Sinatra

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

簡介

注：本文档仅仅是英文版的翻译，会出现内容没有及时更新的情况发生。 如有不一致的地方，请以英文版为准。

Sinatra是一个基于Ruby语言，以最小精力为代价快速创建web应用为目的的DSL（领域专属语言）：

```
# myapp.rb
require 'sinatra'

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

安装gem然后运行：

```
gem install sinatra
ruby myapp.rb
```

在该地址查看： localhost:4567

推荐同时运行`gem install thin`，Sinatra会优先选择thin作为服务器。

路由

在Sinatra中，一个路由是一个HTTP方法与URL匹配范式的配对。 每个路由都与一个代码块关联：

```
get '/' do
  .. 显示一些事物 ..
end

post '/' do
```

```
  .. 创建一些事物 ..
end

put '/' do
  .. 更新一些事物 ..
end

delete '/' do
  .. 消灭一些事物 ..
end

options '/' do
  .. 满足一些事物 ..
end
```


路由按照它们被定义的顺序进行匹配。 第一个与请求匹配的路由会被调用。

路由范式可以包括具名参数，可通过params哈希表获得：

```
get '/hello/:name' do
  # 匹配 "GET /hello/foo" 和 "GET /hello/bar"
  # params[:name] 的值是 'foo' 或者 'bar'
  "Hello #{params[:name]}"
end
```

你同样可以通过代码块参数获得具名参数：

```
get '/hello/:name' do |n|
  "Hello #{n}"
end
```

路由范式也可以包含通配符参数， 可以通过params[:splat]数组获得。

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

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

通过正则表达式匹配的路由：

```
get %r{/hello/([\w]+)} do
  "Hello, #{params[:captures]}"
end
```

或者使用代码块参数：

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

```
  "Hello, #{ }!"
end
```

条件

路由也可以包含多样的匹配条件，比如user agent:

```
get '/foo', :agent => /Songbird (\d\.\d)[\d\/]*?/ do
  "你使用的是Songbird 版本号 #{ :agent 0 }"
end

get '/foo' do
  # 匹配除Songbird以外的浏览器
end
```

其他可选的条件是 host_name 和 provides:

```
get '/', :host_name => /^admin\./ do
  "管理员区域，无权访问！"
end

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

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

你也可以很轻松地定义自己的条件:

```
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
```

返回值

路由代码块的返回值至少决定了返回给HTTP客户端的响应体， 或者至少决定了在Rack堆栈中的下一个中间件。 大多数情况下，将是一个字符串，就像上面的例子中的一样。 但是其他值也是可以接受的。

你可以返回任何对象，或者是一个合理的Rack响应， Rack body对象或者HTTP状态码:

那样，我们可以轻松的实现例如流式传输的例子:

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

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

自定义路由匹配器

如上显示，Sinatra内置了对于使用字符串和正则表达式作为路由匹配的支持。 但是，它并没有只限于此。 你可以非常容易地定义你自己的匹配器:

```
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
```

请注意上面的例子可能超工程了， 因为它也可以用更简单的方式表述:

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

或者，使用消极向前查找:

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

静态文件

静态文件是从 `./public_folder` 目录提供服务。你可以通过设置`:public` 选项设定一个不同的位置:

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

请注意public目录名并没有被包含在URL之中。文件 `./public/css/style.css`是通过 `http://example.com/css/style.css`地址访问的。

视图 / 模板

模板被假定直接位于`./views`目录。 要使用不同的视图目录:

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

请记住一件非常重要的事情，你只可以通过符号引用模板， 即使它们在子目录下 （在这种情况下，使用 `:'subdir/template'`）。 你必须使用一个符号， 因为渲染方法会直接地渲染任何传入的字符串。

Haml模板

需要引入 `haml` gem/library以渲染 HAML 模板:

```
# 你需要在你的应用中引入 haml
require 'haml'

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

渲染 `./views/index.haml`。

Haml的选项 可以通过Sinatra的配置全局设定， 参见 选项和配置， 也可以个别的被覆盖。

```
set :haml, {:format => :html5 } # 默认的Haml输出格式是 :xhtml

get '/' do
  haml :index, :haml_options => {:format => :html4 } # 被覆盖，变成:html4
end
```

Erb模板

```
# 你需要在你的应用中引入 erb
require 'erb'

get '/' do
```

```
  erb :index
end
```

渲染 `./views/index.erb`

Erubis

需要引入 `erubis` gem/library以渲染 erubis 模板:

```
# 你需要在你的应用中引入 erubis
require 'erubis'

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

渲染 `./views/index.erubis`

使用Erubis代替Erb也是可能的:

```
require 'erubis'
Tilt.register :erb, Tilt[:erubis]

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

使用Erubis来渲染 `./views/index.erb`。

Builder 模板

需要引入 `builder` gem/library 以渲染 builder templates:

```
# 需要在你的应用中引入builder
require 'builder'

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

渲染 `./views/index.builder`。

Nokogiri 模板

需要引入 `nokogiri` gem/library 以渲染 nokogiri 模板:

```
# 需要在你的应用中引入 nokogiri
require 'nokogiri'

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

渲染 `./views/index.nokogiri`。

Sass 模板

需要引入 `haml` 或者 `sass` gem/library 以渲染 Sass 模板：

```
# 需要在你的应用中引入 haml 或者 sass
require 'sass'

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

渲染 `./views/stylesheet.sass`。

Sass 的选项 可以通过Sinatra选项全局设定， 参考 选项和配置（英文）, 也可以在个体的基础上覆盖。

```
set :sass, {:style => :compact } # 默认的 Sass 样式是 :nested

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

Scss 模板

需要引入 `haml` 或者 `sass` gem/library 来渲染 Scss templates：

```
# 需要在你的应用中引入 haml 或者 sass
require 'sass'

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

渲染 `./views/stylesheet.scss`。

Scss的选项 可以通过Sinatra选项全局设定， 参考 选项和配置（英文）, 也可以在个体的基础上覆盖。

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

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

Less 模板

需要引入 `less` gem/library 以渲染 Less 模板:

```
# 需要在你的应用中引入 less
require 'less'

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

渲染 `./views/stylesheet.less`。

Liquid 模板

需要引入 `liquid` gem/library 来渲染 Liquid 模板:

```
# 需要在你的应用中引入 liquid
require 'liquid'

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

渲染 `./views/index.liquid`。

因为你不能在Liquid 模板中调用 Ruby 方法 (除了 `yield`)， 你几乎总是需要传递locals给它:

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

Markdown 模板

需要引入 `rdiscount` gem/library 以渲染 Markdown 模板:

```
# 需要在你的应用中引入rdiscount
require "rdiscount"

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

渲染 `./views/index.markdown` (`md` 和 `mkd` 也是合理的文件扩展名)。

在markdown中是不可以调用方法的，也不可以传递 locals给它。 你因此一般会结合其他的渲染引擎来使用它：

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

请注意你也可以从其他模板中调用 markdown 方法：

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

既然你不能在Markdown中调用Ruby，你不能使用Markdown编写的布局。 不过，使用其他渲染引擎作为模版的布局是可能的， 通过传递`:layout_engine`选项:

```
get '/' do
  markdown :index, :layout_engine => :erb
end
```

这将会渲染 `./views/index.md` 并使用 `./views/layout.erb` 作为布局。

请记住你可以全局设定这个选项:

```
set :markdown, :layout_engine => :haml, :layout => :post

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

这将会渲染 `./views/index.markdown` (和任何其他的 Markdown 模版) 并使用 `./views/post.haml` 作为布局.

也可能使用BlueCloth而不是RDiscount来解析Markdown文件:

```
require 'bluecloth'

Tilt.register 'markdown', BlueClothTemplate
Tilt.register 'mkd',      BlueClothTemplate
Tilt.register 'md',       BlueClothTemplate

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

使用BlueCloth来渲染 `./views/index.md` 。

Textile 模板

需要引入 `RedCloth` gem/library 以渲染 Textile 模板：

```
# 在你的应用中引入redcloth
require "redcloth"

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

渲染 `./views/index.textile`。

在textile中是不可以调用方法的，也不可以传递 locals给它。 你因此一般会结合其他的渲染引擎来使用它：

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

请注意你也可以从其他模板中调用`textile`方法：

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

既然你不能在Textile中调用Ruby，你不能使用Textile编写的布局。 不过，使用其他渲染引擎作为模版的布局是可能的， 通过传递`:layout_engine`选项:

```
get '/' do
  textile :index, :layout_engine => :erb
end
```

这将会渲染 `./views/index.textile` 并使用 `./views/layout.erb` 作为布局。

请记住你可以全局设定这个选项:

```
set :textile, :layout_engine => :haml, :layout => :post

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

这将会渲染 `./views/index.textile` (和任何其他的 Textile 模版) 并使用 `./views/post.haml` 作为布局.

RDoc 模板

需要引入 `RDoc` gem/library 以渲染RDoc模板：

```
# 需要在你的应用中引入rdoc/markup/to_html
require "rdoc"
require "rdoc/markup/to_html"

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

渲染 `./views/index.rdoc`。

在rdoc中是不可以调用方法的，也不可以传递locals给它。 你因此一般会结合其他的渲染引擎来使用它：

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

请注意你也可以从其他模板中调用rdoc方法：

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

既然你不能在RDoc中调用Ruby，你不能使用RDoc编写的布局。 不过，使用其他渲染引擎作为模版的布局是可能的， 通过传递`:layout_engine`选项:

```
get '/' do
  rdoc :index, :layout_engine => :erb
end
```

这将会渲染 `./views/index.rdoc` 并使用 `./views/layout.erb` 作为布局。

请记住你可以全局设定这个选项:

```
set :rdoc, :layout_engine => :haml, :layout => :post

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

这将会渲染 `./views/index.rdoc` (和任何其他的 RDoc 模版) 并使用 `./views/post.haml` 作为布局.

Radius 模板

需要引入 `radius` gem/library 以渲染 Radius 模板：

```
# 需要在你的应用中引入radius
require 'radius'

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

渲染 `./views/index.radius`。

因为你不能在Radius 模板中调用 Ruby 方法 (除了 `yield`) ， 你几乎总是需要传递locals给它：

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

Markaby 模板

需要引入markaby gem/library以渲染Markaby模板:

```
#需要在你的应用中引入 markaby
require 'markaby'

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

渲染 ./views/index.mab。

你也可以使用嵌入的 Markaby:

```
get '/' do
  markaby { h1 "Welcome!" }
end
```

Slim 模板

需要引入 slim gem/library 来渲染 Slim 模板:

```
# 需要在你的应用中引入 slim
require 'slim'

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

渲染 ./views/index.slim。

Creole 模板

需要引入 creole gem/library 来渲染 Creole 模板:

```
# 需要在你的应用中引入 creole
require 'creole'

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

渲染 ./views/index.creole。

CoffeeScript 模板

需要引入 `coffee-script` gem/library 并至少满足下面条件一项 以执行Javascript:

- `node` (来自 Node.js) 在你的路径中
- 你正在运行 OSX
- `therubyracer` gem/library

请察看 github.com/josh/ruby-coffee-script 获取更新的选项。

现在你可以渲染 CoffeeScript 模版了:

```
# 需要在你的应用中引入coffee-script
require 'coffee-script'

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

渲染 `./views/application.coffee`。

嵌入模板字符串

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

渲染嵌入模板字符串。

在模板中访问变量

模板和路由执行器在同样的上下文求值。 在路由执行器中赋值的实例变量可以直接被模板访问。

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

或者，显式地指定一个本地变量的哈希：

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

典型的使用情况是在别的模板中按照局部模板的方式来渲染。

内联模板

模板可以在源文件的末尾定义：

```
require 'sinatra'

get '/' do
  haml :index
end

__END__

@@ layout
%html
  = yield

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

注意：引入sinatra的源文件中定义的内联模板才能被自动载入。 如果你在其他源文件中有内联模板， 需要显式执行调用`enable :inline_templates`。

具名模板

模板可以通过使用顶层 `template` 方法定义：

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

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

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

如果存在名为“layout”的模板，该模板会在每个模板渲染的时候被使用。 你可以单独地通过传送 `:layout => false`来禁用， 或者通过`set :haml, :layout => false`来禁用他们。

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

关联文件扩展名

为了关联一个文件扩展名到一个模版引擎，使用 `Tilt.register`。比如，如果你喜欢使用 `tt` 作为Textile模版的扩展名，你可以这样做:

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

添加你自己的模版引擎

首先，通过Tilt注册你自己的引擎，然后创建一个渲染方法:

```
Tilt.register :myat, MyAwesomeTemplateEngine

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

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

渲染 `./views/index.myat`。察看 github.com/rtomayko/tilt 来更多了解Tilt.

过滤器

前置过滤器在每个请求前，在请求的上下文环境中被执行， 而且可以修改请求和响应。 在过滤器中设定的实例变量可以被路由和模板访问:

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

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

后置过滤器在每个请求之后，在请求的上下文环境中执行， 而且可以修改请求和响应。 在前置过滤器和路由中设定的实例变量可以被后置过滤器访问:

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

请注意：除非你显式使用 `body` 方法，而不是在路由中直接返回字符串， 消息体在后置过滤器是不可用的， 因

为它在之后才会生成。

过滤器可以可选地带有范式， 只有请求路径满足该范式时才会执行:

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

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

和路由一样，过滤器也可以带有条件:

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

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

辅助方法

使用顶层的 `helpers` 方法来定义辅助方法， 以便在路由处理器和模板中使用:

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

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

使用 Sessions

Session被用来在请求之间保持状态。如果被激活，每一个用户会话 对应有一个session哈希:

```
enable :sessions

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

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

请注意 enable :sessions 实际上保存所有的数据在一个cookie之中。 这可能不会总是做你想要的（比如，保存大量的数据会增加你的流量）。 你可以使用任何的Rack session中间件，为了这么做， *不要*调用 enable :sessions，而是 按照自己的需要引入你的中间件：

```
use Rack::Session::Pool, :expire_after => 2592000

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

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

挂起

要想直接地停止请求，在过滤器或者路由中使用：

```
halt
```

你也可以指定挂起时的状态码：

```
halt 410
```

或者消息体：

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

或者两者;

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

也可以带消息头：

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

让路

一个路由可以放弃处理，将处理让给下一个匹配的路由，使用 pass：

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

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

路由代码块被直接退出，控制流继续前进到下一个匹配的路由。 如果没有匹配的路由，将返回404。

触发另一个路由

有些时候，pass 并不是你想要的，你希望得到的是另一个路由的结果 。简单的使用 call 可以做到这一点:

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

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

请注意在以上例子中，你可以更加简化测试并增加性能，只要简单的移动

```
<tt>"bar"</tt>到一个被<tt>/foo</tt>
```

和 /bar同时使用的helper。

如果你希望请求被发送到同一个应用，而不是副本， 使用 call! 而不是 call.

察看 Rack specification 如果你想更多了解 call.

设定 消息体，状态码和消息头

通过路由代码块的返回值来设定状态码和消息体不仅是可能的，而且是推荐的。 但是，在某些场景中你可能想在作业流程中的特定点上设置消息体。 你可以通过 body 辅助方法这么做。 如果你这样做了， 你可以在那以后使用该方法获得消息体:

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

after do
  puts body
end
```

也可以传一个代码块给 body，它将会被Rack处理器执行（ 这将可以被用来实现streaming，参见“返回值”）。

和消息体类似，你也可以设定状态码和消息头:

```
```

```
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
```

如同 `body`, 不带参数的 `headers` 和 `status` 可以用来访问 他们你的当前值.

媒体类型

当使用 `send_file` 或者静态文件的场合，你的媒体类型可能 Sinatra并不理解。使用 `mime_type` 通过文件扩展名来注册它们:

```
mime_type :foo, 'text/foo'
```

你也可以通过 `content_type` 辅助方法使用:

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

生成 URL

为了生成URL，你需要使用 `url` 辅助方法， 例如，在Haml中:

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

它会根据反向代理和Rack路由，如果有的话，来计算生成的URL。

这个方法还有一个别名 `to` (见下面的例子).

浏览器重定向

你可以通过 `redirect` 辅助方法触发浏览器重定向:

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

任何额外的参数都会被以 `halt`相同的方式来处理:

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

你可以方便的通过 `redirect back`把用户重定向到来自的页面:

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

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

为了传递参数给redirect，或者加入query:

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

或者使用session:

```
enable :sessions

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

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

缓存控制

正确地设定消息头是恰当的HTTP缓存的基础。

你可以方便的设定 Cache-Control 消息头，像这样:

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

核心提示: 在前置过滤器中设定缓存.

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

如果你正在用 `expires` 辅助方法设定对应的消息头 `Cache-Control` 会自动设定:

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

为了合适地使用缓存，你应该考虑使用 `etag` 和 `last_modified`方法。. 推荐在执行繁重任务*之前*使用这些helpers， 他们会立刻发送响应，如果客户端在缓存中已经有了当前版本。

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

使用 weak ETag 也是有可能的:

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

这些辅助方法并不会为你做任何缓存，而是将必要的信息传送给你的缓存 如果你在寻找缓存的快速解决方案，试试 rack-cache:

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

use Rack::Cache

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

发送文件

为了发送文件，你可以使用 `send_file` 辅助方法:

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

也可以带一些选项:

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

可用的选项有:

filename
: 响应中的文件名，默认是真实文件的名字。

last_modified

Last-Modified 消息头的值，默认是文件的mtime（修改时间）。

type
: 使用的内容类型，如果没有会从文件扩展名猜测。

disposition
: 用于 Content-Disposition，可能的包括： `nil` (默认), `:attachment` 和 `:inline`

length
: Content-Length 的值，默认是文件的大小。

如果Rack处理器支持，Ruby进程除streaming以外的方式会被使用。 如果你使用这个辅助方法， Sinatra会自动处理range请求。

访问请求对象

传入的请求对象可以在请求层（过滤器，路由，错误处理） 通过 `request` 方法被访问：

```
# 在 http://example.com/example 上运行的应用
get '/foo' do
  request.body              # 被客户端设定的请求体（见下）
  request.scheme            # "http"
  request.script_name       # "/example"
  request.path_info         # "/foo"
  request.port              # 80
  request.request_method    # "GET"
  request.query_string      # ""
  request.content_length    # request.body的长度
  request.media_type        # request.body的媒体类型
  request.host              # "example.com"
  request.get?              # true (其他动词也具有类似方法)
  request.form_data?        # false
  request["SOME_HEADER"]    # SOME_HEADER header的值
  request.referrer          # 客户端的referrer 或者 '/'
  request.user_agent        # user agent (被 :agent 条件使用)
  request.cookies           # 浏览器 cookies 哈希
  request.xhr?              # 这是否是ajax请求?
  request.url               # "http://example.com/example/foo"
  request.path              # "/example/foo"
  request.ip                # 客户端IP地址
  request.secure?           # false（如果是ssl则为true）
  request.forwarded?        # true （如果是运行在反向代理之后）
  request.env               # Rack中使用的未处理的env哈希
end
```

一些选项，例如 `script_name` 或者 `path_info` 也是可写的：

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

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

`request.body` 是一个IO或者StringIO对象：

```
post "/api" do
  request.body.rewind  # 如果已经有人读了它
  data = JSON.parse request.body.read
  "Hello #{data['name']}"
end
```

附件

你可以使用 `attachment` 辅助方法来告诉浏览器响应 应当被写入磁盘而不是在浏览器中显示。

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

你也可以传递一个文件名:

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

查找模板文件

`find_template` 辅助方法被用于在渲染时查找模板文件:

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

这并不是很有用。但是在你需要重载这个方法 来实现你自己的查找机制的时候有用。 比如，如果你想支持多于一个视图目录:

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

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

另一个例子是为不同的引擎使用不同的目录:

```
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
```

你可以很容易地包装成一个扩展然后与他人分享!

请注意 `find_template` 并不会检查文件真的存在， 而是对任何可能的路径调用给入的代码块。这并不会带来性能问题， 因为 `render` 会在找到文件的时候马上使用 `break` 。 同样的，模板的路径（和内容）会在除development mode以外的场合 被缓存。你应该时刻提醒自己这一点， 如果你真的想写一个非常疯狂的方法。

配置

运行一次，在启动的时候，在任何环境下:

```
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
```

只当环境 (RACK_ENV environment 变量) 被设定为 `:production`的时候运行:

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

当环境被设定为 `:production` 或者 `:test`的时候运行:

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

你可以使用 `settings` 获得这些配置:

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

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

可选的设置

absolute_redirects

如果被禁用，Sinatra会允许使用相对路径重定向， 但是，Sinatra就不再遵守 RFC 2616标准 (HTTP 1.1)， 该标准只允许绝对路径重定向。

如果你的应用运行在一个未恰当设置的反向代理之后， 你需要启用这个选项。注意 `url` 辅助方法 仍然会生成绝对 URL，除非你传入 `false` 作为第二参数。

默认禁用。

add_charsets

设定 `content_type` 辅助方法会 自动加上字符集信息的多媒体类型。

你应该添加而不是覆盖这个选项: `settings.add_charsets << "application/foobar"`

app_file

主应用文件，用来检测项目的根路径， views和public文件夹和内联模板。

bind

绑定的IP 地址 (默认: 0.0.0.0)。 仅对于内置的服务器有用。

default_encoding

默认编码 (默认为 `"utf-8"`)。

dump_errors

在log中显示错误。

environment

当前环境，默认是 `ENV['RACK_ENV']`， 或者 `"development"` 如果不可用。

logging

使用logger

lock

对每一个请求放置一个锁， 只使用进程并发处理请求。

如果你的应用不是线程安全则需启动。 默认禁用。

method_override

使用 `_method` 魔法以允许在旧的浏览器中在 表单中使用 put/delete 方法

port

监听的端口号。只对内置服务器有用。

prefixed_redirects

是否添加 `request.script_name` 到 重定向请求，如果没有设定绝对路径。那样的话 `redirect '/foo'` 会和 `redirect to('/foo')`起相同作用。默认禁用。

public_folder
: public文件夹的位置。

reload_templates
: 是否每个请求都重新载入模板。 在development mode和 Ruby 1.8.6 中被企业（用来 消除一个Ruby内存泄漏的bug）。

root
: 项目的根目录。

raise_errors
: 抛出异常（应用会停下）。

run
: 如果启用，Sinatra会开启web服务器。 如果使用rackup或其他方式则不要启用。

running
: 内置的服务器在运行吗？ 不要修改这个设置！

server
: 服务器，或用于内置服务器的列表。 默认是 [‘thin’, ‘mongrel’, ‘webrick’], 顺序表明了 优先级。

sessions
: 开启基于cookie的sesson。

show_exceptions
: 在浏览器中显示一个stack trace。

static
: Sinatra是否处理静态文件。 当服务器能够处理则禁用。 禁用会增强性能。 默认开启。

views
: views 文件夹。

错误处理

错误处理在与路由和前置过滤器相同的上下文中运行， 这意味着你可以使用许多好东西，比如 `haml`, `erb`, `halt`，等等。

未找到

当一个 `Sinatra::NotFound` 错误被抛出的时候， 或者响应状态码是404，`not_found` 处理器会被调用：

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

错误

`error` 处理器，在任何路由代码块或者过滤器抛出异常的时候会被调用。 异常对象可以通过`sinatra.error` Rack 变量获得：

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

自定义错误:

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

那么，当这个发生的时候:

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

你会得到:

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

另一种替代方法是，为一个状态码安装错误处理器:

```
error 403 do
  'Access forbidden'
end

get '/secret' do
  403
end
```

或者一个范围:

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

在运行在development环境下时，Sinatra会安装特殊的 `not_found` 和 `error` 处理器。

Rack 中间件

Sinatra 依靠 Rack, 一个面向Ruby web框架的最小标准接口。 Rack的一个最有趣的面向应用开发者的能力是支持“中间件”——坐落在服务器和你的应用之间， 监视 并/或 操作HTTP请求/响应以 提供多样类型的常用功能。

Sinatra 让建立Rack中间件管道异常简单， 通过顶层的 `use` 方法:

```
require 'sinatra'
require 'my_custom_middleware'
```

```
use Rack::Lint
use MyCustomMiddleware

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

use 的语义和在 Rack::Builder DSL(在rack文件中最频繁使用)中定义的完全一样。例如，use 方法接受 多个/可变 参数，包括代码块：

```
use Rack::Auth::Basic do |username, password|
  username == 'admin' && password == 'secret'
end
```

Rack中分布有多样的标准中间件，针对日志， 调试，URL路由，认证和session处理。 Sinatra会自动使用这里面的大部分组件， 所以你一般不需要显示地 use 他们。

测试

Sinatra的测试可以使用任何基于Rack的测试程序库或者框架来编写。 Rack::Test 是推荐候选：

```
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
```

请注意: 内置的 Sinatra::Test 模块和 Sinatra::TestHarness 类 在 0.9.2 版本已废弃。

Sinatra::Base – 中间件，程序库和模块化应用

把你的应用定义在顶层，对于微型应用这会工作得很好， 但是在构建可复用的组件时候会带来客观的不利， 比如构建Rack中间件，Rails metal，带有服务器组件的简单程序库， 或者甚至是Sinatra扩展。顶层的DSL污染了Object命名空间， 并假定了一个微型应用风格的配置 (例如, 单一的应用文件， ./public 和 ./views 目录，日志，异常细节页面，等等）。 这时应该让 Sinatra::Base 走到台前了：

```
require 'sinatra/base'

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

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

Sinatra::Base子类可用的方法实际上就是通过顶层 DSL 可用的方法。 大部分顶层应用可以通过两个改变转换成Sinatra::Base组件：

- 你的文件应当引入 `sinatra/base` 而不是 `sinatra`; 否则，所有的Sinatra的 DSL 方法将会被引进到 主命名空间。
- 把你的应用的路由，错误处理，过滤器和选项放在 一个Sinatra::Base的子类中。

+Sinatra::Base+ 是一张白纸。大部分的选项默认是禁用的， 包含内置的服务器。参见 选项和配置 查看可用选项的具体细节和他们的行为。

模块化 vs. 传统的方式

与通常的认识相反，传统的方式没有任何错误。 如果它适合你的应用，你不需要转换到模块化的应用。

和模块化方式相比只有两个缺点：

- 你对每个Ruby进程只能定义一个Sinatra应用，如果你需要更多， 切换到模块化方式。
- 传统方式使用代理方法污染了 Object 。如果你打算 把你的应用封装进一个 library/gem，转换到模块化方式。

没有任何原因阻止你混合模块化和传统方式。

如果从一种转换到另一种，你需要注意settings中的 一些微小的不同：

```
Setting             Classic                 Modular

app_file            file loading sinatra    nil
run                 $0 == app_file          false
logging             true                    false
method_override     true                    false
inline_templates    true                    false
```

运行一个模块化应用

有两种方式运行一个模块化应用，使用 `run!`来运行:

```
# 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
```

运行:

```
ruby my_app.rb
```

或者使用一个 `config.ru`，允许你使用任何Rack处理器:

```
# config.ru
require './my_app'
run MyApp
```

运行:

```
rackup -p 4567
```

使用config.ru运行传统方式的应用

编写你的应用:

```
# app.rb
require 'sinatra'

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

加入相应的 `config.ru`:

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

什么时候用 config.ru?

以下情况你可能需要使用 `config.ru`:

- 你要使用不同的 Rack 处理器部署 (Passenger, Unicorn, Heroku, ...).
- 你想使用一个或者多个 `Sinatra::Base`的子类.
- 你只想把Sinatra当作中间件使用，而不是端点。

你并不需要切换到`config.ru`仅仅因为你切换到模块化方式， 你同样不需要切换到模块化方式， 仅仅因为要运行 `config.ru`.

把Sinatra当成中间件来使用

不仅Sinatra有能力使用其他的Rack中间件，任何Sinatra 应用程序都可以反过来自身被当作中间件，被加在任何Rack端点前面。 这个端点可以是任何Sinatra应用，或者任何基于Rack的应用程序 (Rails/Ramaze/Camping/...)。

```
require 'sinatra/base'

class LoginScreen < Sinatra::Base
  enable :sessions

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

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

class MyApp < Sinatra::Base
  # 在前置过滤器前运行中间件
  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
```

变量域和绑定

当前所在的变量域决定了哪些方法和变量是可用的。

应用/类 变量域

每个Sinatra应用相当与Sinatra::Base的一个子类。 如果你在使用顶层DSL(`require 'sinatra'`)，那么这个类就是 Sinatra::Application，或者这个类就是你显式创建的子类。 在类层面，你具有的方法类似于 `get` 或者 `before`，但是你不能访问 `request` 对象或者 `session`，因为对于所有的请求， 只有单一的应用类。

通过 `set` 创建的选项是类层面的方法：

```
class MyApp < Sinatra::Base
  # 嘿，我在应用变量域!
  set :foo, 42
  foo # => 42

  get '/foo' do
    # 嘿，我不再处于应用变量域了!
  end
end
```

在下列情况下你将拥有应用变量域的绑定：

- 在应用类中
- 在扩展中定义的方法
- 传递给 `helpers` 的代码块
- 用作`set`值的过程/代码块

你可以访问变量域对象（就是应用类）就像这样：

- 通过传递给代码块的对象 (`configure { |c| ... }`)
- 在请求变量域中使用`settings`

请求/实例 变量域

对于每个进入的请求，一个新的应用类的实例会被创建 所有的处理器代码块在该变量域被运行。在这个变量域中， 你可以访问 `request` 和 `session` 对象，或者调用渲染方法比如 `erb` 或者 `haml`。你可以在请求变量域当中通过`settings`辅助方法 访问应用变量域：

```
class MyApp < Sinatra::Base
  # 嘿，我在应用变量域!
  get '/define_route/:name' do
    # 针对 '/define_route/:name' 的请求变量域
    @value = 42
```

```
    settings.get("/#{params[:name]}") do
      # 针对 "/#{params[:name]}" 的请求变量域
      @value # => nil (并不是相同的请求)
    end

    "Route defined!"
  end
end
```

在以下情况将获得请求变量域:

- get/head/post/put/delete 代码块
- 前置/后置 过滤器
- 辅助方法
- 模板/视图

代理变量域

代理变量域只是把方法转送到类变量域。可是， 他并非表现得100%类似于类变量域, 因为你并不能获得类的绑定: 只有显式地标记为供代理使用的方法才是可用的， 而且你不能和类变量域共享变量/状态。(解释：你有了一个不同的 `self`)。 你可以显式地增加方法代理，通过调用 `Sinatra::Delegator.delegate :method_name`。

在以下情况将获得代理变量域:

- 顶层的绑定，如果你做过 `require "sinatra"`
- 在扩展了 `Sinatra::Delegator` mixin的对象

自己在这里看一下代码: Sinatra::Delegator mixin 已经 被包含进了主命名空间。

命令行

Sinatra 应用可以被直接运行:

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

选项是:

```
-h # help
-p # 设定端口 (默认是 4567)
-o # 设定主机名 (默认是 0.0.0.0)
-e # 设定环境 (默认是 development)
-s # 限定 rack 服务器/处理器 (默认是 thin)
-x # 打开互斥锁 (默认是 off)
```

必要条件

推荐在 Ruby 1.8.7, 1.9.2, JRuby 或者 Rubinius 上安装Sinatra。

下面的Ruby版本是官方支持的:

Ruby 1.8.6
: 不推荐在1.8.6上安装Sinatra， 但是直到Sinatra 1.3.0发布才会放弃对它的支持。 RDoc 和 CoffeScript模板不被这个Ruby版本支持。 1.8.6在它的Hash实现中包含一个内存泄漏问题， 该问题会被1.1.1版本之前的Sinatra引发。 当前版本使用性能下降的代价排除了这个问题。你需要把Rack降级到1.1.x， 因为Rack \>= 1.2不再支持1.8.6。

Ruby 1.8.7
: 1.8.7 被完全支持，但是，如果没有特别原因， 我们推荐你升级到 1.9.2 或者切换到 JRuby 或者 Rubinius.

Ruby 1.9.2
: 1.9.2 被支持而且推荐。注意 Radius 和 Markaby 模板并不和1.9兼容。不要使用 1.9.2p0, 它被已知会产生 segmentation faults.

Rubinius
: Rubinius 被官方支持 (Rubinius \>= 1.2.2)， 除了Textile模板。

JRuby
: JRuby 被官方支持 (JRuby \>= 1.5.6)。 目前未知和第三方模板库有关的问题， 但是，如果你选择了JRuby，请查看一下JRuby rack 处理器， 因为 Thin web 服务器还没有在JRuby上获得支持。

我们也会时刻关注新的Ruby版本。

下面的 Ruby 实现没有被官方支持， 但是已知可以运行 Sinatra:

- JRuby 和 Rubinius 老版本
- MacRuby
- Maglev
- IronRuby
- Ruby 1.9.0 and 1.9.1

不被官方支持的意思是，如果在不被支持的平台上有运行错误， 我们假定不是我们的问题，而是平台的问题。

Sinatra应该会运行在任何支持上述Ruby实现的操作系统。

紧追前沿

如果你喜欢使用 Sinatra 的最新鲜的代码，请放心的使用 master 分支来运行你的程序，它会非常的稳定。

```
cd myapp
```

```
git clone git://github.com/sinatra/sinatra.git
ruby -Isinatra/lib myapp.rb
```

我们也会不定期的发布预发布gems，所以你也可以运行

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

来获得最新的特性。

通过Bundler

如果你想使用最新的Sinatra运行你的应用，通过 Bundler 是推荐的方式。

首先，安装bundler，如果你还没有安装:

```
gem install bundler
```

然后，在你的项目目录下，创建一个 `Gemfile`:

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

# 其他的依赖关系
gem 'haml'                    # 举例，如果你想用haml
gem 'activerecord', '~> 3.0'  # 也许你还需要 ActiveRecord 3.x
```

请注意在这里你需要列出你的应用的所有依赖关系。 Sinatra的直接依赖关系 (Rack and Tilt) 将会， 自动被Bundler获取和添加。

现在你可以像这样运行你的应用:

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

使用自己的

创建一个本地克隆并通过 `sinatra/lib` 目录运行你的应用， 通过 `$LOAD_PATH`:

```
cd myapp
git clone git://github.com/sinatra/sinatra.git
ruby -Isinatra/lib myapp.rb
```

为了在未来更新 Sinatra 源代码:

```
cd myapp/sinatra
git pull
```

全局安装

你可以自行编译 gem：

```
git clone git://github.com/sinatra/sinatra.git
cd sinatra
rake sinatra.gemspec
rake install
```

如果你以root身份安装 gems，最后一步应该是

```
sudo rake install
```

更多

- 项目主页（英文） - 更多的文档， 新闻，和其他资源的链接。
- 贡献 - 找到了一个bug？ 需要帮助？有了一个 patch?
- 问题追踪
- Twitter
- 邮件列表
- IRC: #sinatra on freenode.net

Sinatra

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

Introduction

Attention : Ce document correspond à la traduction de la version anglaise et il n'est peut être plus à jour.

