**[[](http://www.sinatrarb.com/)Sinatra](http://www.sinatrarb.com/)**

* [README](http://www.sinatrarb.com/intro.html)
* [Documentation](http://www.sinatrarb.com/documentation.html)
* [Blog](http://www.sinatrarb.com/blog.html)
* [Contribute](http://www.sinatrarb.com/contributing.html)
* [Crew](http://github.com/sinatra/sinatra/contributors)
* [Code](http://github.com/sinatra/sinatra)
* [About](http://www.sinatrarb.com/about.html)

This page is also available in [Chinese](http://www.sinatrarb.com/intro-zh.html), [French](http://www.sinatrarb.com/intro-fr.html), [German](http://www.sinatrarb.com/intro-de.html), [Hungarian](http://www.sinatrarb.com/intro-hu.html), [Korean](http://www.sinatrarb.com/intro-ko.html), [Portuguese (Brazilian)](http://www.sinatrarb.com/intro-pt-br.html), [Portuguese (European)](http://www.sinatrarb.com/intro-pt-pt.html), [Russian](http://www.sinatrarb.com/intro-ru.html), [Spanish](http://www.sinatrarb.com/intro-es.html) and [Japanese](http://www.sinatrarb.com/intro-jp.html).

**Getting Started**

1. [Routes](http://www.sinatrarb.com/intro.html#Routes)
2. [Conditions](http://www.sinatrarb.com/intro.html#Conditions)
3. [Return Values](http://www.sinatrarb.com/intro.html#Return%20Values)
4. [Custom Route Matchers](http://www.sinatrarb.com/intro.html#Custom%20Route%20Matchers)
5. [Static Files](http://www.sinatrarb.com/intro.html#Static%20Files)
6. [Views / Templates](http://www.sinatrarb.com/intro.html#Views%20/%20Templates)
   1. [Literal Templates](http://www.sinatrarb.com/intro.html#Literal%20Templates)
   2. [Available Template Languages](http://www.sinatrarb.com/intro.html#Available%20Template%20Languages)
      1. [Haml Templates](http://www.sinatrarb.com/intro.html#Haml%20Templates)
      2. [Erb Templates](http://www.sinatrarb.com/intro.html#Erb%20Templates)
      3. [Builder Templates](http://www.sinatrarb.com/intro.html#Builder%20Templates)
      4. [Nokogiri Templates](http://www.sinatrarb.com/intro.html#Nokogiri%20Templates)
      5. [Sass Templates](http://www.sinatrarb.com/intro.html#Sass%20Templates)
      6. [SCSS Templates](http://www.sinatrarb.com/intro.html#SCSS%20Templates)
      7. [Less Templates](http://www.sinatrarb.com/intro.html#Less%20Templates)
      8. [Liquid Templates](http://www.sinatrarb.com/intro.html#Liquid%20Templates)
      9. [Markdown Templates](http://www.sinatrarb.com/intro.html#Markdown%20Templates)
      10. [Textile Templates](http://www.sinatrarb.com/intro.html#Textile%20Templates)
      11. [RDoc Templates](http://www.sinatrarb.com/intro.html#RDoc%20Templates)
      12. [Radius Templates](http://www.sinatrarb.com/intro.html#Radius%20Templates)
      13. [Markaby Templates](http://www.sinatrarb.com/intro.html#Markaby%20Templates)
      14. [RABL Templates](http://www.sinatrarb.com/intro.html#RABL%20Templates)
      15. [Slim Templates](http://www.sinatrarb.com/intro.html#Slim%20Templates)
      16. [Creole Templates](http://www.sinatrarb.com/intro.html#Creole%20Templates)
      17. [CoffeeScript Templates](http://www.sinatrarb.com/intro.html#CoffeeScript%20Templates)
      18. [Stylus Templates](http://www.sinatrarb.com/intro.html#Stylus%20Templates)
      19. [Yajl Templates](http://www.sinatrarb.com/intro.html#Yajl%20Templates)
      20. [WLang Templates](http://www.sinatrarb.com/intro.html#WLang%20Templates)
   3. [Accessing Variables in Templates](http://www.sinatrarb.com/intro.html#Accessing%20Variables%20in%20Templates)
   4. [Templates with yield and nested layouts](http://www.sinatrarb.com/intro.html#Templates%20with%20%3Ccode%3Eyield%3C/code%3E%20and%20nested%20layouts)
   5. [Inline Templates](http://www.sinatrarb.com/intro.html#Inline%20Templates)
   6. [Named Templates](http://www.sinatrarb.com/intro.html#Named%20Templates)
   7. [Associating File Extensions](http://www.sinatrarb.com/intro.html#Associating%20File%20Extensions)
   8. [Adding Your Own Template Engine](http://www.sinatrarb.com/intro.html#Adding%20Your%20Own%20Template%20Engine)
