Sinatra

Put this in your pipe

```
require 'sinatra'

get '/hi' do
  "Hello World!"
end
```

and smoke it

```
$ gem install sinatra
$ ruby hi.rb
== Sinatra has taken the
stage ...
>> Listening on 0.0.0.0:4567
```

Sinatra

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

Getting Started

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.

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, #{        :captures        }!"
end
```

Or with a block parameter:

```
get %r{/hello/([\w]+)} do |c|
  "Hello, #{ }!"
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 #{        :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 "#{ }\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
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.

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.

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 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)
    "#{      }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) "#{      }foo" end
end

module BarUtils
  def bar(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 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, 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:

```
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 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 #{    }"
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 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, 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, with CodeRack 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 '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 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 being extending the main object.

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

Documentation

README

A whirlwind tour of Sinatra's most interesting features.

Configuring Settings

Detailed documentation on all of Sinatra's built-in settings and using `set`, `enable`, and `disable` to configure them.

Testing Sinatra with Rack::Test

Guide to testing Sinatra apps.

Sinatra::Contrib

A collection of common Sinatra extensions, that tries to include the stuff developers ask for a lot and isn't part of Sinatra.

Using Extensions

How to use Sinatra extensions in your application and links to extensions in the wild.

Writing Extensions

How to add new functionality to Sinatra using the extension APIs.

Project Information

Frequently Asked Questions

Answers to those questions most frequently asked on the mailing list and in `#sinatra`.

Release Notes

See the CHANGES file included for release notes about each release:

- 1.4.3 June 7, 2013
- 1.4.2 March 21, 2013
- 1.4.1 March 15, 2013
- 1.4.0 March 15, 2013
- 1.3.6 March 15, 2013
- 1.3.5 February 25, 2013
- 1.3.4 January 26, 2013
- 1.3.3 August 19, 2012
- 1.3.2 December 30, 2011
- 1.3.1 October 4, 2011
- 1.3.0 September 30, 2011
- 1.2.9 March 15, 2013
- 1.2.8 December 30, 2011
- 1.2.7 September 30, 2011
- 1.2.6 May 1, 2011
- 1.2.5 April 30, 2011
- 1.2.4 April 30, 2011
- 1.2.3 April 13, 2011
- 1.2.2 April 08, 2011
- 1.2.1 March 17, 2011
- 1.2.0 March 03, 2011
- 1.1.3 February 20, 2011
- 1.1.2 December 25, 2010
- 1.1.0 October 24, 2010
- 1.0.0 March 23, 2010
- 0.9.6 March 07, 2010
- 0.9.5 March 05, 2010
- 0.9.4 July 26, 2009
- 0.9.3 June 08, 2009
- 0.9.2 March 18, 2009
- 0.9.1 March 02, 2009
- 0.9.0 January 18, 2009
- 0.3.3 November 2, 2008
- 0.3.2 November 2, 2008
- 0.3.1 September 8, 2008
- 0.3.0 August 31, 2008
- 0.2.2 April 15, 2008
- 0.2.1 April 15, 2008
- 0.2.0 April 11, 2008
- 0.1.7 October 23, 2007
- 0.1.6 October 15, 2007
- 0.1.5 October 7, 2007
- 0.1.0 October 4, 2007
- 0.0.1 September 09, 2007

API Documentation

RDoc documentation courtesy of `rubydoc.info`.

In the Wild

List of applications, libraries, websites and companies using Sinatra.

The Sinatra Book

An in-depth look at building and deploying Sinatra applications. Maintained by Chris Schneider and Zachary Scott.

Sinatra Recipes

Community contributed recipes and tutorials for Sinatra.

Screencasts and Presentations

Sinatra, rack and middleware

Ben Schwarz presents Sinatra and his realisations of its inner workings in regard to rack and rack middlware at Melbourne RORO shortly after Railsconf (US).

RubyConf 08: Lightweight Web Services

Adam Wiggins and Blake Mizerany present Sinatra and RestClient at RubyConf 2008. The talk details Sinatra's underlying philosophy and reflects on using Sinatra to build real world applications.

Meet Sinatra (PeepCode)

Dan Benjamin introduces Sinatra in an hour-long screencast. Build an ad server with DataMapper, JavaScript, HAML, and Sinatra. In collaboration with Sinatra creator Blake Mizerany. Only $9.

Classy Web Development with Sinatra (Prag's Screencast Series)

Adam Keys and The Pragmatic Programmers have started a series of screencasts on Sinatra. The first two episodes cover creating a tiny web app and creating a REST service. *$5 a pop.*

Sinatra at Locos x Rails, in Buenos Aires

By Nicolás Sanguinetti, April 2009.

Introduction to Sinatra (screencasts.org)

The first episode in screencasts.org's Sinatra series.

An introduction to sinatra as a rest server from WebNet Conference

Sinatra autopsy by Aleksander Dabrowski

Around The Web

RubyInside's 29 Links and Resources

Peter Cooper's compendium with links to tutorials, example applications, and presentations.

Chris Schneider's Blog: GITTR

Christopher's blog is a treasure-trove of useful Sinatra related information.

Using Compass for CSS in your Sinatra application

Sinatra

Contribute

Want to show Sinatra some love? Help out by contributing!

Find a bug?

Log it in our issue tracker or send a note to the mailing list. Be sure to include all relevant information, like the versions of Sinatra and Ruby you're using. A gist of the code that caused the issue as well as any error messages are also very helpful.

Need help?

The Sinatra mailing list has over 900 subscribers, many of which are happy to help out newbies or talk about potential feature additions. You can also drop by the #sinatra channel on irc.freenode.net.

Have a patch?

Bugs and feature requests that include patches are much more likely to get attention. Here are some guidelines that will help ensure your patch can be applied as quickly as possible:

1. **Use Git and GitHub:** The easiest way to get setup is to fork the sinatra/sinatra repo. Or, the sinatra.github.com repo, if the patch is doc related.

2. **Write unit tests:** If you add or modify functionality, it must include unit tests. If you don't write tests, we have to, and this can hold up acceptance of the patch.

3. **Mind the `README`:** If the patch adds or modifies a major feature, modify the `README.md` file to reflect that. Again, if you don't update the `README`, we have to, and this holds up acceptance.

4. **Push it:** Once you're ready, push your changes to a topic branch and add a note to the ticket with the URL to your branch. Or, say something like, "you can find the patch on johndoe/foobranch". We also gladly accept GitHub pull requests.

NOTE: *we will take whatever we can get.* If you prefer to attach diffs in emails to the mailing list, that's fine; but do know that *someone* will need to take the diff through the process described above and this can hold things up considerably.

Want to write docs?

The process for contributing to Sinatra's website, documentation or the book is the same as contributing code. We use git for versions control and GitHub to track patch requests.

- The sinatra.github.com repo is where the website sources are managed. There are almost always people in `#sinatra` that are happy to discuss, apply, and publish website patches.
- The Book has its own git repository and build process but is managed the same as the website and project codebase.
- Sinatra Recipes is a community project where anyone is free to contribute ideas, recipes and tutorials. Which also has its own git repository.
- The Introduction is generated from Sinatra's README file.
- If you want to help translating the documentation, there already is a Japanese and a German version of the README, which tend to fall behind the English version. Translations into other languages would also be appreciated.

Looking for something to do?

If you'd like to help out but aren't sure how, pick something that looks interesting from the issues list and hack on. Make sure to leave a comment on the ticket noting that you're investigating (a simple "Taking…" is fine).

Sinatra

About

Sinatra was designed and developed by **Blake Mizerany** (bmizerany) in California.

Sinatra would not have been possible without strong company backing. In the past, financial and emotional support have been provided mainly by **Heroku**, **GitHub** and **Engine Yard**, and is now taken care of by **Travis CI**.

Special thanks to the following extraordinary individuals, without whom Sinatra would not be possible:

- **Ryan Tomayko** (rtomayko) for constantly fixing whitespace errors **60d5006**
- **Ezra Zygmuntowicz** (ezmobius) for initial help and letting Blake steal some of merbs internal code.
- **Chris Schneider** (cschneid) for The Book, the blog, **irclogger.com**, and a bunch of useful patches.
- **Markus Prinz** (cypher) for patches over the years, caring about the README, and hanging in there when times were rough.
- **Simon Rozet** (sr) for a ton of doc patches, HAML options, and all that advocacy stuff he's going to do for 1.0.
- **Erik Kastner** (kastner) for fixing `MIME_TYPES` under Rack 0.5.
- **Ben Bleything** (bleything) for caring about HTTP status codes and doc fixes.
- **Igal Koshevoy** (igal) for root path detection under Thin/Passenger.
- **Jon Crosby** (jcrosby) for coffee breaks, doc fixes, and just because, man.
- **Karel Minarik** (karmi) for screaming until the website came back up.
- **Jeremy Evans** (jeremyevans) for unbreaking optional path params (twice!)
- **The GitHub guys** for stealing Blake's table.

- **Nickolas Means** (nmeans) for Sass template support.
- **Victor Hugo Borja** (vic) for splat'n routes specs and doco.
- **Avdi Grimm** (avdi) for basic RSpec support.
- **Jack Danger Canty** for a more accurate root directory and for making me watch **this** just now.
- **Mathew Walker** for making escaped paths work with static files.
- **Millions of Us** for having the problem that led to Sinatra's conception.
- **Songbird** for the problems that helped Sinatra's future become realized.
- **Rick Olson** (technoweenie) for the killer plug at RailsConf '08.
- **Steven Garcia** for the amazing custom artwork you see on 404's and 500's
- **Pat Nakajima** (nakajima) for fixing non-nested params in nested params Hash's.
- **Konstantin Haase** (rkh) for his hard work and ongoing commitment to improving Sinatra, for 1.1.0, 1.2.0 and beyond..
- **Zachary Scott** for adding Konstantin to the AUTHORS file. He also did help writing the book, but mainly for adding Konstantin.
- **Gabriel Andretta** for having people wonder whether our documentation is actually in English or in Spanish.
- **Vasily Polovnyov**, **Nickolay Schwarz**, **Luciano Sousa**, **Wu Jiang**, **Mickael Riga**, **Bernhard Essl**, **Janos Hardi**, **Kouhei Yanagita** and **"burningTyger"** for willingly translating whatever ends up in the README.
- **Wordy** for proofreading our README. **73e137d**
- **cactus** for digging through code and specs, multiple times.
- **Nicolás Sanguinetti** (foca) for strong demand of karma and shaping helpers/register.

And last but not least:

- **Frank Sinatra** (chairman of the board) for having so much class he deserves a web-framework named after him.

Sinatra

Sinatra 1.4.0, 1.3.6, 1.2.9 released!

Posted by Konstantin Haase on **Friday, March 15, 2013** #

I've just released Sinatra 1.4.0, 1.3.6 and 1.2.9.

Find out what's new in 1.4 on my blog.

New feature release, Contrib and Recipes

Posted by Konstantin Haase on **Friday, September 30, 2011** #

We're proud to announce two new releases today: 1.3.0 and 1.2.7. We're also simultaneously releasing sinatra-contrib and would like to officially announce the recently launched Sinatra Recipes project. Read on for more goodness!

Sinatra 1.3.0

We have a new feature release!

From our perspective, one of the biggest additions is a streaming API. A simple version of it ships with Sinatra directly and is extended in sinatra-contrib. The vanilla version looks like this:

```
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
```

The cool thing about this is that it abstracts away the differences between all the different Rack servers, no matter if those are evented (like Thin, Rainbows! or Ebb) or sequential (like Unicorn, Passenger or Mongrel). Only WEBRick has issues with it at the moment (you'll get the response body all at once), but we are looking into this.

What is really interesting about this: If you run on an evented server, like Thin, you can keep the connection open and easily implement messaging services, Server-Sent Events and so on:

```
set :server, :thin
connections = []

get '/' do
  # keep stream open
  stream(:keep_open) { |out| connections << out }
end

post '/' do
  # write to all open streams
  connections.each { |out| out << params[:message] << "\n" }
  "message sent"
end
```

It is worth mentioning that all this works without fibers (which, amongst other things, would significantly limit the stack size).

We added support for the PATCH HTTP verb, which you might be familiar with from the new GitHub API:

```
patch '/' do
  # ... modify a resource ...
end
```

A `logger` helper for logging is available now, and if it writes to the logs depends on whether you enabled or disabled logging:

```
configure(:test) { disable :logging }

get '/' do
  logger.info "I just want to let you know: I'll take care of the request!"
  "Hello World!"
end
```

The `erubis` method has been deprecated. Sinatra will automatically use Erubis for rendering ERB templates if available.

There is more coming with this release, but the rest is mainly bug fixes and improving the behavior. For instance, Sinatra treats ETags and Last-Modified headers properly when they are used as optimistic locking for unsafe HTTP verbs. For more changes, see the 1.3.0 change log.

Special thanks for helping with the 1.3.0 release go to:

Gabriel Andretta, michelc, burningTyger, Tim Felgentreff, Vasily Polovnyov, nashby, kenichi nakamura, Gaku Ueda, Gabriel Horner, Sylvain Desvé, Jacob Burkhart & Josh Lane, aibo (irc), Selman ULUG, Aviv Ben-Yosef, Paolo “Nusco” Perrotta, Marcos Toledo, Matthew Schinckel, Tim Preston, Davide D’Agostino, Simone Carletti, Peter Suschlik, Postmodern, F. Zhang, Rémy Coutable, and David Waite

Sinatra 1.2.7

We will drop support for Rack prior to 1.3 and Ruby 1.8.6 with the Sinatra 1.3.0 release. However, we will continue to supply Sinatra 1.2 with bug fixes until the End-Of-Life of Rack 1.2.x and Sinatra 1.2 will always remain compatible with the latest released 1.8.6 patchlevel.

For the backported patches, see the 1.2.7 change log.

We'd like to thank everyone helping with the 1.2.7 release:

Emanuele Vicentini, Takanori Ishikawa, David Kellum, Gaku Ueda, Lee Reilly, Iain Barnett, pete, Luke Jahnke, John Wolfe, Andrew Armenia, Tim Felgentreff, and Alessandro Dal Grande

Sinatra-Contrib

There are a lot of Sinatra extensions out there, and some of those are used by a large number of apps, like sinatra-content-for or sinatra-reloader. The maintainers of these extensions have to keep up with new Sinatra releases to make sure everything works fine. This poses an issue from time to time, especially since some of these extensions were relying on Sinatra internals. To fix this, we created the sinatra-contrib project (source code at https://github.com/sinatra/sinatra-contrib). This project contains a number of extensions that are probably useful for most Sinatra applications. Some are old extensions that have largely been rewritten to ensure the code quality developers are used to from Sinatra itself and to no longer rely on internals, like sinatra-namespace, and some are new additions of general interest, for example sinatra-respond-with.

Here is the promise: For every Sinatra release, starting with 1.3.0, there will *always* be a fully compatible sinatra-contrib release. Documentation for all the extensions shipping with sinatra-contrib will be made available on the Sinatra website. I would not have managed to take care of this all by my self, so I'm really grateful that Gabriel Andretta jumped in to help with this project.

Documentation is available at sinatrarb.com/contrib.

Sinatra Recipes

Zachary Scott recently launched the Recipes project. It contains community contributed recipes and techniques for Sinatra.

Sinatra Loves Dancer

Posted by Konstantin Haase on **Thursday, July 21, 2011** #

"The only cool PR is provided by one's enemies. They toil incessantly and for free." - Marshall McLuhan

There are a lot of frameworks out there inspired by Sinatra. One of them is Dancer. Some unknown individual or group was posing as different members of the Sinatra core team on CPAN ratings, a website to comment on the quality of Perl libraries.

Those comments were extremely rude and do not represent the opinion of any of the Sinatra developers. Quite the opposite, we love how Sinatra inspired so many of the libraries and are even proud to see Dancer being adopted by Perl developers.

We do not know what the agenda of this one individual is, and honestly, we do not care. The harm done by this person to the Dancer project, the Sinatra project and the people he posed as is unacceptable. The incident has been reported to the CPAN ratings team and we hope that those comments will be removed in due time.

If you are a Perl developer, we suggest you check out Dancer.

What's New in Sinatra 1.2?

Posted by Konstantin Haase on **Thursday, March 3, 2011** #

As announced on the mailing list, we have just released Sinatra 1.2.0. Let's have a closer look at the new features.

Slim support

Sinatra now supports the Slim Template Engine:

```
require 'sinatra'
require 'slim'

get('/') { slim :index }

__END__

@@ index
! doctype html
html
  head
    title Sinatra With Slim
  body
    h1 Slim Is Fun!
    a href="http://haml-lang.com/" A bit like Haml, don't you think?
```

Inline Markaby

Like Builder and Nokogiri templates, Markaby can now be used directly inline:

```
require 'sinatra'
require 'markaby'

get '/' do
  markaby do
    html do
      head { title "Sinatra With Markaby" }
      body { h1 "Markaby Is Fun!" }
    end
  end
end
```

Layout Engines

Whenever you render a template, like `some_page.haml`, Sinatra will use a corresponding layout, like `layout.haml` for you. With the `:layout` option it has always been possible to use a different layout, but it still had to be written in the same template language, Haml in our example. In 1.1 we introduced `markdown`, `textile` and `rdoc` templates. It is not possible to use those for layouts. We therefore added the `:layout_engine` option, which easily allows you to combine one two different template engines:

```
require 'sinatra'
require 'rdiscount'

# for all markdown files, use post.haml as layout
set :markdown, :layout_engine => :haml, :layout => :post

get '/' do
  # use index.haml for readme
  markdown :README, :layout => :index
end

get '/:post' do
  markdown params[:post].to_sym
end
```

This feature should also be handy when migrating from one template language to another, as it allows you to combine Erb with Haml, for instance.

Conditional Filters

We introduced pattern matching filters in 1.1. Now they also support conditions:

```
before :agent => /Song Bird/ do
  # ...
end
```

Those can also be combined with patterns, of course:

```
after '/api/*', :provides => :json do
  # ...
end
```

URL helper

Usually Sinatra does not provide any view helper methods. Those are provided by extensions and would not suit Sinatra's approach of a small but robust core. However, constructing URLs is a use case most people run into sooner or later. It is a bit complicated to construct URLs right. Consider this example:

```
get('/foo') { "<a href='/bar'>Will you make it?</a>" }
get('/bar') { "You made it!" }
```

Feel free to run it. Works, doesn't it? So, what is wrong with it?

Imagine your app is "mounted" by another Rack application, for instance in a `config.ru` like this:

```
map('/there') { run Sinatra::Application }
map('/') { run MyRailsApp::Application }
```

Now the link to `/bar` would end up in a request send to `MyRailsApp` rather than to Sinatra. Injecting `request.script_name` would fix this, but be honest, how often do you do that?

Now, imagine these links are presented out of context, in an RSS feed or embedded on another host. In that case you might want to construct absolute URLs. This is even more cumbersome, as you most certainly either forget to handle reverse proxies, alternative ports/protocols or you end up with lots of URL related code all over the place, while what you should do is use the `url` helper:

```
get('/foo') { "<a href='#{url '/bar'}'>You will make it!</a>" }
get('/bar') { "You made it!" }
```

Since you are likely going to use this with redirects, we also aliased the helper to `to`:

```
get('/foo') { redirect to('/bar') }
get('/bar') { "You made it!" }
```

Named Captures on 1.9

Ruby 1.9 introduced named captures for regular expressions. Sinatra accepts regular expressions for matching paths. Now named captures will automatically end up populating `params`:

```
get %r{/(?<year>\d{4})/(?<month>\d{2})/(?<day>\d{2})/?} do
  date = Date.new params[:year].to_i, params[:month].to_i, params[:day].to_i
  @posts = Post.pubished_on date
  erb :posts
end
```

Templates with different scopes

All rendering methods now accept a `:scope` options:

```
get '/:id' do |id|
  @post = Post.find id

  # without scope
  erb "<%= @post.name %>"

  # with scope
  erb "<%= name %>", :scope => @post
end
```

Note that all Sinatra helper methods and instance variables will *not* be available.

Configurable redirects

In 1.1 we made sure all redirects were absolute URIs, to conform with RFC 2616 (HTTP 1.1). This will result in issues for you if you have a broken Reverse Proxy configuration. If so, you should really fix your configuration. If you are unable to do so, a simple `disable :absolute_redirects` will now give you back the 1.0 behavior. As shown above, you can now use the `to` helper with redirect. If all your redirects are application local, you can now `enable :prefixed_redirects` and skip the `to` altogether:

```
enable :prefixed_redirects
get('/foo') { redirect '/bar' }
get('/bar') { "You made it!" }
```

We did not enable this per default to not break compatibility and to allow you redirects to other Rack endpoints.

Overriding template lookup

One popular feature request is supporting multiple view folders. But everyone wants different semantics. So, instead of choosing one way to go, we gave you means to implement your own lookup logic:

```
helpers do
  def find_template(*)
    puts "looking for index.txt"
    yield "views/index.txt"
    puts "apparently, index.txt doesn't exist, let's try index.html"
    yield "views/index.html"
  end
end

get "/" do
  haml :foo
end
```

Sinatra will call `find_template` to discover the template file. In the above example, we don't care about what template engine to use or what name the template has. It will use `views/index.txt` or `views/index.html` for every template. Let's have a look at the standard implementation:

```
def find_template(views, name, engine)
  Tilt.mappings.each do |ext, klass|
    next unless klass == engine
    yield ::File.join(views, "#{name}.#{ext}")
  end
end
```

As you can see, it will look in the views folder for a file named like the template with any of the file extensions registered for the template engine.

If all you want to change is the folder, you probably should just call `super`:

```
def find_template(views, *a, &b)
  super("#{views}/a", *a, &b)
  super("#{views}/b", *a, &b)
end
```

More examples can be found in the readme.

Other changes

- `send_file` now takes a `:last_modified` option
- improved error handling

Blog revived

Posted by Konstantin Haase on **Thursday, March 3, 2011** #

It has been quiet on this blog recently. Announcements, like the 1.0 and 1.1 releases, have only been made on the mailing list. Plans are to change this and accompany mailing list announcements with in-depth articles about new feature releases.

We Have a FAQ Now

Posted by Ryan Tomayko on **Thursday, January 29, 2009** #

Just getting started with Sinatra and wondering how to do something that should be obvious? Check out The FAQ. We culled a few weeks worth of IRC logs and mailing list threads to compile the initial set of entries and we'll be adding to the list considerably over the coming weeks.

Have an idea for a FAQ entry? Fork the sinatra.github.com repository, add the entry to `faq.markdown`, and push. We'll take care of the rest.

What's New in Sinatra 0.9.0

Posted by Ryan Tomayko on **Sunday, January 18, 2009** #

This is the first in a series of 0.9.x releases designed to move Sinatra toward a rock solid 1.0. While we were able to add a touch of new hotness in this release, the major focus has been on getting the codebase shaped up for future development. Many longstanding bugs and minor annoyances were corrected along the way.

Sinatra's internal classes and methods have changed significantly in this release. Most apps written for 0.3.x will work just fine under the new codebase but we've begun adding deprecation warnings for things slated to be ripped out completely in 1.0. **Please test your apps before upgrading production environments to the 0.9.0 gem.** We're committed to keeping existing apps running through 0.9.x so report compatibility issues through the normal channels.

With that out of the way, here are some of the new features to look out for in 0.9.0.

Sinatra::Base and Proper Rack Citizenship

Sinatra can now be used to build modular / reusable Rack applications and middleware components. This means that multiple Sinatra applications can now be run in isolation and co-exist peacefully with other Rack based frameworks.

Requiring `'sinatra/base'` instead of `'sinatra'` causes a subset of Sinatra's features to be loaded. No methods are added to the top-level and the command-line / auto-running features are disabled. Subclassing `Sinatra::Base` creates a Rack component with the familiar Sinatra DSL methods available in class scope. These classes can then be run as Rack applications or used as middleware components.

Proper documentation on this feature is in the works but here's a quick example for illustration:

```
require 'sinatra/base'

class Foo < Sinatra::Base
  # all options are available for the setting:
  enable :static, :session
  set :root, File.dirname(__FILE__)

  # each subclass has its own private middleware stack:
  use Rack::Deflator

  # instance methods are helper methods and are available from
  # within filters, routes, and views:
  def em(text)
    "<em>#{text}</em>"
  end

  # routes are defined as usual:
  get '/hello/:person' do
    "Hello " + em(params[:person])
  end
end
```

That thing can be plugged in anywhere along a Rack pipeline. For instance, once Rails 2.3 ships, you'll be able to use Sinatra apps to build Rails Metal.

Jesse Newland and Jon Crosby are already experimenting. Very hot.

Nested Params

Form parameters with subscripts are now parsed into a nested/recursive Hash structure. Here's a form:

```
<form method='POST' action='/guestbook/'>
  <input type='text' name='person[name]'>
  <input type='text' name='person[email]'>
  <textarea name='note'></textarea>
</form>
```

It looks like this on the wire:

```
person[name]=Frank&person[email]=frank@theritz.com&message=Stay%20cool
```

Sinatra turns it into a nested Hash structure when accessed through the `params` method:

```
post '/guestbook/' do
  params[:person]    # => { :name => 'Frank', :email => 'frank@theritz.com' }
  "Hi #{person[:name]}! Thanks for signing my guestbook."
end
```

This was a massively popular feature requests. Thanks to Nicolás Sanguinetti for the patch and Michael Fellinger for the original implementation.

Routing with Regular Expressions

The route declaration methods (`get`, `put`, `post`, `put`, `delete`) now take a Regexp as a pattern. Captures are made available to the route block at `params[:captures]`:

```
get %r{/foo/(bar|baz)/(\d+)} do
  # assuming: GET /foo/bar/42
  params[:captures]  # => ['bar', 42]
end
```

Passing on a Route

We added a new request-level `pass` method that immediately exits the current block and passes control to the next matching route. For example:

```
get '/shoot/:person' do
  pass unless %w[Kenny Sherrif].include?(params[:person])
  "You shot #{params[:person]}."
end

get '/shoot/*' do
  "Missed!"
end
```

If no matching route is found after a `pass`, a `NotFound` exception is raised and the application 404s as if no route had matched in the first place.

Refined Test Framework

Sinatra's testing support no longer depends on `Test::Unit` (or any specific test framework for that matter). Requiring `'sinatra/test'` brings in the `Sinatra::Test` module and the `Sinatra::TestHarness` class, which can be used as necessary to simulate requests and make assertions about responses.

You can also require `sinatra/test/unit`, `sinatra/test/spec`, `sinatra/test/rspec`, or `sinatra/test/bacon` to setup a framework-specific testing environment. See the section on "Testing"

in the README for examples.

More

See the CHANGES file for a comprehensive list of enhancements, bug fixes, and deprecations.

Sinatra

Contribute

Want to show Sinatra some love? Help out by contributing!

Find a bug?

Log it in our issue tracker or send a note to the mailing list. Be sure to include all relevant information, like the versions of Sinatra and Ruby you're using. A gist of the code that caused the issue as well as any error messages are also very helpful.

Need help?

The Sinatra mailing list has over 900 subscribers, many of which are happy to help out newbies or talk about potential feature additions. You can also drop by the #sinatra channel on irc.freenode.net.

Have a patch?

Bugs and feature requests that include patches are much more likely to get attention. Here are some guidelines that will help ensure your patch can be applied as quickly as possible:

1. **Use Git and GitHub:** The easiest way to get setup is to fork the sinatra/sinatra repo. Or, the sinatra.github.com repo, if the patch is doc related.

2. **Write unit tests:** If you add or modify functionality, it must include unit tests. If you don't write tests, we have to, and this can hold up acceptance of the patch.

3. **Mind the `README`:** If the patch adds or modifies a major feature, modify the `README.md` file to reflect that. Again, if you don't update the `README`, we have to, and this holds up acceptance.

4. **Push it:** Once you're ready, push your changes to a topic branch and add a note to the ticket with the URL to your branch. Or, say something like, "you can find the patch on johndoe/foobranch". We also gladly accept GitHub pull requests.

NOTE: *we will take whatever we can get.* If you prefer to attach diffs in emails to the mailing list, that's fine; but do know that *someone* will need to take the diff through the process described above and this can hold things up considerably.

Want to write docs?

The process for contributing to Sinatra's website, documentation or the book is the same as contributing code. We use git for versions control and GitHub to track patch requests.

- The sinatra.github.com repo is where the website sources are managed. There are almost always people in `#sinatra` that are happy to discuss, apply, and publish website patches.
- The Book has its own git repository and build process but is managed the same as the website and project codebase.
- Sinatra Recipes is a community project where anyone is free to contribute ideas, recipes and tutorials. Which also has its own git repository.
- The Introduction is generated from Sinatra's README file.
- If you want to help translating the documentation, there already is a Japanese and a German version of the README, which tend to fall behind the English version. Translations into other languages would also be appreciated.

Looking for something to do?

If you'd like to help out but aren't sure how, pick something that looks interesting from the issues list and hack on. Make sure to leave a comment on the ticket noting that you're investigating (a simple "Taking…" is fine).

Sinatra

Configuring Settings

Sinatra includes a number of built-in settings that control whether certain features are enabled. Settings are application-level variables that are modified using one of the `set`, `enable`, or `disable` methods and are available within the request context via the `settings` object. Applications are free to set custom settings as well as the default, built-in settings provided by the framework.

Using `set`, `enable`, and `disable`

In its simplest form, the `set` method takes a setting name and value and creates an attribute on the application. Settings can be accessed within requests via the `settings` object:

```
set :foo, 'bar'

get '/foo' do
  "foo is set to " + settings.foo
end
```

Deferring evaluation

When the setting value is a `Proc`, evaluation is performed when the setting is read so that other settings may be used to calculate the value:

```
set :foo, 'bar'
set :baz, Proc.new { "Hello " + foo }

get '/baz' do
  "baz is set to " + settings.baz
end
```

The `/baz` response should come as “baz is set to Hello bar” unless the `foo` setting is modified.

Configuring multiple settings

Multiple settings can be set by passing a Hash to `set`. The previous example could be rewritten with:

```
set :foo => 'bar', :baz => Proc.new { "Hello " + foo }
```

Setting multiple boolean settings with `enable` and `disable`

The `enable` and `disable` methods are sugar for setting a list of settings to `true` or `false`, respectively. The following two code examples are equivalent:

```
enable  :sessions, :logging
disable :dump_errors, :some_custom_option
```

Using `set`:

```
set :sessions, true
set :logging, true
set :dump_errors, false
set :some_custom_option, false
```

Built-in Settings

`:environment` - configuration/deployment environment

A symbol specifying the deployment environment; typically set to one of `:development`, `:test`, or `:production`. The `:environment` defaults to the value of the `RACK_ENV` environment variable (`ENV['RACK_ENV']`), or `:development` when no `RACK_ENV` environment variable is set.

The environment can be set explicitly:

```
set :environment, :production
```

`:sessions` - enable/disable cookie based sessions

Support for encrypted, cookie-based sessions are included with Sinatra but are disabled by default. Enable them with:

```
set :sessions, true
```

Sessions are implemented by inserting the `Rack::Session::Cookie` component into the application's middleware pipeline.

`:logging` - log requests to STDERR

Writes a single line to `STDERR` in Apache common log format when enabled. This setting is enabled by default in classic style apps and disabled by default in `Sinatra::Base` subclasses.

Internally, the `Rack::CommonLogger` component is used to generate log messages.

`:method_override` - enable/disable the POST `_method` hack

Boolean specifying whether the HTTP POST `_method` parameter hack should be enabled. When `true`, the actual HTTP request method is overridden by the value of the `_method` parameter included in the POST body. The `_method` hack is used to make POST requests look like other request methods (e.g., `PUT`, `DELETE`) and is typically only needed in shitty environments – like HTML form submission – that do not support the full range of HTTP methods.

The POST `_method` hack is implemented by inserting the `Rack::MethodOverride` component into the middleware pipeline.

`:root` – The application's root directory

The directory used as a base for the application. By default, this is assumed to be the directory containing the main application file (`:app_file` setting). The root directory is used to construct the default `:public_folder` and `:views` settings. A common idiom is to set the `:root` setting explicitly in the main application file as follows:

```
set :root, File.dirname(__FILE__)
```

`:static` – enable/disable static file routes

Boolean that determines whether static files should be served from the application's public directory (see the `:public_folder` setting). When `:static` is truthy, Sinatra will check if a static file exists and serve it before checking for a matching route.

The `:static` setting is enabled by default when the `public` directory exists.

`:public_folder` – static files directory

A string specifying the directory where static files should be served from. By default, this is assumed to be a directory named "public" within the root directory (see the `:root` setting). You can set the public directory explicitly with:

```
set :public_folder, '/var/www'
```

The best way to specify an alternative directory name within the root of the application is to use a deferred value that references the `:root` setting:

```
set :public_folder, Proc.new { File.join(root, "static") }
```

`:views` – view template directory

A string specifying the directory where view templates are located. By default, this is assumed to be a directory named "views" within the application's root directory (see the `:root` setting). The best way to specify an alternative directory name within the root of the application is to use a deferred value that references the `:root` setting:

```
set :views, Proc.new { File.join(root, "templates") }
```

:run – enable/disable the built-in web server

Boolean specifying whether the built-in web server is started after the app is fully loaded. By default, this setting is enabled only when the `:app_file` matches `$0`. i.e., when running a Sinatra app file directly with `ruby myapp.rb`. To disable the built-in web server:

```
set :run, false
```

:server – handler used for built-in web server

String or Array of Rack server handler names. When the `:run` setting is enabled, Sinatra will run through the list and start a server with the first available handler. The `:server` setting is set as follows by default:

```
set :server, %w[thin mongrel webrick]
```

:bind – server hostname or IP address

String specifying the hostname or IP address of the interface to listen on when the `:run` setting is enabled. The default value in the development environment is `'localhost'` which means the server is only available from the local machine. In other environments the default is `'0.0.0.0'`, which causes the server to listen on all available interfaces.

To listen on all interfaces in the development environment (for example if you want to test from other computers in your local network) use:

```
set :bind, '0.0.0.0'
```

This can also be set from the command line with the `-o` option. If you set the bind option in your application it will override anything set on the command line.

:port – server port

The port that should be used when starting the built-in web server when the `:run` setting is enabled. The default port is `4567`. To set the port explicitly:

```
set :port, 9494
```

:app_file – main application file

The `:app_file` setting is used to calculate the default `:root`, `:public_folder`, and `:views` setting values. A common idiom is to override the default detection heuristic by setting the `:app_file` explicitly from within the main application file:

```
set :app_file, __FILE__
```

It's also used to detect whether Sinatra should boot a web server when using classic-style applications.

:dump_errors - log exception backtraces to STDERR

Boolean specifying whether backtraces are written to STDERR when an exception is raised from a route or filter. This setting is enabled by default in classic style apps. Disable with:

```
set :dump_errors, false
```

:raise_errors - allow exceptions to propagate outside of the app

Boolean specifying whether exceptions raised from routes and filters should escape the application. When disabled, exceptions are rescued and mapped to error handlers which typically set a 5xx status code and render a custom error page. Enabling the :raise_errors setting causes exceptions to be raised outside of the application where it may be handled by the server handler or Rack middleware, such as Rack::ShowExceptions or Rack::MailExceptions.

:lock - ensure single request concurrency with a mutex lock

Sinatra can be used in threaded environments where more than a single request is processed at a time. However, not all applications and libraries are thread-safe and may cause intermittent errors or general weirdness. Enabling the :lock setting causes all requests to synchronize on a mutex lock, ensuring that only a single request is processed at a time.

The :lock setting is disabled by default.

:show_exceptions - enable classy error pages

Enable error pages that show backtrace and environment information when an unhandled exception occurs. Enabled in development environments by default.

Sinatra

Testing Sinatra with Rack::Test

All examples in the following sections assume that `Test::Unit` is being used in an attempt to be as general as possible. See the Test Framework Examples for information on using the test helpers in other testing environments. To use `Rack::Test` library used when you require `rack/test`, you'll need to install the `rack-test` gem.

gem install rack-test

Example App: `hello_world.rb`

The following example app is used to illustrate testing features. This is assumed to be in a file named `hello_world.rb`:

```
require 'sinatra'

get '/' do
  "Hello World #{params[:name]}".strip
end
```

Using The `Rack::Test::Methods` Mixin

The `Rack::Test::Methods` module includes a variety of helper methods for simulating requests against an application and asserting expectations about the response. It's typically included directly within the test context and makes a few helper methods and attributes available.

The following is a simple example that ensures the hello world app functions properly:

```
ENV['RACK_ENV'] = 'test'

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

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

  def app
    Sinatra::Application
  end
```

```
  def test_it_says_hello_world
    get '/'
    assert last_response.ok?
    assert_equal 'Hello World', last_response.body
  end

  def test_it_says_hello_to_a_person
    get '/', :name => 'Simon'
    assert last_response.body.include?('Simon')
  end
end
```

Using `Rack::Test` without the Mixin

For a variety of reasons you may not want to include `Rack::Test::Methods` into your own classes. `Rack::Test` supports this style of testing as well, here is the above example without using Mixin.

```
ENV['RACK_ENV'] = 'test'

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

class HelloWorldTest < Test::Unit::TestCase

  def test_it_says_hello_world
    browser = Rack::Test::Session.new(Rack::MockSession.new(Sinatra::Application))
    browser.get '/'
    assert browser.last_response.ok?
    assert_equal 'Hello World', browser.last_response.body
  end

  def test_it_says_hello_to_a_person
    browser = Rack::Test::Session.new(Rack::MockSession.new(Sinatra::Application))
    browser.get '/', :name => 'Simon'
    assert browser.last_response.body.include?('Simon')
  end
end
```

Rack::Test's Mock Request Methods

The `get`, `put`, `post`, `delete`, and `head` methods simulate the respective type of request on the application. Tests typically begin with a call to one of these methods followed by one or more assertions against the resulting response.

All mock request methods have the same argument signature:

```
get '/path', params={}, rack_env={}
```

- `/path` is the request path and may optionally include a query string.
- `params` is a Hash of query/post parameters, a String request body, or `nil`.

- `rack_env` is a Hash of Rack environment values. This can be used to set request headers and other request related information, such as session data. See the Rack SPEC for more information on possible key/values.

Asserting Expectations About The Response

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

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

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

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

Optional Test Setup

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

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

```
  def app
    MySinatraApp
  end
```

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

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

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

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

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

```
require 'rack/test'

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

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

Test Framework Examples

As of version `0.9.1`, Sinatra no longer provides testing framework-specific helpers. Those found in `sinatra/test/*.rb` are deprecated and has been removed in Sinatra `1.0`.

RSpec

Sinatra can be tested under plain RSpec. The `Rack::Test` module should be included within the `describe` block:

```
ENV['RACK_ENV'] = 'test'

require 'hello_world'  # <-- your sinatra app
require 'rspec'
require 'rack/test'

describe 'The HelloWorld App' do
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  it "says hello" do
    get '/'
    expect(last_response).to be_ok
    expect(last_response.body).to eq('Hello World')
  end
end
```

Make `Rack::Test` available to all spec contexts by including it via `RSpec`:

```
require 'rspec'
require 'rack/test'

RSpec.configure do |conf|
  conf.include Rack::Test::Methods
end
```

Bacon

Testing with Bacon is similar to `test/unit` and RSpec:

```
ENV['RACK_ENV'] = 'test'
```

```
require 'hello_world'  # <-- your sinatra app
require 'bacon'
require 'rack/test'

describe 'The HelloWorld App' do
  extend Rack::Test::Methods

  def app
    Sinatra::Application
  end

  it "says hello" do
    get '/'
    last_response.should.be.ok
    last_response.body.should.equal 'Hello World'
  end
end
```

Make `Rack::Test` available to all spec contexts by including it in `Bacon::Context`:

```
class Bacon::Context
  include Rack::Test::Methods
end
```

Test::Spec

The `Rack::Test` module should be included within the context of the `describe` block:

```
ENV['RACK_ENV'] = 'test'

require 'hello_world'  # <-- your sinatra app
require 'test/spec'
require 'rack/test'

describe 'The HelloWorld App' do
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  it "says hello" do
    get '/'
    last_response.should.be.ok
    last_response.body.should.equal 'Hello World'
  end
end
```

Make `Rack::Test` available to all spec contexts by including it in `Test::Unit::TestCase`:

```
require 'test/spec'
require 'rack/test'

Test::Unit::TestCase.send :include, Rack::Test::Methods
```

Webrat

From `Webrat`'s wiki where you'll find more examples.

```
ENV['RACK_ENV'] = 'test'

require 'hello_world'  # <-- your sinatra app
require 'rack/test'
require 'test/unit'

Webrat.configure do |config|
  config.mode = :rack
end

class HelloWorldTest < Test::Unit::TestCase
  include Rack::Test::Methods
  include Webrat::Methods
  include Webrat::Matchers

  def app
    Sinatra::Application.new
  end

  def test_it_works
    visit '/'
    assert_contain('Hello World')
  end
end
```

Capybara

`Capybara` will use `Rack::Test` by default. You can use another driver, like `Selenium`, by setting the default_driver.

```
ENV['RACK_ENV'] = 'test'

require 'hello_world'  # <-- your sinatra app
require 'capybara'
require 'capybara/dsl'
require 'test/unit'

class HelloWorldTest < Test::Unit::TestCase
  include Capybara::DSL
  # Capybara.default_driver = :selenium # <-- use Selenium driver

  def setup
    Capybara.app = Sinatra::Application.new
  end

  def test_it_works
    visit '/'
    assert page.has_content?('Hello World')
  end
end
```

See Also

See the source for Rack::Test for more information on `get`, `post`, `put`, `delete` and friends.