Sinatra est un DSL pour créer rapidement et facilement des applications web en Ruby :

```
# mon_application.rb
require 'sinatra'

get '/' do
  'Bonjour le monde !'
end
```

Installez la gem et lancez avec :

```
$ gem install sinatra
$ ruby mon_application.rb
```

Le résultat est visible sur : http://localhost:4567

Il est recommandé d'exécuter également `gem install thin`, pour que Sinatra utilise le server Thin quand il est disponible.

Routes

Dans Sinatra, une route est une méthode HTTP couplée à un masque (pattern) URL. Chaque route est associée à un bloc :

```
get '/' do
  .. montrer quelque chose ..
```

```
end

post '/' do
  .. créer quelque chose ..
end

put '/' do
  .. remplacer quelque chose ..
end

patch '/' do
  .. changer quelque chose ..
end

delete '/' do
  .. effacer quelque chose ..
end

options '/' do
  .. apaiser quelquechose ..
end
```

Les routes sont évaluées dans l'ordre où elles ont été définies. La première route qui correspond à la requête est appelée.

Les masques peuvent inclure des paramètres nommés, accessibles par l'intermédiaire du hash `params` :

```
get '/bonjour/:nom' do
  # répond aux requêtes "GET /bonjour/foo" et "GET /bonjour/bar"
  # params[:nom] est 'foo' ou 'bar'
  "Bonjour #{params[:nom]} !"
end
```

Vous pouvez aussi accéder aux paramètres nommés directement grâce aux paramètres du bloc comme ceci :

```
get '/bonjour/:nom' do |n|
  "Bonjour #{n} !"
end
```

Une route peut contenir un splat (caractère joker), accessible par l'intermédiaire du tableau `params[:splat]` :

```
get '/dire/*/a/*' do
  # répond à /dire/bonjour/a/monde
  params[:splat] # => ["bonjour", "monde"]
end

get '/telecharger/*.*' do
  # répond à /telecharger/chemin/vers/fichier.xml
  params[:splat] # => ["chemin/vers/fichier", "xml"]
end
```

Ou par l'intermédiaire des paramètres du bloc :

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

Une route peut aussi être définie par une expression régulière :

```
get %r{/bonjour/([\w]+)} do
  "Bonjour, #{        :captures        } !"
end
```

Là encore on peut utiliser les paramètres de bloc :

```
get %r{/bonjour/([\w]+)} do |c|
  "Bonjour, #{ }  !"
end
```

Les routes peuvent aussi comporter des paramètres optionnels :

```
get '/posts.?:format?' do
  # répond à "GET /posts" et aussi à "GET /posts.json", "GET /posts.xml" etc...
end
```

A ce propos, à moins d'avoir désactivé la protection contre les attaques par "path transversal" (voir plus loin), l'URL demandée peut avoir été modifiée avant d'être comparée à vos routes.

Conditions

Les routes peuvent définir toutes sortes de conditions, comme par exemple le "user agent" :

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

get '/foo' do
  # Correspond à tous les autres navigateurs
end
```

Les autres conditions disponibles sont `host_name` et `provides` :

```
get '/', :host_name => /^admin\./ do
  "Zone Administrateur, Accès refusé !"
end

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

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

```
  builder :feed
end
```

Vous pouvez facilement définir vos propres conditions :

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

get '/gagner_une_voiture', :probability => 0.1 do
  "Vous avez gagné !"
end

get '/gagner_une_voiture' do
  "Désolé, vous avez perdu."
end
```

Utilisez un splat (caractère joker) dans le cas d'une condition qui prend plusieurs valeurs :

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

get "/mon/compte/", :auth => [:user, :admin] do
  "Informations sur votre compte"
end

get "/reserve/aux/admins/", :auth => :admin do
  "Seuls les administrateurs sont acceptés ici !"
end
```

Valeurs de retour

La valeur renvoyée par le bloc correspondant à une route constitue le corps de la réponse qui sera transmise au client HTTP ou du moins au prochain middleware dans la pile Rack. Le plus souvent, il s'agit d'une chaîne de caractères, comme dans les exemples précédents. Cependant, d'autres valeurs sont acceptées.

Vous pouvez renvoyer n'importe quel objet qu'il s'agisse d'une réponse Rack valide, d'un corps de réponse Rack ou d'un code statut HTTP :

Avec cela, on peut facilement implémenter un streaming par exemple :

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

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

Vous pouvez aussi utiliser le helper `stream` (présenté un peu plus loin) pour éviter la surcharge et intégrer le traitement relatif au streaming dans le bloc de code de la route.

Masques de route spécifiques

Comme cela a été vu auparavant, Sinatra offre la possibilité d'utiliser des masques sous forme de chaines de caractères ou des expressions régulières pour définir les routes. Mais il est possible de faire bien plus. Vous pouvez facilement définir vos propres masques :

```
class MasqueToutSauf
  Masque = Struct.new(:captures)

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

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

def tout_sauf(masque)
  MasqueToutSauf.new(masque)
end

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

Notez que l'exemple ci-dessus est bien trop compliqué et que le même résultat peut être obtenu avec :

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

Ou bien en utilisant la forme négative :

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

Fichiers statiques

Les fichiers du dossier `./public` sont servis de façon statique. Vous avez la possibilité d'utiliser un

autre répertoire en définissant le paramètre `:public_folder` :

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

Notez que le nom du dossier public n'apparait pas dans l'URL. Le fichier `./public/css/style.css` sera appelé via l'URL : `http://exemple.com/css/style.css`.

Utilisez le paramètre `:static_cache_control` pour ajouter l'information d'en-tête `Cache-Control` (voir plus loin).

Vues / Templates

Chaqie langage de template est disponible via sa propre méthode de rendu, lesquelles renvoient tout simplement une chaîne de caractères.

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

Ceci effectue le rendu de la vue `views/index.erb`.

Plutôt que d'utiliser le nom d'un template, vous pouvez directement passer le contenu du template :

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

Les méthodes de templates acceptent un second paramètre, un hash d'options :

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

Ceci effectuera le rendu de la vue `views/index.erb` en l'intégrant au *layout* `views/post.erb` (les vues Erb sont intégrées par défaut au *layout* `views/layout.erb` quand ce fichier existe).

Toute option que Sinatra ne comprend pas sera passée au moteur de rendu :

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

Vous pouvez également définir des options par langage de template de façon générale :

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

get '/' do
```

```
  haml :index
end
```

Les options passées à la méthode de rendu prennent le pas sur les options définies au moyen de `set`.

Options disponibles :

locals Liste de variables locales passées au document. Pratique pour les vues partielles. Exemple : `erb "<%= foo %>", :locals => {:foo => "bar"}`.

default_encoding Encodage de caractères à utiliser en cas d'incertitude. Par défaut, c'est `settings.default_encoding`.

views Dossier de vues dans lequel chercher les templates. Par défaut `settings.views`.

layout S'il faut ou non utiliser un +layout+ (+true+ or +false+). Indique le template à utiliser lorsque c'est un symbole. Exemple : `erb :index, :layout => !request.xhr?`.

content_type Content-Type que le template produit, dépend par défaut du langage de template.

scope Contexte sous lequel effectuer le rendu du template. Par défaut il s'agit de l'instance de l'application. Si vous changez cela, les variables d'instance et les méthodes utilitaires ne seront pas disponibles.

layout_engine Moteur de rendu à utiliser pour le +layout+. Utile pour les langages ne supportant pas les +layouts+. Il s'agit par défaut du moteur utilisé pour le rendu du template. Exemple : `set :rdoc, :layout_engine => :erb`

Les templates sont supposés se trouver directement dans le dossier `./views`. Pour utiliser un dossier de vues différent :

```
set :views, settings.root + '/templates'
```

Il est important de se souvenir que les templates sont toujours référencés sous forme de symboles, même lorsqu'ils sont dans un sous-répertoire (dans ce cas, utilisez `:'sous_repertoire/template'`). Il faut utiliser un symbole car les méthodes de rendu évaluent le contenu des chaînes de caractères au lieu de les considérer comme un chemin vers un fichier.

Langages de template disponibles

Certains langages ont plusieurs implémentations. Pour préciser l'implémentation à utiliser (et garantir l'aspect thread-safe), vous devez simplement l'avoir chargée au préalable :

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

Templates Haml

Dépendances	haml
Extensions de fichier	`.haml`
Exemple	`haml :index, :format => :html5`

Templates Erb

Dépendances	erubis ou erb (inclus avec Ruby)
Extensions de fichier	`.erb`, `.rhtml` ou `.erubis` (Erubis seulement)
Exemple	`erb :index`

Templates Builder

Dépendances	builder
Extensions de fichier	`.builder`
Exemple	`builder { |xml| xml.em "salut" }`

Ce moteur accepte également un bloc pour des templates en ligne (voir exemple).

Templates Nokogiri

Dépendances	nokogiri
Extensions de fichier	`.nokogiri`
Exemple	`nokogiri { |xml| xml.em "salut" }`

Ce moteur accepte également un bloc pour des templates en ligne (voir exemple).

Templates Sass

Dépendances	sass
Extensions de fichier	`.sass`
Exemple	`sass :stylesheet, :style => :expanded`

Templates SCSS

Dépendances	sass
Extensions de fichier	`.scss`
Exemple	`scss :stylesheet, :style => :expanded`

Templates Less

Dépendances	less
Extensions de fichier	`.less`
Exemple	`less :stylesheet`

Templates Liquid

Dépendances	liquid
Extensions de fichier	`.liquid`
Exemple	`liquid :index, :locals => { :key => 'value' }`

Comme vous ne pouvez appeler de méthodes Ruby (autres que `yield`) dans un template Liquid, vous aurez sûrement à lui passer des variables locales.

Templates Markdown

Dépendances	rdiscount, redcarpet, bluecloth, kramdown *ou* maruku
Extensions de fichier	`.markdown`, `.mkd` et `.md`
Exemple	`markdown :index, :layout_engine => :erb`

Il n'est pas possible d'appeler des méthodes depuis markdown, ni de lui passer des variables locales. Par conséquent, il sera souvent utilisé en combinaison avec un autre moteur de rendu :

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

Notez que vous pouvez également appeler la méthode `markdown` au sein d'autres templates :

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

Comme vous ne pouvez pas appeler de Ruby au sein de Markdown, vous ne pouvez pas utiliser de layouts écrits en Markdown. Toutefois, il est possible d'utiliser un moteur de rendu différent pour le template et pour le layout en utilisant l'option `:layout_engine`.

Templates Textile

Dépendances	RedCloth
Extensions de fichier	`.textile`
Exemple	`textile :index, :layout_engine => :erb`

Il n'est pas possible d'appeler des méthodes depuis textile, ni de lui passer des variables locales. Par

conséquent, il sera souvent utilisé en combinaison avec un autre moteur de rendu :

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

Notez que vous pouvez également appeler la méthode `textile` au sein d'autres templates :

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

Comme vous ne pouvez pas appeler de Ruby au sein de Textile, vous ne pouvez pas utiliser de layouts écrits en Textile. Toutefois, il est possible d'utiliser un moteur de rendu différent pour le template et pour le layout en utilisant l'option `:layout_engine`.

Templates RDoc

Dépendances	rdoc
Extensions de fichier	`.rdoc`
Exemple	`rdoc :README, :layout_engine => :erb`

Il n'est pas possible d'appeler des méthodes depuis rdoc, ni de lui passer des variables locales. Par conséquent, il sera souvent utilisé en combinaison avec un autre moteur de rendu :

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

Notez que vous pouvez également appeler la méthode `rdoc` au sein d'autres templates :

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

Comme vous ne pouvez pas appeler de Ruby au sein de RDoc, vous ne pouvez pas utiliser de layouts écrits en RDoc. Toutefois, il est possible d'utiliser un moteur de rendu différent pour le template et pour le layout en utilisant l'option `:layout_engine`.

Templates Radius

Dépendances	radius
Extensions de fichier	`.radius`
Exemple	`radius :index, :locals => { :key => 'value' }`

Comme vous ne pouvez pas appeler de méthodes Ruby depuis un template Radius, vous aurez sûrement à lui passer des variables locales.

Templates Markaby

Dépendances	markaby
Extensions de fichier	`.mab`
Exemple	`markaby { h1 "Bienvenue !" }`

Ce moteur accepte également un bloc pour des templates en ligne (voir exemple).

Templates RABL

Dépendances	rabl
Extensions de fichier	`.rabl`
Exemple	`rabl :index`

Templates Slim

Dépendances	slim
Extensions de fichier	`.slim`
Exemple	`slim :index`

Templates Creole

Dépendances	creole
Extensions de fichier	`.creole`
Exemple	`creole :wiki, :layout_engine => :erb`

Il n'est pas possible d'appeler des méthodes depuis markdown, ni de lui passer des variables locales. Par conséquent, il sera souvent utilisé en combinaison avec un autre moteur de rendu :

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

Notez que vous pouvez également appeler la méthode +markdown+ au sein d'autres templates :

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

Comme vous ne pouvez pas appeler de Ruby au sein de Markdown, vous ne pouvez pas utiliser de +layouts+ écrits en Markdown. Toutefois, il est possible d'utiliser un moteur de rendu différent pour le template et pour le +layout+ en utilisant l'option `:layout_engine`.

Templates CoffeeScript

Dépendances	coffee-script et un [moyen d'exécuter javascript]

	(https://github.com/sstephenson/execjs/blob/master/README.md#readme)
Extensions de fichier	`.coffee`
Exemple	`coffee :index`

Templates Yajl

Dépendances	yajl-ruby
Extensions de fichier	`.yajl`
Exemple	`yajl :index, :locals => { :key => 'qux' }, :callback => 'present', :variable => 'resource'`

Le source du template est évalué en tant que chaine Ruby, puis la variable json obtenue est convertie avec #to_json.

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

Les options `:callback` et `:variable` peuvent être utilisées pour décorer l'objet retourné.

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

Templates WLang

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

L'appel de code ruby au sein des templates n'est pas idiomatique en wlang. L'écriture de templates sans logique est encouragé, via le passage de variables locales. Il est néanmoins possible d'écrire un layout en wlang et d'y utiliser `yield`.

Templates embarqués

```
get '/' do
  haml '%div.title Bonjour le monde'
end
```

Générera le code du template spécifié dans la chaîne de caractères.

Accéder aux variables dans un Template

Un template est évalué dans le même contexte que l'endroit d'où il a été appelé (gestionnaire de route). Les variables d'instance déclarées dans le gestionnaire de route sont directement accessibles dans le template :

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

Alternativement, on peut passer un hash contenant des variables locales :

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

Ceci est généralement utilisé lorsque l'on veut utiliser un template comme partiel (depuis un autre template) et qu'il est donc nécessaire d'adapter les noms de variables.

Templates dans le fichier source

Des templates peuvent être définis dans le fichier source comme ceci :

```
require 'sinatra'

get '/' do
  haml :index
end

__END__

@@ layout
%html
  = yield

@@ index
%div.title Bonjour le monde !
```

NOTE : Les templates du fichier source qui contient `require 'sinatra'` sont automatiquement chargés. Si vous avez des templates dans d'autres fichiers source, il faut explicitement les déclarer avec `enable :inline_templates`.

Templates nommés

Les templates peuvent aussi être définis grâce à la méthode de haut niveau `template` :

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

template :index do
  '%div.title Bonjour le monde !'
end

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

Si un template nommé “layout” existe, il sera utilisé à chaque fois qu’un template sera affiché. Vous pouvez désactivez les layouts au cas par cas en passant `:layout => false` ou bien les désactiver par défaut au moyen de `set :haml, :layout => false` :

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

Associer des extensions de fichier

Pour associer une extension de fichier avec un moteur de rendu, utilisez `Tilt.register`. Par exemple, si vous désirez utiliser l’extension de fichier `tt` pour les templates Textile, vous pouvez faire comme suit :

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

Ajouter son propre moteur de rendu

En premier lieu, déclarez votre moteur de rendu avec Tilt, ensuite créez votre méthode de rendu :

```
Tilt.register :monmoteur, MonMerveilleurMoteurDeRendu

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

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

Utilisera `./views/index.monmoteur`. Voir le dépôt Github pour en savoir plus sur Tilt.

Filtres

Les filtres before sont exécutés avant chaque requête, dans le même contexte que les routes, et permettent de modifier la requête et sa réponse. Les variables d'instance déclarées dans les filtres sont accessibles au niveau des routes et des templates :

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

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

Les filtres after sont exécutés après chaque requête à l'intérieur du même contexte et permettent de modifier la requête et sa réponse. Les variables d'instance déclarées dans les filtres before ou les routes sont accessibles au niveau des filtres after :

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

Note : Le corps de la réponse n'est pas disponible au niveau du filtre after car il ne sera généré que plus tard (sauf dans le cas où vous utilisez la méthode +body+ au lieu de simplement renvoyer une chaine depuis vos routes).

Les filtres peuvent être associés à un masque, ce qui permet de limiter leur exécution aux cas où la requête correspond à ce masque :

```
before '/secret/*' do
  authentification!
end

after '/faire/:travail' do |travail|
  session[:dernier_travail] = travail
end
```

Tout comme les routes, les filtres acceptent également des conditions :

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

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

Helpers

Utilisez la méthode de haut niveau `helpers` pour définir des routines qui seront accessibles dans vos gestionnaires de route et dans vos templates :

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

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

Vous pouvez aussi définir les méthodes helper dans un module séparé :

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

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

helpers FooUtils, BarUtils
```

Cela a le même résultat que d'inclure les modules dans la classe de l'application.

Utiliser les sessions

Une session est utilisée pour conserver un état entre les requêtes. Une fois activées, vous avez un +hash+ de session par session utilisateur :

```
enable :sessions

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

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

Notez que `enable :sessions` enregistre en fait toutes les données dans un +cookie+. Ce n'est pas toujours ce que vous voulez (enregistrer beaucoup de données va augmenter le traffic par exemple). Vous pouvez utiliser n'importe quel +middleware+ Rack de session afin d'éviter cela. N'utiliser *pas* `enable :sessions` dans ce cas mais charger le +middleware+ de votre choix comme vous le feriez pour n'importe quel autre +middleware+ :

```
use Rack::Session::Pool, :expire_after => 2592000

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

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

Pour renforcer la sécurité, les données de session dans le cookie sont signées avec une clé secrète de session. Une clé secrète est générée pour vous au hasard par Sinatra. Toutefois, comme cette clé change à chaque démarrage de votre application, vous pouvez définir cette clé vous-même afin que toutes les instances de votre application la partage :

```
set :session_secret, 'super secret'
```

Si vous souhaitez avoir plus de contrôle, vous pouvez également enregistrer un +hash+ avec des options lors de la configuration de `sessions` :

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

Halt

Pour arrêter immédiatement la requête dans un filtre ou un gestionnaire de route :

```
halt
```

Vous pouvez aussi passer le code retour ...

```
halt 410
```

Ou le texte ...

```
halt 'Ceci est le texte'
```

Ou les deux ...

```
halt 401, 'Partez !'
```

Ainsi que les entêtes ...

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

Bien sûr il est possible de combiner un template avec `halt` :

```
halt erb(:erreur)
```

Passer

Une route peut passer le relais aux autres routes qui correspondent également avec `pass` :

```
get '/devine/:qui' do
  pass unless params[:qui] == 'Frank'
  "Tu m'as eu !"
end

get '/devine/*' do
  'Manqué !'
end
```

On sort donc immédiatement de ce gestionnaire et on continue à chercher, dans les masques suivants, le prochain qui correspond à la requête. Si aucun des masques suivants ne correspond, un code 404 est retourné.

Déclencher une autre route

Parfois, +pass+ n'est pas ce que vous recherchez, au lieu de cela vous souhaitez obtenir le résultat d'une autre route. Pour cela, utilisez simplement call :

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

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

Notez que dans l'exemple ci-dessus, vous faciliterez les tests et améliorerez la performance en déplaçant simplement `"bar"` dans un helper utilisé à la fois par `/foo` et `/bar`.

Si vous souhaitez que la requête soit envoyée à la même instance de l'application plutôt qu'à une copie, utilisez `call!` au lieu de `call`.

Lisez la spécification Rack si vous souhaitez en savoir plus sur `call`.

Définir le corps, le code retour et les entêtes

Il est possible et recommandé de définir le code retour et le corps de la réponse au moyen de la valeur de retour d'un bloc définissant une route. Quoiqu'il en soit, dans certains cas vous pourriez

avoir besoin de définir le coprs de la réponse à un moment arbitraire de l'exécution. Vous pouvez le faire au moyen de la méthode +body+. Si vous faites ainsi, vous pouvez alors utiliser cette même méthode pour accéder au corps de la réponse :

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

after do
  puts body
end
```

Il est également possible de passer un bloc à `body`, qui sera exécuté par le gestionnaire Rack (ceci peut être utilisé pour implémenter un streaming, voir "Valeurs de retour").

Pareillement au corps de la réponse, vous pouvez également définir le code retour et les entêtes :

```
get '/foo' do
  status 418
  headers \
    "Allow"   => "BREW, POST, GET, PROPFIND, WHEN",
    "Refresh" => "Refresh: 20; http://www.ietf.org/rfc/rfc2324.txt"
  body "Je suis une théière !"
end
```

Comme `body` `headers` et `status` peuvent être utilisés sans arguments pour accéder à leurs valeurs.

Faire du streaming

Il y a des cas où vous voulez commencer à renvoyer des données pendant que vous êtes en train de générer le reste de la réponse. Dans les cas les plus extrêmes, vous souhaitez continuer à envoyer des données tant que le client n'abandonne pas la connection. Vous pouvez alors utiliser le helper `stream` pour éviter de créer votre propre système :

```
get '/' do
  stream do |out|
    out << "Ça va être hallu- \n"
    sleep 0.5
    out << " (attends la suite) \n"
    sleep 1
    out << "- cinant ! \n"
  end
end
```

Cela permet d'implémenter des API de streaming ou de Server Sent Events et peut servir de base pour des WebSockets. Vous pouvez aussi l'employer pour augmenter le débit quand une partie du contenu provient d'une resource lente.

Le fonctionnement du streaming, notamment le nombre de requêtes simultanées, dépend énormément du serveur web utilisé. Certains ne prennent pas du tout en charge le streaming

(WEBRick par exemple). Lorsque le serveur ne gère pas le streaming, la partie body de la réponse sera envoyée au client en une seule fois, après que l'exécution du bloc passé au helper +stream+ sera terminée. Le streaming ne fonctionne pas du tout avec Shotgun.

En utilisant le helper +stream+ avec le paramètre +keep_open+, il n'appelera pas la méthode +close+ du flux, vous laissant la possibilité de le fermer à tout moment au cours de l'exécution. Ceci ne fonctionne qu'avec les serveurs evented (ie non threadés) tels que Thin et Rainbows. Les autres serveurs fermeront malgré tout le flux :

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

get '/' do
  # conserve le flux ouvert
  stream(:keep_open) { |out| connections << out }
end

post '/' do
  # écrit dans tous les flux ouverts
  connections.each { |out| out << params[:message] << "\n" }
  "message sent"
end
```

Journalisation (Logging)

Dans le contexte de la requête, la méthode utilitaire +logger+ expose une instance de +logger+ :

```
get '/' do
  logger.info "chargement des données"
  # ...
end
```

Ce logger va automatiquement prendre en compte les paramètres de configuration pour la journalisation de votre gestionnaire Rack. Si la journalisation est désactivée, cette méthode renverra un objet factice et vous n'avez pas à vous en inquiéter dans vos routes en le filtrant.

Notez que la journalisation est seulement activée par défaut pour `Sinatra::Application`, donc si vous héritez de `>Sinatra::Base`, vous aurez à l'activer vous-même :

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

Si vous souhaitez utiliser votre propre logger, vous devez définir le paramètre `logging` à `nil` pour être certain qu'aucun middleware de logging ne sera installé (notez toutefois que +logger+ renverra alors +nil+). Dans ce cas, Sinatra utilisera ce qui sera présent dans `env['rack.logger']`.

Types Mime

Quand vous utilisez `send_file` ou des fichiers statiques, vous pouvez rencontrer des types mime que Sinatra ne connaît pas. Utilisez `mime_type` pour les déclarer par extension de fichier :

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

Vous pouvez également les utiliser avec la méthode `content_type` :

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

Former des URLs

Pour former des URLs, vous devriez utiliser la méthode +url+, par exemple en Haml :

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

Cela prend en compte les proxy inverse et les routeurs Rack, s'ils existent.

Cette méthode est également disponible sous l'alias +to+ (voir ci-dessous pour un exemple).

Redirection du navigateur

Vous pouvez déclencher une redirection du navigateur avec la méthode `redirect` :

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

Tout paramètre additionnel est géré comme des arguments pour la méthode `halt` :

```
redirect to('/bar'), 303
redirect 'http://google.com', 'mauvais endroit mon pote'
```

Vous pouvez aussi rediriger vers la page dont l'utilisateur venait au moyen de `redirect back` :

```
get '/foo' do
  "<a href='/bar'>faire quelque choses</a>"
end
```

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

Pour passer des arguments à une redirection, ajoutez-les soit à la requête :

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

Ou bien utilisez une session :

```
enable :sessions

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

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

Contrôle du cache

Définir correctement vos entêtes à la base pour un bon cache HTTP.

Vous pouvez facilement définir l'entête Cache-Control de la manière suivante :

```
get '/' do
  cache_control :public
  "met le en cache !"
end
```

Conseil de pro : définir le cache dans un filtre +before+ :

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

Si vous utilisez la méthode +expires+ pour définir l'entête correspondant, `Cache-Control` sera alors défini automatiquement :

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

Pour utiliser correctement les caches, vous devriez utiliser +etag+ ou +last_modified+. Il est recommandé d'utiliser ces méthodes *avant* de faire d'importantes modifications, car elles vont immédiatement déclencher la réponse si le client a déjà la version courante dans son cache :

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

Il est également possible d'utiliser un weak ETag :

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

Ces méthodes ne sont pas chargées de mettre des données en cache, mais elles fournissent les informations nécessaires pour votre cache. Si vous êtes à la recherche de solutions rapides pour un reverse-proxy de cache, essayez rack-cache :

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

use Rack::Cache

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

Utilisez le paramètre `:static_cache_control` pour ajouter l'information d'en-tête `Cache-Control` (voir plus loin).

D'après la RFC 2616, votre application devrait se comporter différement lorsque l'en-tête If-Match ou If-None-Match est défini à `*` en tenant compte du fait que la resource demandée existe déjà ou pas. Sinatra considère que les requêtes portant sur des resources sûres (tel que get) ou idempotentes (tel que put) existent déjà et pour les autres resources (par exemple dans le cas de requêtes post) qu'il s'agit de nouvelles resources. Vous pouvez modifier ce comportement en passant une option `:new_resource` :

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

Si vous souhaitez utilisez un ETag faible, utilisez l'option `:kind` :

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

Envoyer des fichiers

Pour envoyer des fichiers, vous pouvez utiliser la méthode `send_file` :

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

Quelques options sont également acceptées :

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

Les options sont :

filename
: le nom du fichier dans la réponse, par défaut le nom du fichier envoyé.

last_modified
: valeur pour l'entête Last-Modified, par défaut la date de modification du fichier

type
: type de contenu à utiliser, deviné à partir de l'extension de fichier si absent

disposition
: utilisé pour Content-Disposition, les valuers possibles étant : `nil` (par défaut), `:attachment` et `:inline`

length
: entête Content-Length, par défaut la taille du fichier

status
: code état à renvoyer. Utile quand un fichier statique sert de page d'erreur.

Si le gestionnaire Rack le supporte, d'autres moyens que le +streaming+ via le processus Ruby seront utilisés. Si vous utilisez cette méthode, Sinatra gérera automatiquement les requêtes de type +range+.

Accéder à l'objet requête

L'objet correspondant à la requête envoyée peut être récupéré dans le contexte de la requête (filtres, routes, gestionnaires d'erreur) au moyen de la méthode +request+ :

```
# application tournant à l'adresse http://exemple.com/exemple
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                # corps de la requête envoyée par le client
                              # (voir ci-dessous)
  request.scheme              # "http"
  request.script_name         # "/exemple"
  request.path_info           # "/foo"
  request.port                # 80
  request.request_method      # "GET"
  request.query_string        # ""
```

```
  request.content_length      # taille de request.body
  request.media_type          # type de média pour request.body
  request.host                # "exemple.com"
  request.get?                # true (méthodes similaires pour les autres
                              # verbes HTTP)
  request.form_data?          # false
  request["UN_ENTETE"]        # valeur de l'entête UN_ENTETE
  request.referer             # référant du client ou '/'
  request.user_agent          # user agent (utilisé par la condition :agent)
  request.cookies             # tableau contenant les cookies du navigateur
  request.xhr?                # requête AJAX ?
  request.url                 # "http://exemple.com/exemple/foo"
  request.path                # "/exemple/foo"
  request.ip                  # adresse IP du client
  request.secure?             # false
  request.forwarded?          # vrai (si on est derrière un proxy inverse)
  request.env                 # tableau brut de l'environnement fourni par
                              # Rack
end
```

Certaines options, telles que `script_name` ou `path_info` peuvent également être modifiées :

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

get "/" do
  "toutes les requêtes arrivent ici"
end
```

`request.body` est un objet IO ou StringIO :

```
post "/api" do
  request.body.rewind  # au cas où il a déjà été lu
  donnees = JSON.parse request.body.read
  "Bonjour #{donnees['nom']} !"
end
```

Fichiers joints

Vous pouvez utiliser la méthode +attachment+ pour indiquer au navigateur que la réponse devrait être stockée sur le disque plutôt qu'affichée :

```
get '/' do
  attachment
  "enregistre-le !"
end
```

Vous pouvez également lui passer un nom de fichier :

```
get '/' do
  attachment "info.txt"
  "enregistre-le !"
end
```

Gérer Date et Time

Sinatra fourni un helper +time_for+ pour convertir une valeur donnée en objet `Time`. Il peut aussi faire la conversion à partir d'objets +DateTime+, `Date` ou de classes similaires :

```
get '/' do
  pass if Time.now > time_for('Dec 23, 2012')
  "encore temps"
end
```

Cette méthode est utilisée en interne par +expires+, +last_modified+ et consorts. Par conséquent, vous pouvez très facilement étendre le fonctionnement de ces méthodes en surchargeant le helper +time_for+ dans votre 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
  "salut"
end
```

Chercher les fichiers de templates

La méthode `find_template` est utilisée pour trouver les fichiers de templates à générer :

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

Ce n'est pas très utilise. En revanche, il est utile de pouvoir surcharger cette méthode afin de définir son propre mécanisme de recherche. Par exemple, vous pouvez utiliser plus d'un répertoire de vues :

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

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

```
  end
end
```

Un autre exemple est d'utiliser des répertoires différents pour des moteurs de rendu différents :

```
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
```

Vous pouvez également écrire cela dans une extension et la partager avec d'autres !

Notez que `find_template` ne vérifie pas que le fichier existe mais va plutôt exécuter le bloc pour tous les chemins possibles. Cela n'induit pas un problème de performance dans le sens où `render` va utiliser +break+ dès qu'un fichier est trouvé. De plus, l'emplacement des templates (et leur contenu) est mis en cache si vous n'êtes pas en mode développement. Vous devriez garder cela en tête si vous écrivez une méthode vraiment dingue.

Configuration

Lancé une seule fois au démarrage de tous les environnements :

```
configure do
  # définir un paramètre
  set :option, 'value'

  # définir plusieurs paramètre
  set :a => 1, :b => 2

  # identique à "set :option, true"
  enable :option

  # identique à "set :option, false""
  disable :option

  # vous pouvez également avoir des paramètres dynamiques avec des blocs
  set(:css_dir) { File.join(views, 'css') }
end
```

Lancé si l'environnement (variable d'environnement RACK_ENV) est défini comme `:production` :

configure :production do ... end

Lancé si l'environnement est `:production` ou `:test` :

configure :production, :test do ... end

Vous pouvez accéder à ces paramètres via `settings` :

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

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

Se protéger des attaques

Sinatra utilise Rack::Protection pour protéger votre application contre les principales attaques opportunistes. Vous pouvez très simplement désactiver cette fonctionnalité (ce qui exposera votre application à beaucoup de vulnerabilités courantes) :