7. [Filters](http://www.sinatrarb.com/intro.html#Filters)
8. [Helpers](http://www.sinatrarb.com/intro.html#Helpers)
   1. [Using Sessions](http://www.sinatrarb.com/intro.html#Using%20Sessions)
   2. [Halting](http://www.sinatrarb.com/intro.html#Halting)
   3. [Passing](http://www.sinatrarb.com/intro.html#Passing)
   4. [Triggering Another Route](http://www.sinatrarb.com/intro.html#Triggering%20Another%20Route)
   5. [Setting Body, Status Code and Headers](http://www.sinatrarb.com/intro.html#Setting%20Body,%20Status%20Code%20and%20Headers)
   6. [Streaming Responses](http://www.sinatrarb.com/intro.html#Streaming%20Responses)
   7. [Logging](http://www.sinatrarb.com/intro.html#Logging)
   8. [Mime Types](http://www.sinatrarb.com/intro.html#Mime%20Types)
   9. [Generating URLs](http://www.sinatrarb.com/intro.html#Generating%20URLs)
   10. [Browser Redirect](http://www.sinatrarb.com/intro.html#Browser%20Redirect)
   11. [Cache Control](http://www.sinatrarb.com/intro.html#Cache%20Control)
   12. [Sending Files](http://www.sinatrarb.com/intro.html#Sending%20Files)
   13. [Accessing the Request Object](http://www.sinatrarb.com/intro.html#Accessing%20the%20Request%20Object)
   14. [Attachments](http://www.sinatrarb.com/intro.html#Attachments)
   15. [Dealing with Date and Time](http://www.sinatrarb.com/intro.html#Dealing%20with%20Date%20and%20Time)
   16. [Looking Up Template Files](http://www.sinatrarb.com/intro.html#Looking%20Up%20Template%20Files)
9. [Configuration](http://www.sinatrarb.com/intro.html#Configuration)
   1. [Configuring attack protection](http://www.sinatrarb.com/intro.html#Configuring%20attack%20protection)
   2. [Available Settings](http://www.sinatrarb.com/intro.html#Available%20Settings)
10. [Environments](http://www.sinatrarb.com/intro.html#Environments)
11. [Error Handling](http://www.sinatrarb.com/intro.html#Error%20Handling)
    1. [Not Found](http://www.sinatrarb.com/intro.html#Not%20Found)
    2. [Error](http://www.sinatrarb.com/intro.html#Error)
12. [Rack Middleware](http://www.sinatrarb.com/intro.html#Rack%20Middleware)
13. [Testing](http://www.sinatrarb.com/intro.html#Testing)
14. [Sinatra::Base - Middleware, Libraries, and Modular Apps](http://www.sinatrarb.com/intro.html#Sinatra::Base%20-%20Middleware,%20Libraries,%20and%20Modular%20Apps)
    1. [Modular vs. Classic Style](http://www.sinatrarb.com/intro.html#Modular%20vs.%20Classic%20Style)
    2. [Serving a Modular Application](http://www.sinatrarb.com/intro.html#Serving%20a%20Modular%20Application)
    3. [Using a Classic Style Application with a config.ru](http://www.sinatrarb.com/intro.html#Using%20a%20Classic%20Style%20Application%20with%20a%20config.ru)
    4. [When to use a config.ru?](http://www.sinatrarb.com/intro.html#When%20to%20use%20a%20config.ru?)
    5. [Using Sinatra as Middleware](http://www.sinatrarb.com/intro.html#Using%20Sinatra%20as%20Middleware)
    6. [Dynamic Application Creation](http://www.sinatrarb.com/intro.html#Dynamic%20Application%20Creation)
15. [Scopes and Binding](http://www.sinatrarb.com/intro.html#Scopes%20and%20Binding)
    1. [Application/Class Scope](http://www.sinatrarb.com/intro.html#Application/Class%20Scope)
    2. [Request/Instance Scope](http://www.sinatrarb.com/intro.html#Request/Instance%20Scope)
    3. [Delegation Scope](http://www.sinatrarb.com/intro.html#Delegation%20Scope)
16. [Command Line](http://www.sinatrarb.com/intro.html#Command%20Line)
17. [Requirement](http://www.sinatrarb.com/intro.html#Requirement)
18. [The Bleeding Edge](http://www.sinatrarb.com/intro.html#The%20Bleeding%20Edge)
    1. [With Bundler](http://www.sinatrarb.com/intro.html#With%20Bundler)
    2. [Roll Your Own](http://www.sinatrarb.com/intro.html#Roll%20Your%20Own)
    3. [Install Globally](http://www.sinatrarb.com/intro.html#Install%20Globally)
19. [Versioning](http://www.sinatrarb.com/intro.html#Versioning)
20. [Further Reading](http://www.sinatrarb.com/intro.html#Further%20Reading)