Sinatra

Sinatra::Contrib

Starting with 1.3.0, every Sinatra release will be followed by a Sinatra::Contrib release which will include collection of common Sinatra extensions. Feel free to fork the source code and start contributing.

This extensions are organized in **common** and **custom extensions**. *Common extensions* will not add significant overhead or change any behavior of already existing APIs, and do not add any dependencies not already installed with Sinatra::Contrib. *Custom extensions*, on the other hand, may add additional dependencies and enhance the behavior of the existing APIs.

Installation

All Sinatra::Contrib extensions are bundled in the `sinatra-contrib` gem.

```
gem install sinatra-contrib
```

Usage

In Classic Style Applications

A single extension (example: sinatra-content-for):

```
require 'sinatra'
require 'sinatra/content_for'
```

Common extensions:

```
require 'sinatra'
require 'sinatra/contrib'
```

All extensions:

```
require 'sinatra'
require 'sinatra/contrib/all'
```

In Modular Style Applications

A single extension (example: sinatra-content-for and sinatra-namespace):

```
require 'sinatra/base'
require 'sinatra/content_for'
require 'sinatra/namespace'

class MyApp < Sinatra::Base
  # Note: Some modules are extensions, some helpers, see the specific
  # documentation or the source
  helpers Sinatra::ContentFor
  register Sinatra::Namespace
end
```

Common extensions:

```
require 'sinatra/base'
require 'sinatra/contrib'

class MyApp < Sinatra::Base
  register Sinatra::Contrib
end
```

All extensions:

```
require 'sinatra/base'
require 'sinatra/contrib/all'

class MyApp < Sinatra::Base
  register Sinatra::Contrib
end
```

What Is Included?

Common Extensions

- Sinatra::ConfigFile: Load your application's configuration from YAML files.
- Sinatra::ContentFor: Adds Rails-style *content_for* helpers to Erb, Erubis, Haml and Slim.
- Sinatra::Cookies: Provides the *cookies* helper for reading and writting cookies.
- Sinatra::JSON: Easily return JSON documents.
- Sinatra::LinkHeader: Helpers for generating link HTML tags and corresponding Link HTTP headers.
- Sinatra::MultiRoute: Create multiple routes with just one statement.
- Sinatra::Namespace: Adds namespaces to Sinatra.

- Sinatra::RespondWith: Choose action and/or template depending on the incoming request.
- Sinatra::Streaming: Improves the streaming API by making the *stream* object immitate an IO object.

Custom Extensions

- Sinatra::Decompile: Recreates path patterns from Sinatra's internal data structures (used by other extensions).
- Sinatra::Reloader: Automatically reloads Ruby files on code changes.

Other Tools

- Sinatra::Extension: Mixin for writing your own Sinatra extensions.

Sinatra

Using Extensions

Extensions provide helper or class methods for Sinatra applications. These methods are customarily listed and described on extensions home pages, many of which are listed below.

Using an extension is usually as simple as installing a gem or library and requiring a file. Consult these steps if you run into problems:

1. Install the gem or vendor the library with your project: `gem install sinatra-prawn`
2. Require the extension in your application: `require 'sinatra/prawn'`
3. If your application is "classic" (i.e., you `require 'sinatra'` and define the application in the main/top-level context), you're done. The extension methods should be available to your application.
4. If your application subclasses `Sinatra::Base`, you have to register the extension in your subclass: `register Sinatra::Prawn`

Helper Extensions

These extensions add helper methods to the request context:

1. sinatra-prawn adds support for PDF rendering with Prawn templates.
2. sinatra-markaby enables rendering of HTML files using markaby templates.
3. sinatra-maruku provides Maruku templates for a Sinatra application.
4. sinatra-rdiscount provides RDiscount templates for a Sinatra application.
5. sinatra-effigy provides Effigy templates and views for a Sinatra application.
6. sinatra-content-for provides `content_for` helper similar to Rails one.
7. sinatra-url-for construct absolute paths and full URLs to actions in a Sinatra application
8. sinatra-static-assets implements `image_tag`, `stylesheet_link_tag`, `javascript_script_tag` and `link_tag` helpers. These helpers construct correct absolute paths for applications dispatched to sub URI.
9. sinatra-mapping implements `map` in the DSL syntax commands which creates dynamically `mapname_path` method.
10. sinatra_more Library with agnostic generators, form builders, named route mappings, easy mailer support among other functionality.
11. sinatra-authorization HTTP auth helpers
12. sinatra-simple-navigation enables creating navigations using the simple-navigation gem.

DSL Extensions

These extensions add methods to Sinatra's application DSL:

1. snap provides support for named routes and helper methods for building URLs for use in links and redirects.

Extensions which handle setup and configuration

1. sinatra-mongoid sets up a MongoDB connection, provides Mongoid to your app, and provides options for configuration.

Add more! See Writing Extensions for more information about creating your own.

Sinatra

Writing Extensions

Sinatra includes an API for extension authors to help ensure that consistent behavior is provided for application developers.

Background

Some knowledge of Sinatra's internal design is required to write good extensions. This section provides a high level overview of the classes and idioms at the core of Sinatra's design.

Sinatra has two distinct modes of use that extensions should be aware of:

1. The "Classic" style, where applications are defined on main / the top-level - most of the examples and documentation target this usage. Classic applications are often single-file, standalone apps that are run directly from the command line or with a minimal rackup file. When an extension is required in a classic application, the expectation is that all extension functionality should be present without additional setup on the application developers part (like included/extending modules).

2. The "Modular" style, where `Sinatra::Base` is subclassed explicitly and the application is defined within the subclass's scope. These applications are often bundled as libraries and used as components within a larger Rack-based system. Modular applications must include any desired extensions explicitly by calling `register ExtensionModule` within the application's class scope.

Most extensions are relevant to both styles but care must be taken by extension authors to ensure that extensions do the right thing under each style. The extension API (`Sinatra.register` and `Sinatra.helpers`) is provided to help extension authors with this task.

Important: The following notes on `Sinatra::Base` and `Sinatra::Application` are provided for background only - extension authors should not need to modify these classes directly.

`Sinatra::Base`

The `Sinatra::Base` class provides the context for all evaluation in a Sinatra application. The top-level DSLish stuff exists in class scope while request-level stuff exists at instance scope.

Applications are *defined* within the class scope of a `Sinatra::Base` subclass. The "DSL" (e.g., `get`, `post`, `before`, `configure`, `set`, etc.) is simply a set of class methods defined on `Sinatra::Base`.

Extending the DSL is achieved by adding class methods to `Sinatra::Base` or one of its subclasses. However, Base classes should not be extended with `extend`; the `Sinatra.register` method (described below) is provided for this task.

Requests are evaluated within a new `Sinatra::Base` instance - routes, before filters, views, helpers, and error pages all share this same context. The default set of request-level helper methods (e.g, `erb`, `haml`, `halt`, `content_type`, etc.) are simple instance methods defined on `Sinatra::Base` or within modules that are included in `Sinatra::Base`. Providing new functionality at the request level is achieved by adding instance methods to `Sinatra::Base`.

As with DSL extensions, helper modules should not be added directly to `Sinatra::Base` by extension authors with `include`; the `Sinatra.helpers` method (described below) is provided for this task.

`Sinatra::Application`

The `Sinatra::Application` class provides the default execution context for *classic style applications*. It is a simple subclass of `Sinatra::Base` that provides default option values and other behavior tailored for top-level apps. When a classic style application is run, all `Sinatra::Application` public class methods are exported to the top-level.

Rules for Extensions

1. Never modify `Sinatra::Base` directly. You should not include or extend, change option values, or modify its behavior elsewise. Modular style applications will include your extension in their subclass explicitly using the `register` method.

2. Never `require 'sinatra'` in your extension. You should only ever need to `require 'sinatra/base'`. The reason for this is that `require 'sinatra'` is what triggers the classic style - extensions should never trigger the classic style.

3. Use the APIs described below where possible. You should not need to include or extend modules directly or define methods directly on a core Sinatra class. The specialized methods below handle all of that for you.

4. Extensions *should* be defined in separate modules under the `Sinatra` module. For example, an extension that added basic authentication primitives might be named `Sinatra::BasicAuth`.

Extending The Request Context with `Sinatra.helpers`

The most common type of extension is one that adds methods for use in routes, views, and helper methods.