```
disable :protection
```

Pour désactiver seulement un type de protection, vous pouvez définir `protection` avec un hash d'options :

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

Vous pouvez également lui passer un tableau pour désactiver plusieurs types de protection :

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

Paramètres disponibles

absolute_redirects
: Si désactivé, Sinatra permettra les redirections relatives. Toutefois, Sinatra ne sera plus conforme à la RFC 2616 (HTTP 1.1), qui n'autorise que les redirections absolues. Activez si votre application tourne derrière un proxy inverse qui n'a pas été correctement configuré. Notez que la méthode `url` continuera de produire des URLs absolues, sauf si vous lui passez `false` comme second argument.

  Désactivé par défaut.

add_charsets
: types mime pour lesquels la méthode `content_type` va automatiquement ajouter l'information du `charset`.

Vous devriez lui ajouter des valeurs plutôt que de l'écraser :

```
settings.add_charsets >> "application/foobar"
```

app_file

chemin pour le fichier de l'application principale, utilisé pour détecter la racine du projet, les dossiers public et vues, et les templates en ligne.

bind
adresse IP sur laquelle se brancher (par défaut : 0.0.0.0). Utiliser seulement pour le serveur intégré.
default_encoding
encodage à utiliser si inconnu (par défaut `"utf-8"`)
dump_errors
afficher les erreurs dans le `log`.
environment
environnement courant, par défaut `ENV['RACK_ENV']`, ou `"development"` si absent.
logging
utiliser le `logger`.
lock

Place un `lock` autour de chaque requête, n'exécutant donc qu'une seule requête par processus Ruby.

Activé si votre application n'est pas `thread-safe`. Désactivé par défaut.

method_override
utilise la magie de `_method` afin de permettre des formulaires put/delete dans des navigateurs qui ne le permettent pas.
port
port à écouter. Utiliser seulement pour le serveur intégré.
prefixed_redirects
si oui ou non `request.script_name` doit être inséré dans les redirections si un chemin non absolu est utilisé. Ainsi, `redirect '/foo'` se comportera comme `redirect to('/foo')`. Désactivé par défaut.
protection
défini s'il faut activer ou non la protection contre les attaques web. Voir la section protection précédente.
public_dir
alias pour `public_folder`. Voir ci-dessous.
public_folder
chemin pour le dossier à partir duquel les fichiers publics sont servis. Utilisé seulement si les fichiers statiques doivent être servis (voir le paramètre `static`). Si non défini, il découle du paramètre `app_file`.
reload_templates
si oui ou non les templates doivent être rechargés entre les requêtes. Activé en mode développement.
root

chemin pour le dossier racine du projet. Si non défini, il découle du paramètre `app_file`.

raise_errors
: soulever les erreurs (ce qui arrêtera l'application). Désactivé par défaut sauf lorsque `environment` est défini à `"test"`.

run
: si activé, Sinatra s'occupera de démarrer le serveur, ne pas activer si vous utiliser rackup ou autres.

running
: est-ce que le serveur intégré est en marche ? ne changez pas ce paramètre !

server
: serveur ou liste de serveurs à utiliser pour le serveur intégré. Par défaut ['thin', 'mongrel', 'webrick'], l'ordre indiquant la priorité.

sessions
: active le support des sessions basées sur les cookies, en utilisant `Rack::Session::Cookie`. Reportez-vous à la section 'Utiliser les sessions' pour plus d'informations.

show_exceptions
: affiche la trace de l'erreur dans le navigateur lorsqu'une exception se produit. Désactivé par défaut sauf lorsque `environment` est défini à `"development"`.

static
: Si oui ou non Sinatra doit s'occuper de servir les fichiers statiques. Désactivez si vous utilisez un serveur capable de le gérer lui même. Le désactiver augmentera la performance. Activé par défaut pour le style classique, désactivé pour le style modulaire.

static_cache_control
: A définir quand Sinatra rend des fichiers statiques pour ajouter les en-têtes `Cache-Control`. Utilise le helper `cache_control`. Désactivé par défaut. Utiliser un array explicite pour définir des plusieurs valeurs : `set :static_cache_control, [:public, :max_age => 300]`

threaded
: à définir à `true` pour indiquer à Thin d'utiliser `EventMachine.defer` pour traiter la requête.

views
: chemin pour le dossier des vues. Si non défini, il découle du paramètre `app_file`.

Environements

Il existe trois environnements prédéfinis : `"development"`, `"production"` et `"test"`. Les environements peuvent être sélectionné via la variable d'environnement +RACK_ENV+. Sa valeur par défaut est `"development"`. Dans ce mode, tous les templates sont rechargés à chaque requête. Des handlers spécifiques pour `not_found` et `error` sont installés pour vous permettre d'avoir une pile de trace dans votre navigateur. En mode `"production"` et `"test"` les templates sont mis en cache par défaut.

Pour exécuter votre application dans un environnement différent, utilisez l'option `-e` de Ruby :

```
$ ruby mon_application.rb -e [ENVIRONMENT]
```

Vous pouvez utiliser une des méthodes +development?+, +test?+ et +production?+ pour déterminer quel est l'environnement en cours.

Gérer les erreurs

Les gestionnaires d'erreur s'exécutent dans le même contexte que les routes ou les filtres, ce qui veut dire que vous avez accès (entre autres) aux bons vieux `haml`, `erb`, `halt`, etc.

NotFound

Quand une exception `Sinatra::NotFound` est soulevée, ou que le code retour est 404, le gestionnaire `not_found` est invoqué :

```
not_found do
  'Pas moyen de trouver ce que vous cherchez'
end
```

Error

Le gestionnaire +error+ est invoqué à chaque fois qu'une exception est soulevée dans une route ou un filtre. L'objet exception est accessible via la variable Rack `sinatra.error` :

```
error do
  'Désolé mais une méchante erreur est survenue - ' + env['sinatra.error'].name
end
```

Erreur sur mesure :

```
error MonErreurSurMesure do
  'Donc il est arrivé ceci ...' + env['sinatra.error'].message
end
```

Donc si ceci arrive :

```
get '/' do
  raise MonErreurSurMesure, 'quelque chose de mal'
end
```

Vous obtenez ça :

Donc il est arrivé ceci... quelque chose de mal

Alternativement, vous pouvez avoir un gestionnaire d'erreur associé à un code particulier :

```
error 403 do
  'Accès interdit'
end
```

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

Ou un intervalle :

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

Sinatra installe pour vous quelques gestionnaires `not_found` et `error` génériques lorsque vous êtes en environnement `development`.

Les Middlewares Rack

Sinatra tourne avec Rack, une interface standard et minimale pour les web frameworks Ruby. Un des points forts de Rack est le support de ce que l'on appelle des "middlewares" – composant qui vient se situer entre le serveur et votre application, et dont le but est de visualiser/manipuler la requête/réponse HTTP, et d'offrir diverses fonctionnalités classiques.

Sinatra permet de construire facilement des middlewares Rack via la méthode de haut niveau +use+ :

```
require 'sinatra'
require 'mon_middleware_perso'

use Rack::Lint
use MonMiddlewarePerso

get '/bonjour' do
  'Bonjour le monde'
end
```

La sémantique de +use+ est identique à celle définie dans le DSL de Rack::Builder (le plus souvent utilisé dans un fichier rackup). Par exemple, la méthode +use+ accepte divers arguments ainsi que des blocs :

```
use Rack::Auth::Basic do |login, password|
  login == 'admin' && password == 'secret'
end
```

Rack est distribué avec une bonne variété de middlewares standards pour les logs, débuguer, faire du routage URL, de l'authentification, gérer des sessions. Sinatra utilise beaucoup de ces composants automatiquement via la configuration, donc pour ceux-ci vous n'aurez pas à utiliser la méthode `use`.

Tester

Les tests pour Sinatra peuvent être écrit avec n'importe quelle bibliothèque basée sur Rack. Rack::Test est recommandé :

```
require 'mon_application_sinatra'
require 'test/unit'
require 'rack/test'

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

  def app
    Sinatra::Application
  end

  def test_ma_racine
    get '/'
    assert_equal 'Bonjour le monde !', last_response.body
  end

  def test_avec_des_parametres
    get '/rencontrer', :name => 'Frank'
    assert_equal 'Salut Frank !', last_response.body
  end

  def test_avec_rack_env
    get '/', {}, 'HTTP_USER_AGENT' => 'Songbird'
    assert_equal "Vous utilisez Songbird !", last_response.body
  end
end
```

Sinatra::Base - Les Middlewares, Bibliothèques, et Applications Modulaires

Définir votre application au niveau supérieur fonctionne bien dans le cas des micro-applications mais présente pas mal d'inconvénients pour créer des composants réutilisables sous forme de middlewares Rack, de Rails metal, de simples librairies avec un composant serveur ou même d'extensions Sinatra. Le niveau supérieur suppose une configuration dans le style des micro-applications (une application d'un seul fichier, des répertoires `./public` et `./views`, des logs, une page d'erreur, etc...). C'est là que `Sinatra::Base` prend tout son intérêt :

```
require 'sinatra/base'

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

  get '/' do
    'Bonjour le monde !'
  end
end
```

Les méthodes de la classe `Sinatra::Base` sont parfaitement identiques à celles disponibles via le DSL de haut niveau. Il suffit de deux modifications pour transformer la plupart des applications de haut niveau en un composant `Sinatra::Base` :

- Votre fichier doit charger `sinatra/base` au lieu de `sinatra`, sinon toutes les méthodes du DSL Sinatra seront importées dans l'espace de nom principal.
- Les gestionnaires de routes, la gestion d'erreur, les filtres et les options doivent être placés dans une classe héritant de `Sinatra::Base`.

`Sinatra::Base` est une page blanche. La plupart des options sont désactivées par défaut, y compris le serveur intégré. Reportez-vous à Options et Configuration pour plus d'informations sur les options et leur fonctionnement.

Style modulaire vs. style classique

Contrairement aux idées reçues, il n'y a rien de mal à utiliser le style classique. Si c'est ce qui convient pour votre application, vous n'avez pas aucune raison de passer à une application modulaire.

Le principal inconvénient du style classique sur le style modulaire est que vous ne pouvez avoir qu'une application Ruby par processus Ruby. Si vous pensez en utiliser plus, passez au style modulaire. Et rien ne vous empêche de mixer style classique et style modulaire.

Si vous passez d'un style à l'autre, souvenez-vous des quelques différences mineures en ce qui concerne les paramètres par défaut :

Paramètre Classique Modulaire

app_file fichier chargeant sinatra fichier héritant de Sinatra::Base run $0 == app_file false logging true false method_override true false inline_templates true false static true false

Servir une application modulaire

Il y a deux façons de faire pour démarrer une application modulaire, démarrez avec `run!` :

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

class MyApp < Sinatra::Base
  # ... code de l'application ici ...

  # démarre le serveur si ce fichier est directement exécuté
  run! if app_file == $0
end
```

Démarrez ensuite avec :

```
$ ruby my_app.rb
```

Ou alors avec un fichier `config.ru`, qui permet d'utiliser n'importe quel gestionnaire Rack :

```
# config.ru
require './my_app'
run MyApp
```

Exécutez :

```
$ rackup -p 4567
```

Utiliser une application de style classique avec un fichier config.ru

Ecrivez votre application :

```
# app.rb
require 'sinatra'

get '/' do
  'Bonjour le monde !'
end
```

Et un fichier `config.ru` correspondant :

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

Quand utiliser un fichier config.ru ?

Quelques cas où vous devriez utiliser un fichier `config.ru` :

- Vous souhaitez déployer avec un autre gestionnaire Rack (Passenger, Unicorn, Heroku, ...).
- Vous souhaitez utiliser plus d'une sous-classe de `Sinatra::Base`.
- Vous voulez utiliser Sinatra comme un middleware, non en tant que endpoint.

Il n'est pas nécessaire de passer par un fichier `config.ru` pour la seule raison que vous êtes passé au style modulaire, et vous n'avez pas besoin de passer au style modulaire pour utiliser un fichier `config.ru`.

Utiliser Sinatra comme Middleware

Non seulement Sinatra peut utiliser d'autres middlewares Rack, il peut également être à son tour utilisé au-dessus de n'importe quel endpoint Rack en tant que middleware. Ce endpoint peut très bien être une autre application Sinatra, ou n'importe quelle application basée sur Rack (Rails/Ramaze/Camping/...) :

```
require 'sinatra/base'

class EcranDeConnexion < Sinatra::Base
  enable :sessions

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

  post('/connexion') do
    if params[:nom] = 'admin' && params[:motdepasse] = 'admin'
      session['nom_utilisateur'] = params[:nom]
    else
      redirect '/connexion'
    end
  end
end

class MonApp < Sinatra::Base
  # le middleware sera appelé avant les filtres
  use EcranDeConnexion

  before do
    unless session['nom_utilisateur']
      halt "Accès refusé, merci de vous <a href='/connexion'>connecter</a>."
    end
  end

  get('/') { "Bonjour #{session['nom_utilisateur']}." }
end
```

Création dynamique d'applications

Il se peut que vous ayez besoin de créer une nouvelle application à l'exécution sans avoir à les assigner à une constante, vous pouvez le faire grâce à `Sinatra.new` :

```
require 'sinatra/base'
mon_app = Sinatra.new { get('/') { "salut" } }
mon_app.run!
```

L'application dont elle hérite peut être passé en argument optionnel :

```
# config.ru
require 'sinatra/base'

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

    run!
```

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

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

C'est notamment utile pour tester des extensions à Sinatra ou bien pour utiliser Sinatra dans votre propre bibliothèque.

Cela permet également d'utiliser très facilement Sinatra comme middleware :

```
require 'sinatra/base'

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

run RailsProject::Application
```

Contextes et Binding

Le contexte dans lequel vous êtes détermine les méthodes et variables disponibles.

Contexte de l'application/classe

Toute application Sinatra correspond à une sous-classe de `Sinatra::Base`. Si vous utilisez le DSL haut niveau (`require 'sinatra'`), alors cette classe est `Sinatra::Application`, sinon il s'agit de la sous-classe que vous avez définie. Dans le contexte de la classe, vous avez accès aux méthodes telles que `get` ou `before`, mais vous n'avez pas accès aux objets +request+ ou +session+ car c'est la même classe d'application qui traitera toutes les requêtes.

Les options définies au moyen de +set+ deviennent des méthodes de classe :

```
class MonApp < Sinatra::Base
  # Eh, je suis dans le contexte de l'application !
  set :foo, 42
  foo # => 42

  get '/foo' do
    # Eh, je ne suis plus dans le contexte de l'application !
  end
end
```

Vous avez le binding du contexte de l'application dans :

- Le corps de la classe d'application

- Les méthodes définies par les extensions
- Le bloc passé à `helpers`
- Les procs/blocs utilisés comme argument pour `set`
- Le bloc passé à `Sinatra.new`

Vous pouvez atteindre ce contexte (donc la classe) de la façon suivante :

- Via l'objet passé dans les blocs `configure` (`configure { |c| ... }`)
- En utilisant `settings` dans le contexte de la requête

Contexte de la requête/instance

Pour tout traitement d'une requête, une nouvelle instance de votre classe d'application est créée et tous vos gestionnaires sont exécutés dans ce contexte. Dans ce dernier, vous pouvez accéder aux objets `request` et `session` et faire appel aux fonctions de rendu telles que `erb` ou `haml`. Vous pouvez accéder au contexte de l'application depuis le contexte de la requête au moyen de `settings` :

```
class MonApp < Sinatra::Base
  # Eh, je suis dans le contexte de l'application !
  get '/ajouter_route/:nom' do
    # Contexte de la requête pour '/ajouter_route/:nom'
    @value = 42

    settings.get("/#{params[:nom]}") do
      # Contexte de la requête pour "/#{params[:nom]}"
      @value # => nil (on est pas au sein de la même requête)
    end

    "Route ajoutée !"
  end
end
```

Vous avez le binding du contexte de la requête dans :

- les blocs get/head/post/put/delete/options
- les filtres before/after
- les méthodes utilitaires (définies au moyen de `helpers`)
- les vues/templates

Le contexte de délégation

Le contexte de délégation se contente de transmettre les appels de méthodes au contexte de classe. Toutefois, il ne se comporte pas à 100% comme le contexte de classe car vous n'avez pas le binding de la classe : seules les méthodes spécifiquement déclarées pour délégation sont disponibles et il n'est pas possible de partager des variables/états avec le contexte de classe (comprenez : `self` n'est pas le même). Vous pouvez ajouter des délégation de méthodes en appelant `Sinatra::Delegator.delegate :method_name`.

Vous avez le binding du contexte de délégation dans :

- Le binding de haut niveau, si vous avez utilisé `require "sinatra"`
- Un objet qui inclut le module `Sinatra::Delegator`

Jetez un oeil pour vous faire une idée : voici le mixin Sinatra::Delegator qui étend l'objet principal.

Ligne de commande

Les applications Sinatra peuvent être lancées directement :

```
$ ruby mon_application.rb [-h] [-x] [-e ENVIRONNEMENT] [-p PORT] [-o HOTE] [-s SERVEUR]
```

Les options sont :

```
-h # aide
-p # déclare le port (4567 par défaut)
-o # déclare l'hôte (0.0.0.0 par défaut)
-e # déclare l'environnement (+development+ par défaut)
-s # déclare le serveur/gestionnaire à utiliser (thin par défaut)
-x # active le mutex lock (off par défaut)
```

Configuration nécessaire

Les versions suivantes de Ruby sont officiellement supportées :

Ruby 1.8.7
: 1.8.7 est complètement supporté, toutefois si rien ne vous en empêche, nous vous recommandons de passer à 1.9.2 ou bien de passer à JRuby ou Rubinius. Le support de Ruby 1.8.7 ne sera pas supprimé avant la sortie de Sinatra 2.0 et de Ruby 2.0, à moins qu'un improbable Ruby 1.8.8 apparaisse. Et même dans ce cas, nous pourrions continuer à le supporter. **Ruby 1.8.6 n'est plus supporté**. Si vous souhaitez utiliser la version 1.8.6, vous devez revenir à Sinatra 1.2 qui continuera à recevoir des corrections de bugs tant que Sinatra 1.4.0 ne sera pas livré.

Ruby 1.9.2
: 1.9.2 est totalement supporté et recommandé. N'utilisez pas 1.9.2p0 car il provoque des erreurs de segmentation à l'exécution de Sinatra. Son support continuera au minimum jusqu'à la sortie de Ruby 1.9.4/2.0 et le support de la dernière version 1.9 se poursuivra aussi longtemps que la core team de Ruby la supportera.

Ruby 1.9.3
: 1.9.3 est totalement supporté et recommandé. Nous vous rappelons que passer à 1.9.3 depuis une version précédente annulera toutes les sessions.

Rubinius
: Rubinius est officiellement supporté (Rubinius <= 1.2.4), tout fonctionne, y compris tous les langages de template. La version 2.0 à venir est également supportée.

JRuby
: JRuby est officiellement supporté (JRuby <= 1.6.5). Aucune anomalie avec des bibliothèques de templates tierces ne sont connues. Toutefois, si vous choisissez JRuby, alors tournez vous vers des gestionnaires Rack JRuby car le serveur Thin n'est pas complètement supporté par JRuby. Le support des extensions C dans JRuby est encore expérimental, ce qui n'affecte que RDiscount, Redcarpet and RedCloth pour l'instant.

Nous gardons également un oeil sur les versions Ruby à venir.

Les implémentations Ruby suivantes ne sont pas officiellement supportées mais sont malgré tout connues pour permettre de faire fonctionner Sinatra :

- Versions plus anciennes de JRuby et Rubinius
- Ruby Enterprise Edition
- MacRuby, Maglev, IronRuby
- Ruby 1.9.0 et 1.9.1 (mais nous déconseillons leur utilisation)

Le fait de ne pas être officiellement supporté signifie que si quelque chose ne fonctionne pas uniquement sur cette plateforme alors c'est un problème de la plateforme et pas un bug de Sinatra.

Nous lançons également notre intégration continue (CI) avec ruby-head (la future 2.0.0) et la branche 1.9.4, mais étant donné les évolutions continuelles, nous ne pouvont rien garantir, si ce n'est que les versions 1.9.4p0 et 2.0.0p0 seront supportées.

Sinatra devrait fonctionner sur n'importe quel système d'exploitation supportant l'implémentation Ruby choisie.

Il n'est pas possible d'utiliser Sinatra sur Cardinal, SmallRuby, Blueuby ou toute version de Ruby antérieure à 1.8.7 à l'heure actuelle.

Essuyer les plâtres

Si vous voulez utiliser la toute dernière version de Sinatra, n'ayez pas peur de faire tourner votre application sur la branche master, cela devrait être stable.

Nous publions également une gem de +prerelease+ de temps en temps que vous pouvez installer comme suit :

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

afin d'avoir les toutes dernières fonctionnalités.

Avec Bundler

Si vous voulez faire tourner votre application avec le tout dernier Sinatra, Bundler est recommandé.

Tout d'abord, installer bundler si vous ne l'avez pas :

```
$ gem install bundler
```

Ensuite, dans le dossier de votre projet, créez un fichier +Gemfile+ :

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

# autres dépendances
gem 'haml'                    # par exemple, si vous utilisez haml
gem 'activerecord', '~> 3.0'  # peut-être que vous avez également besoin
                              # de ActiveRecord 3.x
```

Notez que vous aurez à lister toutes les dépendances de votre application dans ce fichier. Les dépendances directes de Sinatra (Rack et Tilt) seront automatiquement téléchargées et ajoutées par Bundler.

Maintenant, vous pouvez faire tourner votre application de la façon suivante :

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

Faites le vous-même

Créez un clone local et démarrez votre application avec le dossier `sinatra/lib` dans le `$LOAD_PATH` :

```
$ cd myapp
$ git clone git://github.com/sinatra/sinatra.git
$ ruby -Isinatra/lib myapp.rb

A l'avenir, pour mettre à jour le code source de Sinatra :

~~~~bash
$ cd myapp/sinatra
$ git pull
```

Installez globalement

Vous pouvez construire la gem vous-même :

```
$ git clone git://github.com/sinatra/sinatra.git
$ cd sinatra
$ rake sinatra.gemspec
$ rake install
```

Si vous installez les gems en tant que +root+, la dernière étape sera :

```
$ sudo rake install
```

Versions

Sinatra se conforme aux (versions sémantiques)[http://semver.org/], aussi bien SemVer que SemVerTag.

Mais encore

- Site internet – Plus de documentation, de news, et des liens vers d'autres ressources.
- Contribuer – Vous avez trouvé un bug ? Besoin d'aide ? Vous avez un patch ?
- Suivi des problèmes
- Twitter
- [Mailing List])(http://groups.google.com/group/sinatrarb/topics)
- IRC : #sinatra sur http://freenode.net
- IRC : #sinatra on http://freenode.net
- Sinatra Book Tutoriels et recettes
- Sinatra Recipes trucs et astuces rédigés par la communauté
- Documentation API de la dernière version ou du HEAD courant sur http://rubydoc.info
- CI server

Sinatra

Einführung

Wichtig: Dieses Dokument ist eine Übersetzung aus dem Englischen und unter Umständen nicht auf dem aktuellen Stand.

Sinatra ist eine DSL, die das schnelle Erstellen von Webanwendungen in Ruby mit minimalem Aufwand ermöglicht:

```
# myapp.rb
require 'sinatra'
get '/' do
  'Hallo Welt!'
end
```

Einfach via `rubygems` installieren und starten:

```
gem install sinatra
ruby myapp.rb
```

Die Seite kann nun unter http://localhost:4567 betrachtet werden.

Es wird empfohlen, den Thin-Server via `gem install thin` zu installieren, den Sinatra dann, soweit vorhanden, automatisch verwendet.

Inhalt

Routen

In Sinatra wird eine Route durch eine HTTP-Methode und ein URL-Muster definiert. Jeder dieser Routen wird ein Ruby-Block zugeordnet:

```
get '/' do
  .. zeige etwas ..
end

post '/' do
  .. erstelle etwas ..
end

put '/' do
  .. update etwas ..
end

delete '/' do
  .. entferne etwas ..
end

options '/' do
  .. zeige, was wir können ..
end

link '/' do
  .. verbinde etwas ..
end

unlink '/' do
  .. trenne etwas ..
end
```


Die Routen werden in der Reihenfolge durchlaufen, in der sie definiert wurden. Das erste Routen-Muster, das mit dem Request übereinstimmt, wird ausgeführt.

Die Muster der Routen können benannte Parameter beinhalten, die über den `params`-Hash zugänglich gemacht werden:

```
get '/hallo/:name' do
  # passt auf "GET /hallo/foo" und "GET /hallo/bar"
  # params[:name] ist 'foo' oder 'bar'
  "Hallo #{params[:name]}"
end
```

Man kann auf diese auch mit Block-Parametern zugreifen:

```
get '/hallo/:name' do |n|
  "Hallo #{n}"
end
```

Routen-Muster können auch mit Splat- oder Wildcard-Parametern über das `params[:splat]`-Array angesprochen werden:

```
```

```
get '/sag/*/zu/*' do
  # passt auf /sag/hallo/zu/welt
  params[:splat] # => ["hallo", "welt"]
end

get '/download/*.*' do
  # passt auf /download/pfad/zu/datei.xml
  params[:splat] # => ["pfad/zu/datei", "xml"]
end
```

Oder mit Block-Parametern:

```
get '/download/*.*' do |pfad, endung|
  [pfad, endung] # => ["Pfad/zu/Datei", "xml"]
end
```

Routen mit regulären Ausdrücken sind auch möglich:

```
get %r{/hallo/([\w]+)} do
  "Hallo, #{params[:captures].first}!"
end
```

Und auch hier können Block-Parameter genutzt werden:

```
get %r{/hallo/([\w]+)} do |c|
  "Hallo, #{c}!"
end
```

Routen-Muster können auch mit optionalen Parametern ausgestattet werden:

```
get '/posts.?:format?' do
  # passt auf "GET /posts" sowie jegliche Erweiterung
  # wie "GET /posts.json", "GET /posts.xml" etc.
end
```

Anmerkung: Solange man den sog. Path Traversal Attack-Schutz nicht deaktiviert (siehe weiter unten), kann es sein, dass der Request-Pfad noch vor dem Abgleich mit den Routen modifiziert wird.

Bedingungen

An Routen können eine Vielzahl von Bedingungen angehängt werden, die erfüllt sein müssen, damit der Block ausgeführt wird. Möglich wäre etwa eine Einschränkung des User-Agents:

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

get '/foo' do
  # passt auf andere Browser
```

```
end
```

Andere mitgelieferte Bedingungen sind `host_name` und `provides`:

```
get '/', :host_name => /^admin\./ do
  "Adminbereich, Zugriff verweigert!"
end

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

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

Es können auch andere Bedingungen relativ einfach hinzugefügt werden:

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

get '/auto_gewinnen', :probability => 0.1 do
  "Du hast gewonnen!"
end

get '/auto_gewinnen' do
  "Tut mir leid, verloren."
end
```

Bei Bedingungen, die mehrere Werte annehmen können, sollte ein Splat verwendet werden:

```
set(:auth) do |*roles|   # <- hier kommt der Splat ins Spiel
  condition do
    unless logged_in? && roles.any? {|role| current_user.in_role? role }
      redirect "/login/", 303
    end
  end
end

get "/mein/account/", :auth => [:user, :admin] do
  "Mein Account"
end

get "/nur/admin/", :auth => :admin do
  "Nur Admins dürfen hier rein!"
end
```

Rückgabewerte

Durch den Rückgabewert eines Routen-Blocks wird mindestens der Response-Body festgelegt, der an den HTTP-Client, bzw. die nächste Rack-Middleware, weitergegeben wird. Im Normalfall handelt es sich hierbei, wie in den vorangehenden Beispielen zu sehen war, um einen String. Es werden allerdings auch andere Werte akzeptiert.

Es kann jedes gültige Objekt zurückgegeben werden, bei dem es sich entweder um einen Rack-Rückgabewert, einen Rack-Body oder einen HTTP-Status-Code handelt:

- Ein Array mit drei Elementen: `[Status (Fixnum), Headers (Hash), Response-Body (antwortet auf #each)]`.
- Ein Array mit zwei Elementen: `[Status (Fixnum), Response-Body (antwortet auf #each)]`.
- Ein Objekt, das auf `#each` antwortet und den an diese Methode übergebenen Block nur mit Strings als Übergabewerte aufruft.
- Ein Fixnum, das den Status-Code festlegt.

Damit lässt sich relativ einfach Streaming implementieren:

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

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

Ebenso kann die `stream`-Helfer-Methode (s.u.) verwendet werden, die Streaming direkt in die Route integriert.

Eigene Routen-Muster

Wie oben schon beschrieben, ist Sinatra von Haus aus mit Unterstützung für String-Muster und Reguläre Ausdrücke zum Abgleichen von Routen ausgestattet. Das muss aber noch nicht alles sein, es können ohne großen Aufwand eigene Routen-Muster erstellt werden:

```
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
```

Beachte, dass das obige Beispiel etwas übertrieben wirkt. Es geht auch einfacher:

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

Oder unter Verwendung eines negativen look ahead:

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

Statische Dateien

Statische Dateien werden aus dem `./public`-Ordner ausgeliefert. Es ist möglich, einen anderen Ort zu definieren, indem man die `:public_folder`-Option setzt:

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

Zu beachten ist, dass der Ordnername public nicht Teil der URL ist. Die Datei `./public/css/style.css` ist unter `http://example.com/css/style.css` zu finden.

Um den `Cache-Control`-Header mit Informationen zu versorgen, verwendet man die `:static_cache_control`-Einstellung (s.u.).

Views/Templates

Alle Templatesprachen verwenden ihre eigene Renderingmethode, die jeweils einen String zurückgibt:

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

Dieses Beispiel rendert `views/index.erb`.

Anstelle eines Templatenamens kann man auch direkt die Templatesprache verwenden:

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

Templates nehmen ein zweite Argument an, den Options-Hash:

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

Dieses Beispiel rendert `views/index.erb` eingebettet in `views/post.erb` (Voreinstellung ist `views/layout.erb`, sofern es vorhanden ist.)

Optionen, die Sinatra nicht versteht, werden an das Template weitergereicht:

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

Für alle Templates können auch generelle Einstellungen festgelegt werden:

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

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

Optionen, die an die Rendermethode weitergegeben werden, überschreiben die Einstellungen, die mit `set` festgelegt wurden.

Einstellungen:

locals
: Liste von lokalen Variablen, die and das Dokument weitergegeben werden. Praktisch für Partials: `erb "<%= foo %>", :locals => {:foo => "bar"}`

default_encoding
: Gibt die Stringkodierung an, die verwendet werden soll. Voreingestellt auf `settings.default_encoding`.

views
: Ordner, aus dem die Templates heraus geladen werden. Voreingestellt auf `settings.views`.

layout
: Legt fest, ob ein Layouttemplate verwendet werden soll oder nicht (`true` oder`false`). Ist es ein Symbol, dann legt es fest, welches Template als Layout verwendet wird: `erb :index, :layout => !request.xhr?`

content_type
: Content-Type den das Template ausgibt. Voreinstellung hängt von der Templatesprache ab.

scope
: Scope, in dem das Template gerendert wird. Liegt standardmäßig innerhalb der App-Instanz. Wird Scope geändert, sind Instanzvariablen und Helfermethoden nicht verfügbar.

layout_engine
: Legt fest, welcher Renderer für das Layout verantwortlich ist. Hilfreich für Sprachen, die sonst keine Templates unterstützen. Voreingestellt auf den Renderer, der für das Template verwendet wird: `set :rdoc, :layout_engine => :erb`

layout_options
: Besondere Einstellungen, die nur für das Rendering verwendet werden: `set :rdoc, :layout_options => { :views => 'views/layouts' }`

Sinatra geht davon aus, dass die Templates sich im `./views` Verzeichnis befinden. Es kann jedoch ein anderer Ordner festgelegt werden:

```
set :views, settings.root + '/templates'
```

Es ist zu beachten, dass immer mit Symbolen auf Templates verwiesen werden muss, auch dann, wenn sie sich in einem Unterordner befinden:

```
haml :'unterverzeichnis/template'
```

Rendering-Methoden rendern jeden String direkt.

Direkte Templates