Sinatra is a [DSL](http://en.wikipedia.org/wiki/Domain-specific_language) 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.

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 %r{/hello/([\w]+)} **do**

"Hello, **#{**params[:captures].first**}**!"

**end**

Or with a block parameter:

get %r{/hello/([\w]+)} **do** |c|

"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**

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

**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**

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.

**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](http://haml.info/) |
| File Extension | .haml |
| Example | haml :index, :format => :html5 |

**Erb Templates**

|  |  |
| --- | --- |
| Dependency | [erubis](http://www.kuwata-lab.com/erubis/) or erb (included in Ruby) |
| File Extensions | .erb, .rhtml or .erubis (Erubis only) |
| Example | erb :index |

**Builder Templates**

|  |  |
| --- | --- |
| Dependency | [builder](http://builder.rubyforge.org/) |
| File Extension | .builder |
| Example | builder { |xml| xml.em "hi" } |

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

**Nokogiri Templates**

|  |  |
| --- | --- |
| Dependency | [nokogiri](http://nokogiri.org/) |
| File Extension | .nokogiri |
| Example | nokogiri { |xml| xml.em "hi" } |

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

**Sass Templates**

|  |  |
| --- | --- |
| Dependency | [sass](http://sass-lang.com/) |
| File Extension | .sass |
| Example | sass :stylesheet, :style => :expanded |

**SCSS Templates**

|  |  |
| --- | --- |
| Dependency | [sass](http://sass-lang.com/) |
| File Extension | .scss |
| Example | scss :stylesheet, :style => :expanded |

**Less Templates**

|  |  |
| --- | --- |
| Dependency | [less](http://www.lesscss.org/) |
| File Extension | .less |
| Example | less :stylesheet |

**Liquid Templates**

|  |  |
| --- | --- |
| Dependency | [liquid](http://www.liquidmarkup.org/) |
| 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](https://github.com/rtomayko/rdiscount), [RedCarpet](https://github.com/vmg/redcarpet), [BlueCloth](http://deveiate.org/projects/BlueCloth), [kramdown](http://kramdown.rubyforge.org/), [maruku](http://maruku.rubyforge.org/) |
| 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](http://redcloth.org/) |
| 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](http://rdoc.rubyforge.org/) |
| 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.