For example, suppose you wanted to write an extension that added an `h` method that escaped reserved HTML characters (such as those found in other popular Ruby web frameworks).

```
require 'sinatra/base'

module Sinatra
  module HTMLEscapeHelper
```

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

  helpers HTMLEscapeHelper
end
```

The call to `Sinatra.helpers` includes the module in `Sinatra::Application`, making all methods defined in the module available to classic style applications. Using this extension in classic style apps is a simple as requiring the extension and using the new method:

```
require 'sinatra'
require 'sinatra/htmlescape'

get "/hello" do
  h "1 < 2"     # => "1 &lt; 2"
end
```

`Sinatra::Base` subclasses, on the other hand, must require *and* include the module explicitly using the `helpers` method:

```
require 'sinatra/base'
require 'sinatra/htmlescape'

class HelloApp < Sinatra::Base
  helpers Sinatra::HTMLEscapeHelper

  get "/hello" do
    h "1 < 2"
  end
end
```

Extending The DSL (class) Context with `Sinatra.register`

Extensions can also extend Sinatra's class level DSL using the `Sinatra.register` method. Here's an extension that adds a `block_links_from` macro that checks the referer on each request for a app provided pattern and sends back a `403 Forbidden` response when a match is detected:

```
require 'sinatra/base'

module Sinatra
  module LinkBlocker
    def block_links_from(host)
      before {
        halt 403, "Go Away!" if request.referer.match(host)
      }
    end
  end

  register LinkBlocker
end
```

`Sinatra.register` adds all public methods in the module(s) given as class methods on `Sinatra::Application`. It also handles exporting public methods to the top-level when classic style apps are executed.

A classic style application would use this extension as follows:

```
require 'sinatra'
require 'sinatra/linkblocker'

block_links_from 'digg.com'

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

Modular style applications must register the extension explicitly in their `Sinatra::Base` subclasses:

```
require 'sinatra/base'
require 'sinatra/linkblocker'

class Hello < Sinatra::Base
  register Sinatra::LinkBlocker

  block_links_from 'digg.com'

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

Setting Options and Other Extension Setup

Extensions can define options, routes, before filters, and error handlers by defining a `registered` method on the extension module. The `Module.registered` method is called immediately after the extension module is added to the `Sinatra::Base` subclass and is passed the class that the module was registered with.

The following example creates a very simple extension that adds basic session auth support. Options are added for the username and password, routes are defined for logging in, and helper methods are provided for determining whether a user has been authorized:

```
require 'sinatra/base'

module Sinatra
  module SessionAuth

    module Helpers
      def authorized?
        session[:authorized]
      end

      def authorize!
        redirect '/login' unless authorized?
      end
```

```
      def logout!
        session[:authorized] = false
      end
    end

    def self.registered(app)
      app.helpers SessionAuth::Helpers

      app.set :username, 'frank'
      app.set :password, 'changeme'

      app.get '/login' do
        "<form method='POST' action='/login'>" +
        "<input type='text' name='user'>" +
        "<input type='text' name='pass'>" +
        "</form>"
      end

      app.post '/login' do
        if params[:user] == options.username && params[:pass] == options.password
          session[:authorized] = true
          redirect '/'
        else
          session[:authorized] = false
          redirect '/login'
        end
      end
    end
  end

  register SessionAuth
end
```

A classic application would use this extension by requiring the extension library, overriding options, and using the helpers provided:

```
require 'sinatra'
require 'sinatra/sessionauth'

set :password, 'hoboken'

get '/public' do
  if authorized?
    "Hi. I know you."
  else
    "Hi. We haven't met. <a href='/login'>Login, please.</a>"
  end
end

get '/private' do
  authorize!
  'Thanks for logging in.'
end
```

A modular application is different only in that it must register the extension module explicitly:

```
require 'sinatra/base'
```

```
require 'sinatra/sessionauth'

class MyApp < Sinatra::Base
  register Sinatra::SessionAuth

  set :password, 'hoboken'

  get '/public' do
    if authorized?
      "Hi. I know you."
    else
      "Hi. We haven't met. <a href='/login'>Login, please.</a>"
    end
  end

  get '/private' do
    authorize!
    'Thanks for logging in.'
  end
end
```

Building and Packaging Extensions

Sinatra extensions should be built as separate libraries and packaged as gems or as single files that can be included within an application's `lib` directory. The ideal process for using an extensions is installing a gem and requiring a single file.

The following is an example file layout for a typical extension packaged as a gem:

```
sinatra-fu
|-- README
|-- LICENSE
|-- Rakefile
|-- lib
|   `-- sinatra
|       `-- fu.rb
|-- test
|   `-- spec_sinatra_fu.rb
`-- sinatra-fu.gemspec
```

Sinatra

Frequently Asked Questions

How do I make my Sinatra app reload on changes?

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

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

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

```
$ gem install rerun
```

Now if you start your Sinatra app like this:

```
$ ruby app.rb
```

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

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

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

```
$ rerun 'rackup'
```

You get the idea.

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

What are my deployment options?

See the book.

How do I use sessions?

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

```
enable :sessions

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

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

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

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

How do I use session-based flash?

Use Rack::Flash.

Can I run Sinatra under Ruby 1.9?

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

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

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

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

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

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

How do I access helpers from within my views?

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

In `hello.rb`:

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

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

In `views/hello.haml`:

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

How do I render partials?

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

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

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

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

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

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

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

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

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

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

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

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

Sure:

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

Seriously.

How do I make the trailing slash optional?

Put a question mark after it:

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

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

How do I render templates nested in subdirectories?

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

with:

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

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

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

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

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

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

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

How do I send email from Sinatra?

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

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

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

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

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

Frank
```

And in `mailerapp.rb`:

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

How do I escape HTML?

Use Rack::Utils in your helpers as follows:

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

Now you can escape HTML in your templates like this:

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

Thanks to Chris Schneider for the tip!

How do I automatically escape HTML?

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

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

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

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

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

How do I use ActiveRecord migrations?

From Adam Wiggins's blog:

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

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

How do I use HTTP authentication?

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

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

```
require 'sinatra'

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

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

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

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

```
require 'sinatra'

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

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

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

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

How do I test HTTP authentication?

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

```
require 'sinatra'

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

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

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

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

require 'application'

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

  def app
    Sinatra::Application
  end

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

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

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

Sinatra

NOTE: Feel free to fork Sinatra's website on GitHub and add your own entry to the `wild.markdown` file. Just push and send a pull request. We'll take care of the rest.

Applications

- Flip Text A simple app used to flip text upside down.
- SequenceServer An easy-to-setup BLAST sequence search server
- Flickr Group Viewer is a (nicer?) way to view photos in a flickr group – on GitHub.
- OnBoard a web interface to manage Linux-based network appliances.
- Tweetrad.io text to speech conversion for twitter
- Tweeps conference app to aggregate tweets by attendees
- Integrity the easy and fun Continuous Integration server
- Sum is an email-based budgeting application
- Marley the blog engine without <textareas>
- Postview a simple blog-engine that render text files written in Markdown.
- Scanty an even more minimalist blogging engine
- Scanty using CouchDB
- Scanty using CouchDB 0.9.0a and CouchRest 0.16
- Haze a lightweight and minimalist blogging engine using flat text files.
- Wind HTML5 blog engine focused in a easy, extensible and fast admin.
- git-wiki a git-powered wiki
- Wordnatra an interface to Princeton's WordNet lexical dictionary
- GitHub-FogBuz logs your GitHub commits with FogBuz
- Notable listens on Jabber to take your notes and then displays them back to you
- Dash Sinatra provides a Ruby API to push metrics from an app to the FiveRuns Dash service
- Giftsmas Gift Tracking Website using Sinatra, Sequel, and Scaffolding Extensions
- Toopaste Pastebin site created with Sinatra and DataMapper
- Columnlog Tiny feed fetching app
- Kapow Comics on your iPhone
- Knapsack Pack pages into data URIs
- Thumblmonks Delayed HTTP Requests
- weaky a basic CouchDB wiki
- Amnesia an app that presents memcached server stats
- So Nice is a Small web interface to control iTunes, Rhythmbox or MPD
- sinatra-rubygems A complete reimplementation of the gem server command as a Sinatra application
- Nesta is an SEO oriented CMS/blog for developers
- Hancock A Single Sign On provider based on OpenID

