)/ do
  "Hello, #{params['captures'].first}!"
end
```

Or with a block parameter:

```
get %r{/hello/([\w]+)} do |c|
  # Matches "GET /meta/hello/world", "GET /hello/world/1234" etc.
  "Hello, #{c}!"
end
```

Route patterns may have optional parameters:

```
get '/posts/:format?' do
  # matches "GET /posts/" and any extension "GET /posts/json", "GET /posts/xml" etc
end
```

Routes may also utilize query parameters:

```
get '/posts' do
  # matches "GET /posts?title=foo&author=bar"
  title = params['title']
  author = params['author']
  # uses title and author variables; query is optional to the /posts route
end
```

By the way, unless you disable the path traversal attack protection (see below), the request path might be modified before matching against your routes.

You may customize the Mustermann options used for a given route by passing in a `:mustermann_opts` hash:

```
get '\A/posts\z', :mustermann_opts => { :type => :regexp, :check_anchors => false } do
  # matches /posts exactly, with explicit anchoring
  "If you match an anchored pattern clap your hands!"
end
```

It looks like a condition, but it isn't one! These options will be merged into the global `:mustermann_opts` hash described below.

Conditions

Routes may include a variety of matching conditions, such as the 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
```

Other available conditions are `host_name` and `provides`:

```
get '/', :host_name => /^admin\./ do
  "Admin Area, Access denied!"
end

get '/', :provides => 'html' do
  haml :index
end

get '/', :provides => ['rss', 'atom', 'xml'] do
  builder :feed
end
```

`provides` searches the request's Accept header.

You can easily define your own conditions:

```
set(:probability) { |value| condition { rand <= value } }
```

```
get '/win_a_car', :probability => 0.1 do
  "You won!"
end

get '/win_a_car' do
  "Sorry, you lost."
end
```

For a condition that takes multiple values use a splat:

```
set(:auth) do |*roles|   # <- notice the splat here
  condition do
    unless logged_in? && roles.any? {|role| current_user.in_role? role }
      redirect "/login/", 303
    end
  end
end

get "/my/account/", :auth => [:user, :admin] do
  "Your Account Details"
end

get "/only/admin/", :auth => :admin do
  "Only admins are allowed here!"
end
```

Return Values

The return value of a route block determines at least the response body passed on to the HTTP client, or at least the next middleware in the Rack stack. Most commonly, this is a string, as in the above examples. But other values are also accepted.

You can return any object that would either be a valid Rack response, Rack body object or HTTP status code:

- An Array with three elements: `[status (Fixnum), headers (Hash), response body (responds to #each)]`
- An Array with two elements: `[status (Fixnum), response body (responds to #each)]`
- An object that responds to `#each` and passes nothing but strings to the given block
- A Fixnum representing the status code

That way we can, for instance, easily implement a streaming example:

```
class Stream
  def each
    100.times { |i| yield "#{i}\n" }
  end
end
```

```
get('/') { Stream.new }
```

You can also use the `stream` helper method (described below) to reduce boiler plate and embed the streaming logic in the route.

Custom Route Matchers

As shown above, Sinatra ships with built-in support for using String patterns and regular expressions as route matches. However, it does not stop there. You can easily define your own matchers:

```
class AllButPattern
  Match = Struct.new(:captures)

  def initialize(except)
    @except   = except
    @captures = Match.new([])
  end

  def match(str)
    @captures unless @except === str
  end
end

def all_but(pattern)
  AllButPattern.new(pattern)
end

get all_but("/index") do
  # ...
end
```

Note that the above example might be over-engineered, as it can also be expressed as:

```
get // do
  pass if request.path_info == "/index"
  # ...
end
```

Or, using negative look ahead:

```
get %r{(?!/index)} do
  # ...
end
```

Static Files

Static files are served from the `./public` directory. You can specify a different location by setting the `:public_folder` option:

```
set :public_folder, File.dirname(__FILE__) + '/static'
```

Note that the public directory name is not included in the URL. A file `./public/css/style.css` is made available as `http://example.com/css/style.css`.

Use the `:static_cache_control` setting (see below) to add `Cache-Control` header info.

Views / Templates

Each template language is exposed via its own rendering method. These methods simply return a string:

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

This renders `views/index.erb`.

Instead of a template name, you can also just pass in the template content directly:

```
get '/' do
  code = "<%= Time.now %>"
  erb code
end
```

Templates take a second argument, the options hash:

```
get '/' do
  erb :index, :layout => :post
end
```

This will render `views/index.erb` embedded in the `views/post.erb` (default is `views/layout.erb`, if it exists).

Any options not understood by Sinatra will be passed on to the template engine:

```
get '/' do
  haml :index, :format => :html5
end
```

You can also set options per template language in general:

```
set :haml, :format => :html5

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

Options passed to the render method override options set via `set`.

Available Options:

locals
: List of locals passed to the document. Handy with partials. Example: `erb "<%= foo %>", :locals => {:foo => "bar"}`

default_encoding
: String encoding to use if uncertain. Defaults to `settings.default_encoding`.

views
: Views folder to load templates from. Defaults to `settings.views`.

layout
: Whether to use a layout (`true` or `false`). If it's a Symbol, specifies what template to use. Example: `erb :index, :layout => !request.xhr?`

content_type
: Content-Type the template produces. Default depends on template language.

scope
: Scope to render template under. Defaults to the application instance. If you change this, instance variables and helper methods will not be available.

layout_engine
: Template engine to use for rendering the layout. Useful for languages that do not support layouts otherwise. Defaults to the engine used for the template. Example: `set :rdoc, :layout_engine => :erb`

layout_options
: Special options only used for rendering the layout. Example: `set :rdoc, :layout_options => { :views => 'views/layouts' }`

Templates are assumed to be located directly under the `./views` directory. To use a different views directory:

```
set :views, settings.root + '/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'` or `'subdir/template'.to_sym`). You must use a symbol because otherwise rendering methods will render any strings passed to them directly.