**Radius Templates**

|  |  |
| --- | --- |
| Dependency | [Radius](http://radius.rubyforge.org/) |
| 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](http://markaby.github.com/) |
| File Extension | .mab |
| Example | markaby { h1 "Welcome!" } |

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

**RABL Templates**

|  |  |
| --- | --- |
| Dependency | [Rabl](https://github.com/nesquena/rabl) |
| File Extension | .rabl |
| Example | rabl :index |

**Slim Templates**

|  |  |
| --- | --- |
| Dependency | [Slim Lang](http://slim-lang.com/) |
| File Extension | .slim |
| Example | slim :index |

**Creole Templates**

|  |  |
| --- | --- |
| Dependency | [Creole](https://github.com/minad/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.

**CoffeeScript Templates**

|  |  |
| --- | --- |
| Dependency | [CoffeeScript](https://github.com/josh/ruby-coffee-script) and a [way to execute javascript](https://github.com/sstephenson/execjs/blob/master/README.md#readme) |
| File Extension | .coffee |
| Example | coffee :index |

**Stylus Templates**

|  |  |
| --- | --- |
| Dependency | [Stylus](https://github.com/lucasmazza/ruby-stylus) and a [way to execute javascript](https://github.com/sstephenson/execjs/blob/master/README.md#readme) |
| 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](https://github.com/brianmario/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](https://github.com/blambeau/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 method 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.

**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 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**

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, do **not** call enable :sessions, but instead pull in your middleware of choice as you would any other middleware:

use **Rack**::**Session**::**Pool**, :expire\_after => 2592000

get '/' **do**

"value = " << session[:value].inspect

**end**

get '/:value' **do**

session[:value] = params[:value]

**end**

To improve security, the session data in the cookie is signed with a session secret. A random secret is generated for you by Sinatra. However, since this secret will change with every start of your application, you might want to set the secret yourself, so all your application instances share it:

set :session\_secret, 'super secret'

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

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

**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](http://dev.w3.org/html5/eventsource/) and can be used as the basis for [WebSockets](http://en.wikipedia.org/wiki/WebSocket). 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, like WEBRick, 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) { |out| connections << out }

# purge dead connections

connections.reject!(&:closed?)

# acknowledge

"subscribed"

**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**

**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 in your routes and filters about it.

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://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](http://en.wikipedia.org/wiki/HTTP_ETag#Strong_and_weak_validation):

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](https://github.com/rtomayko/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 for 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**

For sending files, 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, in response, defaults to the real file name.

last\_modified

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

type

content type to use, guessed from the file extension if missing.

disposition

used for Content-Disposition, possible value: nil (default), :attachment and :inline

length

Content-Length header, defaults to file size.

status

Status code to be send. 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, 2012')

"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 (RACK\_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](https://github.com/rkh/rack-protection#readme) 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 has been enabled. Sometimes you want to set up sessions on your own, though. In that case you can get it to set up session based protections by passing the :session option:

use **Rack**::**Session**::**Pool**

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\_charsets

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\_charsets << "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['RACK\_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 per default.

method\_override

use \_method magic to allow put/delete forms in browsers that don't support it.

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 per 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.

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.

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 per 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.

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 RACK\_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 RACK\_ENV environment variable:

RACK\_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. The exception object can be obtained from the sinatra.error Rack variable:

error **do**

'Sorry there was a nasty error - ' + env['sinatra.error'].name

**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](http://rack.rubyforge.org/), 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](http://rack.rubyforge.org/doc/classes/Rack/Builder.html) 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](https://github.com/rack/rack/tree/master/lib/rack), [rack-contrib](https://github.com/rack/rack-contrib#readm), with [CodeRack](http://coderack.org/) or in the [Rack wiki](https://github.com/rack/rack/wiki/List-of-Middleware).