Hancock Client Rack middleware client for the Hancock server, written in Sinatra
- Rails Searchable API Doc runs on Sinatra
- Sinatra Saucer JRuby web application frontend for Flying Saucer, which converts XHTML into a PDF
- WineAdds micro-app that calculates common additions to wine
- Deltacloud Deltacloud protects your apps from cloud API changes and incompatibilies. REST API is written using Sinatra
- Jaconda A simple team collaboration service that allows you to create chat rooms for groups through Gtalk/Jabber protocol. REST API is written in Sinatra
- Soxer Soxer is a lean but infinitely extensible web publishing tool.
- SinMagick a front-end for image processing and thumbnailing with flexible storage options.
- Fundry - Crowdfunding for Software Development - Fundry is designed around software projects, helping developers get paid for developing new features, and enabling your community to pledge to get the features they want. The site and API (coming soon) is all written in Sinatra.
- Picky - a fast & clever semantic search engine.
- Headhunter - giving Twitter profile pics a permanent URL
- blitz.io - Making load and performance testing a fun sport
- GitHub High Scores - A fun way to rank GitHub repository contributors in a 8-bit, 80's-tastic aracdey viewing environment.
- Git-Webby - Git Smart HTTP Ruby/Sinatra implementation with useful features.
- Trudy - A Nabaztag server.
- Markdown Tree - Serve a Hierarchy of Markdown files simply (like mksite or other site generators, only dynamic)
- Fracture.it URL shortening service [Source]
- streetart.io streetart.io allows you to share and discover interesting street art, public art and sculptures around you and your city. API is powered by Sinatra and Redis.
- TaskwarriorWeb - A Sinatra-based web interface for the Taskwarrior todo application. Because being a neck-beard is only fun sometimes.
- Travis CI - Free Hosted Continuous Integration Platform for the Open Source Community

Libraries and extensions

- Sinatra::SimpleNavigation Easy creation of navigations (with multiple levels) with sinatra and the simple-navigation gem
- sinatra_more Library with agnostic generators, form builders, named route mappings, easy mailer support, and more.
- Sinatra's Hat Mount models as web services in Sinatra with ease
- Classy Resources Think resources controller, except for sinatra
- Sinatra Ditties A collection of plugins and useful helpers
- Chicago runtime and testing helpers, extensions, and macros commonly used by Thumblemonks
- Capinatra Quickly deploy Sinatra apps to Passenger.
- Frankie Easy creation of Facebook applications with Sinatra and the Facebook gem
- Sinatra Application Template Base app with DataMapper, Haml and RSpec. Fork and build.
- Sinatra OAuth Provider an OAuth provider implemented with Sinatra
- Spork Executes some aynchronous code using Sinatra running under Passenger
- Retweet Create Twitter apps like tweetdreams.org with ease

- Rack Flash Simple flash hash implementation for Rack with extra sugar for Sinatra.
- Sinatra-REST generates RESTful routes for your models (ActiveRecord, DataMapper, Stone)
- R18n Agnostic i18n library with Sinatra extension.
- url_for Construct absolute paths and full URLs for current application.
- WebIRC Web-based IRC client.
- content_for Rails-like `content_for` helper for your views (supports ERB and Haml.)
- Sinatra Effigy Sinatra extension for Effigy (HTML in .html files, Ruby in .rb files) views.
- Sinatra Mongoid Sinatra extension for Mongoid (MongoDB ORM).
- Haml::More Adds more functionality to Haml and Sass
- Sinatra::AdvancedRoutes Makes routes first class objects
- Sinatra::Compass Integrates the Compass stylesheet framework
- Sinatra::ConfigFile Adds YAML config file support
- Sinatra::MoreServer Adds support for more web servers to Sinatra::Base#run!
- Sinatra::Namespace Adds namespaces, allows namespaces to have local helpers.
- Sinatra::Reloader Advanced and fast code reloader
- Sinatra::Sugar Extensions for Sinatra's standard methods, like #set or #register
- Sinatra::TestHelper Adds helper methods and better integration for various testing frameworks
- Sinatra::WebInspector Allows you to inspect a running app
- Yard::Sinatra Displays Sinatra routes (including comments) in YARD output
- BigBand Pre-configured replacement for Sinatra::Base setting up useful extensions
- Sinatra::Jsonp JSONP output helper for Sinatra.
- Bowtie Admin generator for DataMapper models, on Sinatra.
- Sinatra::Kittens Random kitten pictures for Sinatra not_found pages.
- Sinatra::FuzzyLayout Extension for enabling or disabling layout template for views using regex.

Websites

- LinkedIn is a social network for business contacts
- I Shoot Film presents Funkaoshi's film photography posted to Flickr organized by film rolls - on GitHub.
- Speed of Animals: imagine being a spotted hyena...
- Bloor and Lansdowne is Blansdowne is a photoblog about a neighbourhood in Toronto.
- NYLS Upload Tool Web-based replacement of FTP
- Fabulously40 Mobile Sinatra powered mobile site of Fabulously40
- UploadBooth Sinatra powered Upload Service
- PasteBooth Sinatra powered multi-lingual Paste Service
- ShrinkBooth Sinatra powered multi-lingual URL shortening Service
- Calendar About Nothing Seinfeld calendar for hackers
- irclogger Logging urz IRC channel #sinatra
- URL Unwind Don't get RickRolled again
- Tweetdreams Twitter dream journal
- Does Follow Twitter tool for figuring out who follows whom
- portfolio.quirkey.com
- Gusg.us Blog - Ridding the long bus
- Deleanotes
- themoviedb.org The name says it all :)
- columnlog Tiny feed fetching app
- Your Fiveruns Dashboard

- IronRuby Watch their great stats here
- MooURL The web's cutest URL shortening service
- FlexCode iPhone development
- California PATH program - UC Berkley In-vehicle applications on their Vehicle-Infrastructure Integration server
- TweepDiff Compare friends/followers of two Twitter users
- Effectif Development
- Chirp-chirp Write a Sinatra-based Twitter clone in 200 lines of Ruby code
- Rdoc.info Auto-generate and host Ruby library documentation using GitHub web hooks
- All Sorts An aggregator for novel *collective nouns*, driven by Twitter
- Congrelate Congress data mashup visualization tool.
- Scholas Social file-sharing for academics.
- Gemcutter Awesome gem hosting.
- SavvyFranchises Franchise Search + Discovery for the Savvy Entrepreneur.
- The Setup The setup is a bunch of nerdy interviews (Source)
- A Good Company An awesome little portfolio for a great design company.
- dbpedia lite takes some of the structured data in Wikipedia and presents it as Linked Data (Source).
- Evan Lecklider Website showing off work, code examples and experiments built with Sinatra and Compass.
- Brightspoke.com matches people with bicycles.
- Kittygram shows cute cat photos from around the world.
- happens.co.za Add countdown times for any "happening" or event and share it with others.
- DarkArts Studios Projects and source code of Clive Crous
- Ant genome database
- GitHub High Scores A fun way to rank GitHub repository contributors in a 8-bit, 80's-tastic aracdey viewing environment.
- Conker Software Micro-company of Tom Philip that makes Web, Android and Boxee apps.
- Similander extracts keywords from a given URL to find similar pages (Source).
- Sombras de Papel Website of a pop-rock band from Barcelona (Spain) entirely made using Sinatra + HTML5. There's also an automatic mobile site.
- streetart.io streetart.io allows you to share and discover interesting street art, public art and sculptures around you and your city. The website is a sinatra app which pulls data via the API.
- Scrum Primer - A short introduction to Scrum. The code is at Scrum Primer @ Github. Also uses r18n for Sinatra
- Jembatan Keledai A collection of mnemonics in Indonesian.

Companies using Sinatra

- Heroku
- GitHub
- Apple
- BBC
- LinkedIn
- Agency Rainford
- Songbird
- Curbed Network
- Meticulo

- Rock-n-code
- Solutious
- Centro
- Apartment Therapy
- Inventive Labs
- ENTP
- FiveRuns
- Engine Yard
- Citrusbyte
- Lomographic Society International
- Swipht Technologies
- Nth Metal Interactive
- Osborne Brook
- Robot Mode
- thoughtbot
- Forward
- WyeWorks
- Idealist Digital
- A Good Company
- Ylastic
- Fork
- Tractor Feed
- Ant genomics database - Fourmidable
- Mu Dynamics
- Scribble Squad
- Stanford
- Hiboux
- PentesterLab
- Cedar Creek IT Solutions
- Animoto
- Red Hat
- Travis CI
- Clearhaus
- Odd-e
- Digital Deployment