```
get '/' do
  haml '%div.title Hallo Welt'
end
```

Hier wird der String direkt gerendert.

Verfügbare Templatesprachen

Einige Sprachen haben mehrere Implementierungen. Um festzulegen, welche verwendet wird (und dann auch Thread-sicher ist), verwendet man am besten zu Beginn ein `'require'`:

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

Haml Templates

Abhängigkeit	haml
Dateierweiterung	`.haml`
Beispiel	`haml :index, :format => :html5`

Erb Templates

Abhängigkeit	erubis oder erb (Standardbibliothek von Ruby)
Dateierweiterungen	`.erb`, `.rhtml` oder `.erubis` (nur Erubis)
Beispiel	`erb :index`

Builder Templates

Abhängigkeit	builder
Dateierweiterung	`.builder`
Beispiel	`builder { |xml| xml.em "Hallo" }`

Nimmt ebenso einen Block für Inline-Templates entgegen (siehe Beispiel).

Nokogiri Templates

Abhängigkeit	nokogiri
Dateierweiterung	`.nokogiri`
Beispiel	`nokogiri { |xml| xml.em "Hallo" }`

Nimmt ebenso einen Block für Inline-Templates entgegen (siehe Beispiel).

Sass Templates

Abhängigkeit	sass
Dateierweiterung	`.sass`
Beispiel	`sass :stylesheet, :style => :expanded`

SCSS Templates

Abhängigkeit	sass
Dateierweiterung	`.scss`
Beispiel	`scss :stylesheet, :style => :expanded`

Less Templates

Abhängigkeit	less
Dateierweiterung	`.less`
Beispiel	`less :stylesheet`

Liquid Templates

Abhängigkeit	liquid
Dateierweiterung	`.liquid`
Beispiel	`liquid :index, :locals => { :key => 'Wert' }`

Da man aus dem Liquid-Template heraus keine Ruby-Methoden aufrufen kann (ausgenommen `yield`), wird man üblicherweise locals verwenden wollen, mit denen man Variablen weitergibt.

Markdown Templates

Abhängigkeit	Eine der folgenden Bibliotheken: RDiscount, RedCarpet, BlueCloth, kramdown oder maruku
Dateierweiterungen	`.markdown`, `.mkd` und `.md`
Beispiel	`markdown :index, :layout_engine => :erb`

Da man aus den Markdown-Templates heraus keine Ruby-Methoden aufrufen und auch keine locals verwenden kann, wird man Markdown üblicherweise in Kombination mit anderen Renderern verwenden wollen:

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

Beachte, dass man die `markdown`-Methode auch aus anderen Templates heraus aufrufen kann:

```
%h1 Gruß von Haml!
%p= markdown(:Grüße)
```

Da man Ruby nicht von Markdown heraus aufrufen kann, können auch Layouts nicht in Markdown geschrieben werden. Es ist aber möglich, einen Renderer für die Templates zu verwenden und einen anderen für das Layout, indem die `:layout_engine`-Option verwendet wird.

Textile Templates

Abhängigkeit	RedCloth
Dateierweiterung	`.textile`
Beispiel	`textile :index, :layout_engine => :erb`

Da man aus dem Textile-Template heraus keine Ruby-Methoden aufrufen und auch keine locals verwenden kann, wird man Textile üblicherweise in Kombination mit anderen Renderern verwenden wollen:

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

Beachte, dass man die `textile`-Methode auch aus anderen Templates heraus aufrufen kann:

```
%h1 Gruß von Haml!
%p= textile(:Grüße)
```

Da man Ruby nicht von Textile heraus aufrufen kann, können auch Layouts nicht in Textile geschrieben werden. Es ist aber möglich, einen Renderer für die Templates zu verwenden und einen anderen für das Layout, indem die `:layout_engine`-Option verwendet wird.

RDoc Templates

Abhängigkeit	rdoc
Dateierweiterung	`.rdoc`
Beispiel	`textile :README, :layout_engine => :erb`

Da man aus dem RDoc-Template heraus keine Ruby-Methoden aufrufen und auch keine locals verwenden kann, wird man RDoc üblicherweise in Kombination mit anderen Renderern verwenden wollen:

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

Beachte, dass man die `rdoc`-Methode auch aus anderen Templates heraus aufrufen kann:

```
%h1 Gruß von Haml!
%p= rdoc(:Grüße)
```

Da man Ruby nicht von RDoc heraus aufrufen kann, können auch Layouts nicht in RDoc geschrieben werden. Es ist aber möglich, einen Renderer für die Templates zu verwenden und einen anderen für das Layout, indem die `:layout_engine`-Option verwendet wird.

Radius Templates

Abhängigkeit	radius
Dateierweiterung	`.radius`
Beispiel	`radius :index, :locals => { :key => 'Wert' }`

Da man aus dem Radius-Template heraus keine Ruby-Methoden aufrufen kann, wird man üblicherweise locals verwenden wollen, mit denen man Variablen weitergibt.

Markaby Templates

Abhängigkeit	markaby
Dateierweiterung	`.mab`
Beispiel	`markaby { h1 "Willkommen!" }`

Nimmt ebenso einen Block für Inline-Templates entgegen (siehe Beispiel).

RABL Templates

Abhängigkeit	rabl
Dateierweiterung	`.rabl`
Beispiel	`rabl :index`

Slim Templates

Abhängigkeit	slim
Dateierweiterung	`.slim`
Beispiel	`slim :index`

Creole Templates

Abhängigkeit	creole
Dateierweiterung	`.creole`
Beispiel	`creole :wiki, :layout_engine => :erb`

Da man aus dem Creole-Template heraus keine Ruby-Methoden aufrufen und auch keine locals verwenden kann, wird man Creole üblicherweise in Kombination mit anderen Renderern verwenden wollen:

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

Beachte, dass man die `creole`-Methode auch aus anderen Templates heraus aufrufen kann:

```
%h1 Gruß von Haml!
%p= creole(:Grüße)
```

Da man Ruby nicht von Creole heraus aufrufen kann, können auch Layouts nicht in Creole geschrieben werden. Es ist aber möglich, einen Renderer für die Templates zu verwenden und einen anderen für das Layout, indem die `:layout_engine`-Option verwendet wird.

CoffeeScript Templates

Abhängigkeit	coffee-script und eine Möglichkeit JavaScript auszuführen.
Dateierweiterung	`.coffee`
Beispiel	`coffee :index`

Stylus Templates

Abhängigkeit	Stylus und eine Möglichkeit JavaScript auszuführen.
Dateierweiterung	`.styl`
Beispiel	`stylus :index`

Um Stylus-Templates ausführen zu können, müssen `stylus` und `stylus/tilt` zuerst geladen werden:

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

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

Yajl Templates

Abhängigkeit	yajl-ruby
Dateierweiterung	`.yajl`
Beispiel	`yajl :index, :locals => { :key => 'qux' }, :callback => 'present', :variable => 'resource'`

Die Template-Quelle wird als Ruby-String evaluiert. Die daraus resultierende json Variable wird mit Hilfe von `#to_json` umgewandelt:

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

Die `:callback` und `:variable` Optionen können mit dem gerenderten Objekt verwendet werden:

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

WLang Templates

Abhängigkeit	wlang
Dateierweiterung	`.wlang`
Beispiel	`wlang :index, :locals => { :key => 'value' }`

Ruby-Methoden in wlang aufzurufen entspricht nicht den idiomatischen Vorgaben von wlang, es bietet sich deshalb an, `:locals` zu verwenden. Layouts, die wlang und `yield` verwenden, werden aber trotzdem unterstützt.

Rendert den eingebetteten Template-String.

Auf Variablen in Templates zugreifen

Templates werden in demselben Kontext ausgeführt wie Routen. Instanzvariablen in Routen sind auch direkt im Template verfügbar:

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

Oder durch einen expliziten Hash von lokalen Variablen:

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

Dies wird typischerweise bei Verwendung von Subtemplates (partials) in anderen Templates eingesetzt.

Templates mit `yield` und verschachtelte Layouts

Ein Layout ist üblicherweise ein Template, dass ein `yield` aufruft. Ein solches Template kann entweder wie oben beschrieben über die `:template` option verwendet werden oder mit einem Block gerendert werden:

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

Dieser Code entspricht weitestgehend `erb :index, :layout => :post`.

Blöcke an Render-Methoden weiterzugeben ist besonders bei verschachtelten Layouts hilfreich:

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

Der gleiche Effekt kann auch mit weniger Code erreicht werden:

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

```
  erb :user
end
```

Zur Zeit nehmen folgende Renderer Blöcke an: `erb`, `haml`, `liquid`, `slim` und `wlang`.

Das gleich gilt auch für die allgemeine `render` Methode.

Inline-Templates

Templates können auch am Ende der Datei definiert werden:

```
require 'sinatra'

get '/' do
  haml :index
end

__END__

@@ layout
%html
  = yield

@@ index
%div.title Hallo Welt!!!!!
```

Anmerkung: Inline-Templates, die in der Datei definiert sind, die `require 'sinatra'` aufruft, werden automatisch geladen. Um andere Inline-Templates in anderen Dateien aufzurufen, muss explizit `enable :inline_templates` verwendet werden.

Benannte Templates

Templates können auch mit der Top-Level `template`-Methode definiert werden:

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

template :index do
  '%div.title Hallo Welt!'
end

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

Wenn ein Template mit dem Namen “layout” existiert, wird es bei jedem Aufruf verwendet. Durch `:layout => false` kann das Ausführen verhindert werden:

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

Dateiendungen zuordnen

Um eine Dateiendung einer Template-Engine zuzuordnen, kann `Tilt.register` genutzt werden. Wenn etwa die Dateiendung `tt` für Textile-Templates genutzt werden soll, lässt sich dies wie folgt bewerkstelligen:

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

Eine eigene Template-Engine hinzufügen

Zu allererst muss die Engine bei Tilt registriert und danach eine Rendering-Methode erstellt werden:

```
Tilt.register :mtt, MeineTolleTemplateEngine

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

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

Dieser Code rendert `./views/application.mtt`. Siehe github.com/rtomayko/tilt, um mehr über Tilt zu erfahren.

Filter

Before-Filter werden vor jedem Request in demselben Kontext, wie danach die Routen, ausgeführt. So können etwa Request und Antwort geändert werden. Gesetzte Instanzvariablen in Filtern können in Routen und Templates verwendet werden:

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

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

After-Filter werden nach jedem Request in demselben Kontext ausgeführt und können ebenfalls Request und Antwort ändern. In Before-Filtern gesetzte Instanzvariablen können in After-Filtern verwendet werden:

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

Filter können optional auch mit einem Muster ausgestattet werden, welches auf den Request-Pfad passen muss, damit der Filter ausgeführt wird:

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

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

Ähnlich wie Routen können Filter auch mit weiteren Bedingungen eingeschränkt werden:

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

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

Helfer

Durch die Top-Level `helpers`-Methode werden sogenannte Helfer-Methoden definiert, die in Routen und Templates verwendet werden können:

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

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

Sessions verwenden

Sessions werden verwendet, um Zustände zwischen den Requests zu speichern. Sind sie aktiviert,

kann ein Session-Hash je Benutzer-Session verwendet werden:

```
enable :sessions

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

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

Beachte, dass `enable :sessions` alle Daten in einem Cookie speichert. Unter Umständen kann dies negative Effekte haben, z.B. verursachen viele Daten höheren, teilweise überflüssigen Traffic. Um das zu vermeiden, kann eine Rack- Session-Middleware verwendet werden. Dabei wird auf `enable :sessions` verzichtet und die Middleware wie üblich im Programm eingebunden:

```
use Rack::Session::Pool, :expire_after => 2592000

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

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

Um die Sicherheit zu erhöhen, werden Cookies, die Session-Daten führen, mit einem sogenannten Session-Secret signiert. Da sich dieses Geheimwort bei jedem Neustart der Applikation automatisch ändert, ist es sinnvoll, ein eigenes zu wählen, damit sich alle Instanzen der Applikation dasselbe Session-Secret teilen:

```
set :session_secret, 'super secret'
```

Zur weiteren Konfiguration kann man einen Hash mit Optionen in den `sessions` Einstellungen ablegen.

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

Anhalten

Zum sofortigen Stoppen eines Request in einem Filter oder einer Route:

```
halt
```

Der Status kann beim Stoppen auch angegeben werden:

```
halt 410
```

Oder auch den Response-Body:

```
halt 'Hier steht der Body'
```

Oder beides:

```
halt 401, 'verschwinde!'
```

Sogar mit Headern:

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

Natürlich ist es auch möglich, ein Template mit `halt` zu verwenden:

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

Weiterspringen

Eine Route kann mittels `pass` zu der nächsten passenden Route springen:

```
get '/raten/:wer' do
  pass unless params[:wer] == 'Frank'
  'Du hast mich!'
end

get '/raten/*' do
  'Du hast mich nicht!'
end
```

Der Block wird sofort verlassen und es wird nach der nächsten treffenden Route gesucht. Ein 404-Fehler wird zurückgegeben, wenn kein treffendes Routen-Muster gefunden wird.

Eine andere Route ansteuern

Manchmal entspricht `pass` nicht den Anforderungen, wenn das Ergebnis einer anderen Route gefordert wird. Um das zu erreichen, lässt sich `call` nutzen:

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

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

Beachte, dass in dem oben angegeben Beispiel die Performance erheblich erhöht werden kann, wenn `"bar"` in eine Helfer-Methode umgewandelt wird, auf die `/foo` und `/bar` zugreifen können.

Wenn der Request innerhalb derselben Applikations-Instanz aufgerufen und keine Kopie der Instanz erzeugt werden soll, kann `call!` anstelle von `call` verwendet werden.

Die Rack-Spezifikationen enthalten weitere Informationen zu `call`.

Body, Status-Code und Header setzen

Es ist möglich und empfohlen, den Status-Code sowie den Response-Body mit einem Returnwert in der Route zu setzen. In manchen Situationen kann es jedoch sein, dass der Body an irgendeiner anderen Stelle während der Ausführung gesetzt wird. Das lässt sich mit der Helfer-Methode `body` bewerkstelligen. Wird `body` verwendet, lässt sich der Body jederzeit über diese Methode aufrufen:

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

after do
  puts body
end
```

Ebenso ist es möglich, einen Block an `body` weiterzureichen, der dann vom Rack-Handler ausgeführt wird (lässt sich z.B. zur Umsetzung von Streaming einsetzen, siehe auch “Rückgabewerte”).

Vergleichbar mit `body` lassen sich auch Status-Code und Header setzen:

```
get '/foo' do
  status 418
  headers \
    "Allow"   => "BREW, POST, GET, PROPFIND, WHEN",
    "Refresh" => "Refresh: 20; http://www.ietf.org/rfc/rfc2324.txt"
  halt "Ich bin ein Teekesselchen"
end
```

Genau wie bei `body` liest ein Aufrufen von `headers` oder `status` ohne Argumente den aktuellen Wert aus.

Response-Streams

In manchen Situationen sollen Daten bereits an den Client zurückgeschickt werden, bevor ein vollständiger Response bereit steht. Manchmal will man die Verbindung auch erst dann beenden und Daten so lange an den Client zurückschicken, bis er die Verbindung abbricht. Für diese Fälle gibt es die `stream`-Helfer-Methode, die es einem erspart eigene Lösungen zu schreiben:

```
```

```
get '/' do
  stream do |out|
    out << "Das ist ja mal wieder fanta - \n"
    sleep 0.5
    out << " (bitte warten ..) \n"
    sleep 1
    out << " - stisch!\n"
  end
end
```

Damit lassen sich Streaming-APIs realisieren, sog. Server Sent Events die als Basis für WebSockets dienen. Ebenso können sie verwendet werden, um den Durchsatz zu erhöhen, wenn ein Teil der Daten von langsamen Ressourcen abhängig ist.

Es ist zu beachten, dass das Verhalten beim Streaming, insbesondere die Anzahl nebenläufiger Anfragen, stark davon abhängt, welcher Webserver für die Applikation verwendet wird. Einige Server, z.B. WEBRick, unterstützen Streaming nicht oder nur teilweise. Sollte der Server Streaming nicht unterstützen, wird ein vollständiger Response-Body zurückgeschickt, sobald der an `stream` weitergegebene Block abgearbeitet ist. Mit Shotgun funktioniert Streaming z.B. überhaupt nicht.

Ist der optionale Parameter `keep_open` aktiviert, wird beim gestreamten Objekt `close` nicht aufgerufen und es ist einem überlassen dies an einem beliebigen späteren Zeitpunkt nachholen. Die Funktion ist jedoch nur bei Event-gesteuerten Serven wie Thin oder Rainbows möglich, andere Server werden trotzdem den Stream beenden:

```
# Durchgehende Anfrage (long polling)

set :server, :thin
connections = []

get '/subscribe' do
  # Client-Registrierung beim Server, damit Events mitgeteilt werden können
  stream(:keep_open) { |out| connections << out }

  # tote Verbindungen entfernen
  connections.reject!(&:closed?)

  # Rückmeldung
  "Angemeldet"
end

post '/message' do
  connections.each do |out|
    # Den Client über eine neue Nachricht in Kenntnis setzen
    # notify client that a new message has arrived
    out << params[:message] << "\n"

    # Den Client zur erneuten Verbindung auffordern
    out.close
  end

  # Rückmeldung
  "Mitteilung erhalten"
end
```

Logger

Im Geltungsbereich eines Request stellt die `logger` Helfer-Methode eine `Logger` Instanz zur Verfügung:

```
get '/' do
  logger.info "es passiert gerade etwas"
  # ...
end
```

Der Logger übernimmt dabei automatisch alle im Rack-Handler eingestellten Log-Vorgaben. Ist Loggen ausgeschaltet, gibt die Methode ein Leerobjekt zurück. In den Routen und Filtern muss man sich also nicht weiter darum kümmern.

Beachte, dass das Loggen standardmäßig nur für `Sinatra::Application` voreingestellt ist. Wird über `Sinatra::Base` vererbt, muss es erst aktiviert werden:

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

Damit auch keine Middleware das Logging aktivieren kann, muss die `logging` Einstellung auf `nil` gesetzt werden. Das heißt aber auch, dass `logger` in diesem Fall `nil` zurückgeben wird. Üblicherweise wird das eingesetzt, wenn ein eigener Logger eingerichtet werden soll. Sinatra wird dann verwenden, was in `env['rack.logger']` eingetragen ist.

Mime-Types

Wenn `send_file` oder statische Dateien verwendet werden, kann es vorkommen, dass Sinatra den Mime-Typ nicht kennt. Registriert wird dieser mit `mime_type` per Dateiendung:

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

Es kann aber auch der `content_type`-Helfer verwendet werden:

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

URLs generieren

Zum Generieren von URLs sollte die `url`-Helfer-Methode genutzen werden, so z.B. beim Einsatz von Haml:

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

Soweit vorhanden, wird Rücksicht auf Proxys und Rack-Router genommen.

Diese Methode ist ebenso über das Alias `to` zu erreichen (siehe Beispiel unten).

Browser-Umleitung

Eine Browser-Umleitung kann mithilfe der `redirect`-Helfer-Methode erreicht werden:

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

Weitere Parameter werden wie Argumente der `halt`-Methode behandelt:

```
redirect to('/bar'), 303
redirect 'http://google.com', 'Hier bist du falsch'
```

Ebenso leicht lässt sich ein Schritt zurück mit dem Alias `redirect back` erreichen:

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

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

Um Argumente an ein Redirect weiterzugeben, können sie entweder dem Query übergeben:

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

oder eine Session verwendet werden:

```
enable :sessions

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

get '/bar' do
```

```
  session[:secret]
end
```

Cache einsetzen

Ein sinnvolles Einstellen von Header-Daten ist die Grundlage für ein ordentliches HTTP-Caching.

Der Cache-Control-Header lässt sich ganz einfach einstellen:

```
get '/' do
  cache_control :public
  "schon gecached"
end
```

Profitipp: Caching im before-Filter aktivieren

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

Bei Verwendung der `expires`-Helfermethode zum Setzen des gleichnamigen Headers, wird `Cache-Control` automatisch eigestellt:

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

Um alles richtig zu machen, sollten auch `etag` oder `last_modified` verwendet werden. Es wird empfohlen, dass diese Helfer aufgerufen werden **bevor** die eigentliche Arbeit anfängt, da sie sofort eine Antwort senden, wenn der Client eine aktuelle Version im Cache vorhält:

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

ebenso ist es möglich einen schwachen ETag zu verwenden:

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

Diese Helfer führen nicht das eigentliche Caching aus, sondern geben die dafür notwendigen Informationen an den Cache weiter. Für schnelle Reverse-Proxy Cache-Lösungen bietet sich z.B. rack-cache an:

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

```
use Rack::Cache

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

Um den `Cache-Control`-Header mit Informationen zu versorgen, verwendet man die `:static_cache_control`-Einstellung (s.u.).

Nach RFC 2616 sollte sich die Anwendung anders verhalten, wenn ein If-Match oder ein If-None_match Header auf `*` gesetzt wird in Abhängigkeit davon, ob die Resource bereits existiert. Sinatra geht davon aus, dass Ressourcen bei sicheren Anfragen (z.B. bei get oder Idempotenten Anfragen wie put) bereits existieren, wobei anderen Ressourcen (besipielsweise bei post), als neue Ressourcen behandelt werden. Dieses Verhalten lässt sich mit der `:new_resource` Option ändern:

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

Soll das schwache ETag trotzdem verwendet werden, verwendet man die `:kind` Option:

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

Dateien versenden

Zum Versenden von Dateien kann die `send_file`-Helfer-Methode verwendet werden:

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

Für `send_file` stehen einige Hash-Optionen zur Verfügung:

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

filename
: Dateiname als Response. Standardwert ist der eigentliche Dateiname.

last_modified
: Wert für den Last-Modified-Header, Standardwert ist `mtime` der Datei.

type
: Content-Type, der verwendet werden soll. Wird, wenn nicht angegeben, von der Dateiendung abgeleitet.

disposition
: Verwendet für Content-Disposition. Mögliche Werte sind: `nil` (Standard), `:attachment` und

```
    :inline.
length
    Content-Length-Header. Standardwert ist die Dateigröße.
```

Soweit vom Rack-Handler unterstützt, werden neben der Übertragung über den Ruby-Prozess auch andere Möglichkeiten genutzt. Bei Verwendung der `send_file`-Helfer-Methode kümmert sich Sinatra selbstständig um die Range-Requests.

Das Request-Objekt

Auf das `request`-Objekt der eigehenden Anfrage kann vom Anfrage-Scope aus zugegriffen werden:

```
# App läuft unter 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 des Client (siehe unten)
  request.scheme              # "http"
  request.script_name         # "/example"
  request.path_info           # "/foo"
  request.port                # 80
  request.request_method      # "GET"
  request.query_string        # ""
  request.content_length      # Länge des request.body
  request.media_type          # Medientypus von request.body
  request.host                # "example.com"
  request.get?                # true (ähnliche Methoden für andere Verben)
  request.form_data?          # false
  request["irgendein_param"]  # Wert von einem Parameter; [] ist die Kurzform für den
params Hash
  request.referrer            # Der Referrer des Clients oder '/'
  request.user_agent          # User-Agent (verwendet in der :agent Bedingung)
  request.cookies             # Hash des Browser-Cookies
  request.xhr?                # Ist das hier ein Ajax-Request?
  request.url                 # "http://example.com/example/foo"
  request.path                # "/example/foo"
  request.ip                  # IP-Adresse des Clients
  request.secure?             # false (true wenn SSL)
  request.forwarded?          # true (Wenn es hinter einem Reverse-Proxy verwendet wird)
  request.env                 # vollständiger env-Hash von Rack übergeben
end
```

Manche Optionen, wie etwa `script_name` oder `path_info`, sind auch schreibbar:

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

get "/" do
  "Alle Anfragen kommen hier an!"
end
```

Der `request.body` ist ein IO- oder StringIO-Objekt:

```
post "/api" do
  request.body.rewind # falls schon jemand davon gelesen hat
  daten = JSON.parse request.body.read
  "Hallo #{daten['name']}"
end
```

Anhänge

Damit der Browser erkennt, dass ein Response gespeichert und nicht im Browser angezeigt werden soll, kann der `attachment`-Helfer verwendet werden:

```
get '/' do
  attachment
  "Speichern!"
end
```

Ebenso kann eine Dateiname als Parameter hinzugefügt werden:

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

Umgang mit Datum und Zeit

Sinatra bietet eine `time_for`-Helfer-Methode, die aus einem gegebenen Wert ein Time-Objekt generiert. Ebenso kann sie nach `DateTime`, `Date` und ähnliche Klassen konvertieren:

```
get '/' do
  pass if Time.now > time_for('Dec 23, 2012')
  "Noch Zeit"
end
```

Diese Methode wird intern für +expires, `last_modiefied` und ihresgleichen verwendet. Mit ein paar Handgriffen lässt sich diese Methode also in ihrem Verhalten erweitern, indem man `time_for` in der eigenen Applikation überschreibt:

```
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
  "Hallo"
end
```

Nachschlagen von Template-Dateien

Die `find_template`-Helfer-Methode wird genutzt, um Template-Dateien zum Rendern aufzufinden:

```
find_template settings.views, 'foo', Tilt[:haml] do |file|
  puts "könnte diese hier sein: #{     }"
end
```

Das ist zwar nicht wirklich brauchbar, aber wenn man sie überschreibt, kann sie nützlich werden, um eigene Nachschlage-Mechanismen einzubauen. Zum Beispiel dann, wenn mehr als nur ein view-Verzeichnis verwendet werden soll:

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

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

Ein anderes Beispiel wäre, verschiedene Vereichnisse für verschiedene Engines zu verwenden:

```
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
```

Ebensogut könnte eine Extension aber auch geschrieben und mit anderen geteilt werden!

Beachte, dass `find_template` nicht prüft, ob eine Datei tatsächlich existiert. Es wird lediglich der angegebene Block aufgerufen und nach allen möglichen Pfaden gesucht. Das ergibt kein Performance-Problem, da `render block` verwendet, sobald eine Datei gefunden wurde. Ebenso werden Template-Pfade samt Inhalt gecached, solange nicht im Entwicklungsmodus gearbeitet wird. Das sollte im Hinterkopf behalten werden, wenn irgendwelche verrückten Methoden zusammenbastelt werden.

Konfiguration

Wird einmal beim Starten in jedweder Umgebung ausgeführt:

```
configure do
  # setze eine Option
  set :option, 'wert'

  # setze mehrere Optionen
  set :a => 1, :b => 2

  # das gleiche wie `set :option, true`
  enable :option

  # das gleiche wie `set :option, false`
  disable :option

  # dynamische Einstellungen mit Blöcken
  set(:css_dir) { File.join(views, 'css') }
end
```

Läuft nur, wenn die Umgebung (RACK_ENV-Umgebungsvariable) auf `:production` gesetzt ist:

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

Läuft nur, wenn die Umgebung auf `:production` oder auf `:test` gesetzt ist:

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

Diese Einstellungen sind über `settings` erreichbar:

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

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

Einstellung des Angriffsschutzes

Sinatra verwendet Rack::Protection, um die Anwendung vor häufig vorkommenden Angriffen zu schützen. Diese Voreinstellung lässt sich selbstverständlich deaktivieren, der damit verbundene Geschwindigkeitszuwachs steht aber in keinem Verhätnis zu den möglichen Risiken.

```
disable :protection
```

Um einen bestimmten Schutzmechanismus zu deaktivieren, fügt man `protection` einen Hash mit Optionen hinzu:

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

Neben Strings akzeptiert `:except` auch Arrays, um gleich mehrere Schutzmechanismen zu deaktivieren:

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

Mögliche Einstellungen

absolute_redirects
: Wenn ausgeschaltet, wird Sinatra relative Redirects zulassen. Jedoch ist Sinatra dann nicht mehr mit RFC 2616 (HTTP 1.1) konform, das nur absolute Redirects zulässt. Sollte eingeschaltet werden, wenn die Applikation hinter einem Reverse-Proxy liegt, der nicht ordentlich eingerichtet ist. Beachte, dass die `url`-Helfer-Methode nach wie vor absolute URLs erstellen wird, es sei denn, es wird als zweiter Parameter `false` angegeben. Standardmäßig nicht aktiviert.

add_charsets
: Mime-Types werden hier automatisch der Helfer-Methode `content_type` zugeordnet. Es empfielt sich, Werte hinzuzufügen statt sie zu überschreiben: `settings.add_charsets << "application/foobar"`

app_file
: Pfad zur Hauptdatei der Applikation. Wird verwendet, um das Wurzel-, Inline-, View- und öffentliche Verzeichnis des Projekts festzustellen.

bind
: IP-Address, an die gebunden wird (Standardwert: `0.0.0.0`). Wird nur für den eingebauten Server verwendet.

default_encoding
: Das Encoding, falls keines angegeben wurde. Standardwert ist `"utf-8"`.

dump_errors
: Fehler im Log anzeigen.

environment
: Momentane Umgebung. Standardmäßig auf `content_type` oder `"development"` eingestellt, soweit ersteres nicht vorhanden.

logging
: Den Logger verwenden.

lock
: Jeder Request wird gelocked. Es kann nur ein Request pro Ruby-Prozess gleichzeitig verarbeitet werden. Eingeschaltet, wenn die Applikation threadsicher ist. Standardmäßig nicht aktiviert.

method_override
: Verwende `_method`, um put/delete-Formulardaten in Browsern zu verwenden, die dies normalerweise nicht unterstützen.

port
: Port für die Applikation. Wird nur im internen Server verwendet.

prefixed_redirects
: Entscheidet, ob `request.script_name` in Redirects eingefügt wird oder nicht, wenn kein absoluter Pfad angegeben ist. Auf diese Weise verhält sich `redirect '/foo'` so, als wäre es ein `redirect to('/foo')`. Standardmäßig nicht aktiviert.

protection
: Legt fest, ob der Schutzmechanismus für häufig Vorkommende Webangriffe auf Webapplikationen aktiviert wird oder nicht. Weitere Informationen im vorhergehenden Abschnitt.

public_folder
: Das öffentliche Verzeichnis, aus dem Daten zur Verfügung gestellt werden können. Wird nur dann verwendet, wenn statische Daten zur Verfügung gestellt werden können (s.u. `static` Option). Leitet sich von der `app_file` Einstellung ab, wenn nicht gesetzt.

public_dir
: Alias für `public_folder`, s.o.

reload_templates
: Im development-Modus aktiviert.

root
: Wurzelverzeichnis des Projekts. Leitet sich von der `app_file` Einstellung ab, wenn nicht gesetzt.

raise_errors
: Einen Ausnahmezustand aufrufen. Beendet die Applikation. Ist automatisch aktiviert, wenn die Umgebung auf `"test"` eingestellt ist. Ansonsten ist diese Option deaktiviert.

run
: Wenn aktiviert, wird Sinatra versuchen, den Webserver zu starten. Nicht verwenden, wenn Rackup oder anderes verwendet werden soll.

running
: Läuft der eingebaute Server? Diese Einstellung nicht ändern!

server
: Server oder Liste von Servern, die als eingebaute Server zur Verfügung stehen. Standardmäßig auf `['thin', 'mongrel', 'webrick']` voreingestellt. Die Anordnung gibt die Priorität vor.

sessions
: Sessions auf Cookiebasis mittels `Rack::Session::Cookie`aktivieren. Für weitere Infos bitte in der Sektion 'Sessions verwenden' nachschauen.

show_exceptions
: Bei Fehlern einen Stacktrace im Browseranzeigen. Ist automatisch aktiviert, wenn die Umgebung auf `"development"` eingestellt ist. Ansonsten ist diese Option deaktiviert. Kann auch auf `:after_handler` gestellt werden, um eine anwendungsspezifische Fehlerbehandlung auszulösen, bevor der Fehlerverlauf im Browser angezeigt wird.

static
: Entscheidet, ob Sinatra statische Dateien zur Verfügung stellen soll oder nicht. Sollte nicht aktiviert werden, wenn ein Server verwendet wird, der dies auch selbstständig erledigen kann. Deaktivieren wird die Performance erhöhen. Standardmäßig aktiviert.

static_cache_control
: Wenn Sinatra statische Daten zur Verfügung stellt, können mit dieser Einstellung die `Cache-Control` Header zu den Responses hinzugefügt werden. Die Einstellung verwendet dazu die `cache_control` Helfer-Methode. Standardmäßig deaktiviert. Ein Array wird verwendet, um mehrere Werte gleichzeitig zu übergeben: `set :static_cache_control, [:public, :max_age => 300]`

threaded

Wird es auf `true` gesetzt, wird Thin aufgefordert `EventMachine.defer` zur Verarbeitung des Requests einzusetzen.

views
: Verzeichnis der Views. Leitet sich von der `app_file` Einstellung ab, wenn nicht gesetzt.

x_cascade
: Einstellung, ob der X-Cascade Header bei fehlender Route gesetzt wird oder nicht. Standardeinstellung ist `true`.

Umgebungen

Es gibt drei voreingestellte Umgebungen in Sinatra: `"development"`, `"production"` und `"test"`. Umgebungen können über die `RACK_ENV` Umgebungsvariable gesetzt werden. Die Standardeinstellung ist `"development"`. In diesem Modus werden alle Templates zwischen Requests neu geladen. Dazu gibt es besondere Fehlerseiten für 404 Stati und Fehlermeldungen. In `"production"` und `"test"` werden Templates automatisch gecached.

Um die Anwendung in einer anderen Umgebung auszuführen kann man die `-e` Option verwenden:

```
ruby my_app.rb -e [ENVIRONMENT]
```

In der Anwendung kann man die die Methoden `development?`, `test?` und `production?` verwenden, um die aktuelle Umgebung zu erfahren.

Fehlerbehandlung

Error-Handler laufen in demselben Kontext wie Routen und Filter, was bedeutet, dass alle Goodies wie `haml`, `erb`, `halt`, etc. verwendet werden können.

Nicht gefunden

Wenn eine `Sinatra::NotFound`-Exception geworfen wird oder der Statuscode 404 ist, wird der `not_found`-Handler ausgeführt:

```
not_found do
  'Seite kann nirgendwo gefunden werden.'
end
```

Fehler

Der `error`-Handler wird immer ausgeführt, wenn eine Exception in einem Routen-Block oder in

einem Filter geworfen wurde. Die Exception kann über die `sinatra.error`-Rack-Variable angesprochen werden:

```
error do
  'Entschuldige, es gab einen hässlichen Fehler - ' + env['sinatra.error'].name
end
```

Benutzerdefinierte Fehler:

```
error MeinFehler do
  'Au weia, ' + env['sinatra.error'].message
end
```

Dann, wenn das passiert:

```
get '/' do
  raise MeinFehler, 'etwas Schlimmes ist passiert'
end
```

bekommt man dieses:

```
Au weia, etwas Schlimmes ist passiert
```

Alternativ kann ein Error-Handler auch für einen Status-Code definiert werden:

```
error 403 do
  'Zugriff verboten'
end

get '/geheim' do
  403
end
```

Oder ein Status-Code-Bereich:

```
error 400..510 do
  'Halt?'
end
```

Sinatra setzt verschiedene `not_found`- und `error`-Handler in der Development-Umgebung ein, um hilfreiche Debugging Informationen und Stack Traces anzuzeigen.

Rack-Middleware

Sinatra baut auf Rack, einem minimalistischen Standard-Interface für Ruby-Webframeworks. Eines der interessantesten Features für Entwickler ist der Support von Middlewares, die zwischen den Server und die Anwendung geschaltet werden und so HTTP-Request und/oder Antwort überwachen und/oder manipulieren können.

Sinatra macht das Erstellen von Middleware-Verkettungen mit der Top-Level-Methode `use` zu einem Kinderspiel:

```
require 'sinatra'
require 'meine_middleware'

use Rack::Lint
use MeineMiddleware

get '/hallo' do
  'Hallo Welt'
end
```

Die Semantik von `use` entspricht der gleichnamigen Methode der Rack::Builder-DSL (meist verwendet in Rackup-Dateien). Ein Beispiel dafür ist, dass die `use`-Methode mehrere/verschiedene Argumente und auch Blöcke entgegennimmt:

```
use Rack::Auth::Basic do |username, password|
  username == 'admin' && password == 'geheim'
end
```

Rack bietet eine Vielzahl von Standard-Middlewares für Logging, Debugging, URL-Routing, Authentifizierung und Session-Verarbeitung. Sinatra verwendet viele von diesen Komponenten automatisch, abhängig von der Konfiguration. So muss `use` häufig nicht explizit verwendet werden.

Hilfreiche Middleware gibt es z.B. hier: rack, rack-contrib, mit CodeRack oder im Rack wiki.

Testen

Sinatra-Tests können mit jedem auf Rack aufbauendem Test-Framework geschrieben werden. Rack::Test wird empfohlen:

```
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 'Hallo Welt!', 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 "Du verwendest Songbird!", last_response.body
  end
end
```

Hinweis: Wird Sinatra modular verwendet, muss `Sinatra::Application` mit dem Namen der Applikations-Klasse ersetzt werden.

Sinatra::Base - Middleware, Bibliotheken und modulare Anwendungen

Das Definieren einer Top-Level-Anwendung funktioniert gut für Mikro-Anwendungen, hat aber Nachteile, wenn wiederverwendbare Komponenten wie Middleware, Rails Metal, einfache Bibliotheken mit Server-Komponenten oder auch Sinatra-Erweiterungen geschrieben werden sollen.

Das Top-Level geht von einer Konfiguration für eine Mikro-Anwendung aus (wie sie z.B. bei einer einzelnen Anwendungsdatei, `./public` und `./views` Ordner, Logging, Exception-Detail-Seite, usw.). Genau hier kommt `Sinatra::Base` ins Spiel:

```
require 'sinatra/base'

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

  get '/' do
    'Hallo Welt!'
  end
end
```

Die MyApp-Klasse ist eine unabhängige Rack-Komponente, die als Middleware, Endpunkt oder via Rails Metal verwendet werden kann. Verwendet wird sie durch `use` oder `run` von einer Rackup-`config.ru`-Datei oder als Server-Komponente einer Bibliothek:

```
MyApp.run! :host => 'localhost', :port => 9090
```

Die Methoden der `Sinatra::Base`-Subklasse sind genau dieselben wie die der Top-Level-DSL. Die meisten Top-Level-Anwendungen können mit nur zwei Veränderungen zu `Sinatra::Base` konvertiert werden:

- Die Datei sollte `require 'sinatra/base'` anstelle von `require 'sinatra/base'` aufrufen, ansonsten werden alle von Sinatras DSL-Methoden in den Top-Level-Namespace importiert.
- Alle Routen, Error-Handler, Filter und Optionen der Applikation müssen in einer Subklasse von `Sinatra::Base` definiert werden.

`Sinatra::Base` ist ein unbeschriebenes Blatt. Die meisten Optionen sind per Standard deaktiviert. Das

betrifft auch den eingebauten Server. Siehe Optionen und Konfiguration für Details über mögliche Optionen.

Modularer vs. klassischer Stil

Entgegen häufiger Meinungen gibt es nichts gegen den klassischen Stil einzuwenden. Solange es die Applikation nicht beeinträchtigt, besteht kein Grund, eine modulare Applikation zu erstellen.

Der größte Nachteil der klassischen Sinatra Anwendung gegenüber einer modularen ist die Einschränkung auf eine Sinatra Anwendung pro Ruby-Prozess. Sollen mehrere zum Einsatz kommen, muss auf den modularen Stil umgestiegen werden. Dabei ist es kein Problem klassische und modulare Anwendungen miteinander zu vermischen.

Bei einem Umstieg, sollten einige Unterschiede in den Einstellungen beachtet werden:

Szenario	Classic	Modular
app_file	Sinatra ladende Datei	Sinatra::Base subklassierende Datei
run	$0 == app_file	false
logging	true	false
method_override	true	false
inline_templates	true	false
static	true	false

Eine modulare Applikation bereitstellen

Es gibt zwei übliche Wege, eine modulare Anwendung zu starten. Zum einen über `run!`:

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

class MeinApp < Sinatra::Base
  # ... Anwendungscode hierhin ...

  # starte den Server, wenn die Ruby-Datei direkt ausgeführt wird
  run! if app_file == $0
end
```

Starte mit:

```
ruby mein_app.rb
```

Oder über eine `config.ru`-Datei, die es erlaubt, einen beliebigen Rack-Handler zu verwenden:

```
# config.ru (mit rackup starten)
require './mein_app'
run MeineApp
```

Starte:

```
rackup -p 4567
```

Eine klassische Anwendung mit einer config.ru verwenden

Schreibe eine Anwendungsdatei:

```
# app.rb
require 'sinatra'

get '/' do
  'Hallo Welt!'
end
```

sowie eine dazugehörige `config.ru`-Datei:

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

Wann sollte eine config.ru-Datei verwendet werden?

Anzeichen dafür, dass eine `config.ru`-Datei gebraucht wird:

- Es soll ein anderer Rack-Handler verwendet werden (Passenger, Unicorn, Heroku, ...).
- Es gibt mehr als nur eine Subklasse von `Sinatra::Base`.
- Sinatra soll als Middleware verwendet werden, nicht als Endpunkt.

Es gibt keinen Grund, eine `config.ru`-Datei zu verwenden, nur weil eine Anwendung im modularen Stil betrieben werden soll. Ebenso wird keine Anwendung mit modularem Stil benötigt, um eine `config.ru`-Datei zu verwenden.

Sinatra als Middleware nutzen

Es ist nicht nur möglich, andere Rack-Middleware mit Sinatra zu nutzen, es kann außerdem jede Sinatra-Anwendung selbst als Middleware vor jeden beliebigen Rack-Endpunkt gehangen werden. Bei diesem Endpunkt muss es sich nicht um eine andere Sinatra-Anwendung handeln, es kann jede andere Rack-Anwendung sein (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 wird vor Filtern ausgeführt
  use LoginScreen

  before do
    unless session['user_name']
      halt "Zugriff verweigert, bitte <a href='/login'>einloggen</a>."
    end
  end

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

Dynamische Applikationserstellung

Manche Situationen erfordern die Erstellung neuer Applikationen zur Laufzeit, ohne dass sie einer Konstanten zugeordnet werden. Dies lässt sich mit `Sinatra.new` erreichen:

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

Die Applikation kann mit Hilfe eines optionalen Parameters erstellt werden:

```
# config.ru
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
```

Das ist besonders dann interessant, wenn Sinatra-Erweiterungen getestet werden oder Sinatra in

einer Bibliothek Verwendung findet.

Ebenso lassen sich damit hervorragend Sinatra-Middlewares erstellen:

```
require 'sinatra/base'

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

run RailsProject::Application
```

Geltungsbereich und Bindung

Der Geltungsbereich (Scope) legt fest, welche Methoden und Variablen zur Verfügung stehen.

Anwendungs- oder Klassen-Scope

Jede Sinatra-Anwendung entspricht einer `Sinatra::Base`-Subklasse. Falls die Top- Level-DSL verwendet wird (`require 'sinatra'`), handelt es sich um `Sinatra::Application`, andernfalls ist es jene Subklasse, die explizit angelegt wurde. Auf Klassenebene stehen Methoden wie `get` oder `before` zur Verfügung, es gibt aber keinen Zugriff auf das `request`-Object oder die `session`, da nur eine einzige Klasse für alle eingehenden Anfragen genutzt wird.

Optionen, die via `set` gesetzt werden, sind Methoden auf Klassenebene:

```
class MyApp < Sinatra::Base
  # Hey, ich bin im Anwendungsscope!
  set :foo, 42
  foo # => 42

  get '/foo' do
    # Hey, ich bin nicht mehr im Anwendungs-Scope!
  end
end
```

Im Anwendungs-Scope befindet man sich:

- In der Anwendungs-Klasse.
- In Methoden, die von Erweiterungen definiert werden.
- Im Block, der an `helpers` übergeben wird.
- In Procs und Blöcken, die an `set` übergeben werden.
- Der an `Sinatra.new` übergebene Block

Auf das Scope-Objekt (die Klasse) kann wie folgt zugegriffen werden:

- Über das Objekt, das an den `configure`-Block übergeben wird (`configure { |c| ... }`).

`settings` aus den anderen Scopes heraus.

Anfrage- oder Instanz-Scope

Für jede eingehende Anfrage wird eine neue Instanz der Anwendungs-Klasse erstellt und alle Handler in diesem Scope ausgeführt. Aus diesem Scope heraus kann auf `request` oder `session` zugegriffen und Methoden wie `erb` oder `haml` aufgerufen werden. Außerdem kann mit der `settings`-Method auf den Anwendungs-Scope zugegriffen werden:

```
class MyApp < Sinatra::Base
  # Hey, ich bin im Anwendungs-Scope!
  get '/neue_route/:name' do
    # Anfrage-Scope für '/neue_route/:name'
    @value = 42

    settings.get "/#{params[:name]}" do
      # Anfrage-Scope für "/#{params[:name]}"
      @value # => nil (nicht dieselbe Anfrage)
    end

    "Route definiert!"
  end
end
```

Im Anfrage-Scope befindet man sich:

- In get/head/post/put/delete-Blöcken
- In before/after-Filtern
- In Helfer-Methoden
- In Templates

Delegation-Scope

Vom Delegation-Scope aus werden Methoden einfach an den Klassen-Scope weitergeleitet. Dieser verhält sich jedoch nicht 100%ig wie der Klassen-Scope, da man nicht die Bindung der Klasse besitzt: Nur Methoden, die explizit als delegierbar markiert wurden, stehen hier zur Verfügung und es kann nicht auf die Variablen des Klassenscopes zugegriffen werden (mit anderen Worten: es gibt ein anderes `self`). Weitere Delegationen können mit `Sinatra::Delegator.delegate :methoden_name` hinzugefügt werden.

Im Delegation-Scop befindet man sich:

- Im Top-Level, wenn `require 'sinatra'` aufgerufen wurde.
- In einem Objekt, das mit dem `Sinatra::Delegator`-Mixin erweitert wurde.

Schau am besten im Code nach: Hier ist Sinatra::Delegator mixin definiert und wird in den [globalen Namespace eingebunden](http://github.com/sinatra/sinatra/blob/master/lib/sinatra/main.rb

Kommandozeile

Sinatra-Anwendungen können direkt von der Kommandozeile aus gestartet werden:

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

Die Optionen sind:

```
-h # Hilfe
-p # Port setzen (Standard ist 4567)
-h # Host setzen (Standard ist 0.0.0.0)
-e # Umgebung setzen (Standard ist development)
-s # Rack-Server/Handler setzen (Standard ist thin)
-x # Mutex-Lock einschalten (Standard ist off)
```

Systemanforderungen

Die folgenden Versionen werden offiziell unterstützt:

Ruby 1.8.7
: 1.8.7 wird vollständig unterstützt, aber solange nichts dagegen spricht, wird ein Update auf 1.9.2 oder ein Umstieg auf JRuby/Rubinius empfohlen. Unterstützung für 1.8.7 wird es mindestens bis Sinatra 2.0 und Ruby 2.0 geben, es sei denn, dass der unwahrscheinliche Fall eintritt und 1.8.8 rauskommt. Doch selbst dann ist es eher wahrscheinlich, dass 1.8.7 weiterhin unterstützt wird. **Ruby 1.8.6 wird nicht mehr unterstützt.** Soll Sinatra unter 1.8.6 eingesetzt werden, muss Sinatra 1.2 verwendet werden, dass noch bis zum Release von Sinatra 1.4.0 fortgeführt wird.

Ruby 1.9.2
: 1.9.2 wird voll unterstützt und empfohlen. Version 1.9.2p0 sollte nicht verwendet werden, da unter Sinatra immer wieder Segfaults auftreten. Unterstützung wird es mindestens bis zum Release von Ruby 1.9.4/2.0 geben und das letzte Sinatra Release für 1.9 wird so lange unterstützt, wie das Ruby Core-Team 1.9 pflegt.

Ruby 1.9.3
: 1.9.3 wird vollständig unterstützt und empfohlen. Achtung, bei einem Wechsel zu 1.9.3 werden alle Sessions ungültig.

Rubinius
: Rubinius (rbx >= 1.2.4) wird offiziell unter Einbezug aller Templates unterstützt. Die kommende 2.0 Version wird ebenfalls unterstützt, samt 1.9 Modus.

JRuby
: JRuby wird offiziell unterstützt (JRuby >= 1.6.7). Probleme mit Template- Bibliotheken Dritter sind nicht bekannt. Falls JRuby zum Einsatz kommt, sollte aber darauf geachtet werden, dass ein JRuby-Rack-Handler zum Einsatz kommt – die Thin und Mongrel Web-Server werden bisher nicht unterstütz. JRubys Unterstützung für C-Erweiterungen sind zur Zeit ebenfalls experimenteller Natur, betrifft im Moment aber nur die RDiscount, Redcarpet, RedCloth und

[[Yajl]] Templates.

Weiterhin werden wir die kommende Ruby-Versionen im Auge behalten.

Die nachfolgend aufgeführten Ruby-Implementierungen werden offiziell nicht von Sinatra unterstützt, funktionieren aber normalerweise:

- Ruby Enterprise Edition
- Ältere Versionen von JRuby und Rubinius
- MacRuby, Maglev, IronRuby
- Ruby 1.9.0 und 1.9.1 (wird jedoch nicht empfohlen, s.o.)

Nicht offiziell unterstützt bedeutet, dass wenn Sachen nicht funktionieren, wir davon ausgehen, dass es nicht an Sinatra sondern an der jeweiligen Implentierung liegt.

Im Rahmen unserer CI (Kontinuierlichen Integration) wird bereits ruby-head (das kommende Ruby 2.0.0) und 1.9.4 mit eingebunden. Da noch alles im Fluss ist, kann zur Zeit für nichts garantiert werden. Es kann aber erwartet werden, dass Ruby 2.0.0p0 und 1.9.4p0 von Sinatra unterstützt werden wird.

Sinatra sollte auf jedem Betriebssystem laufen, dass den gewählten Ruby- Interpreter unterstützt.

Sinatra wird aktuell nicht unter Cardinal, SmallRuby, BleuRuby oder irgendeiner Version von Ruby vor 1.8.7 laufen.

Der neuste Stand (The Bleeding Edge)

Um auf dem neusten Stand zu bleiben, kann der Master-Branch verwendet werden. Er sollte recht stabil sein. Ebenso gibt es von Zeit zu Zeit prerelease Gems, die so installiert werden:

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

Mit Bundler

Wenn die Applikation mit der neuesten Version von Sinatra und Bundler genutzt werden soll, empfehlen wir den nachfolgenden Weg.

Soweit Bundler noch nicht installiert ist:

```
gem install bundler
```

Anschließend wird eine `Gemfile`-Datei im Projektverzeichnis mit folgendem Inhalt erstellt:

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

```
# evtl. andere Abhängigkeiten
gem 'haml'                    # z.B. wenn du Haml verwendest...
gem 'activerecord', '~> 3.0'  # ...oder ActiveRecord 3.x
```

Beachte: Hier sollten alle Abhängigkeiten eingetragen werden. Sinatras eigene, direkte Abhängigkeiten (Tilt und Rack) werden von Bundler automatisch aus dem Gemfile von Sinatra hinzugefügt.

Jetzt kannst du deine Applikation starten:

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

Eigenes Repository

Um auf dem neuesten Stand von Sinatras Code zu sein, kann eine lokale Kopie angelegt werden. Gestartet wird in der Anwendung mit dem `sinatra/lib`- Ordner im `LOAD_PATH`:

```
cd myapp
git clone git://github.com/sinatra/sinatra.git
ruby -Isinatra/lib myapp.rb
```

Alternativ kann der `sinatra/lib`-Ordner zum `LOAD_PATH` in der Anwendung hinzugefügt werden:

```
$LOAD_PATH.unshift File.dirname(__FILE__) + '/sinatra/lib'
require 'rubygems'
require 'sinatra'

get '/ueber' do
  "Ich laufe auf Version " + Sinatra::VERSION
end
```

Um Sinatra-Code von Zeit zu Zeit zu aktualisieren:

```
cd myproject/sinatra
git pull
```

Gem erstellen

Aus der eigenen lokalen Kopie kann nun auch ein globales Gem gebaut werden:

```
git clone git://github.com/sinatra/sinatra.git
cd sinatra
rake sinatra.gemspec
rake install
```

Falls Gems als Root installiert werden sollen, sollte die letzte Zeile folgendermaßen lauten:

```
sudo rake install
```

Versions-Verfahren

Sinatra folgt dem sogenannten Semantic Versioning, d.h. SemVer und SemVerTag.

Mehr

- Projekt-Website - Ergänzende Dokumentation, News und Links zu anderen Ressourcen.
- Mitmachen - Einen Fehler gefunden? Brauchst du Hilfe? Hast du einen Patch?
- Issue-Tracker
- Twitter
- Mailing-Liste
- #sinatra auf http://freenode.net
- Sinatra Book Kochbuch Tutorial
- Sinatra Recipes Sinatra-Rezepte aus der Community
- API Dokumentation für die aktuelle Version oder für HEAD auf http://rubydoc.info
- CI Server

Sinatra

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

Bevezetés

Fontos megjegyzés: Ez a dokumentum csak egy fordítása az angol nyelvű változat, és lehet, hogy nem naprakész.

A Sinatra egy DSL webalkalmazások Ruby nyelven történő fejlesztéséhez, minimális energiabefektetéssel:

```
# myapp.rb
require 'sinatra'
get '/' do
  'Hello Világ!'
end
```

Telepítsd a gem-et és indítsd el az alkalmazást a következőképpen:

```
sudo gem install sinatra
ruby myapp.rb
```

Az alkalmazás elérhető lesz itt: `http://localhost:4567`

Útvonalak (routes)

A Sinatrában az útvonalat egy HTTP metódus és egy URL-re illeszkedő minta párosa alkotja. Minden egyes útvonalhoz tartozik egy blokk:

```
get '/' do
  .. megjelenítünk valamit ..
end

post '/' do
  .. létrehozunk valamit ..
end
```

```
put '/' do
  .. frissítünk valamit ..
end

delete '/' do
  .. törlünk valamit ..
end
```

Az útvonalak illeszkedését a rendszer a definiálásuk sorrendjében ellenőrzi. Sorrendben mindig az első illeszkedő útvonalhoz tartozó metódus kerül meghívásra.

Az útvonalminták tartalmazhatnak paramétereket is, melyeket a `params` hash-ből érhetünk el:

```
get '/hello/:name' do
  # illeszkedik a "GET /hello/foo" és a "GET /hello/bar" útvonalakra
  # ekkor params[:name] értéke 'foo' vagy 'bar' lesz
  "Hello #{params[:name]}"
end
```

A kulcsszavas argumentumokat (named parameters) blokk paraméterek útján is el tudod érni:

```
get '/hello/:name' do |n|
  "Hello #{n}"
end
```

Az útvonalmintákban szerepelhetnek joker paraméterek is, melyeket a `params[:splat]` tömbön keresztül tudunk elérni.

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

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

Reguláris kifejezéseket is felvehetünk az útvonalba:

```
get %r{/hello/([\w]+)} do
  "Hello, #{params[:captures]}"
end
```

Vagy blokk paramétereket:

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

Az útvonalak azonban számos egyéb illeszkedési feltétel szerint is tervezhetők, így például az user agent karakterláncot alapul véve:

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

get '/foo' do
  # illeszkedik az egyéb user agentekre
end
```

Statikus állományok

A statikus fájlok kiszolgálása a `./public` könyvtárból történik, de természetesen más könyvtárat is megadhatsz erre a célra, mégpedig a :public_folder kapcsoló beállításával:

set :public_folder, File.dirname(**FILE**) + '/static'

Fontos mgejegyezni, hogy a nyilvános könyvtár neve nem szerepel az URL-ben. A ./public/css/style.css fájl az `http://example.com/css/style.css` URL-en lesz elérhető.

Nézetek és Sablonok

A sablonfájlokat rendszerint a `./views` könyvtárba helyezzük, de itt is lehetőség nyílik egyéb könyvtár használatára:

set :views, File.dirname(**FILE**) + '/templates'

Nagyon fontos észben tartani, hogy a sablononkra mindig szimbólumokkal hivatkozunk, még akkor is, ha egyéb (ebben az esetben a :'subdir/template') könyvtárban tároljuk őket. A renderelő metódusok minden, nekik közvetlenül átadott karakterláncot megjelenítenek.

Haml sablonok

HAML sablonok rendereléséhez szükségünk lesz a haml gem-re vagy könyvtárra:

```
# Importáljuk be a haml-t az alkalmazásba
require 'haml'

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

Ez szépen lerendereli a `./views/index.haml` sablont.

A Haml kapcsolói globálisan is beállíthatók a Sinatra konfigurációi között, lásd az Options and Configurations lapot. A globális beállításokat lehetőségünk van felülírni metódus szinten is.

```
set :haml, {:format => :html5 } # az alapértelmezett Haml formátum az :xhtml

get '/' do
  haml :index, :haml_options => {:format => :html4 } # immár felülírva
end
```

Erb sablonok

Importáljuk be az erb-t az alkalmazásba

```
require 'erb'

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

Ez a `./views/index.erb` sablont fogja lerenderelni.

Builder sablonok

Szükségünk lesz a builder gem-re vagy könyvtárra a builder sablonok rendereléséhez:

Importáljuk be a builder-t az alkalmazásba

```
require 'builder'

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

Ez pedig a `./views/index.builder` állományt fogja renderelni.

Sass sablonok

Sass sablonok használatához szükség lesz a haml gem-re vagy könyvtárra:

Be kell importálni a haml, vagy a sass könyvtárat

```
require 'sass'

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

Így a `./views/stylesheet.sass` fájl máris renderelhető.

A Sass kapcsolói globálisan is beállíthatók a Sinatra konfigurációi között, lásd az Options and Configurations lapot. A globális beállításokat lehetőségünk van felülírni metódus szinten is.

```
set :sass, {:style => :compact } # az alapértelmezett Sass stílus a :nested

get '/stylesheet.css' do
  sass :stylesheet, :sass_options => {:style => :expanded } # felülírva
end
```

Beágyazott sablonok

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

Lerendereli a beágyazott sablon karakerláncát.

Változók elérése a sablonokban

A sablonok ugyanabban a kontextusban kerülnek kiértékelésre, mint az útvonal metódusok (route handlers). Az útvonal metódusokban megadott változók közvetlenül elérhetőek lesznek a sablonokban:

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

De megadhatod egy lokális változókat tartalmazó explicit hash-ben is:

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

Ezt leginkább akkor érdemes megtenni, ha partial-eket akarunk renderelni valamely más sablonból.

Fájlon belüli sablonok

Sablonokat úgy is megadhatunk, hogy egyszerűen az alkalmazás fájl végére begépeljük őket:

```
require 'rubygems'
```

```
require 'sinatra'

get '/' do
  haml :index
end

__END__

@@ layout
%html
  = yield

@@ index
%div.title Helló Világ!!!!!
```

Megjegyzés: azok a fájlon belüli sablonok, amelyek az alkalmazás fájl végére kerülnek és függnek a sinatra könyvtártól, automatikusan betöltődnek. Ha ugyanezt más alkalmazásfájlban is szeretnéd megtenni, hívd meg a `use_in_file_templates!` metódust az adott fájlban.

Kulcsszavas sablonok

Sablonokat végül a felsőszintű `template` metódussal is definiálhatunk:

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

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

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

Ha létezik “layout” nevű sablon, akkor az minden esetben meghívódik, amikor csak egy sablon renderelésre kerül. A layoutokat ki lehet kapcsolni a `:layout => false` meghívásával.

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

Helperek

Használd a felső szintű `helpers` metódust azokhoz a helper függvényekhez, amiket az útvonal metódusokban és a sablonokban akarsz használni:

```
helpers do
  def bar(name)
```

```
      "#{      }bar"
    end
  end

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

Szűrők (filters)

Az előszűrők (before filter) az adott hívás kontextusában minden egyes kérés alkalmával kiértékelődnek, így módosíthatják a kérést és a választ egyaránt. A szűrőkbe felvett példányváltozók elérhetőek lesznek az útvonalakban és a sablonokban is:

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

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

Az utószűrők az egyes kérések után, az adott kérés kontextusában kerülnek kiértékelésre, így ezek is képesek módosítani a kérést és a választ egyaránt. Az előszűrőkben és úvonalakban létrehozott példányváltozók elérhetőek lesznek az utószűrők számára:

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

Megállítás

Egy kérés szűrőben vagy útvonalban történő azonnal blokkolásához használd a következő parancsot:

halt

A megállításkor egy blokktörzset is megadhatsz …

halt ‘ez fog megjelenni a törzsben’

Vagy állítsd be a HTTP státuszt és a törzset is egyszerre …

halt 401, ‘menj innen!’

Passzolás

Az útvonalak továbbadhatják a végrehajtást egy másik útvonalnak a `pass` függvényhívással:

```
get '/guess/:who' do
  pass unless params[:who] == 'Frici'
  "Eltaláltad!"
end

get '/guess/*' do
  "Elhibáztad!"
end
```

Az útvonal blokkja azonnal kilép és átadja a vezérlést a következő illeszkedő útvonalnak. Ha nem talál megfelelő útvonalat, a Sinatra egy 404-es hibával tér vissza.

Beállítások

Csak indításkor, de minden környezetre érvényesen fusson le:

```
configure do
  ...
end
```

Csak akkor fusson le, ha a környezet (a RACK_ENV környezeti változóban) `:production`-ra van állítva:

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

Csak akkor fusson le, ha a környezet `:production` vagy `:test`:

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

Hibakezelés

A hibakezelők ugyanabban a kontextusban futnak le, mint az útvonalak és előszűrők, ezért számukra is elérhetőek mindazok a könyvtárak, amelyek az utóbbiak rendelkezésére is állnak; így például a `haml`, az `erb`, a `halt` stb.

Nem található

Amikor a `Sinatra::NotFound` kivétel fellép, vagy a válasz HTTP státuszkódja 404-es, mindig a `not_found` metódus hívódik meg.

```
not_found do
  'Sehol sem találom, amit keresel'
end
```

Hiba

Az `error` metódus hívódik meg olyankor, amikor egy útvonal, blokk vagy előszűrő kivételt vált ki. A kivétel objektum lehívható a `sinatra.error` Rack változótól:

```
error do
  'Elnézést, de valami szörnyű hiba lépett fel - ' + env['sinatra.error'].name
end
```

Egyéni hibakezelés:

```
error MyCustomError do
  'Szóval az van, hogy...' + env['sinatra.error'].message
end
```

És amikor fellép:

```
get '/' do
  raise MyCustomError, 'valami nem stimmel!'
end
```

Ez fog megjelenni:

Szóval az van, hogy... valami nem stimmel!

A Sinatra speciális `not_found` és `error` hibakezelőket használ, amikor a futtatási környezet fejlesztői módba van kapcsolva.

Mime típusok

A `send_file` metódus használatakor, vagy statikus fájlok kiszolgálásakor előfordulhat, hogy a Sinatra nem ismeri fel a fájlok mime típusát. Ilyenkor használd a +mime_type+ kapcsolót a fájlkiterjesztés bevezetéséhez:

```
mime_type :foo, 'text/foo'
```

Rack Middleware

A Sinatra egy Ruby keretrendszerek számára kifejlesztett egyszerű és szabványos interfészre, a Rack -re épül. A Rack fejlesztői szempontból egyik legérdekesebb jellemzője, hogy támogatja az úgynevezett “middleware” elnevezésű komponenseket, amelyek beékelődnek a szerver és az alkalmazás közé, így képesek megfigyelni és/vagy módosítani a HTTP kéréseket és válaszokat. Segítségükkel különféle, egységesen működő funkciókat építhetünk be rendszerünkbe.

A Sinatra keretrendszerben gyerekjáték a Rack middleware-ek behúzása a use metódus segítségével:

```
require 'sinatra'
require 'my_custom_middleware'

use Rack::Lint
use MyCustomMiddleware

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

A use metódus szemantikája megegyezik a Rack::Builder DSL-ben használt +use+ metóduséval (az említett DSL-t leginkább rackup állományokban használják). Hogy egy példát említsünk, a use metódus elfogad változókat és blokkokat egyaránt, akár kombinálva is ezeket:

```
use Rack::Auth::Basic do |username, password|
  username == 'admin' && password == 'titkos'
end
```

A Rack terjesztéssel egy csomó alap middleware komponens is érkezik, amelyekkel a naplózás, URL útvonalak megadása, autentikáció és munkamenet-kezelés könnyen megvalósítható. A Sinatra ezek közül elég sokat automatikusan felhasznál a beállításoktól függően, így ezek explicit betöltésével (+use+) nem kell bajlódnod.

Tesztelés

Sinatra teszteket bármely Rack alapú tesztelő könyvtárral vagy keretrendszerrel készíthetsz. Mi a Rack::Test könyvtárat ajánljuk:

```
require 'my_sinatra_app'
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 Világ!', last_response.body
  end

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

  def test_with_rack_env
    get '/', {}, 'HTTP_USER_AGENT' => 'Songbird'
    assert_equal "Songbird-öt használsz!", last_response.body
  end
end
```

Megjegyzés: A beépített Sinatra::Test és Sinatra::TestHarness osztályok a 0.9.2-es kiadástól kezdve elavultnak számítanak.

Sinatra::Base - Middleware-ek, könyvtárak és moduláris alkalmazások

Az alkalmazást felső szinten építeni megfelelhet mondjuk egy kisebb app esetén, ám kifejezetten károsnak bizonyulhat olyan komolyabb, újra felhasználható komponensek készítésekor, mint például egy Rack middleware, Rails metal, egyszerűbb kiszolgáló komponenssel bíró könyvtárak vagy éppen Sinatra kiterjesztések. A felső szintű DSL bepiszkítja az Objektum névteret, ráadásul kisalkalmazásokra szabott beállításokat feltételez (így például egyetlen alkalmazásfájl, `./public` és `./views` könyvtár meglétét, naplózást, kivételkezelő oldalt stb.). Itt jön a képbe a Sinatra::Base osztály:

```
require 'sinatra/base'

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

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

A MyApp osztály immár önálló Rack komponensként, mondjuk Rack middleware-ként vagy alkalmazásként, esetleg Rails metal-ként is tud működni. Közvetlenül használhatod (`use`) vagy futtathatod (`run`) az osztályodat egy rackup konfigurációs állományban (`config.ru`), vagy egy szerverkomponenst tartalmazó könyvtár vezérlésekor:

```
MyApp.run! :host => 'localhost', :port => 9090
```

A Sinatra::Base gyermekosztályaiban elérhető metódusok egyúttal a felső szintű DSL-en keresztül is hozzáférhetők. A legtöbb felső szintű alkalmazás átalakítható Sinatra::Base alapú komponensekké

két lépésben:

A `Sinatra::Base` osztály igazából egy üres lap: a legtöbb funkció alapból ki van kapcsolva, beleértve a beépített szervert is. A beállításokkal és az egyes kapcsolók hatásával az Options and Configuration lap foglalkozik.

Széljegyzet: A Sinatra felső szintű DSL-je egy egyszerű delegációs rendszerre épül. A Sinatra::Application osztály – a Sinatra::Base egy speciális osztályaként – fogadja az összes :get, :put, :post, :delete, :before, :error, :not_found, :configure és :set üzenetet, ami csak a felső szintre beérkezik. Érdemes utánanézned a kódban, miképp kerül be a Sinatra::Delegator mixin a fő névtérbe.