**Testing**

Sinatra tests can be written using any Rack-based testing library or framework. [Rack::Test](http://rdoc.info/github/brynary/rack-test/master/frames) is recommended:

require 'my\_sinatra\_app'

require 'test/unit'

require 'rack/test'

**class** **MyAppTest** < **Test**::**Unit**::**TestCase**

include **Rack**::**Test**::**Methods**

**def** **app**

**Sinatra**::**Application**

**end**

**def** **test\_my\_default**

get '/'

assert\_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\_rack\_env**

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 [Options and Configuration](http://sinatra.github.com/configuration.html) for details on available options and their behavior.

**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** |
| app\_file | file loading sinatra | file subclassing Sinatra::Base |
| run | $0 == app\_file | false |
| logging | true | false |
| method\_override | true | false |
| inline\_templates | true | false |
| static | true | false |

**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/Ramaze/Camping/…):

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](https://github.com/sinatra/sinatra/blob/ca06364/lib/sinatra/base.rb#L1609-1633) being [extending the main object](https://github.com/sinatra/sinatra/blob/ca06364/lib/sinatra/main.rb#L28-30).

**Command Line**

Sinatra applications can be run directly:

ruby myapp.rb [-h] [-x] [-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)

-x # turn on the mutex lock (default is off)

**Requirement**

The following Ruby versions are officially supported:

Ruby 1.8.7

1.8.7 is fully supported, however, if nothing is keeping you from it, we recommend upgrading or switching to JRuby or Rubinius. Support for 1.8.7 will not be dropped before Sinatra 2.0. Ruby 1.8.6 is no longer supported.

Ruby 1.9.2

1.9.2 is fully supported. Do not use 1.9.2p0, as it is known to cause segmentation faults when running Sinatra. Official support will continue at least until the release of Sinatra 1.5.

Ruby 1.9.3

1.9.3 is fully supported and recommended. Please note that switching to 1.9.3 from an earlier version will invalidate all sessions. 1.9.3 will be supported until the release of Sinatra 2.0.

Ruby 2.0.0

2.0.0 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 recommendended 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.

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 (the upcoming 2.1.0), but we can’t guarantee anything, since it is constantly moving. Expect 2.1.0 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 1.8.7.

**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](http://gembundler.com/) 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

gem 'activerecord', '~> 3.0' # maybe you also need ActiveRecord 3.x

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

**Roll Your Own**

Create a local clone and run your app with the sinatra/lib directory on the $LOAD\_PATH:

cd myapp

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

ruby -I sinatra/lib myapp.rb

To update the Sinatra sources in the future:

cd myapp/sinatra

git pull

**Install Globally**

You can build the gem on your own:

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

cd sinatra

rake sinatra.gemspec

rake install

If you install gems as root, the last step should be

sudo rake install

**Versioning**

Sinatra follows [Semantic Versioning](http://semver.org/), both SemVer and SemVerTag.

**Further Reading**

* [Project Website](http://www.sinatrarb.com/) - Additional documentation, news, and links to other resources.
* [Contributing](http://www.sinatrarb.com/contributing) - Find a bug? Need help? Have a patch?
* [Issue tracker](http://github.com/sinatra/sinatra/issues)
* [Twitter](http://twitter.com/sinatra)
* [Mailing List](http://groups.google.com/group/sinatrarb/topics)
* IRC: [#sinatra](irc://chat.freenode.net/#sinatra) on http://freenode.net
* [Sinatra Book](http://sinatra-book.gittr.com) Cookbook Tutorial
* [Sinatra Recipes](http://recipes.sinatrarb.com/) Community contributed recipes
* API documentation for the [latest release](http://rubydoc.info/gems/sinatra) or the [current HEAD](http://rubydoc.info/github/sinatra/sinatra) on http://rubydoc.info
* [CI server](http://travis-ci.org/sinatra/sinatra)