Literal Templates

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

Renders the template string. You can optionally specify `:path` and `:line` for a clearer backtrace if there is a filesystem path or line associated with that string:

```
get '/' do
  haml '%div.title Hello World', :path => 'examples/file.haml', :line => 3
end
```

Available Template Languages

Some languages have multiple implementations. To specify what implementation to use (and to be thread-safe), you should simply require it first:

```
require 'rdiscount' # or require 'bluecloth'
get('/') { markdown :index }
```

Haml Templates

Dependency	haml
File Extension	`.haml`
Example	`haml :index, :format => :html5`

Erb Templates

Dependency	erubis or erb (included in Ruby)
File Extensions	`.erb`, `.rhtml` or `.erubis` (Erubis only)
Example	`erb :index`

Builder Templates

Dependency	builder
File Extension	`.builder`
Example	`builder { |xml| xml.em "hi" }`

It also takes a block for inline templates (see example).

Nokogiri Templates

Dependency	nokogiri
File Extension	`.nokogiri`
Example	`nokogiri { |xml| xml.em "hi" }`

It also takes a block for inline templates (see example).

Sass Templates

Dependency	sass
File Extension	`.sass`

Example	`sass :stylesheet, :style => :expanded`

SCSS Templates

Dependency	sass
File Extension	`.scss`
Example	`scss :stylesheet, :style => :expanded`

Less Templates

Dependency	less
File Extension	`.less`
Example	`less :stylesheet`

Liquid Templates

Dependency	liquid
File Extension	`.liquid`
Example	`liquid :index, :locals => { :key => 'value' }`

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

Markdown Templates

Dependency	Anyone of: RDiscount, RedCarpet, BlueCloth, kramdown, maruku
File Extensions	`.markdown`, `.mkd` and `.md`
Example	`markdown :index, :layout_engine => :erb`

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)
```

Since you cannot call Ruby from Markdown, you cannot use layouts written in Markdown. However, it is possible to use another rendering engine for the template than for the layout by passing the `:layout_engine` option.

Textile Templates

Dependency	RedCloth
File Extension	`.textile`
Example	`textile :index, :layout_engine => :erb`

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)
```

Since you cannot call Ruby from Textile, you cannot use layouts written in Textile. However, it is possible to use another rendering engine for the template than for the layout by passing the `:layout_engine` option.

RDoc Templates

Dependency	RDoc
File Extension	`.rdoc`
Example	`rdoc :README, :layout_engine => :erb`

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)
```

Since you cannot call Ruby from RDoc, you cannot use layouts written in RDoc. However, it is possible to use another rendering engine for the template than for the layout by passing the `:layout_engine` option.

AsciiDoc Templates

Dependency	Asciidoctor
File Extension	`.asciidoc`, `.adoc` and `.ad`
Example	`asciidoc :README, :layout_engine => :erb`

Since you cannot call Ruby methods directly from an AsciiDoc template, you almost always want to pass locals to it.

Radius Templates

Dependency	Radius
File Extension	`.radius`
Example	`radius :index, :locals => { :key => 'value' }`

Since you cannot call Ruby methods directly from a Radius template, you almost always want to pass locals to it.

Markaby Templates

Dependency	Markaby
File Extension	`.mab`
Example	`markaby { h1 "Welcome!" }`

It also takes a block for inline templates (see example).

RABL Templates

Dependency	Rabl
File Extension	`.rabl`
Example	`rabl :index`

Slim Templates

Dependency	Slim Lang
File Extension	`.slim`
Example	`slim :index`

Creole Templates

Dependency	Creole
File Extension	`.creole`
Example	`creole :wiki, :layout_engine => :erb`

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

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

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

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

Since you cannot call Ruby from Creole, you cannot use layouts written in Creole. However, it is possible to use another rendering engine for the template than for the layout by passing the `:layout_engine` option.

MediaWiki Templates

Dependency	WikiCloth
File Extension	`.mediawiki` and `.mw`
Example	`mediawiki :wiki, :layout_engine => :erb`

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

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

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

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

Since you cannot call Ruby from MediaWiki, you cannot use layouts written in MediaWiki. However, it is possible to use another rendering engine for the template than for the layout by passing the `:layout_engine` option.

CoffeeScript Templates

Dependency	CoffeeScript and a way to execute javascript
File Extension	`.coffee`
Example	`coffee :index`

Stylus Templates

Dependency	Stylus and a way to execute javascript
File Extension	`.styl`
Example	`stylus :index`

Before being able to use Stylus templates, you need to load `stylus` and `stylus/tilt` first:

```
require 'sinatra'
require 'stylus'
require 'stylus/tilt'

get '/' do
  stylus :example
end
```

Yajl Templates

Dependency	yajl-ruby
File Extension	`.yajl`
Example	`yajl :index, :locals => { :key => 'qux' }, :callback => 'present', :variable => 'resource'`

The template source is evaluated as a Ruby string, and the resulting json variable is converted using `#to_json`:

```
json = { :foo => 'bar' }
json[:baz] = key
```

The `:callback` and `:variable` options can be used to decorate the rendered object:

```
var resource = {"foo":"bar","baz":"qux"};
present(resource);
```

WLang Templates

Dependency	WLang
File Extension	`.wlang`
Example	`wlang :index, :locals => { :key => 'value' }`

Since calling ruby methods is not idiomatic in WLang, you almost always want to pass locals to it. Layouts written in WLang and `yield` are supported, though.

Accessing Variables in Templates

Templates are evaluated within the same context as route handlers. Instance variables set in route handlers are directly 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= bar.name', :locals => { :bar => foo }
end
```

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

Templates with `yield` and nested layouts

A layout is usually just a template that calls `yield`. Such a template can be used either through the `:template` option as described above, or it can be rendered with a block as follows:

```
erb :post, :layout => false do
  erb :index
end
```

This code is mostly equivalent to `erb :index, :layout => :post`.

Passing blocks to rendering methods is most useful for creating nested layouts:

```
erb :main_layout, :layout => false do
  erb :admin_layout do
    erb :user
  end
end
```

This can also be done in fewer lines of code with:

```
erb :admin_layout, :layout => :main_layout do
  erb :user
end
```

Currently, the following rendering methods accept a block: `erb`, `haml`, `liquid`, `slim` , `wlang`. Also the general `render` method accepts a block.