Parancssori lehetőségek

Sinatra alkalmazásokat közvetlenül futtathatunk:

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

Az alábbi kapcsolókat ismeri fel a rendszer:

-h # segítség -p # a port beállítása (alapértelmezés szerint ez a 4567-es) -e # a környezet beállítása (alapértelmezés szerint ez a development) -s # a rack szerver/handler beállítása (alapértelmezetten ez a thin) -x # a mutex lock bekapcsolása (alapértelmezetten ki van kapcsolva)

Fejlesztői változat

Ha a Sinatra legfrissebb, fejlesztői változatát szeretnéd használni, készíts egy helyi másolatot és indítsd az alkalmazásodat úgy, hogy a `sinatra/lib` könyvtár elérhető legyen a `LOAD_PATH`-on:

```
cd myapp
git clone git://github.com/sinatra/sinatra.git
ruby -Isinatra/lib myapp.rb
```

De hozzá is adhatod a `sinatra/lib` könyvtárat a `LOAD_PATH`-hoz az alkalmazásodban:

```
$LOAD_PATH.unshift File.dirname(__FILE__) + '/sinatra/lib'
require 'rubygems'
require 'sinatra'

get '/about' do
  "A következő változatot futtatom " + Sinatra::VERSION
end
```

A Sinatra frissítését később így végezheted el:

```
cd myproject/sinatra
git pull
```

További információk

- A projekt weboldala – Kiegészítő dokumentáció, hírek, hasznos linkek
- Közreműködés – Hibát találtál? Segítségre van szükséged? Foltot küldenél be?
- Lighthouse – Hibakövetés és kiadások
- Twitter
- Levelezőlista
- IRC: #sinatra a http://freenode.net címen

Sinatra

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

소개

주의: 이 문서는 영문판의 번역본이며 최신판 문서와 다를 수 있음.

Sinatra는 최소한의 노력으로 루비 기반 웹 애플리케이션을 신속하게 만들 수 있게 해 주는 DSL이다:

```
# myapp.rb
require 'sinatra'

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

다음과 같이 젬을 설치하고 실행한다:

```
gem install sinatra
ruby myapp.rb
```

확인: http://localhost:4567

gem install thin도 함께 실행하기를 권장하며, 그럴 경우 Sinatra는 thin을 부른다.

라우터(Routes)

Sinatra에서, 라우터(route)는 URL-매칭 패턴과 쌍을 이루는 HTTP 메서드다. 각각의 라우터는 블록과 연결된다:

```
get '/' do
  .. 무언가 보여주기(show) ..
end

post '/' do
```

```
  .. 무언가 만들기(create) ..
end

put '/' do
  .. 무언가 대체하기(replace) ..
end

patch '/' do
  .. 무언가 수정하기(modify) ..
end

delete '/' do
  .. 무언가 없애기(annihilate) ..
end

options '/' do
  .. 무언가 주기(appease) ..
end
```

라우터는 정의된 순서에 따라 매치되며 매칭된 첫 번째 라우터가 호출된다.

라우터 패턴에는 이름을 가진 매개변수가 포함될 수있으며, params 해시로 접근할 수 있다:

```
get '/hello/:name' do
  # "GET /hello/foo" 및 "GET /hello/bar"와 매치
  # params[:name]은 'foo' 또는 'bar'
  "Hello #{params[:name]}"
end
```

또한 블록 매개변수를 통하여도 이름을 가진 매개변수에 접근할 수 있다:

```
get '/hello/:name' do |n|
  "Hello #{n}"
end
```

라우터 패턴에는 스플랫(splat, 또는 와일드카드)도 포함될 수 있으며, 이럴 경우 params[:splat] 배열로 접근할 수 있다:

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

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

또는 블록 매개변수도 가능하다:

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

```
end
```

정규표현식을 이용한 라우터 매칭:

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

또는 블록 매개변수로도 가능:

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

라우터 패턴에는 선택적인(optional) 매개변수도 올 수 있다:

```
get '/posts(.:format)?' do
  # "GET /posts" 및 "GET /posts.json", "GET /posts.xml" 와 같은 어떤 확장자와도 매칭
end
```

한편, 경로 탐색 공격 방지(path traversal attack protection, 아래 참조)를 비활성화시키지 않았다면, 요청 경로는 라우터와 매칭되기 이전에 수정될 수 있다.

조건(Conditions)

라우터는 예를 들면 사용자 에이전트(user agent)와 같은 다양한 매칭 조건을 포함할 수 있다:

```
get '/foo', :agent => /Songbird (\d\.\d)[\d\/]*?/ do
  "Songbird 버전 #{        :agent  0 }을 사용하고 있군!"
end

get '/foo' do
  # songbird 브라우저가 아닌 경우 매치
end
```

그 밖에 다른 조건으로는 host_name과 provides가 있다:

```
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
```

여러분만의 조건도 쉽게 정의할 수 있다:

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

get '/win a car', :probability => 0.1 do
  "당첨 축하!"
end

get '/win a car' do
  "안타깝지 않습니다."
end
```

여러 값을 받는 조건에는 스플랫(splat)을 사용하자:

```
set(:auth) do |*roles|   # <- 이게 스플랫
  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
  "내 계정 정보"
end

get "/only/admin/", :auth => :admin do
  "관리자 외 접근불가!"
end
```

반환값(Return Values)

라우터 블록의 반환값은 HTTP 클라이언트로 전달되는 응답 본문을 결정하거나, 또는 Rack 스택에서 다음 번 미들웨어를 결정한다. 대부분의 경우, 이 반환값은 위의 예제에서 보듯 문자열이지만, 다른 값도 가능하다.

유효한 Rack 응답, Rack 본문 객체 또는 HTTP 상태 코드가 되는 어떠한 객체라도 반환할 수 있다:

이에 따라 우리는, 예를 들면, 스트리밍(streaming) 예제를 쉽게 구현할 수 있다:

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

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

이런 번거로움을 줄이기 위해 `stream` 헬퍼 메서드(아래 참조)를 사용하여 스트리밍 로직을 라우터 속에 둘 수도 있다.

커스텀 라우터 매처(Custom Route Matchers)

위에서 보듯, Sinatra에는 문자열 패턴 및 정규표현식을 이용한 라우터 매칭 지원이 내장되어 있다. 그렇지만, 그게 끝이 아니다. 여러분 만의 매처(matcher)도 쉽게 정의할 수 있다:

```
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
```

사실 위의 예제는 조금 과하게 작성된 면이 있다. 다음과 같이 표현할 수도 있다:

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

또는 네거티브 룩어헤드(negative look ahead)를 사용할 수도 있다:

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

정적 파일(Static Files)

정적 파일들은 `./public`에서 제공된다. 위치를 다른 곳으로 변경하려면 `:public_folder` 옵션을 사용하면 된다:

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

이 때 public 디렉터리명은 URL에 포함되지 않는다는 점에 유의. `./public/css/style.css` 파일은 `http://example.com/css/style.css` 로 접근할 수 있다.

`Cache-Control` 헤더 정보를 추가하려면 `:static_cache_control` 설정(아래 참조)을 사용하면 된다.

뷰 / 템플릿(Views / Templates)

각 템플릿 언어는 그들만의 고유한 렌더링 메서드를 통해 표출된다. 이들 메서드는 단순히 문자열을 반환한다.

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

이 메서드는 `views/index.erb`를 렌더한다.

템플릿 이름 대신 템플릿의 내용을 직접 전달할 수도 있다:

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

템플릿은 두 번째 인자로 옵션값의 해시를 받는다:

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

이렇게 하면 `views/post.erb` 속에 내장된 `views/index.erb`를 렌더한다. (기본값은 `views/layout.erb`이며, 이 파일이 존재할 경우에만 먹는다).

Sinatra가 이해하지 못하는 모든 옵션값들은 템플릿 엔진으로 전달될 것이다:

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

옵션값은 템플릿 언어별로 일반적으로 설정할 수도 있다:

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

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

render 메서드에서 전달된 옵션값들은 `set`을 통해 설정한 옵션값을 덮어 쓴다.

가능한 옵션값들:

locals
: 문서로 전달되는 local 목록. 파셜과 함께 사용하기 좋음. 예제: `erb "<%= foo %>", :locals => {:foo => "bar"}`

default_encoding

불확실한 경우에 사용할 문자열 인코딩. 기본값은 `settings.default_encoding`.

views
: 템플릿을 로드할 뷰 폴더. 기본값은 `settings.views`.

layout
: 레이아웃을 사용할지 여부 (`true` 또는 `false`), 만약 이 값이 심볼일 경우, 사용할 템플릿을 지정. 예제:
`erb :index, :layout => !request.xhr?`

content_type
: 템플릿이 생성하는 Content-Type, 기본값은 템플릿 언어에 의존.

scope
: 템플릿을 렌더링하는 범위. 기본값은 어플리케이션 인스턴스. 만약 이 값을 변경하면, 인스턴스 변수와 헬퍼 메서드들을 사용할 수 없게 됨.

layout_engine
: 레이아웃 렌더링에 사용할 템플릿 엔진. 레이아웃을 지원하지 않는 언어인 경우에 유용. 기본값은 템플릿에서 사용하는 엔진. 예제: `set :rdoc, :layout_engine => :erb`

템플릿은 `./views` 아래에 놓이는 것으로 가정됨. 만약 뷰 디렉터리를 다른 곳으로 두려면:

```
set :views, settings.root + '/templates'
```

꼭 알아야 할 중요한 점 한 가지는 템플릿은 언제나 심볼로 참조된다는 것이며, 템플릿이 하위 디렉터리에 위치한 경우라도 마찬가지임(그럴 경우에는 `:'subdir/template'`을 사용). 반드시 심볼이어야 하는 이유는, 만약 그렇게 하지 않으면 렌더링 메서드가 전달된 문자열을 직접 렌더하려 할 것이기 때문임.

가능한 템플릿 언어들(Available Template Languages)

일부 언어는 여러 개의 구현이 있음. 어느 구현을 사용할지 저정하려면(그리고 스레드-안전thread-safe 모드로 하려면), 먼저 require 시키기만 하면 됨:

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

Haml 템플릿

의존	haml
파일 확장자	`.haml`
예	`haml :index, :format => :html5`

Erb 템플릿

의존	erubis 또는 erb (루비 속에 포함)
파일 확장자	`.erb`, `.rhtml` 또는 `.erubis` (Erubis만 해당)
예제	`erb :index`

Builder 템플릿

의존	builder
파일 확장자	`.builder`
Example	`builder { |xml| xml.em "hi" }`

인라인 템플릿으로 블록을 받음(예제 참조).

Nokogiri 템플릿

의존	nokogiri
파일 확장자	`.nokogiri`
예제	`nokogiri { |xml| xml.em "hi" }`

인라인 템플릿으로 블록을 받음(예제 참조).

Sass 템플릿

의존	sass
파일 확장자	`.sass`
예제	`sass :stylesheet, :style => :expanded`

SCSS 템플릿

의존	sass
파일 확장자	`.scss`
예제	`scss :stylesheet, :style => :expanded`

Less 템플릿

의존	less
파일 확장자	`.less`
예제	`less :stylesheet`

Liquid 템플릿

의존	liquid
파일 확장자	`.liquid`
예제	`liquid :index, :locals => { :key => 'value' }`

Liquid 템플릿에서는 루비 메서드(`yield` 제외)를 호출할 수 없기 때문에, 거의 대부분의 경우 locals를 전달해

야 함.

Markdown 템플릿

의존	rdiscount, redcarpet, bluecloth, kramdown *또는* maruku
파일 확장	`.markdown`, `.mkd`, `.md`
예제	`markdown :index, :layout_engine => :erb`

마크다운에서는 메서드 호출 뿐 아니라 locals 전달도 안됨. 따라서 일반적으로는 다른 렌더링 엔진과 함께 사용하게 될 것임:

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

또한 다른 템플릿 속에서 `markdown` 메서드를 호출할 수도 있음:

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

Markdown에서 루비를 호출할 수 없기 때문에, Markdown으로 작성된 레이아웃은 사용할 수 없음. 단, `:layout_engine` 옵션으로 템플릿의 레이아웃은 다른 렌더링 엔진을 사용하는 것은 가능.

Textile 템플릿

의존	RedCloth
파일 확장자	`.textile`
예제	`textile :index, :layout_engine => :erb`

Textile에서 메서드를 호출하거나 locals를 전달하는 것은 불가능함. 따라서 일반적으로 다른 렌더링 엔진과 함께 사용하게 될 것임:

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

또한 다른 템플릿 속에서 `textile` 메서드를 호출할 수도 있음:

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

Textile에서 루비를 호출할 수 없기 때문에, Textile로 작성된 레이아웃은 사용할 수 없음. 단, `:layout_engine` 옵션으로 템플릿의 레이아웃은 다른 렌더링 엔진을 사용하는 것은 가능.

RDoc 템플릿

의존	rdoc

파일 확장자 `.rdoc`
예제 `rdoc :README, :layout_engine => :erb`

rdoc에서 메서드를 호출하거나 locals를 전달하는 것은 불가능함. 따라서 일반적으로 다른 렌더링 엔진과 함께 사용하게 될 것임:

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

또한 다른 템플릿 속에서 `rdoc` 메서드를 호출할 수도 있음:

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

RDoc에서 루비를 호출할 수 없기 때문에, RDoc로 작성된 레이아웃은 사용할 수 없음. 단, `:layout_engine` 옵션으로 템플릿의 레이아웃은 다른 렌더링 엔진을 사용하는 것은 가능. ### Radius 템플릿

의존 radius
파일 확장자 `.radius`
예제 `radius :index, :locals => { :key => 'value' }`

Radius 템플릿에서는 루비 메서드를 호출할 수 없기 때문에, 거의 대부분의 경우 locals로 전달하게 될 것임.

Markaby 템플릿

의존 markaby
파일확장 `.mab`
예제 `markaby { h1 "Welcome!" }`

인라인 템플릿으로 블록을 받을 수도 있음(예제 참조).

RABL 템플릿

의존 rabl
파일 확장자 `.rabl`
예제 `rabl :index`

Slim 템플릿

의존 slim
파일 확장자 `.slim`
예제 `slim :index`

Creole 템플릿

의존	creole
파일 확장자	`.creole`
예제	`creole :wiki, :layout_engine => :erb`

creole에서는 루비 메서드를 호출할 수 없고 locals도 전달할 수 없음. 따라서 일반적으로는 다른 렌더링 엔진과 함께 사용하게 될 것임.

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

또한 다른 템플릿 속에서 `creole` 메서드를 호출할 수도 있음:

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

Creole에서 루비를 호출할 수 없기 때문에, Creole로 작성된 레이아웃은 사용할 수 없음. 단, `:layout_engine` 옵션으로 템플릿의 레이아웃은 다른 렌더링 엔진을 사용하는 것은 가능.

CoffeeScript 템플릿

의존성	coffee-script 와 자바스크립트 실행법
파일 확장자	`.coffee`
예제	`coffee :index`

Yajl 템플릿

의존	yajl-ruby
파일 확장자	`.yajl`
예제	`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 #to_json. 템플릿 소스는 루비 문자열로 평가(evaluate)되고, 결과인 json 변수는 #to_json으로 변환됨.

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

`:callback`과 `:variable` 옵션은 렌더된 객체를 꾸미는데(decorate) 사용할 수 있음.

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

내장된(Embedded) 템플릿

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

내장된 템플릿 문자열을 렌더함.

템플릿에서 변수에 접근하기

Templates are evaluated within the same context as route handlers. Instance variables set in route handlers are directly accessible by templates: 템플릿은 라우터 핸들러와 같은 맥락(context)에서 평가된다. 라우터 핸들러에서 설정한 인스턴스 변수들은 템플릿에서 접근 가능하다:

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

또는, 명시적으로 로컬 변수의 해시를 지정:

```
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. 이 방법은 통상적으로 템플릿을 다른 템플릿 속에서 파셜(partial)로 렌더링할 때 사용된다.

인라인 템플릿

템플릿은 소스 파일의 마지막에서 정의할 수도 있다:

```
require 'sinatra'

get '/' do
  haml :index
end

__END__

@@ layout
%html
= yield

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

참고: require sinatra 시킨 소스 파일에 정의된 인라인 템플릿은 자동으로 로드된다. 다른 소스 파일에서 인라인 템플릿을 사용하려면 명시적으로 `enable :inline_templates`을 호출하면 됨.

이름을 가지는 템플릿(Named Templates)

템플릿은 톱 레벨(top-level)에서 `template`메서드를 사용하여 정의할 수 있다:

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

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

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

"layout"이라는 이름의 템플릿이 존재하면, 매번 템플릿이 렌더될 때마다 사용될 것이다. 이 때 `:layout => false`를 전달하여 개별적으로 레이아웃을 비활성시키거나 또는 `set :haml, :layout => false`으로 기본값을 비활성으로 둘 수 있다:

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

파일 확장자 연결하기

어떤 파일 확장자를 특정 템플릿 엔진과 연결하려면, `Tilt.register`를 사용하면 된다. 예를 들어, `tt`라는 파일 확장자를 Textile 템플릿과 연결하고 싶다면, 다음과 같이 하면 된다:

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

나만의 고유한 템플릿 엔진 추가하기

우선, Tilt로 여러분 엔진을 등록하고, 그런 다음 렌더링 메서드를 생성하자:

```
Tilt.register :myat, MyAwesomeTemplateEngine

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

get '/' do
```

```
  myat :index
end
```

`./views/index.myat` 를 렌더함. Tilt에 대한 더 자세한 내용은 https://github.com/rtomayko/tilt 참조.

필터(Filters)

사전 필터(before filter)는 라우터와 동일한 맥락에서 매 요청 전에 평가되며 요청과 응답을 변형할 수 있다. 필터에서 설정된 인스턴스 변수들은 라우터와 템플릿 속에서 접근 가능하다:

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

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

사후 필터(after filter)는 라우터와 동일한 맥락에서 매 요청 이후에 평가되며 마찬가지로 요청과 응답을 변형할 수 있다. 사전 필터와 라우터에서 설정된 인스턴스 변수들은 사후 필터에서 접근 가능하다:

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

참고: 만약 라우터에서 body 메서드를 사용하지 않고 그냥 문자열만 반환한 경우라면, body는 나중에 생성되는 탓에, 아직 사후 필터에서 사용할 수 없을 것이다.

필터는 선택적으로 패턴을 취할 수 있으며, 이 경우 요청 경로가 그 패턴과 매치할 경우에만 필터가 평가될 것이다.

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

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

라우터와 마찬가지로, 필터 역시 조건을 갖는다:

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

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

헬퍼(Helpers)

톱-레벨의 `helpers` 메서드를 사용하여 라우터 핸들러와 템플릿에서 사용할 헬퍼 메서드들을 정의할 수 있다:

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

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

또는, 헬퍼 메서드는 별도의 모듈 속에 정의할 수도 있다:

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

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

helpers FooUtils, BarUtils
```

이 경우 모듈을 애플리케이션 클래스에 포함(include)시킨 것과 동일한 효과를 갖는다.

세션(Sessions) 사용하기

세션은 요청 동안에 상태를 유지하기 위해 사용한다. 세션이 활성화되면, 사용자 세션 당 session 해시 하나씩을 갖게 된다:

```
enable :sessions

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

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

`enable :sessions`은 실은 모든 데이터를 쿠키 속에 저장함에 유의하자. 항상 이렇게 하고 싶지 않을 수도 있을 것이다(예를 들어, 많은 양의 데이터를 저장하게 되면 트래픽이 높아진다). 이 때는 여러 가지 랙 세션 미들웨어(Rack session middleware)를 사용할 수 있을 것이다: 이렇게 할 경우라면, `enable :sessions`을 호출하지 말고, 대신 여러분이 선택한 미들웨어를 다른 모든 미들웨어들처럼 포함시키면 된다:

```
use Rack::Session::Pool, :expire_after => 2592000

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

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

보안을 위해서, 쿠키 속의 세션 데이터는 세션 시크릿(secret)으로 사인(sign)된다. Sinatra는 여러분을 위해 무작위 시크릿을 생성한다. 그렇지만, 이 시크릿은 여러분 애플리케이션 시작 시마다 변경될 수 있기 때문에, 여러분은 여러분 애플리케이션의 모든 인스턴스들이 공유할 시크릿을 직접 만들고 싶을 수도 있다:

```
set :session_secret, 'super secret'
```

조금 더 세부적인 설정이 필요하다면, sessions 설정에서 옵션이 있는 해시를 저장할 수도 있을 것이다:

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

중단하기(Halting)

필터나 라우터에서 요청을 즉각 중단하고 싶을 때 사용하라:

```
halt
```

중단할 때 상태를 지정할 수도 있다:

```
halt 410
```

또는 본문을 넣을 수도 있다:

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

또는 둘 다도 가능하다:

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

헤더를 추가할 경우에는 다음과 같이 하면 된다:

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

물론 halt를 템플릿과 결합하는 것도 가능하다:

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

넘기기(Passing)

라우터는 pass를 사용하여 다음 번 매칭되는 라우터로 처리를 넘길 수 있다:

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

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

이 때 라우터 블록에서 즉각 빠져나오게 되고 제어는 다음 번 매칭되는 라우터로 넘어간다. 만약 매칭되는 라우터를 찾지 못하면, 404가 반환된다.

다른 라우터 부르기(Triggering Another Route)

경우에 따라서는 pass가 아니라, 다른 라우터를 호출한 결과를 얻고 싶은 경우도 있을 것이다. 이 때는 간단하게 +call+을 사용하면 된다:

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

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

위 예제의 경우, "bar"를 헬퍼로 옮겨 /foo와 /bar 모두에서 사용하도록 함으로써 테스팅을 쉽게 하고 성능을 높일 수 있을 것이다.

만약 그 요청이 사본이 아닌 바로 그 동일 인스턴스로 보내지도록 하고 싶다면, call 대신 call!을 사용하면 된다.

call에 대한 더 자세한 내용은 Rack 명세를 참고하면 된다.

본문, 상태 코드 및 헤더 설정하기

라우터 블록의 반환값과 함께 상태 코드(status code)와 응답 본문(response body)을 설정하는 것은 가능하기도 하거니와 권장되는 방법이다. 그렇지만, 경우에 따라서는 본문을 실행 흐름 중의 임의 지점에서 설정하고 싶을 수도 있다. 이 때는 body 헬퍼 메서드를 사용하면 된다. 이렇게 하면, 그 순간부터 본문에 접근할 때 그 메서드를 사용할 수가 있다:

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

after do
  puts body
end
```

body로 블록을 전달하는 것도 가능하며, 이 블록은 랙(Rack) 핸들러에 의해 실행될 것이다. (이 방법은 스트리밍을 구현할 때 사용할 수 있는데, "값 반환하기"를 참고).

본문와 마찬가지로, 상태코드와 헤더도 설정할 수 있다:

```
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
```

body처럼, header와 status도 매개변수 없이 사용하여 그것의 현재 값을 액세스하는 데 사용될 수 있다.

응답 스트리밍(Streaming Responses)

응답 본문의 일정 부분을 계속 생성하는 가운데 데이터를 내보내기 시작하고 싶을 경우도 있을 것이다. 극단적인 예제로, 클라이언트가 접속을 끊기 전까지 계속 데이터를 내보내고 싶을 수도 있다. 여러분만의 래퍼(wrapper)를 만들기 싫다면 stream 헬퍼를 사용하면 된다:

```
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
```

이렇게 하면 스트리밍 API나 서버 발송 이벤트Server Sent Events를 구현할 수 있게 해 주며, WebSockets을 위한 기반으로 사용될 수 있다. 또한 이 방법은 일부 콘텐츠가 느린 자원에 의존하는 경우에 스로풋(throughtput)을 높이기 위해 사용될 수도 있다.

스트리밍 동작, 특히 동시 요청의 수는 애플리케이션을 서빙하는 웹서버에 크게 의존적이다. 어떤 서버, 예컨대 WEBRick 같은 경우는 아예 스트리밍을 지원조차 하지 못할 것이다. 만약 서버가 스트리밍을 지원하지 않는다면, 본문은 stream 으로 전달된 블록이 수행을 마친 후에 한꺼번에 반환될 것이다. 스트리밍은 Shotgun에서는 작동하지 않는다.

만약 선택적 매개변수 keep_open이 설정되어 있다면, 스트림 객체에서 close를 호출하지 않을 것이고, 따라서 여러분은 나중에 실행 흐름 상의 어느 시점에서 스트림을 닫을 수 있다. 이 옵션은 Thin과 Rainbow 같은 이벤

트 기반 서버에서만 작동한다. 다른 서버들은 여전히 스트림을 닫을 것이다:

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

get '/' do
  # 스트림을 열린 채 유지
  stream(:keep_open) { |out| connections << out }
end

post '/' do
  # 모든 열린 스트림에 쓰기
  connections.each { |out| out << params[:message] << "\n" }
  "message sent"
end
```

로깅(Logging)

In the request scope, the `logger` helper exposes a `Logger` instance: 요청 스코프(request scope) 내에서, `Logger`의 인스턴스인 `logger` 헬퍼를 사용할 수 있다:

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

이 로거는 여러분이 Rack 핸들러에서 설정한 로그 셋팅을 자동으로 참고한다. 만약 로깅이 비활성이라면, 이 메서드는 더미(dummy) 객체를 반환할 것이며, 따라서 여러분은 라우터나 필터에서 이 부분에 대해 걱정할 필요는 없다.

로깅은 `Sinatra::Application`에서만 기본으로 활성화되어 있음에 유의하자. 만약 `Sinatra::Base`로부터 상속받은 경우라면 직접 활성화시켜 줘야 한다:

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

어떠한 로깅 미들웨어도 설정되지 않게 하려면, `logging` 설정을 `nil`로 두면 된다. 그렇지만, 이럴 경우 `logger`는 `nil`을 반환할 것임에 유의하자. 통상적인 유스케이스는 여러분만의 로거를 사용하고자 할 경우일 것이다. Sinatra는 `env['rack.logger']`에서 찾은 것을 사용할 것이다.

마임 타입(Mime Types)

`send_file`이나 정적인 파일을 사용할 때에 Sinatra가 인식하지 못하는 마임 타입이 있을 수 있다. 이 경우 `mime_type`을 사용하여 파일 확장자를 등록하면 된다:

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

또는 content_type 헬퍼와 함께 사용할 수도 있다:

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

URL 생성하기

URL을 생성하려면 url 헬퍼 메서드를 사용해야 한다. 예를 들어 Haml에서:

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

이것은 리버스 프록시(reverse proxies)와 Rack 라우터를, 만약 존재한다면, 참고한다.

This method is also aliased to to (see below for an example). 이 메서드는 to라는 별칭으로도 사용할 수 있다 (아래 예제 참조).

브라우저 재지정(Browser Redirect)

redirect 헬퍼 메서드를 사용하여 브라우저 리다이렉트를 촉발시킬 수 있다:

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

여타 부가적인 매개변수들은 halt에서 전달한 인자들처럼 다루어 진다:

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

redirect back을 사용하면 사용자가 왔던 페이지로 다시 돌아가는 리다이렉트도 쉽게 할 수 있다:

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

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

리다이렉트와 함께 인자를 전달하려면, 쿼리에 붙이거나:

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

또는 세션을 사용하면 된다:

```
enable :sessions

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

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

캐시 컨트롤(Cache Control)

헤더를 정확하게 설정하는 것은 적절한 HTTP 캐싱의 기본이다.

Cache-Control 헤더를 다음과 같이 간단하게 설정할 수 있다:

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

프로 팁: 캐싱은 사전 필터에서 설정하라:

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

expires 헬퍼를 사용하여 그에 상응하는 헤더를 설정한다면, Cache-Control이 자동으로 설정될 것이다:

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

캐시를 잘 사용하려면, etag 또는 last_modified의 사용을 고려해야 할 것이다. 무거운 작업을 하기 전에 이들 헬퍼를 호출할 것을 권장하는데, 이러면 만약 클라이언트 캐시에 현재 버전이 이미 들어 있을 경우엔 즉각 응답을 반환(flush)하게 될 것이다:

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

```
end
```

약한 ETag를 사용하는 것도 가능하다:

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

이들 헬퍼는 어떠한 캐싱도 하지 않으며, 대신 필요한 정보를 캐시에 제공한다. 여러분이 만약 손쉬운 리버스 프록시(reverse-proxy) 캐싱 솔루션을 찾고 있다면, rack-cache를 써보라:

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

use Rack::Cache

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

정적 파일에 Cache-Control 헤더 정보를 추가하려면 :static_cache_control 설정(아래 참조)을 사용하라:

RFC 2616에 따르면 If-Match 또는 If-None-Match 헤더가 *로 설정된 경우 요청한 리소스(resource)가 이미 존재하느냐 여부에 따라 다르게 취급해야 한다고 되어 있다. Sinatra는 (get 처럼) 안전하거나 (put 처럼) 멱등인 요청에 대한 리소스는 이미 존재한다고 가정하며, 반면 다른 리소스(예를 들면 post 요청 같은)의 경우는 새 리소스로 취급한다. 이런 설정은 :new_resource 옵션으로 전달하여 변경할 수 있다:

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

여전히 약한 ETag를 사용하고자 한다면, :kind으로 전달하자:

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

파일 전송하기(Sending Files)

파일을 전송하려면, send_file 헬퍼 메서드를 사용하면 된다:

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

이 메서드는 몇 가지 옵션을 받는다:

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

옵션들:

filename
: 응답에서의 파일명. 기본값은 실제 파일명이다.

last_modified
: Last-Modified 헤더값. 기본값은 파일의 mtime.

type
: 사용할 컨텐츠 유형. 없으면 파일 확장자로부터 유추된다.

disposition
: Content-Disposition에서 사용됨. 가능한 값들: `nil` (기본값), `:attachment` 및 `:inline`

length
: Content-Length, 기본값은 파일 크기.

status
: 전송할 상태 코드. 오류 페이지로 정적 파일을 전송할 경우에 유용.

Rack 핸들러가 지원할 경우, Ruby 프로세스로부터의 스트리밍이 아닌 다른 수단을 사용할 수 있다. 만약 이 헬퍼 메서드를 사용하게 되면, Sinatra는 자동으로 범위 요청(range request)을 처리할 것이다.