Inline Templates

Templates may be defined at the end of the source file:

```
require 'sinatra'

get '/' do
  haml :index
end

__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 individually disable layouts by passing `:layout => false` or disable them by default via `set :haml, :layout => false`:

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

Associating File Extensions

To associate a file extension with a template engine, use `Tilt.register`. For instance, if you like to use the file extension `tt` for Textile templates, you can do the following:

```
Tilt.register :tt, Tilt[:textile]
```

Adding Your Own Template Engine

First, register your engine with Tilt, then create a rendering method:

```
Tilt.register :myat, MyAwesomeTemplateEngine

helpers do
  def myat(*args) render(:myat, *args) end
end

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

Renders `./views/index.myat`. See https://github.com/rtomayko/tilt to learn more about Tilt.

Using Custom Logic for Template Lookup

To implement your own template lookup mechanism you can write your own `#find_template` method:

```
configure do
  set :views [ './views/a', './views/b' ]
end

def find_template(views, name, engine, &block)
  Array(views).each do |v|
    super(v, name, engine, &block)
  end
end
```

Filters

Before filters are evaluated before each request within the same context as the routes will be and can modify the request and response. Instance variables set in filters are accessible by routes and templates:

```
before do
  @note = 'Hi!'
  request.path_info = '/foo/bar/baz'
end

get '/foo/*' do
  @note #=> 'Hi!'
  params['splat'] #=> 'bar/baz'
end
```

After filters are evaluated after each request within the same context as the routes will be and can also modify the request and response. Instance variables set in before filters and routes are accessible by after filters:

```
after do
  puts response.status
end
```

Note: Unless you use the `body` method rather than just returning a String from the routes, the body will not yet be available in the after filter, since it is generated later on.

Filters optionally take 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
```

Like routes, filters also take conditions:

```
before :agent => /Songbird/ do
  # ...
end

after '/blog/*', :host_name => 'example.com' do
  # ...
end
```

Helpers

Use the top-level `helpers` method to define helper methods for use in route handlers and templates:

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

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

Alternatively, helper methods can be separately defined in a module:

```
module FooUtils
  def foo(name) "#{name}foo" end
end

module BarUtils
  def bar(name) "#{name}bar" end
end

helpers FooUtils, BarUtils
```

The effect is the same as including the modules in the application class.

Using Sessions

A session is used to keep state during requests. If activated, you have one session hash per user session:

```
enable :sessions
```

```
get '/' do
  "value = " << session[:value].inspect
end

get '/:value' do
  session['value'] = params['value']
end
```

Session Secret Security

To improve security, the session data in the cookie is signed with a session secret using `HMAC-SHA1`. This session secret should optimally be a cryptographically secure random value of an appropriate length which for `HMAC-SHA1` is greater than or equal to 64 bytes (512 bits, 128 hex characters). You would be advised not to use a secret that is less than 32 bytes of randomness (256 bits, 64 hex characters). It is therefore **very important** that you don’t just make the secret up, but instead use a secure random number generator to create it. Humans are extremely bad at generating random values.

By default, a 32 byte secure random session secret is generated for you by Sinatra, but it will change with every restart of your application. If you have multiple instances of your application, and you let Sinatra generate the key, each instance would then have a different session key which is probably not what you want.

For better security and usability it’s recommended that you generate a secure random secret and store it in an environment variable on each host running your application so that all of your application instances will share the same secret. You should periodically rotate this session secret to a new value. Here are some examples of how you might create a 64 byte secret and set it:

Session Secret Generation

```
$ ruby -e "require 'securerandom'; puts SecureRandom.hex(64)"
99ae8af...snip...ec0f262ac
```

Session Secret Generation (Bonus Points)

Use the sysrandom gem to prefer use of system RNG facilities to generate random values instead of userspace `OpenSSL` which MRI Ruby currently defaults to:

```
$ gem install sysrandom
Building native extensions.  This could take a while...
Successfully installed sysrandom-1.x
1 gem installed

$ ruby -e "require 'sysrandom/securerandom'; puts SecureRandom.hex(64)"
99ae8af...snip...ec0f262ac
```

Session Secret Environment Variable

Set a `SESSION_SECRET` environment variable for Sinatra to the value you generated. Make this value persistent across reboots of your host. Since the method for doing this will vary across systems this is for illustrative purposes only:

```
# echo "export SESSION_SECRET=99ae8af...snip...ec0f262ac" >> ~/.bashrc
```

Session Secret App Config

Setup your app config to fail-safe to a secure random secret if the `SESSION_SECRET` environment variable is not available.

For bonus points use the sysrandom gem here as well:

```
require 'securerandom'
# -or- require 'sysrandom/securerandom'
set :session_secret, ENV.fetch('SESSION_SECRET') { SecureRandom.hex(64) }
```

Session Config

If you want to configure it further, you may also store a hash with options in the `sessions` setting:

```
set :sessions, :domain => 'foo.com'
```

To share your session across other apps on subdomains of foo.com, prefix the domain with a *.* like this instead:

```
set :sessions, :domain => '.foo.com'
```

Choosing Your Own Session Middleware

Note that `enable :sessions` actually stores all data in a cookie. This might not always be what you want (storing lots of data will increase your traffic, for instance). You can use any Rack session middleware in order to do so, one of the following methods can be used:

```
enable :sessions
set :session_store, Rack::Session::Pool
```

Or to set up sessions with a hash of options:

```
set :sessions, :expire_after => 2592000
set :session_store, Rack::Session::Pool
```

Another option is to **not** call `enable :sessions`, but instead pull in your middleware of choice as you would any other middleware.

It is important to note that when using this method, session based protection **will not be enabled by default**.

The Rack middleware to do that will also need to be added:

```
use Rack::Session::Pool, :expire_after => 2592000
use Rack::Protection::RemoteToken
use Rack::Protection::SessionHijacking
```

See ‘Configuring attack protection’ for more information.

Halting

To immediately stop a request within a filter or route use:

```
halt
```

You can also specify the status when halting:

```
halt 410
```

Or the body:

```
halt 'this will be the body'
```

Or both:

```
halt 401, 'go away!'
```

With headers:

```
halt 402, {'Content-Type' => 'text/plain'}, 'revenge'
```

It is of course possible to combine a template with `halt`:

```
halt erb(:error)
```

Passing

A route can punt processing to the next matching route using `pass`:

```
get '/guess/:who' do
  pass unless params['who'] == 'Frank'
  'You got me!'
end

get '/guess/*' do
  'You missed!'
end
```

The route block is immediately exited and control continues with the next matching route. If no matching route is found, a 404 is returned.

Triggering Another Route

Sometimes `pass` is not what you want, instead you would like to get the result of calling another route. Simply use `call` to achieve this:

```
get '/foo' do
  status, headers, body = call env.merge("PATH_INFO" => '/bar')
  [status, headers, body.map(&:upcase)]
end

get '/bar' do
  "bar"
end
```

Note that in the example above, you would ease testing and increase performance by simply moving `"bar"` into a helper used by both `/foo` and `/bar`.

If you want the request to be sent to the same application instance rather than a duplicate, use `call!` instead of `call`.

Check out the Rack specification if you want to learn more about `call`.

Setting Body, Status Code and Headers

It is possible and recommended to set the status code and response body with the return value of the route block. However, in some scenarios you might want to set the body at an arbitrary point in the execution flow. You can do so with the `body` helper method. If you do so, you can use that method from there on to access the body:

```
get '/foo' do
  body "bar"
end

after do
  puts body
end
```

It is also possible to pass a block to `body`, which will be executed by the Rack handler (this can be used to implement streaming, see “Return Values”).

Similar to the body, you can also set the status code and headers:

```
get '/foo' do
  status 418
  headers \
    "Allow"   => "BREW, POST, GET, PROPFIND, WHEN",
    "Refresh" => "Refresh: 20; http://www.ietf.org/rfc/rfc2324.txt"
  body "I'm a tea pot!"
end
```

Like body, headers and status with no arguments can be used to access their current values.

Streaming Responses

Sometimes you want to start sending out data while still generating parts of the response body. In extreme examples, you want to keep sending data until the client closes the connection. You can use the stream helper to avoid creating your own wrapper:

```
get '/' do
  stream do |out|
    out << "It's gonna be legen -\n"
    sleep 0.5
    out << " (wait for it) \n"
    sleep 1
    out << "- dary!\n"
  end
end
```

This allows you to implement streaming APIs, Server Sent Events, and can be used as the basis for WebSockets. It can also be used to increase throughput if some but not all content depends on a slow resource.

Note that the streaming behavior, especially the number of concurrent requests, highly depends on the web server used to serve the application. Some servers might not even support streaming at all. If the server does not support streaming, the body will be sent all at once after the block passed to stream finishes executing. Streaming does not work at all with Shotgun.

If the optional parameter is set to keep_open, it will not call close on the stream object, allowing you to close it at any later point in the execution flow. This only works on evented servers, like Thin and Rainbows. Other servers will still close the stream:

```
# long polling

set :server, :thin
connections = []

get '/subscribe' do
  # register a client's interest in server events
  stream(:keep_open) do |out|
    connections << out
    # purge dead connections
    connections.reject!(&:closed?)
  end
end

post '/:message' do
  connections.each do |out|
    # notify client that a new message has arrived
    out << params['message'] << "\n"
    # indicate client to connect again
    out.close
  end

  # acknowledge
  "message received"
end
```

It’s also possible for the client to close the connection when trying to write to the socket. Because of this, it’s recommended to check `out.closed?` before trying to write.

Logging

In the request scope, the `logger` helper exposes a `Logger` instance:

```
get '/' do
  logger.info "loading data"
  # ...
end
```

This logger will automatically take your Rack handler’s logging settings into account. If logging is disabled, this method will return a dummy object, so you do not have to worry about it in your routes and filters.

Note that logging is only enabled for `Sinatra::Application` by default, so if you inherit from `Sinatra::Base`, you probably want to enable it yourself:

```
class MyApp < Sinatra::Base
  configure :production, :development do
    enable :logging
  end
end
```

To avoid any logging middleware to be set up, set the `logging` setting to `nil`. However, keep in mind that `logger` will in that case return `nil`. A common use case is when you want to set your own logger. Sinatra will use whatever it will find in `env['rack.logger']`.

Mime Types

When using `send_file` or static files you may have mime types Sinatra doesn’t understand. Use `mime_type` to register them by file extension:

```
configure do
  mime_type :foo, 'text/foo'
end
```

You can also use it with the `content_type` helper:

```
get '/' do
  content_type :foo
  "foo foo foo"
end
```

Generating URLs

For generating URLs you should use the `url` helper method, for instance, in Haml:

```
%a{:href => url('/foo')} foo
```

It takes reverse proxies and Rack routers into account, if present.

This method is also aliased to `to` (see below for an example).

Browser Redirect

You can trigger a browser redirect with the `redirect` helper method:

```
get '/foo' do
  redirect to('/bar')
end
```

Any additional parameters are handled like arguments passed to `halt`:

```
redirect to('/bar'), 303
redirect 'http://www.google.com/', 'wrong place, buddy'
```

You can also easily redirect back to the page the user came from with `redirect back`:

```
get '/foo' do
  "<a href='/bar'>do something</a>"
end

get '/bar' do
  do_something
  redirect back
end
```

To pass arguments with a redirect, either add them to the query:

```
redirect to('/bar?sum=42')
```

Or use a session:

```
enable :sessions
```

```
get '/foo' do
  session[:secret] = 'foo'
  redirect to('/bar')
end

get '/bar' do
  session[:secret]
end
```

Cache Control

Setting your headers correctly is the foundation for proper HTTP caching.

You can easily set the Cache-Control header like this:

```
get '/' do
  cache_control :public
  "cache it!"
end
```

Pro tip: Set up caching in a before filter:

```
before do
  cache_control :public, :must_revalidate, :max_age => 60
end
```

If you are using the `expires` helper to set the corresponding header, `Cache-Control` will be set automatically for you:

```
before do
  expires 500, :public, :must_revalidate
end
```

To properly use caches, you should consider using `etag` or `last_modified`. It is recommended to call those helpers *before* doing any heavy lifting, as they will immediately flush a response if the client already has the current version in its cache:

```
get "/article/:id" do
  @article = Article.find params['id']
  last_modified @article.updated_at
  etag @article.sha1
  erb :article
end
```

It is also possible to use a weak ETag:

```
etag @article.sha1, :weak
```

These helpers will not do any caching for you, but rather feed the necessary information to your cache. If you are looking for a quick reverse-proxy caching solution, try rack-cache:

```
require "rack/cache"
require "sinatra"

use Rack::Cache

get '/' do
  cache_control :public, :max_age => 36000
  sleep 5
  "hello"
end
```

Use the `:static_cache_control` setting (see below) to add `Cache-Control` header info to static files.

According to RFC 2616, your application should behave differently if the If-Match or If-None-Match header is set to `*`, depending on whether the resource requested is already in existence. Sinatra assumes resources for safe (like get) and idempotent (like put) requests are already in existence, whereas other resources (for instance post requests) are treated as new resources. You can change this behavior by passing in a `:new_resource` option:

```
get '/create' do
  etag '', :new_resource => true
  Article.create
  erb :new_article
end
```

If you still want to use a weak ETag, pass in a `:kind` option:

```
etag '', :new_resource => true, :kind => :weak
```

Sending Files

To return the contents of a file as the response, you can use the `send_file` helper method:

```
get '/' do
  send_file 'foo.png'
end
```

It also takes options:

```
send_file 'foo.png', :type => :jpg
```

The options are:

filename
: File name to be used in the response, defaults to the real file name.

last_modified
    Value for Last-Modified header, defaults to the file's mtime.

type
    Value for Content-Type header, guessed from the file extension if missing.

disposition
    Value for Content-Disposition header, possible values: `nil` (default), `:attachment` and `:inline`

length
    Value for Content-Length header, defaults to file size.

status
    Status code to be sent. Useful when sending a static file as an error page. If supported by the Rack handler, other means than streaming from the Ruby process will be used. If you use this helper method, Sinatra will automatically handle range requests.

Accessing the Request Object

The incoming request object can be accessed from request level (filter, routes, error handlers) through the `request` method:

```
# app running on http://example.com/example
get '/foo' do
  t = %w[text/css text/html application/javascript]
  request.accept              # ['text/html', '*/*']
  request.accept? 'text/xml'  # true
  request.preferred_type(t)   # 'text/html'
  request.body                # request body sent by the client (see below)
  request.scheme              # "http"
  request.script_name         # "/example"
  request.path_info           # "/foo"
  request.port                # 80
  request.request_method      # "GET"
  request.query_string        # ""
  request.content_length      # length of request.body
  request.media_type          # media type of request.body
  request.host                # "example.com"
  request.get?                # true (similar methods for other verbs)
  request.form_data?          # false
  request["some_param"]       # value of some_param parameter. [] is a shortcut to the params hash.
  request.referrer            # the referrer of the client or '/'
  request.user_agent          # user agent (used by :agent condition)
  request.cookies             # hash of browser cookies
  request.xhr?                # is this an ajax request?
  request.url                 # "http://example.com/example/foo"
  request.path                # "/example/foo"
  request.ip                  # client IP address
  request.secure?             # false (would be true over ssl)
  request.forwarded?          # true (if running behind a reverse proxy)
  request.env                 # raw env hash handed in by Rack
end
```

Some options, like `script_name` or `path_info`, can also be written:

```
before { request.path_info = "/" }

get "/" do
  "all requests end up here"
end
```

The `request.body` is an IO or StringIO object:

```
post "/api" do
  request.body.rewind  # in case someone already read it
  data = JSON.parse request.body.read
  "Hello #{data['name']}!"
end
```

Attachments

You can use the `attachment` helper to tell the browser the response should be stored on disk rather than displayed in the browser:

```
get '/' do
  attachment
  "store it!"
end
```

You can also pass it a file name:

```
get '/' do
  attachment "info.txt"
  "store it!"
end
```

Dealing with Date and Time

Sinatra offers a `time_for` helper method that generates a Time object from the given value. It is also able to convert `DateTime`, `Date` and similar classes:

```
get '/' do
  pass if Time.now > time_for('Dec 23, 2016')
  "still time"
end
```

This method is used internally by `expires`, `last_modified` and akin. You can therefore easily extend the behavior of those methods by overriding `time_for` in your application:

```
helpers do
  def time_for(value)
    case value
    when :yesterday then Time.now - 24*60*60
    when :tomorrow  then Time.now + 24*60*60
    else super
    end
  end
end

get '/' do
  last_modified :yesterday
  expires :tomorrow
  "hello"
end
```

Looking Up Template Files

The `find_template` helper is used to find template files for rendering:

```
find_template settings.views, 'foo', Tilt[:haml] do |file|
  puts "could be #{file}"
end
```

This is not really useful. But it is useful that you can actually override this method to hook in your own lookup mechanism. For instance, if you want to be able to use more than one view directory:

```
set :views, ['views', 'templates']

helpers do
  def find_template(views, name, engine, &block)
    Array(views).each { |v| super(v, name, engine, &block) }
  end
end
```

Another example would be using different directories for different engines:

```
set :views, :sass => 'views/sass', :haml => 'templates', :default => 'views'

helpers do
  def find_template(views, name, engine, &block)
    _, folder = views.detect { |k,v| engine == Tilt[k] }
    folder ||= views[:default]
    super(folder, name, engine, &block)
  end
end
```

You can also easily wrap this up in an extension and share with others!

Note that `find_template` does not check if the file really exists but rather calls the given block for all possible paths. This is not a performance issue, since `render` will use `break` as soon as a file is found. Also, template locations (and content) will be cached if you are not running in development mode. You should keep that in mind if you write a really crazy method.

Configuration

Run once, at startup, in any environment:

```
configure do
  # setting one option
  set :option, 'value'