요청 객체에 접근하기(Accessing the Request Object)

인입되는 요청 객에는 요청 레벨(필터, 라우터, 오류 핸들러)에서 `request` 메서드를 통해 접근 가능하다:

```
# http://example.com/example 상에서 실행 중인 앱
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                # 클라이언트로부터 전송된 요청 본문 (아래 참조)
  request.scheme              # "http"
  request.script_name         # "/example"
  request.path_info           # "/foo"
  request.port                # 80
  request.request_method      # "GET"
  request.query_string        # ""
  request.content_length      # request.body의 길이
  request.media_type          # request.body의 미디어 유형
  request.host                # "example.com"
  request.get?                # true (다른 동사에 대해 유사한 메서드 있음)
  request.form_data?          # false
  request["SOME_HEADER"]      # SOME_HEADER 헤더의 값
  request.referrer            # 클라이언트의 리퍼러 또는 '/'
  request.user_agent          # 사용자 에이전트 (:agent 조건에서 사용됨)
  request.cookies             # 브라우저 쿠키의 해시
  request.xhr?                # 이게 ajax 요청인가요?
  request.url                 # "http://example.com/example/foo"
  request.path                # "/example/foo"
  request.ip                  # 클라이언트 IP 주소
  request.secure?             # false (ssl 접속인 경우 true)
  request.forwarded?          # true (리버스 프록시 하에서 작동 중이라면)
  request.env                 # Rack에 의해 처리되는 로우(raw) env 해시
end
```

일부 옵션들, script_name 또는 path_info와 같은 일부 옵션은 쓸 수도 있다:

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

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

request.body는 IO 또는 StringIO 객체이다:

```
post "/api" do
  request.body.rewind  # 누군가 이미 읽은 경우
  data = JSON.parse request.body.read
  "Hello #{data['name']}"
end
```

첨부(Attachments)

attachment 헬퍼를 사용하여 브라우저에게 응답이 브라우저에 표시되는 게 아니라 디스크에 저장되어야 함을 알릴 수 있다:

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

이 때 파일명을 전달할 수도 있다:

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

날짜와 시간 다루기

Sinatra는 time_for_ 헬퍼 메서드를 제공하는데, 이 메서드는 주어진 값으로부터 Time 객체를 생성한다. DateTime 이나 Date 또는 유사한 클래스들도 변환 가능하다:

```
get '/' do
  pass if Time.now > time_for('Dec 23, 2012')
  "still time"
end
```

이 메서드는 내부적으로 expires 나 last_modified 같은 곳에서 사용된다. 따라서 여러분은 애플리케이션에서 time_for를 오버라이딩하여 이들 메서드의 동작을 쉽게 확장할 수 있다:

```
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
```

템플릿 파일 참조하기

find_template는 렌더링할 템플릿 파일을 찾는데 사용된다:

```
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: 이건 별로 유용하지 않다. 그렇지만 이 메서드를 오버라이드하여 여러분만의 참조 메커니즘에서 가로채는 것은 유용하다. 예를 들어, 하나 이상의 뷰 디렉터리를 사용하고자 한다면:

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

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

또다른 예제는 각각의 엔진마다 다른 디렉터리를 사용할 경우다:

```
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
```

여러분은 이것을 간단하게 확장(extension)으로 만들어 다른 사람들과 공유할 수 있다!

은 그 파일이 실제 존재하는지 검사하지 않음에 유의하자 대신 모든 가능한 경로에 대해 주어진

`find_template` .
블록을 호출할 뿐이다. 이것은 성능 문제는 아닌 것이, `render`는 파일이 발견되는 즉시 `break`를 사용할 것이기 때문이다. 또한, 템플릿 위치(그리고 콘텐츠)는 개발 모드에서 실행 중이 아니라면 캐시될 것이다. 정말로 멋진 메세드를 작성하고 싶다면 이 점을 명심하자.

설정(Configuration)

모든 환경에서, 시작될 때, 한번만 실행:

```
configure do
  # 옵션 하나 설정
  set :option, 'value'

  # 여러 옵션 설정
  set :a => 1, :b => 2

  # `set :option, true`와 동일
  enable :option

  # `set :option, false`와 동일
  disable :option

  # 블록으로 동적인 설정을 할 수도 있음
  set(:css_dir) { File.join(views, 'css') }
end
```

환경(RACK_ENV 환경 변수)이 `:production`일 때만 실행:

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

환경이 `:production` 또는 `:test`일 때 실행:

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

이들 옵션은 `settings`를 통해 접근 가능하다:

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

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

공격 방어 설정하기(Configuring attack protection)

Sinatra는 Rack::Protection을 사용하여 일반적인, 일어날 수 있는 공격에 대비한다. 이 부분은 간단하게 비활성시킬 수 있다(성능 향상 효과를 가져올 것이다):

```
disable :protection
```

하나의 방어층만 스킵하려면, 옵션 해시에 `protection`을 설정하면 된다:

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

방어막 여러 개를 비활성하려면, 배열로 주면 된다:

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

가능한 설정들(Available Settings)

absolute_redirects
: 만약 비활성이면, Sinatra는 상대경로 리다이렉트를 허용할 것이지만, 이렇게 되면 Sinatra는 더 이상 오직 절대경로 리다이렉트만 허용하고 있는 RFC 2616(HTTP 1.1)에 위배될 것이다. 적정하게 설정되지 않은 리버스 프록시 하에서 앱을 실행 중이라면 활성화시킬 것. `rul` 헬퍼는, 만약 두 번째 매개변수로 `false`를 전달하지만 않는다면, 여전히 절대경로 URL을 생성할 것임에 유의하자. 기본값은 비활성.

add_charsets
: `content_type`가 문자셋 정보에 자동으로 추가하게 될 마임(mime) 타입. 이 옵션은 오버라이딩하지 말고 추가해야 한다: `settings.add_charsets << "application/foobar"`

app_file
: 메인 애플리케이션 파일의 경로. 프로젝트 루트와 뷰, 그리고 public 폴더, 인라인 템플릿을 파악할 때 사용됨.

bind
: 바인드할 IP 주소(기본값: 0.0.0.0). 오직 빌트인(built-in) 서버에서만 사용됨.

default_encoding
: 모를 때 가정할 인코딩 (기본값은 `"utf-8"`).

dump_errors
: 로그로 에러 출력.

environment
: 현재 환경, 기본값은 `ENV['RACK_ENV']` 또는 알 수 없을 경우 "development".

logging
: 로거(logger) 사용.

lock
: 매 요청에 걸쳐 잠금(lock)을 설정. Ruby 프로세스 당 요청을 동시에 할 경우. 앱이 스레드 안전(thread-safe)이 아니라면 활성화시킬 것. 기본값은 비활성.

method_override
: put/delete를 지원하지 않는 브라우저에서 put/delete 폼을 허용하는 `_method` 꼼수 사용.

port
: 접속 포트 빌트인 서버에서만 사용됨

. .

prefixed_redirects
: 절대경로가 주어지지 않은 리다이렉트에 request.script_name를 삽입할지 여부. 이렇게 하면 redirect '/foo'는 redirect to('/foo') 처럼 동작. 기본값은 비활성.

protection
: 웹 공격 방어를 활성화시킬 건지 여부. 위의 보안 섹션 참조.

public_folder
: public 파일이 제공될 폴더의 경로. static 파일 제공이 활성화된 경우만 사용됨(아래 static참조). 만약 설정이 없으면 app_file로부터 유추됨.

reload_templates
: 요청 간에 템플릿을 리로드(reload)할 건지 여부. 개발 모드에서는 활성됨.

root
: 프로젝트 루트 디렉터리 경로. 설정이 없으면 app_file 설정으로부터 유추됨.

raise_errors
: 예외 발생(애플리케이션은 중단됨). 기본값은 environment가 "test"인 경우는 활성, 그렇지 않으면 비활성.

run
: 활성화되면, Sinatra가 웹서버의 시작을 핸들링. rackup 또는 다른 도구를 사용하는 경우라면 활성화시키지 말 것.

running
: 빌트인 서버가 실행 중인지? 이 설정은 변경하지 말 것!

server
: 빌트인 서버로 사용할 서버 또는 서버 목록. 기본값은 ['thin', 'mongrel', 'webrick']이며 순서는 우선순위를 의미.

sessions
: Rack::Session::Cookie를 사용한 쿠키 기반 세션 활성화. 보다 자세한 정보는 '세션 사용하기' 참조.

show_exceptions
: 예외 발생 시에 브라우저에 스택 추적을 보임. 기본값은 environment가 "development"인 경우는 활성, 나머지는 비활성.

static
: Sinatra가 정적(static) 파일을 핸들링할 지 여부. 이 기능을 수행하는 서버를 사용하는 경우라면 비활성시킬 것. 비활성시키면 성능이 올라감. 기본값은 전통적 방식에서는 활성, 모듈 앱에서는 비활성.

static_cache_control
: Sinatra가 정적 파일을 제공하는 경우, 응답에 Cache-Control 헤더를 추가할 때 설정. cache_control 헬퍼를 사용. 기본값은 비활성. 여러 값을 설정할 경우는 명시적으로 배열을 사용할 것: set :static_cache_control, [:public, :max_age => 300]

threaded
: true로 설정하면, Thin이 요청을 처리하는데 있어 EventMachine.defer를 사용하도록 함.

views
: 뷰 폴더 경로. 설정하지 않은 경우 app_file로부터 유추됨.

환경(Environments)

환경은 RACK_ENV 환경 변수를 통해서도 설정할 수 있다. 기본값은 “development”다. 이 모드에서, 모든 템플릿들은 요청 간에 리로드된다. 특별한 not_found 와 error 핸들러가 이 환경에 설치되기 때문에 브라우저에서 스택 추적을 볼 수 있을 것이다. "production"과 "test"에서는 템플릿은 캐시되는 게 기본값이다.

다른 환경으로 실행시키려면 -e옵션을 사용하면 된다:

```
ruby my_app.rb -e [ENVIRONMENT]
```

현재 설정된 환경이 무엇인지 검사하기 위해 사전 정의된 development?, test? 및 production? 메서드를 사용할 수 있다.

예외 처리(Error Handling)

예외 핸들러는 라우터 및 사전 필터와 동일한 맥락에서 실행된다. 이 말인즉, 이들이 제공하는 모든 것들을 사용할 수 있다는 말이다. 예를 들면 haml, erb, halt, 등등.

찾을 수 없음(Not Found)

Sinatra::NotFound 예외가 발생하거나 또는 응답의 상태 코드가 404라면, not_found 핸들러가 호출된다:

```
not_found do
  '[redacted]'
end
```

오류(Error)

error 핸들러는 라우터 또는 필터에서 뭐든 오류가 발생할 경우에 호출된다. 예외 객체는 Rack 변수 sinatra.error로부터 얻을 수 있다:

```
error do
  '[redacted]' + env['sinatra.error'].name
end
```

사용자 정의 오류:

```
error MyCustomError do
  '[redacted]' + env['sinatra.error'].message
end
```

그런 다음, 이 오류가 발생하면:

```
get '/' do
  raise MyCustomError, '[redacted]'
end
```

다음을 얻는다:

```
무슨 일이 생겼냐면요... 안좋은 일
```

또는, 상태 코드에 대해 오류 핸들러를 설치할 수 있다:

```
error 403 do
  'Access forbidden'
end

get '/secret' do
  403
end
```

Or a range:

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

Sinatra는 개발 환경에서 동작할 경우에 특별한 not_found 와 error 핸들러를 설치한다.

Rack 미들웨어(Rack Middleware)

Sinatra는 Rack 위에서 동작하며, Rack은 루비 웹 프레임워크를 위한 최소한의 표준 인터페이스이다. Rack이 애플리케이션 개발자들에게 제공하는 가장 흥미로운 기능 중 하나가 바로 "미들웨어(middleware)"에 대한 지원이며, 여기서 미들웨어란 서버와 여러분의 애플리케이션 사이에 위치하면서 HTTP 요청/응답을 모니터링하거나/또는 조작함으로써 다양한 유형의 공통 기능을 제공하는 컴포넌트(component)다.

Sinatra는 톱레벨의 use 메서드를 사용하여 Rack 미들웨어의 파이프라인을 만드는 일을 식은 죽 먹기로 만든다:

```
require 'sinatra'
require 'my_custom_middleware'

use Rack::Lint
use MyCustomMiddleware

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

use의 의미는 Rack::Builder DSL (rackup 파일에서 가장 많이 사용된다)에서 정의한 것들과 동일하다. 예를 들어, use 메서드는 블록 뿐 아니라 여러 개의/가변적인 인자도 받는다:

```
use Rack::Auth::Basic do |username, password|
  username == 'admin' && password == 'secret'
end
```

Rack은 로깅, 디버깅, URL 라우팅, 인증, 그리고 세션 핸들링을 위한 다양한 표준 미들웨어로 분산되어 있다.

Sinatra는 설정에 기반하여 이들 컴포넌트들 중 많은 것들을 자동으로 사용하며, 따라서 여러분은 일반적으로는 use를 명시적으로 사용할 필요가 없을 것이다.

유용한 미들웨어들은 rack, rack-contrib, CodeRack 또는 Rack wiki 에서 찾을 수 있다.

테스팅(Testing)

Sinatra 테스트는 Rack 기반 어떠한 테스팅 라이브러리 또는 프레임워크를 사용하여도 작성할 수 있다. Rack::Test를 권장한다:

```
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
```

Sinatra::Base - 미들웨어(Middleware), 라이브러리(Libraries), 그리고 모듈 앱(Modular Apps)

톱레벨에서 앱을 정의하는 것은 마이크로 앱(micro-app) 수준에서는 잘 동작하지만, Rack 미들웨어나, Rails 메탈(metal) 또는 서버 컴포넌트를 갖는 간단한 라이브러리, 또는 더 나아가 Sinatra 익스텐션(extension) 같은 재사용 가능한 컴포넌트들을 구축할 경우에는 심각한 약점을 가진다. 톱레벨은 마이크로 앱 스타일의 설정을 가정한다(즉, 하나의 단일 애플리케이션 파일과 `./public` 및 `./views` 디렉터리, 로깅, 예외 상세 페이지 등등). 이게 바로 `Sinatra::Base`가 필요한 부분이다:

```
require 'sinatra/base'

class MyApp < Sinatra::Base
```

```
  set :sessions, true
  set :foo, 'bar'

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

Sinatra::Base 서브클래스에서 사용가능한 메서드들은 톱레벨 DSL로 접근 가능한 것들과 동일하다. 대부분의 톱레벨 앱들이 다음 두 가지만 수정하면 Sinatra::Base 컴포넌트로 변환 가능하다:

- 파일은 sinatra가 아닌 sinatra/base를 require해야 하며, 그렇지 않으면 모든 Sinatra의 DSL 메서드들이 메인 네임스페이스에 불러지게 된다.
- 앱의 라우터, 예외 핸들러, 필터, 그리고 옵션들을 Sinatra::Base의 서브클래스에 둘 것.

Sinatra::Base는 빈서판(blank slate)이다. 빌트인 서버를 비롯한 대부분의 옵션들이 기본값으로 꺼져 있다. 가능한 옵션들과 그 작동에 대한 상세는 Options and Configuration을 참조할 것.

모듈(Modular) vs. 전통적 방식(Classic Style)

일반적인 믿음과는 반대로, 전통적 방식에 잘못된 부분은 없다. 여러분 애플리케이션에 맞다면, 모듈 애플리케이션으로 전환할 필요는 없다.

모듈 방식이 아닌 전통적 방식을 사용할 경우 생기는 주된 단점은 루비 프로세스 당 오직 하나의 Sinatra 애플리케이션만 사용할 수 있다는 점이다. 만약 하나 이상을 사용할 계획이라면, 모듈 방식으로 전환하라. 모듈 방식과 전통적 방식을 섞어쓰지 못할 이유는 없다.

하나의 방식에서 다른 것으로 전환할 경우에는, 기본값 설정의 미묘한 차이에 유의해야 한다:

설정전통적 방식 모듈 방식

Setting	Classic	Modular
app_file	sinatra를 로딩하는 파일	Sinatra::Base를 서브클래싱한 파일
run	$0 == app_file	false
logging	true	false
method_override	true	false
inline_templates	true	false
static	true	false

모듈 애플리케이션(Modular Application) 제공하기

모듈 앱을 시작하는 두 가지 일반적인 옵션이 있는데, 공격적으로 run!으로 시작하거나:

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

```
class MyApp < Sinatra::Base
  # ... 여기에 앱 코드가 온다 ...

  # 루비 파일이 직접 실행될 경우에 서버를 시작
  run! if app_file == $0
end
```

다음과 같이 시작:

```
ruby my_app.rb
```

또는 config.ru와 함께 사용하며, 이 경우는 어떠한 Rack 핸들러라도 사용할 수 있다:

```
# config.ru
require './my_app'
run MyApp
```

실행:

```
rackup -p 4567
```

config.ru로 전통적 방식의 애플리케이션 사용하기

앱 파일을 다음과 같이 작성하고:

```
# app.rb
require 'sinatra'

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

대응하는 config.ru는 다음과 같이 작성:

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

언제 config.ru를 사용할까?

Good signs you probably want to use a config.ru: 다음은 config.ru를 사용하게 될 징후들이다:

- 다른 Rack 핸들러(Passenger, Unicorn, Heroku, ...)로 배포하고자 할 때.
- 하나 이상의 Sinatra::Base 서브클래스를 사용하고자 할 때.
- Sinatra를 최종점(endpoint)이 아니라, 오로지 미들웨어로만 사용하고자 할 때.

모듈 방식으로 전환했다는 이유만으로 config.ru로 전환할 필요는 없으며, 또한 config.ru를 사용한다고 해

서 모듈 방식을 사용해야 하는 것도 아니다.

Sinatra를 미들웨어로 사용하기

Sinatra에서 다른 Rack 미들웨어를 사용할 수 있을 뿐 아니라, 모든 Sinatra 애플리케이션은 순차로 어떠한 Rack 종착점 앞에 미들웨어로 추가될 수 있다. 이 종착점은 다른 Sinatra 애플리케이션이 될 수도 있고, 또는 Rack 기반의 어떠한 애플리케이션(Rails/Ramaze/Camping/...)이라도 가능하다:

```
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
  # 미들웨어는 사전 필터보다 앞서 실행됨
  use LoginScreen

  before do
unless session['user_name']
  halt "<a href='/login'>로그인</a> 하세요."
end
  end

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

동적인 애플리케이션 생성(Dynamic Application Creation)

경우에 따라선 어떤 상수에 할당하지 않고 런타임에서 새 애플리케이션들을 생성하고 싶을 수도 있을 것인데, 이 때는 Sinatra.new를 쓰면 된다:

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

이것은 선택적 인자로 상속할 애플리케이션을 받는다:

```
# config.ru
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
```

이것은 Sintra 익스텐션을 테스팅하거나 또는 여러분의 라이브러리에서 Sinatra를 사용할 경우에 특히 유용하다.

또한 이 방법은 Sinatra를 미들웨어로 사용하는 것을 아주 쉽게 만들어 준다:

```
require 'sinatra/base'

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

run RailsProject::Application
```

범위(Scopes)와 바인딩(Binding)

현재 어느 범위에 있느냐가 어떤 메서드와 변수를 사용할 수 있는지를 결정한다.

애플리케이션/클래스 범위

모든 Sinatra 애플리케이션은 `Sinatra::Base`의 서브클래스에 대응된다. 만약 톱레벨 DSL (`require 'sinatra'`)을 사용한다면, 이 클래스는 `Sinatra::Application`이며, 그렇지 않을 경우라면 여러분이 명시적으로 생성한 그 서브클래스가 된다. 클래스 레벨에서는 `get` 이나 `before` 같은 메서드들을 가지나, `request` 객체나 `session` 에는 접근할 수 없다. 왜냐면 모든 요청에 대해 애플리케이션 클래스는 오직 하나이기 때문이다.

`set`으로 생성한 옵션들은 클래스 레벨의 메서드들이다:

```
class MyApp < Sinatra::Base
  # 이봐요, 저는 애플리케이션 범위에 있다구요!
  set :foo, 42
  foo # => 42

  get '/foo' do
# 저기요, 전 이제 더 이상 애플리케이션 범위 속에 있지 않아요!
  end
end
```

다음 속에 있을 때 애플리케이션 범위가 된다:

- 애플리케이션 클래스 본문
- 확장으로 정의된 메서드
- helpers로 전달된 블록
- set의 값으로 사용된 Procs/blocks
- Sinatra.new로 전달된 블록

범위 객체 (클래스)는 다음과 같이 접근할 수 있다:

- configure 블록으로 전달된 객체를 통해(configure { |c| ... })
- 요청 범위 내에서 settings

요청/인스턴스 범위

매 요청마다, 애플리케이션 클래스의 새 인스턴스가 생성되고 모든 핸들러 블록은 그 범위 내에서 실행된다. 이 범위 내에서 여러분은 request 와 session 객체에 접근하거나 erb 나 haml 같은 렌더링 메서드를 호출할 수 있다. 요청 범위 내에서 애플리케이션 범위는 settings 헬퍼를 통해 접근 가능하다:

```
class MyApp < Sinatra::Base
  # 이봐요, 전 애플리케이션 범위에 있다구요!
  get '/define_route/:name' do
# '/define_route/:name'의 요청 범위
@value = 42

settings.get("/#{params[:name]}") do
  # "/#{params[:name]}"의 요청 범위
  @value # => nil (동일한 요청이 아님)
end

"[illegible]"
  end
end
```

다음 속에 있을 때 요청 범위 바인딩이 된다:

- get/head/post/put/delete/options 블록
- before/after 필터
- 헬퍼(helper) 메서드
- 템플릿/뷰

위임 범위(Delegation Scope)

위임 범위(delegation scope)는 메서드를 단순히 클래스 범위로 보낸다(forward). 그렇지만, 100% 클래스 범위처럼 움직이진 않는데, 왜냐면 클래스 바인딩을 갖지 않기 때문이다. 오직 명시적으로 위임(delegation) 표시된 메서드들만 사용 가능하며 또한 클래스 범위와 변수/상태를 공유하지 않는다 (유의: self가 다르다). Sinatra::Delegator.delegate :method_name을 호출하여 메서드 위임을 명시적으로 추가할 수 있다.

다음 속에 있을 때 위임 범위 바인딩을 갖는다:

- 톱레벨 바인딩, `require "sinatra"`를 한 경우
- `Sinatra::Delegator` 믹스인으로 확장된 객체

직접 코드를 살펴보길 바란다: Sinatra::Delegator 믹스인 코드는 메인 객체를 확장한 것이다.

명령행(Command Line)

Sinatra 애플리케이션은 직접 실행할 수 있다:

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

옵션들:

```
-h # 도움말
-p # 포트 설정 (기본값은 4567)
-o # 호스트 설정 (기본값은 0.0.0.0)
-e # 환경 설정 (기본값은 development)
-s # rack 서버/핸들러 지정 (기본값은 thin)
-x # mutex 잠금 켜기 (기본값은 off)
```

요구사항(Requirement)

다음의 루비 버전은 공식적으로 지원한다:

Ruby 1.8.7
: 1.8.7은 완전하게 지원되지만, 꼭 그래야할 특별한 이유가 없다면, 1.9.2로 업그레이드하거나 또는 JRuby나 Rubinius로 전환할 것을 권장한다. 1.8.7에 대한 지원은 Sinatra 2.0과 Ruby 2.0 이전에는 중단되지 않을 것이다. 또한 그때도, 우리는 계속 지원할 것이다. **Ruby 1.8.6**은 더이상 지원되지 않는다**.** 만약 1.8.6으로 실행하려 한다면, Sinatra 1.2로 다운그레이드하라. Sinatra 1.4.0이 릴리스될 때 까지는 버그 픽스를 받을 수 있을 것이다.

Ruby 1.9.2
: 1.9.2는 완전하게 지원되면 권장된다. Radius와 Maraby는 현재 1.9와 호환되지 않음에 유의하라. 1.9.2p0은, Sinatra를 실행했을 때 세그먼트 오류가 발생한다고 알려져 있으니 사용하지 말라. Ruby 1.9.4/2.0 릴리스까지는 적어도 지원을 계속할 것이며, 최신 1.9 릴리스에 대한 지원은 Ruby 코어팀이 지원하고 있는 한 계속 지원할 것이다.

Ruby 1.9.3
: 1.9.3은 완전하게 지원된다. 그렇지만 프로덕션에서의 사용은 보다 상위의 패치 레벨이 릴리스될 때까지 기다리길 권장한다(현재는 p0). 이전 버전에서 1.9.3으로 전환할 경우 모든 세션이 무효화된다는 점을 유의하라.

Rubinius
: Rubinius는 공식적으로 지원되며 (Rubinius >= 1.2.4), 모든 템플릿 언어를 포함한 모든 것들이 작동한다. 조만간 출시될 2.0 릴리스 역시 지원할 것이다.

JRuby
: JRuby는 공식적으로 지원된다 (JRuby >= 1.6.5). 서드 파티 템플릿 라이브러리와의 문제는 알려진 바 없지만, 만약 JRuby를 사용하기로 했다면, JRuby rack 핸들러를 찾아보길 바란다. Thin 웹 서버는 JRuby에서 완전하게 지원되지 않는다. JRuby의 C 확장 지원은 아직 실험 단계이며, RDiscount, Redcarpet 및 RedCloth가 현재 이 영향을 받는다.

또한 우리는 새로 나오는 루비 버전을 주시한다.

다음 루비 구현체들은 공식적으로 지원하지 않지만 여전히 Sinatra를 실행할 수 있는 것으로 알려져 있다:

- JRuby와 Rubinius 예전 버전
- Ruby Enterprise Edition
- MacRuby, Maglev, IronRuby
- Ruby 1.9.0 및 1.9.1 (그러나 이 버전들은 사용하지 말 것을 권함)

공식적으로 지원하지 않는다는 것의 의미는 무언가가 그쪽에서만 잘못되고 지원되는 플랫폼에서는 그러지 않을 경우, 우리의 문제가 아니라 그쪽의 문제로 간주한다는 뜻이다.

또한 우리는 CI를 ruby-head (곧 나올 2.0.0) 과 1.9.4 브랜치에 맞춰 실행하지만, 계속해서 변하고 있기 때문에 아무 것도 보장할 수는 없다. 1.9.4p0와 2.0.0p0가 지원되길 기대한다.

Sinatra는 선택한 루비 구현체가 지원하는 어떠한 운영체제에서도 작동해야 한다.

현재 Cardinal, SmallRuby, BlueRuby 또는 1.8.7 이전의 루비 버전에서는 Sinatra를 실행할 수 없을 것이다.

최신(The Bleeding Edge)

Sinatra의 가장 최근 코드를 사용하고자 한다면, 여러분 애플리케이션을 마스터 브랜치에 맞춰 실행하면 되지만, 덜 안정적일 것임에 분명하다.

또한 우리는 가끔 사전배포(prerelease) 젬을 푸시하기 때문에, 다음과 같이 할 수 있다

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

최신 기능들을 얻기 위해선

Bundler를 사용하여

여러분 애플리케이션을 최신 Sinatra로 실행하고자 한다면, Bundler를 사용할 것을 권장한다.

우선, 아직 설치하지 않았다면 bundler를 설치한다:

```
gem install bundler
```

그런 다음, 프로젝트 디렉터리에서, `Gemfile`을 하나 만든다:

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

# 다른 의존관계들
gem 'haml'# 예를 들어, haml을 사용한다면
gem 'activerecord', '~> 3.0'  # 아마도 ActiveRecord 3.x도 필요할 것
```

이 속에 애플리케이션의 모든 의존관계를 나열해야 함에 유의하자. 그렇지만, Sinatra가 직접적인 의존관계에 있는 것들 (Rack과 Tilt)은 Bundler가 자동으로 추출하여 추가할 것이다.

이제 여러분은 다음과 같이 앱을 실행할 수 있다:

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

직접 하기(Roll Your Own)

로컬 클론(clone)을 생성한 다음 `$LOAD_PATH`에 `sinatra/lib` 디렉터리를 주고 여러분 앱을 실행한다:

```
cd myapp
git clone git://github.com/sinatra/sinatra.git
ruby -Isinatra/lib myapp.rb
```

이후에 Sinatra 소스를 업데이트하려면:

```
cd myapp/sinatra
git pull
```

전역으로 설치(Install Globally)

젬을 직접 빌드할 수 있다:

```
git clone git://github.com/sinatra/sinatra.git
cd sinatra
rake sinatra.gemspec
rake install
```

만약 젬을 루트로 설치한다면, 마지막 단계는 다음과 같이 해야 한다

```
sudo rake install
```

버저닝(Versioning)

Sinatra는 시맨틱 버저닝Semantic Versioning을 준수한다. SemVer 및 SemVerTag 둘 다 해당된.

더 읽을 거리(Further Reading)

- 프로젝트 웹사이트 - 추가 문서들, 뉴스, 그리고 다른 리소스들에 대한 링크.
- 기여하기 - 버그를 찾았나요? 도움이 필요한가요? 패치를 하셨나요?
- 이슈 트래커
- 트위터
- Mailing List
- IRC: #sinatra http://freenode.net
- Sinatra Book Cookbook 튜토리얼
- Sinatra Recipes 커뮤니티가 만드는 레시피
- http://rubydoc.info에 있는 최종 릴리스 또는 current HEAD에 대한 API 문서
- CI server

Sinatra

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

Introdução

Atenção: Este documento é apenas uma tradução da versão em inglês e pode estar desatualizado.

Sinatra é uma DSL para criar aplicações web em Ruby com o mínimo de esforço e rapidez:

```
# minhaapp.rb
require 'sinatra'

get '/' do
  'Olá Mundo!'
end
```

Instale a gem e execute como:

```
gem install sinatra
ruby minhaapp.rb
```

Acesse em: localhost:4567

Recomendamos a execução de `gem install thin`. Caso esteja disponível, o Sinatra irá usar.

Rotas

No Sinatra, uma rota é um método HTTP emparelhado com um padrão de URL. Cada rota possui um bloco de execução:

```
get '/' do
  .. mostrando alguma coisa ..
end

post '/' do
```

```
  .. criando alguma coisa ..
end

put '/' do
  .. atualizando alguma coisa ..
end

patch '/' do
  .. modificando alguma coisa ..
end

delete '/' do
  .. removendo alguma coisa ..
end

options '/' do
  .. estabelecendo alguma coisa ..
end
```

As rotas são interpretadas na ordem em que são definidos. A primeira rota encontrada responde ao pedido.

Padrões de rota podem conter parâmetros nomeados, acessível através do hash `params`:

```
get '/ola/:nome' do
  # corresponde a "GET /ola/foo" e "GET /ola/bar"
  # params[:nome] é 'foo' ou 'bar'
  "Olá #{params[:nome]}!"
end
```

Você também pode acessar parâmetros nomeados através dos parâmetros de um bloco:

```
get '/ola/:nome' do |n|
  "Olá #{n}!"
end
```

Padrões de rota também podem conter parâmetros splat (wildcard), acessível através do array `params[: splat]`:

```
get '/diga/*/para/*' do
  # corresponde a /diga/ola/para/mundo
  params[:splat] # => ["ola", "mundo"]
end

get '/download/*.*' do
  # corresponde a /download/pasta/do/arquivo.xml
  params[:splat] # => ["pasta/do/arquivo", "xml"]
end
```

Ou com parâmetros de um bloco:

```
get '/download/*.*' do |pasta, ext|
  [pasta, ext] # => ["pasta/do/arquivo", "xml"]
end
```

Rotas podem corresponder com expressões regulares:

```
get %r{/ola/([\w]+)} do
  "Olá, #{        :captures        }"
end
```

Ou com parâmetros de um bloco:

```
get %r{/ola/([\w]+)} do |c|
  "Olá, #{ }"
end
```

Padrões de rota podem contar com parâmetros opcionais:

```
get '/posts?:formato?' do
  # corresponde a "GET /posts" e qualquer extensão "GET /posts.json", "GET /posts.xml",
etc.
end
```

A propósito, a menos que você desative a proteção contra ataques (veja abaixo), o caminho solicitado pode ser alterado antes de concluir a comparação com as suas rotas.

Condições

Rotas podem incluir uma variedade de condições, tal como o `user agent`:

```
get '/foo', :agent => /Songbird (\d\.\d)[\d\/]*?/ do
  "Você está usando o Songbird versão #{        :agent  0 }"
end

get '/foo' do
  # Correspondente a navegadores que não sejam Songbird
end
```

Outras condições disponíveis são `host_name` e `provides`:

```
get '/', :host_name => /^admin\./ do
  "Área administrativa. Acesso negado!"
end

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

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

Você pode facilmente definir suas próprias condições:

```
set(:probabilidade) { |valor| condition { rand <= valor } }

get '/ganha_um_carro', :probabilidade => 0.1 do
  "Você ganhou!"
end

get '/ganha_um_carro' do
  "Sinto muito, você perdeu."
end
```

Use splat, para uma condição que levam vários valores:

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

get "/minha/conta/", :auth => [:usuario, :administrador] do
  "Detalhes da sua conta"
end

get "/apenas/administrador/", :auth => :administrador do
  "Apenas administradores são permitidos aqui!"
end
```

Retorno de valores

O valor de retorno do bloco de uma rota determina pelo menos o corpo da resposta passado para o cliente HTTP, ou pelo menos o próximo middleware na pilha Rack. Frequentemente, isto é uma `string`, tal como nos exemplos acima. Mas, outros valores também são aceitos.

Você pode retornar uma resposta válida ou um objeto para o Rack, sendo eles de qualquer tipo de objeto que queira. Além disto, é possível retornar um código de status HTTP.

Dessa forma, podemos implementar facilmente um exemplo de streaming:

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

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

Você também pode usar o método auxiliar `stream` (descrito abaixo) para incorporar a lógica de streaming na rota.

Custom Route Matchers

Como apresentado acima, a estrutura do Sinatra conta com suporte embutido para uso de padrões de String e expressões regulares como validadores de rota. No entanto, ele não pára por aí. Você pode facilmente definir os seus próprios validadores:

```
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 que o exemplo acima pode ser robusto e complicado em excesso. Pode também ser implementado como:

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

Ou, usando algo mais denso à frente:

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

Arquivos estáticos

Arquivos estáticos são disponibilizados a partir do diretório `./public`. Você pode especificar um local diferente pela opção `:public_folder`

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

Note que o nome do diretório público não é incluido na URL. Um arquivo `./public/css/style.css` é disponibilizado como `http://example.com/css/style.css`.

Views / Templates

Templates presumem-se estar localizados sob o diretório `./views`. Para utilizar um diretório view diferente:

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

Uma coisa importante a ser lembrada é que você sempre tem as referências dos templates como símbolos, mesmo se eles estiverem em um sub-diretório (nesse caso utilize `:'subdir/template'`). Métodos de renderização irão processar qualquer string passada diretamente para elas.

Haml Templates

A gem/biblioteca haml é necessária para renderizar templates HAML:

```
# Você precisa do 'require haml' em sua aplicação.
require 'haml'

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

Renderiza `./views/index.haml`.

Opções Haml podem ser setadas globalmente através das configurações do sinatra, veja Opções e Configurações, e substitua em uma requisição individual.

```
set :haml, {:format => :html5 } # o formato padrão do Haml é :xhtml

get '/' do
  haml :index, :haml_options => {:format => :html4 } # substituido
end
```

Erb Templates

```
# Você precisa do 'require erb' em sua aplicação
require 'erb'

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

Renderiza `./views/index.erb`

Erubis

A gem/biblioteca erubis é necessária para renderizar templates erubis:

```
# Você precisa do 'require erubis' em sua aplicação.
require 'erubis'

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

Renderiza `./views/index.erubis`

Builder Templates

A gem/biblioteca builder é necessária para renderizar templates builder:

```
# Você precisa do 'require builder' em sua aplicação.
require 'builder'

get '/' do
  content_type 'application/xml', :charset => 'utf-8'
  builder :index
end
```

Renderiza `./views/index.builder`.

Sass Templates

A gem/biblioteca sass é necessária para renderizar templates sass:

```
# Você precisa do 'require haml' ou 'require sass' em sua aplicação.
require 'sass'

get '/stylesheet.css' do
  content_type 'text/css', :charset => 'utf-8'
  sass :stylesheet
end
```

Renderiza `./views/stylesheet.sass`.

Opções Sass podem ser setadas globalmente através das configurações do sinatra, veja Opções e Configurações, e substitua em uma requisição individual.

```
set :sass, {:style => :compact } # o estilo padrão do Sass é :nested

get '/stylesheet.css' do
  content_type 'text/css', :charset => 'utf-8'
  sass :stylesheet, :style => :expanded # substituido
end
```

Less Templates

A gem/biblioteca less é necessária para renderizar templates Less:

```
# Você precisa do 'require less' em sua aplicação.
require 'less'

get '/stylesheet.css' do
  content_type 'text/css', :charset => 'utf-8'
  less :stylesheet
end
```

Renderiza `./views/stylesheet.less`.

Inline Templates

```
get '/' do
  haml '%div.title Olá Mundo'
end
```

Renderiza a string, em uma linha, no template.