  # setting multiple options
  set :a => 1, :b => 2

  # same as `set :option, true`
  enable :option

  # same as `set :option, false`
  disable :option

  # you can also have dynamic settings with blocks
  set(:css_dir) { File.join(views, 'css') }
end
```

Run only when the environment (`APP_ENV` environment variable) is set to `:production`:

```
configure :production do
  ...
end
```

Run when the environment is set to either `:production` or `:test`:

```
configure :production, :test do
  ...
end
```

You can access those options via `settings`:

```
configure do
  set :foo, 'bar'
end

get '/' do
  settings.foo? # => true
  settings.foo  # => 'bar'
  ...
end
```

Configuring attack protection

Sinatra is using Rack::Protection to defend your application against common, opportunistic attacks. You can easily disable this behavior (which will open up your application to tons of common vulnerabilities):

```
disable :protection
```

To skip a single defense layer, set `protection` to an options hash:

```
set :protection, :except => :path_traversal
```

You can also hand in an array in order to disable a list of protections:

```
set :protection, :except => [:path_traversal, :session_hijacking]
```

By default, Sinatra will only set up session based protection if `:sessions` have been enabled. See ‘Using Sessions’. Sometimes you may want to set up sessions “outside” of the Sinatra app, such as in the config.ru or with a separate `Rack::Builder` instance. In that case you can still set up session based protection by passing the `:session` option:

```
set :protection, :session => true
```

Available Settings

absolute_redirects

If disabled, Sinatra will allow relative redirects, however, Sinatra will no longer conform with RFC 2616 (HTTP 1.1), which only allows absolute redirects.
Enable if your app is running behind a reverse proxy that has not been set up properly. Note that the `url` helper will still produce absolute URLs, unless you pass in `false` as the second parameter.
Disabled by default.

add_charset

Mime types the `content_type` helper will automatically add the charset info to. You should add to it rather than overriding this option: `settings.add_charset << "application/foobar"`

app_file

Path to the main application file, used to detect project root, views and public folder and inline templates.

bind

IP address to bind to (default: `0.0.0.0` *or* `localhost` if your `environment` is set to development). Only used for built-in server.

default_encoding

Encoding to assume if unknown (defaults to `"utf-8"`).

dump_errors

Display errors in the log.

environment
: Current environment. Defaults to `ENV['APP_ENV']`, or `"development"` if not available.

logging
: Use the logger.

lock
: Places a lock around every request, only running processing on request per Ruby process concurrently. Enabled if your app is not thread-safe. Disabled by default.

method_override
: Use `_method` magic to allow put/delete forms in browsers that don't support it.

mustermann_opts
: A default hash of options to pass to Mustermann.new when compiling routing paths.

port
: Port to listen on. Only used for built-in server.

prefixed_redirects
: Whether or not to insert `request.script_name` into redirects if no absolute path is given. That way `redirect '/foo'` would behave like `redirect to('/foo')`. Disabled by default.

protection
: Whether or not to enable web attack protections. See protection section above.

public_dir
: Alias for `public_folder`. See below.

public_folder
: Path to the folder public files are served from. Only used if static file serving is enabled (see `static` setting below). Inferred from `app_file` setting if not set.

quiet
: Disables logs generated by Sinatra's start and stop commands. `false` by default.

reload_templates
: Whether or not to reload templates between requests. Enabled in development mode.

root
: Path to project root folder. Inferred from `app_file` setting if not set.

raise_errors
: Raise exceptions (will stop application). Enabled by default when `environment` is set to `"test"`, disabled otherwise.

run
: If enabled, Sinatra will handle starting the web server. Do not enable if using rackup or other means.

running
: Is the built-in server running now? Do not change this setting!

server
: Server or list of servers to use for built-in server. Order indicates priority, default depends on Ruby implementation.

sessions
: Enable cookie-based sessions support using `Rack::Session::Cookie`. See 'Using Sessions' section for more information.

session_store
: The Rack session middleware used. Defaults to `Rack::Session::Cookie`. See 'Using Sessions' section for more information.

show_exceptions
: Show a stack trace in the browser when an exception happens. Enabled by default when `environment` is set to `"development"`, disabled otherwise.
: Can also be set to `:after_handler` to trigger app-specified error handling before showing a stack trace

in the browser.

static

Whether Sinatra should handle serving static files.
Disable when using a server able to do this on its own.
Disabling will boost performance.
Enabled by default in classic style, disabled for modular apps.

static_cache_control

When Sinatra is serving static files, set this to add `Cache-Control` headers to the responses. Uses the `cache_control` helper. Disabled by default.
Use an explicit array when setting multiple values: `set :static_cache_control, [:public, :max_age => 300]`

threaded

If set to `true`, will tell Thin to use `EventMachine.defer` for processing the request.

traps

Whether Sinatra should handle system signals.

views

Path to the views folder. Inferred from `app_file` setting if not set.

x_cascade

Whether or not to set the X-Cascade header if no route matches. Defaults to `true`.

Environments

There are three predefined `environments`: `"development"`, `"production"` and `"test"`. Environments can be set through the `APP_ENV` environment variable. The default value is `"development"`. In the `"development"` environment all templates are reloaded between requests, and special `not_found` and `error` handlers display stack traces in your browser. In the `"production"` and `"test"` environments, templates are cached by default.

To run different environments, set the `APP_ENV` environment variable:

```
APP_ENV=production ruby my_app.rb
```

You can use predefined methods: `development?`, `test?` and `production?` to check the current environment setting:

```
get '/' do
  if settings.development?
    "development!"
  else
    "not development!"
  end
end
```

Error Handling

Error handlers run within the same context as routes and before filters, which means you get all the goodies it has to offer, like `haml`, `erb`, `halt`, etc.

Not Found

When a `Sinatra::NotFound` exception is raised, or the response's status code is 404, the `not_found` handler is invoked:

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

Error

The `error` handler is invoked any time an exception is raised from a route block or a filter. But note in development it will only run if you set the show exceptions option to `:after_handler`:

```
set :show_exceptions, :after_handler
```

The exception object can be obtained from the `sinatra.error` Rack variable:

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

Custom errors:

```
error MyCustomError do
  'So what happened was...' + env['sinatra.error'].message
end
```

Then, if this happens:

```
get '/' do
  raise MyCustomError, 'something bad'
end
```

You get this:

```
So what happened was... something bad
```

Alternatively, you can install an error handler for a status code:

```
error 403 do
  'Access forbidden'
end
```

```
get '/secret' do
  403
end
```

Or a range:

```
error 400..510 do
  'Boom'
end
```

Sinatra installs special `not_found` and `error` handlers when running under the development environment to display nice stack traces and additional debugging information in your browser.

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 these components automatically based on configuration so you typically don't have to `use` them explicitly.

You can find useful middleware in rack, rack-contrib, or in the Rack wiki.

Testing

Sinatra tests can be written using any Rack-based testing library or framework. Rack::Test is recommended:

```
require 'my_sinatra_app'
require 'minitest/autorun'
require 'rack/test'

class MyAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_my_default
    get '/'
    assert_equal 'Hello World!', last_response.body
  end

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

  def test_with_user_agent
    get '/', {}, 'HTTP_USER_AGENT' => 'Songbird'
    assert_equal "You're using Songbird!", last_response.body
  end
end
```

Note: If you are using Sinatra in the modular style, replace `Sinatra::Application` above with the class name of your app.

Sinatra::Base - Middleware, Libraries, and Modular Apps

Defining your app at the top-level works well for micro-apps but has considerable drawbacks when building reusable components such as Rack middleware, Rails metal, simple libraries with a server component, or even Sinatra extensions. The top-level assumes a micro-app style configuration (e.g., a single application file, `./public` and `./views` directories, logging, exception detail page, etc.). That’s where `Sinatra::Base` comes into play:

```
require 'sinatra/base'

class MyApp < Sinatra::Base
  set :sessions, true
  set :foo, 'bar'

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

The methods available to `Sinatra::Base` subclasses are exactly the same as those available via the top-level DSL. Most top-level apps can be converted to `Sinatra::Base` components with two modifications:

- Your file should require `sinatra/base` instead of `sinatra`; otherwise, all of Sinatra's DSL methods are imported into the main namespace.
- Put your app's routes, error handlers, filters, and options in a subclass of `Sinatra::Base`.

`Sinatra::Base` is a blank slate. Most options are disabled by default, including the built-in server. See Configuring Settings for details on available options and their behavior. If you want behavior more similar to when you define your app at the top level (also known as Classic style), you can subclass `Sinatra::Application`:

```
require 'sinatra/base'

class MyApp < Sinatra::Application
  get '/' do
    'Hello world!'
  end
end
```

Modular vs. Classic Style

Contrary to common belief, there is nothing wrong with the classic style. If it suits your application, you do not have to switch to a modular application.

The main disadvantage of using the classic style rather than the modular style is that you will only have one Sinatra application per Ruby process. If you plan to use more than one, switch to the modular style. There is no reason you cannot mix the modular and the classic styles.

If switching from one style to the other, you should be aware of slightly different default settings:

Setting	Classic	Modular	Modular
app_file	file loading sinatra	file subclassing Sinatra::Base	file subclassing Sinatra::Application
run	$0 == app_file	false	false
logging	true	false	true
method_override	true	false	true
inline_templates	true	false	true
static	true	File.exist?(public_folder)	true

Serving a Modular Application

There are two common options for starting a modular app, actively starting with `run!`:

```
# my_app.rb
require 'sinatra/base'

class MyApp < Sinatra::Base
  # ... app code here ...

  # start the server if ruby file executed directly
  run! if app_file == $0
end
```

Start with:

```
ruby my_app.rb
```

Or with a `config.ru` file, which allows using any Rack handler:

```
# config.ru (run with rackup)
require './my_app'
run MyApp
```

Run:

```
rackup -p 4567
```

Using a Classic Style Application with a config.ru

Write your app file:

```
# app.rb
require 'sinatra'

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

And a corresponding `config.ru`:

```
require './app'
run Sinatra::Application
```

When to use a config.ru?

A `config.ru` file is recommended if:

- You want to deploy with a different Rack handler (Passenger, Unicorn, Heroku, …).

- You want to use more than one subclass of `Sinatra::Base`.
- You want to use Sinatra only for middleware, and not as an endpoint.

There is no need to switch to a `config.ru` simply because you switched to the modular style, and you don’t have to use the modular style for running with a `config.ru`.

Using Sinatra as Middleware

Not only is Sinatra able to use other Rack middleware, any Sinatra application can in turn be added in front of any Rack endpoint as middleware itself. This endpoint could be another Sinatra application, or any other Rack-based application (Rails/Hanami/Roda/…):

```
require 'sinatra/base'

class LoginScreen < Sinatra::Base
  enable :sessions

  get('/login') { haml :login }

  post('/login') do
    if params['name'] == 'admin' && params['password'] == 'admin'
      session['user_name'] = params['name']
    else
      redirect '/login'
    end
  end
end

class MyApp < Sinatra::Base
  # middleware will run before filters
  use LoginScreen

  before do
    unless session['user_name']
      halt "Access denied, please <a href='/login'>login</a>."
    end
  end

  get('/') { "Hello #{session['user_name']}." }
end
```

Dynamic Application Creation

Sometimes you want to create new applications at runtime without having to assign them to a constant. You can do this with `Sinatra.new`:

```
require 'sinatra/base'
my_app = Sinatra.new { get('/') { "hi" } }
```

```
my_app.run!
```

It takes the application to inherit from as an optional argument:

```
# config.ru (run with rackup)
require 'sinatra/base'

controller = Sinatra.new do
  enable :logging
  helpers MyHelpers
end

map('/a') do
  run Sinatra.new(controller) { get('/') { 'a' } }
end

map('/b') do
  run Sinatra.new(controller) { get('/') { 'b' } }
end
```

This is especially useful for testing Sinatra extensions or using Sinatra in your own library.

This also makes using Sinatra as middleware extremely easy:

```
require 'sinatra/base'

use Sinatra do
  get('/') { ... }
end

run RailsProject::Application
```

Scopes and Binding

The scope you are currently in determines what methods and variables are available.

Application/Class Scope

Every Sinatra application corresponds to a subclass of `Sinatra::Base`. If you are using the top-level DSL (`require 'sinatra'`), then this class is `Sinatra::Application`, otherwise it is the subclass you created explicitly. At class level you have methods like `get` or `before`, but you cannot access the `request` or `session` objects, as there is only a single application class for all requests.

Options created via `set` are methods at class level:

```
class MyApp < Sinatra::Base
  # Hey, I'm in the application scope!
  set :foo, 42
  foo # => 42

  get '/foo' do
    # Hey, I'm no longer in the application scope!
  end
end
```

You have the application scope binding inside:

- Your application class body
- Methods defined by extensions
- The block passed to `helpers`
- Procs/blocks used as value for `set`
- The block passed to `Sinatra.new`

You can reach the scope object (the class) like this:

- Via the object passed to configure blocks (`configure { |c| ... }`)
- `settings` from within the request scope

Request/Instance Scope

For every incoming request, a new instance of your application class is created, and all handler blocks run in that scope. From within this scope you can access the `request` and `session` objects or call rendering methods like `erb` or `haml`. You can access the application scope from within the request scope via the `settings` helper:

```
class MyApp < Sinatra::Base
  # Hey, I'm in the application scope!
  get '/define_route/:name' do
    # Request scope for '/define_route/:name'
    @value = 42

    settings.get("/#{params['name']}") do
      # Request scope for "/#{params['name']}"
      @value # => nil (not the same request)
    end

    "Route defined!"
  end
end
```

You have the request scope binding inside:

- get, head, post, put, delete, options, patch, link and unlink blocks
- before and after filters
- helper methods
- templates/views

Delegation Scope

The delegation scope just forwards methods to the class scope. However, it does not behave exactly like the class scope, as you do not have the class binding. Only methods explicitly marked for delegation are available, and you do not share variables/state with the class scope (read: you have a different `self`). You can explicitly add method delegations by calling `Sinatra::Delegator.delegate :method_name`.

You have the delegate scope binding inside:

- The top level binding, if you did `require "sinatra"`
- An object extended with the `Sinatra::Delegator` mixin

Have a look at the code for yourself: here's the Sinatra::Delegator mixin being extending the main object.

Command Line

Sinatra applications can be run directly:

```
ruby myapp.rb [-h] [-x] [-q] [-e ENVIRONMENT] [-p PORT] [-o HOST] [-s HANDLER]
```

Options are:

```
-h # help
-p # set the port (default is 4567)
-o # set the host (default is 0.0.0.0)
-e # set the environment (default is development)
-s # specify rack server/handler (default is thin)
-q # turn on quiet mode for server (default is off)
-x # turn on the mutex lock (default is off)
```

Multi-threading

Paraphrasing from this StackOverflow answer by Konstantin

Sinatra doesn't impose any concurrency model, but leaves that to the underlying Rack handler (server) like Thin, Puma or WEBrick. Sinatra itself is thread-safe, so there won't be any problem if the Rack handler uses a threaded model of concurrency. This would mean that when starting the server, you'd have to specify the correct invocation method for the specific Rack handler. The following example is a demonstration of how to start a multi-threaded Thin server:

```
# app.rb

require 'sinatra/base'

class App < Sinatra::Base
  get '/' do
    "Hello, World"
  end
end

App.run!
```

To start the server, the command would be:

```
thin --threaded start
```

Requirement

The following Ruby versions are officially supported:

Ruby 2.2

2.2 is fully supported and recommended. There are currently no plans to drop official support for it.

Rubinius

Rubinius is officially supported (Rubinius >= 2.x). It is recommended to `gem install puma`.

JRuby

The latest stable release of JRuby is officially supported. It is not recommended to use C extensions with JRuby. It is recommended to `gem install trinidad`.

Versions of Ruby prior to 2.2.2 are no longer supported as of Sinatra 2.0.

We also keep an eye on upcoming Ruby versions.

The following Ruby implementations are not officially supported but still are known to run Sinatra:

- Older versions of JRuby and Rubinius
- Ruby Enterprise Edition
- MacRuby, Maglev, IronRuby
- Ruby 1.9.0 and 1.9.1 (but we do recommend against using those)

Not being officially supported means if things only break there and not on a supported platform, we assume it’s not our issue but theirs.

We also run our CI against ruby-head (future releases of MRI), but we can’t guarantee anything, since it is constantly moving. Expect upcoming 2.x releases to be fully supported.

Sinatra should work on any operating system supported by the chosen Ruby implementation.

If you run MacRuby, you should `gem install control_tower`.

Sinatra currently doesn’t run on Cardinal, SmallRuby, BlueRuby or any Ruby version prior to 2.2.

The Bleeding Edge

If you would like to use Sinatra's latest bleeding-edge code, feel free to run your application against the master branch, it should be rather stable.

We also push out prerelease gems from time to time, so you can do a

```
gem install sinatra --pre
```

to get some of the latest features.

With Bundler

If you want to run your application with the latest Sinatra, using Bundler is the recommended way.

First, install bundler, if you haven't:

```
gem install bundler
```

Then, in your project directory, create a `Gemfile`:

```
source 'https://rubygems.org'
gem 'sinatra', :github => 'sinatra/sinatra'

# other dependencies
gem 'haml'                    # for instance, if you use haml
```

Note that you will have to list all your application's dependencies in the `Gemfile`. Sinatra's direct dependencies (Rack and Tilt) will, however, be automatically fetched and added by Bundler.

Now you can run your app like this:

```
bundle exec ruby myapp.rb
```

Versioning

Sinatra follows Semantic Versioning, both SemVer and SemVerTag.

Further Reading

- Project Website - Additional documentation, news, and links to other resources.
- Contributing - Find a bug? Need help? Have a patch?
- Issue tracker
- Twitter
- Mailing List
- IRC: #sinatra on http://freenode.net
- Sinatra & Friends on Slack and see here for an invite.

- Sinatra Book Cookbook Tutorial
- Sinatra Recipes Community contributed recipes
- API documentation for the latest release or the current HEAD on http://www.rubydoc.info/
- CI server