Acessando Variáveis nos Templates

Templates são avaliados dentro do mesmo contexto como manipuladores de rota. Variáveis de instância setadas em rotas manipuladas são diretamente acessadas por templates:

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

Ou, especifique um hash explícito para variáveis locais:

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

Isso é tipicamente utilizando quando renderizamos templates como partials dentro de outros templates.

Templates Inline

Templates podem ser definidos no final do arquivo fonte(.rb):

```
require 'rubygems'
require 'sinatra'

get '/' do
  haml :index
end

__END__

@@ layout
%html
  = yield

@@ index
%div.title Olá Mundo!!!!!
```

NOTA: Templates inline definidos no arquivo fonte são automaticamente carregados pelo sinatra. Digite `enable :inline_templates` se você tem templates inline no outro arquivo fonte.

Templates nomeados

Templates também podem ser definidos utilizando o método top-level `template`:

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

template :index do
  '%div.title Olá Mundo!'
end

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

Se existir um template com nome “layout”, ele será utilizado toda vez que um template for renderizado. Você pode desabilitar layouts passando `:layout => false`.

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

Helpers

Use o método de alto nível `helpers` para definir métodos auxiliares para utilizar em manipuladores de rotas e modelos:

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

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

Filtros

Filtros Before são avaliados antes de cada requisição dentro do contexto da requisição e pode modificar a requisição e a reposta. Variáveis de instância setadas nos filtros são acessadas através de rotas e templates:

```
before do
  @nota = 'Oi!'
  request.path_info = '/foo/bar/baz'
end

get '/foo/*' do
  @nota #=> 'Oi!'
  params[:splat] #=> 'bar/baz'
end
```

Filtros After são avaliados após cada requisição dentro do contexto da requisição e também podem modificar o pedido e a resposta. Variáveis de instância definidas nos filtros before e rotas são acessadas através dos filtros after:

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

Filtros opcionalmente tem um padrão, fazendo com que sejam avaliados somente se o caminho do pedido coincidir com esse padrão:

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

after '/create/:slug' do |slug|
```

```
  session[:last_slug] = slug
end
```

Halting

Para parar imediatamente uma requisição com um filtro ou rota utilize:

```
halt
```

Você também pode especificar o status quando parar...

```
halt 410
```

Ou com corpo de texto...

```
halt 'isso será o corpo do texto'
```

Ou também...

```
halt 401, 'vamos embora!'
```

Com cabeçalhos...

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

Passing

Uma rota pode processar aposta para a próxima rota correspondente usando `pass`:

```
get '/adivinhar/:quem' do
  pass unless params[:quem] == 'Frank'
  'Você me pegou!'
end

get '/adivinhar/.' do
  'Você falhou!'
end
```

O bloqueio da rota é imediatamente encerrado e o controle continua com a próxima rota de parâmetro. Se o parâmetro da rota não for encontrado, um 404 é retornado.

Configuração

Rodando uma vez, na inicialização, em qualquer ambiente:

```
configure do
  ...
end
```

Rodando somente quando o ambiente (`RACK_ENV` environment variável) é setado para `:production`:

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

Rodando quando o ambiente é setado para `:production` ou `:test`:

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

Tratamento de Erros

Tratamento de erros rodam dentro do mesmo contexto como rotas e filtros before, o que significa que você pega todos os presentes que tem para oferecer, como `haml`, `erb`, `halt`, etc.

Não Encontrado

Quando um `Sinatra::NotFound` exception é levantado, ou o código de status da reposta é 404, o `not_found` manipulador é invocado:

```
not_found do
  'Isto está longe de ser encontrado'
end
```

Erro

O manipulador `error` é invocado toda a vez que uma exceção é lançada a partir de um bloco de rota ou um filtro. O objeto da exceção pode ser obtido a partir da variável Rack `sinatra.error`:

```
error do
  'Desculpe, houve um erro desagradável - ' + env['sinatra.error'].name
end
```

Erros customizados:

```
error MeuErroCustomizado do
  'Então que aconteceu foi...' + env['sinatra.error'].message
end
```

Então, se isso acontecer:

```
get '/' do
  raise MeuErroCustomizado, 'alguma coisa ruim'
end
```

Você receberá isso:

```
Então que aconteceu foi... alguma coisa ruim
```

Alternativamente, você pode instalar manipulador de erro para um código de status:

```
error 403 do
  'Acesso negado'
end

get '/secreto' do
  403
end
```

Ou um range:

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

O Sinatra instala os manipuladores especiais `not_found` e `error` quando roda sobre o ambiente de desenvolvimento.

Mime Types

Quando utilizamos `send_file` ou arquivos estáticos você pode ter mime types Sinatra não entendidos. Use `mime_type` para registrar eles por extensão de arquivos:

```
mime_type :foo, 'text/foo'
```

Você também pode utilizar isto com o helper `content_type`:

```
content_type :foo
```

Middleware Rack

O Sinatra roda no Rack, uma interface padrão mínima para frameworks web em Ruby. Um das capacidades mais interessantes do Rack para desenvolver aplicativos é suporte a “middleware” - componentes que ficam entre o servidor e sua aplicação monitorando e/ou manipulando o request/response do HTTP para prover vários tipos de funcionalidades comuns.

O Sinatra faz construtores pipelines do middleware Rack facilmente em um nível superior utilizando o método `use`:

```
require 'sinatra'
require 'meu_middleware_customizado'

use Rack::Lint
use MeuMiddlewareCustomizado

get '/ola' do
  'Olá mundo'
end
```

A semântica de `use` é idêntica aquela definida para a DSL Rack::Builder (mais frequentemente utilizada para arquivos rackup). Por exemplo, o método `use` aceita múltiplos argumentos/variáveis bem como blocos:

```
use Rack::Auth::Basic do |usuario, senha|
  usuario == 'admin' && senha == 'secreto'
end
```

O Rack é distribuido com uma variedade de middleware padrões para logs, debugs, rotas de URL, autenticação, e manipuladores de sessão. Sinatra utilizada muitos desses componentes automaticamente baseando sobre configuração, então, tipicamente você não tem `use` explicitamente.

Testando

Testes no Sinatra podem ser escritos utilizando qualquer biblioteca ou framework de teste baseados no Rack. Rack::Test é recomendado:

```
require 'minha_aplicacao_sinatra'
require 'rack/test'

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

  def app
    Sinatra::Application
  end

  def meu_test_default
```

```
    get '/'
    assert_equal 'Ola Mundo!', last_response.body
  end

  def teste_com_parametros
    get '/atender', :name => 'Frank'
    assert_equal 'Olá Frank!', last_response.bodymeet
  end

  def test_com_ambiente_rack
    get '/', {}, 'HTTP_USER_AGENT' => 'Songbird'
    assert_equal "Você está utilizando o Songbird!", last_response.body
  end
end
```

NOTA: Os módulos de classe embutidos `Sinatra::Test` e `Sinatra::TestHarness` são depreciados na versão 0.9.2.

Sinatra::Base - Middleware, Bibliotecas e aplicativos modulares

Definir sua aplicação em um nível superior de trabalho funciona bem para micro aplicativos, mas tem consideráveis incovenientes na construção de componentes reutilizáveis como um middleware Rack, metal Rails, bibliotecas simples como um componente de servidor, ou mesmo extensões Sinatra. A DSL de nível superior polui o espaço do objeto e assume um estilo de configuração de micro aplicativos (exemplo: uma simples arquivo de aplicação, diretórios `./public` e `./views`, logs, página de detalhes de exceção, etc.). É onde o `Sinatra::Base` entra em jogo:

```
require 'sinatra/base'

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

  get '/' do
    'Ola mundo!'
  end
end
```

A classe `MinhaApp` é um componente Rack independente que pode agir como um middleware Rack, uma aplicação Rack, ou metal Rails. Você pode utilizar ou executar esta classe com um arquivo rackup `config.ru`; ou, controlar um componente de servidor fornecendo como biblioteca:

```
MinhaApp.run! :host => 'localhost', :port => 9090
```

Os métodos disponíveis para subclasses `Sinatra::Base` são exatamente como aqueles disponíveis via a DSL de nível superior. Aplicações de nível mais alto podem ser convertidas para componentes `Sinatra::Base` com duas modificações:

- Seu arquivo deve requerer `sinatra/base` ao invés de `sinatra`; outra coisa, todos os métodos DSL do Sinatra são importados para o espaço principal.

- Coloque as rotas da sua aplicação, manipuladores de erro, filtros e opções na subclasse de um `Sinatra::Base`.

`Sinatra::Base` é um quadro branco. Muitas opções são desabilitadas por padrão, incluindo o servidor embutido. Veja Opções e Configurações para detalhes de opções disponíveis e seus comportamentos.

SIDEBAR: A DSL de alto nível do Sinatra é implementada utilizando um simples sistema de delegação. A classe `Sinatra::Application` - uma subclasse especial da `Sinatra::Base` - recebe todos os `:get`, `:put`, `:post`, `:delete`, `:before`, `:error`, `:not_found`, `:configure`, e `:set messages` enviados para o alto nível. Dê uma olhada no código você mesmo: aqui está o Sinatra::Delegator mixin sendo incluido dentro de um espaço principal

Linha de Comando

Aplicações Sinatra podem ser executadas diretamente:

```
ruby minhaapp.rb [-h] [-x] [-e AMBIENTE] [-p PORTA] [-o HOST] [-s SERVIDOR]
```

As opções são:

```
-h # ajuda
-p # define a porta (padrão é 4567)
-o # define o host (padrão é 0.0.0.0)
-e # define o ambiente (padrão é development)
-s # especifica o servidor/manipulador rack (padrão é thin)
-x # ativa o bloqueio (padrão é desligado)
```

A última versão

Se você gostaria de utilizar o código da última versão do Sinatra, crie um clone local e execute sua aplicação com o diretório `sinatra/lib` no `LOAD_PATH`:

```
cd minhaapp
git clone git://github.com/sinatra/sinatra.git
ruby -I sinatra/lib minhaapp.rb
```

Alternativamente, você pode adicionar o diretório do `sinatra/lib` no `LOAD_PATH` do seu aplicativo:

```
$LOAD_PATH.unshift File.dirname(__FILE__) + '/sinatra/lib'
require 'rubygems'
require 'sinatra'

get '/sobre' do
  "Estou rodando a versão" + Sinatra::VERSION
end
```

Para atualizar o código do Sinatra no futuro:

```
cd meuprojeto/sinatra
git pull
```

Mais

- Website do Projeto - Documentação adicional, novidades e links para outros recursos.
- Contribuir - Encontrar um bug? Precisa de ajuda? Tem um patch?
- Acompanhar Questões
- Twitter
- Lista de Email
- IRC: #sinatra em freenode.net

Sinatra

Introdução

Atenção: Este documento é apenas uma tradução da versão em inglês e pode estar desatualizado.

Sinatra é uma DSL para criar aplicações web em Ruby com o mínimo de esforço e rapidez:

```
# minhaapp.rb
require 'sinatra'

get '/' do
  'Olá Mundo!'
end
```

Instale a gem e execute como:

```
gem install sinatra
ruby minhaapp.rb
```

Acesse em: localhost:4567

Recomendamos a execução de `gem install thin`. Caso esteja disponível, o Sinatra irá usar.

Rotas

No Sinatra, uma rota é um método HTTP emparelhado com um padrão de URL. Cada rota possui um bloco de execução:

```
get '/' do
  .. mostrando alguma coisa ..
end

post '/' do
```

```
  .. criando alguma coisa ..
end

put '/' do
  .. atualizando alguma coisa ..
end

patch '/' do
  .. modificando alguma coisa ..
end

delete '/' do
  .. removendo alguma coisa ..
end

options '/' do
  .. estabelecendo alguma coisa ..
end
```

As rotas são interpretadas na ordem em que são definidos. A primeira rota encontrada responde ao pedido.

Padrões de rota podem conter parâmetros nomeados, acessível através do hash `params`:

```
get '/ola/:nome' do
  # corresponde a "GET /ola/foo" e "GET /ola/bar"
  # params[:nome] é 'foo' ou 'bar'
  "Olá #{params[:nome]}!"
end
```

Você também pode acessar parâmetros nomeados através dos parâmetros de um bloco:

```
get '/ola/:nome' do |n|
  "Olá #{n}!"
end
```

Padrões de rota também podem conter parâmetros splat (wildcard), acessível através do array `params[:splat]`:

```
get '/diga/*/para/*' do
  # corresponde a /diga/ola/para/mundo
  params[:splat] # => ["ola", "mundo"]
end

get '/download/*.*' do
  # corresponde a /download/pasta/do/arquivo.xml
  params[:splat] # => ["pasta/do/arquivo", "xml"]
end
```

Ou com parâmetros de um bloco:

```
get '/download/*.*' do |pasta, ext|
  [pasta, ext] # => ["pasta/do/arquivo", "xml"]
end
```

Rotas podem corresponder com expressões regulares:

```
get %r{/ola/([\w]+)} do
  "Olá, #{        :captures        }"
end
```

Ou com parâmetros de um bloco:

```
get %r{/ola/([\w]+)} do |c|
  "Olá, #{ }"
end
```

Padrões de rota podem contar com parâmetros opcionais:

```
get '/posts?:formato?' do
  # corresponde a "GET /posts" e qualquer extensão "GET /posts.json", "GET /posts.xml",
etc.
end
```

A propósito, a menos que você desative a proteção contra ataques (veja abaixo), o caminho solicitado pode ser alterado antes de concluir a comparação com as suas rotas.

Condições

Rotas podem incluir uma variedade de condições, tal como o `user agent`:

```
get '/foo', :agent => /Songbird (\d\.\d)[\d\/]*?/ do
  "Você está usando o Songbird versão #{        :agent  0 }"
end

get '/foo' do
  # Correspondente a navegadores que não sejam Songbird
end
```

Outras condições disponíveis são `host_name` e `provides`:

```
get '/', :host_name => /^admin\./ do
  "Área administrativa. Acesso negado!"
end

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

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

Você pode facilmente definir suas próprias condições:

```
set(:probabilidade) { |valor| condition { rand <= valor } }

get '/ganha_um_carro', :probabilidade => 0.1 do
  "Você ganhou!"
end

get '/ganha_um_carro' do
  "Sinto muito, você perdeu."
end
```

Use splat, para uma condição que levam vários valores:

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

get "/minha/conta/", :auth => [:usuario, :administrador] do
  "Detalhes da sua conta"
end

get "/apenas/administrador/", :auth => :administrador do
  "Apenas administradores são permitidos aqui!"
end
```

Retorno de valores

O valor de retorno do bloco de uma rota determina pelo menos o corpo da resposta passado para o cliente HTTP, ou pelo menos o próximo middleware na pilha Rack. Frequentemente, isto é uma `string`, tal como nos exemplos acima. Mas, outros valores também são aceitos.

Você pode retornar uma resposta válida ou um objeto para o Rack, sendo eles de qualquer tipo de objeto que queira. Além disto, é possível retornar um código de status HTTP.

Dessa forma, podemos implementar facilmente um exemplo de streaming:

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

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

Você também pode usar o método auxiliar `stream` (descrito abaixo) para incorporar a lógica de streaming na rota.

Custom Route Matchers

Como apresentado acima, a estrutura do Sinatra conta com suporte embutido para uso de padrões de String e expressões regulares como validadores de rota. No entanto, ele não pára por aí. Você pode facilmente definir os seus próprios validadores:

```
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 que o exemplo acima pode ser robusto e complicado em excesso. Pode também ser implementado como:

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

Ou, usando algo mais denso à frente:

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

Arquivos estáticos

Arquivos estáticos são disponibilizados a partir do diretório `./public`. Você pode especificar um local diferente pela opção `:public_folder`

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

Note que o nome do diretório público não é incluido na URL. Um arquivo `./public/css/style.css` é disponibilizado como `http://example.com/css/style.css`.

Views / Templates

Templates presumem-se estar localizados sob o diretório `./views`. Para utilizar um diretório view diferente:

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

Uma coisa importante a ser lembrada é que você sempre tem as referências dos templates como símbolos, mesmo se eles estiverem em um sub-diretório (nesse caso utilize `:'subdir/template'`). Métodos de renderização irão processar qualquer string passada diretamente para elas.

Haml Templates

A gem/biblioteca haml é necessária para renderizar templates HAML:

```
# Você precisa do 'require haml' em sua aplicação.
require 'haml'

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

Renderiza `./views/index.haml`.

Opções Haml podem ser setadas globalmente através das configurações do sinatra, veja Opções e Configurações, e substitua em uma requisição individual.

```
set :haml, {:format => :html5 } # o formato padrão do Haml é :xhtml

get '/' do
  haml :index, :haml_options => {:format => :html4 } # substituido
end
```

Erb Templates

```
# Você precisa do 'require erb' em sua aplicação
require 'erb'

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

Renderiza `./views/index.erb`

Erubis

A gem/biblioteca erubis é necessária para renderizar templates erubis:

```
# Você precisa do 'require erubis' em sua aplicação.
require 'erubis'

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

Renderiza `./views/index.erubis`

Builder Templates

A gem/biblioteca builder é necessária para renderizar templates builder:

```
# Você precisa do 'require builder' em sua aplicação.
require 'builder'

get '/' do
  content_type 'application/xml', :charset => 'utf-8'
  builder :index
end
```

Renderiza `./views/index.builder`.

Sass Templates

A gem/biblioteca sass é necessária para renderizar templates sass:

```
# Você precisa do 'require haml' ou 'require sass' em sua aplicação.
require 'sass'

get '/stylesheet.css' do
  content_type 'text/css', :charset => 'utf-8'
  sass :stylesheet
end
```

Renderiza `./views/stylesheet.sass`.

Opções Sass podem ser setadas globalmente através das configurações do sinatra, veja Opções e Configurações, e substitua em uma requisição individual.

```
set :sass, {:style => :compact } # o estilo padrão do Sass é :nested

get '/stylesheet.css' do
  content_type 'text/css', :charset => 'utf-8'
  sass :stylesheet, :style => :expanded # substituido
end
```

Less Templates

A gem/biblioteca less é necessária para renderizar templates Less:

```
# Você precisa do 'require less' em sua aplicação.
require 'less'

get '/stylesheet.css' do
  content_type 'text/css', :charset => 'utf-8'
  less :stylesheet
end
```

Renderiza `./views/stylesheet.less`.

Inline Templates

```
get '/' do
  haml '%div.title Olá Mundo'
end
```

Renderiza a string, em uma linha, no template.

Acessando Variáveis nos Templates

Templates são avaliados dentro do mesmo contexto como manipuladores de rota. Variáveis de instância setadas em rotas manipuladas são diretamente acessadas por templates:

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

Ou, especifique um hash explícito para variáveis locais:

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

Isso é tipicamente utilizando quando renderizamos templates como partials dentro de outros templates.

Templates Inline

Templates podem ser definidos no final do arquivo fonte(.rb):

```
require 'rubygems'
require 'sinatra'

get '/' do
  haml :index
end

__END__

@@ layout
%html
  = yield

@@ index
%div.title Olá Mundo!!!!!
```

NOTA: Templates inline definidos no arquivo fonte são automaticamente carregados pelo sinatra. Digite `enable :inline_templates` se você tem templates inline no outro arquivo fonte.

Templates nomeados

Templates também podem ser definidos utilizando o método top-level `template`:

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

template :index do
  '%div.title Olá Mundo!'
end

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

Se existir um template com nome “layout”, ele será utilizado toda vez que um template for renderizado. Você pode desabilitar layouts passando `:layout => false`.

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

Helpers

Use o método de alto nível `helpers` para definir métodos auxiliares para utilizar em manipuladores de rotas e modelos:

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

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

Filtros

Filtros Before são avaliados antes de cada requisição dentro do contexto da requisição e pode modificar a requisição e a reposta. Variáveis de instância setadas nos filtros são acessadas através de rotas e templates:

```
before do
  @nota = 'Oi!'
  request.path_info = '/foo/bar/baz'
end

get '/foo/*' do
  @nota #=> 'Oi!'
  params[:splat] #=> 'bar/baz'
end
```

Filtros After são avaliados após cada requisição dentro do contexto da requisição e também podem modificar o pedido e a resposta. Variáveis de instância definidas nos filtros before e rotas são acessadas através dos filtros after:

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

Filtros opcionalmente tem um padrão, fazendo com que sejam avaliados somente se o caminho do pedido coincidir com esse padrão:

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

after '/create/:slug' do |slug|
```

```
  session[:last_slug] = slug
end
```

Halting

Para parar imediatamente uma requisição com um filtro ou rota utilize:

```
halt
```

Você também pode especificar o status quando parar...

```
halt 410
```

Ou com corpo de texto...

```
halt 'isso será o corpo do texto'
```

Ou também...

```
halt 401, 'vamos embora!'
```

Com cabeçalhos...

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

Passing

Uma rota pode processar aposta para a próxima rota correspondente usando `pass`:

```
get '/adivinhar/:quem' do
  pass unless params[:quem] == 'Frank'
  'Você me pegou!'
end

get '/adivinhar/.' do
  'Você falhou!'
end
```

O bloqueio da rota é imediatamente encerrado e o controle continua com a próxima rota de parâmetro. Se o parâmetro da rota não for encontrado, um 404 é retornado.

Configuração

Rodando uma vez, na inicialização, em qualquer ambiente:

```
configure do
  ...
end
```

Rodando somente quando o ambiente (`RACK_ENV` environment variável) é setado para `:production`:

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

Rodando quando o ambiente é setado para `:production` ou `:test`:

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

Tratamento de Erros

Tratamento de erros rodam dentro do mesmo contexto como rotas e filtros before, o que significa que você pega todos os presentes que tem para oferecer, como `haml`, `erb`, `halt`, etc.

Não Encontrado

Quando um `Sinatra::NotFound` exception é levantado, ou o código de status da reposta é 404, o `not_found` manipulador é invocado:

```
not_found do
  'Isto está longe de ser encontrado'
end
```

Erro

O manipulador `error` é invocado toda a vez que uma exceção é lançada a partir de um bloco de rota ou um filtro. O objeto da exceção pode ser obtido a partir da variável Rack `sinatra.error`:

```
error do
  'Desculpe, houve um erro desagradável - ' + env['sinatra.error'].name
end
```

Erros customizados:

```
error MeuErroCustomizado do
  'Então que aconteceu foi ...' + env['sinatra.error'].message
end
```

Então, se isso acontecer:

```
get '/' do
  raise MeuErroCustomizado, 'alguma coisa ruim'
end
```

Você receberá isso:

```
Então que aconteceu foi... alguma coisa ruim
```

Alternativamente, você pode instalar manipulador de erro para um código de status:

```
error 403 do
  'Acesso negado'
end

get '/secreto' do
  403
end
```

Ou um range:

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

O Sinatra instala os manipuladores especiais `not_found` e `error` quando roda sobre o ambiente de desenvolvimento.

Mime Types

Quando utilizamos `send_file` ou arquivos estáticos você pode ter mime types Sinatra não entendidos. Use `mime_type` para registrar eles por extensão de arquivos:

```
mime_type :foo, 'text/foo'
```

Você também pode utilizar isto com o helper `content_type`:

```
content_type :foo
```

Middleware Rack

O Sinatra roda no Rack, uma interface padrão mínima para frameworks web em Ruby. Um das capacidades mais interessantes do Rack para desenvolver aplicativos é suporte a “middleware” - componentes que ficam entre o servidor e sua aplicação monitorando e/ou manipulando o request/response do HTTP para prover vários tipos de funcionalidades comuns.

O Sinatra faz construtores pipelines do middleware Rack facilmente em um nível superior utilizando o método `use`:

```
require 'sinatra'
require 'meu_middleware_customizado'

use Rack::Lint
use MeuMiddlewareCustomizado

get '/ola' do
  'Olá mundo'
end
```

A semântica de `use` é idêntica aquela definida para a DSL Rack::Builder (mais frequentemente utilizada para arquivos rackup). Por exemplo, o método `use` aceita múltiplos argumentos/variáveis bem como blocos:

```
use Rack::Auth::Basic do |usuario, senha|
  usuario == 'admin' && senha == 'secreto'
end
```

O Rack é distribuido com uma variedade de middleware padrões para logs, debugs, rotas de URL, autenticação, e manipuladores de sessão. Sinatra utilizada muitos desses componentes automaticamente baseando sobre configuração, então, tipicamente você não tem `use` explicitamente.

Testando

Testes no Sinatra podem ser escritos utilizando qualquer biblioteca ou framework de teste baseados no Rack. Rack::Test é recomendado:

```
require 'minha_aplicacao_sinatra'
require 'rack/test'

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

  def app
    Sinatra::Application
  end

  def meu_test_default
```

```
    get '/'
    assert_equal 'Ola Mundo!', last_response.body
  end

  def teste_com_parametros
    get '/atender', :name => 'Frank'
    assert_equal 'Olá Frank!', last_response.bodymeet
  end

  def test_com_ambiente_rack
    get '/', {}, 'HTTP_USER_AGENT' => 'Songbird'
    assert_equal "Você está utilizando o Songbird!", last_response.body
  end
end
```

NOTA: Os módulos de classe embutidos `Sinatra::Test` e `Sinatra::TestHarness` são depreciados na versão 0.9.2.

Sinatra::Base - Middleware, Bibliotecas e aplicativos modulares

Definir sua aplicação em um nível superior de trabalho funciona bem para micro aplicativos, mas tem consideráveis incovenientes na construção de componentes reutilizáveis como um middleware Rack, metal Rails, bibliotecas simples como um componente de servidor, ou mesmo extensões Sinatra. A DSL de nível superior polui o espaço do objeto e assume um estilo de configuração de micro aplicativos (exemplo: uma simples arquivo de aplicação, diretórios `./public` e `./views`, logs, página de detalhes de exceção, etc.). É onde o `Sinatra::Base` entra em jogo:

```
require 'sinatra/base'

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

  get '/' do
    'Ola mundo!'
  end
end
```

A classe `MinhaApp` é um componente Rack independente que pode agir como um middleware Rack, uma aplicação Rack, ou metal Rails. Você pode utilizar ou executar esta classe com um arquivo rackup `config.ru`; ou, controlar um componente de servidor fornecendo como biblioteca:

```
MinhaApp.run! :host => 'localhost', :port => 9090
```

Os métodos disponíveis para subclasses `Sinatra::Base` são exatamente como aqueles disponíveis via a DSL de nível superior. Aplicações de nível mais alto podem ser convertidas para componentes `Sinatra::Base` com duas modificações:

- Seu arquivo deve requerer `sinatra/base` ao invés de `sinatra`; outra coisa, todos os métodos DSL do Sinatra são importados para o espaço principal.

- Coloque as rotas da sua aplicação, manipuladores de erro, filtros e opções na subclasse de um `Sinatra::Base`.

`Sinatra::Base` é um quadro branco. Muitas opções são desabilitadas por padrão, incluindo o servidor embutido. Veja Opções e Configurações para detalhes de opções disponíveis e seus comportamentos.

SIDEBAR: A DSL de alto nível do Sinatra é implementada utilizando um simples sistema de delegação. A classe `Sinatra::Application` - uma subclasse especial da `Sinatra::Base` - recebe todos os `:get`, `:put`, `:post`, `:delete`, `:before`, `:error`, `:not_found`, `:configure`, e `:set messages` enviados para o alto nível. Dê uma olhada no código você mesmo: aqui está o Sinatra::Delegator mixin sendo incluido dentro de um espaço principal

Linha de Comando

Aplicações Sinatra podem ser executadas diretamente:

```
ruby minhaapp.rb [-h] [-x] [-e AMBIENTE] [-p PORTA] [-o HOST] [-s SERVIDOR]
```

As opções são:

```
-h # ajuda
-p # define a porta (padrão é 4567)
-o # define o host (padrão é 0.0.0.0)
-e # define o ambiente (padrão é development)
-s # especifica o servidor/manipulador rack (padrão é thin)
-x # ativa o bloqueio (padrão é desligado)
```

A última versão

Se você gostaria de utilizar o código da última versão do Sinatra, crie um clone local e execute sua aplicação com o diretório `sinatra/lib` no `LOAD_PATH`:

```
cd minhaapp
git clone git://github.com/sinatra/sinatra.git
ruby -I sinatra/lib minhaapp.rb
```

Alternativamente, você pode adicionar o diretório do `sinatra/lib` no `LOAD_PATH` do seu aplicativo:

```
$LOAD_PATH.unshift File.dirname(__FILE__) + '/sinatra/lib'
require 'rubygems'
require 'sinatra'

get '/sobre' do
  "Estou rodando a versão" + Sinatra::VERSION
end
```

Para atualizar o código do Sinatra no futuro:

```
cd meuprojeto/sinatra
git pull
```

Mais

- Website do Projeto - Documentação adicional, novidades e links para outros recursos.
- Contribuir - Encontrar um bug? Precisa de ajuda? Tem um patch?
- Acompanhar Questões
- Twitter
- Lista de Email
- IRC: #sinatra em freenode.net

Sinatra

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

Введение

Внимание: Этот документ является переводом английской версии и может быть устаревшим

Sinatra — это предметно-ориентированный каркас (DSL) для быстрого создания функциональных веб-приложений на Ruby с минимумом усилий:

```
# myapp.rb
require 'sinatra'

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

Установите gem:

```
gem install sinatra
```

и запустите приложение с помощью:

```
ruby myapp.rb
```

Оцените результат: http://localhost:4567

Рекомендуется также установить Thin, сделать это можно командой: `gem install thin`. Thin — это более производительный и функциональный сервер для разработки приложений на Sinatra.

Маршруты

В Sinatra маршрут — это пара: <HTTP метод> и

<шаблон URL>. Каждый маршрут связан с блоком кода:

```
get '/' do
  # .. что-то показать ..
end

post '/' do
  # .. что-то создать ..
end

put '/' do
  # .. что-то заменить ..
end

patch '/' do
  # .. что-то изменить ..
end

delete '/' do
  # .. что-то удалить ..
end

options '/' do
  # .. что-то ответить ..
end
```


Маршруты сверяются с запросом в порядке очередности их записи в файле приложения. Первый же совпавший с запросом маршрут и будет вызван.

Шаблоны маршрутов могут включать в себя именованные параметры, доступные в хэше `params`:

```
get '/hello/:name' do
  # соответствует "GET /hello/foo" и "GET /hello/bar",
  # где params[:name] 'foo' или 'bar'
  "Hello #{ :name }"
end
```

Также можно использовать именованные параметры в качестве переменных блока:

```
get '/hello/:name' do |n|
  "Hello #{ }"
end
```

Шаблоны маршрутов также могут включать в себя splat (или ‘*’ маску, обозначающую любой символ) параметры, доступные в массиве `params[:splat]`:

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

get '/download/*.*' do
```

```
  # соответствует /download/path/to/file.xml
  params[:splat] # => ["path/to/file", "xml"]
end
```

Или с параметрами блока:

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

Регулярные выражения в качестве шаблонов маршрутов:

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

Или с параметром блока:

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

Шаблоны маршрутов могут иметь необязательные параметры:

```
get '/posts.?:format?' do
  # соответствует "GET /posts", "GET /posts.json", "GET /posts.xml" и т.д.
end
```

Кстати, если вы не отключите защиту от обратного пути в директориях (path traversal, см. ниже), путь запроса может быть изменен до начала поиска подходящего маршрута.

Условия

Маршруты могут включать различные условия совпадений, например, клиентское приложение (user agent):

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

get '/foo' do
  # соответствует не-songbird браузерам
end
```

Другими доступными условиями являются `host_name` и `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
```

Вы можете задать собственные условия:

```
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
```

Для условия, которое принимает несколько параметров, используйте звездочку:

```
set(:auth) do |*roles|   # <- обратите внимание на звездочку
  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
```

Возвращаемые значения

Возвращаемое значение блока маршрута ограничивается телом ответа, которое будет передано HTTP клиенту, или следующей “прослойкой” (middleware) в Rack стеке. Чаще всего это строка, как в примерах выше. Но также приемлемы и другие значения.

Вы можете вернуть любой объект, который будет либо корректным Rack ответом, объектом Rack body, либо кодом состояния HTTP:

Таким образом, легко можно реализовать, например, поточный пример:

```
class Stream
  def each
```

```
    100.times { |i| yield "#{ }\n" }
  end
end

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

Вы также можете использовать метод `stream` (описываемый ниже), чтобы уменьшить количество дублируемого кода и держать логику стриминга прямо в маршруте.

Собственные детекторы совпадений для маршрутов

Как показано выше, Sinatra поставляется со встроенной поддержкой строк и регулярных выражений в качестве шаблонов URL. Но и это еще не все. Вы можете легко определить свои собственные детекторы совпадений (matchers) для маршрутов:

```
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
```

Заметьте, что предыдущий пример, возможно, чересчур усложнен, потому что он может быть реализован так:

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

Или с использованием негативного просмотра вперед:

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

Статические файлы

Статические файлы отдаются из `./public` директории. Вы можете указать другое место, используя опцию `:public_folder`:

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

Учтите, что имя директории со статическими файлами не включено в URL. Например, файл `./public/css/style.css` будет доступен как `http://example.com/css/style.css`.

Используйте опцию `:static_cache_control` (см. ниже), чтобы добавить заголовок `Cache-Control`.

Представления / Шаблоны

Каждый шаблонизатор представлен своим собственным методом. Эти методы попросту возвращают строку:

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

Отобразит `views/index.erb`.

Вместо имени шаблона вы так же можете передавать непосредственно само содержимое шаблона:

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

Эти методы принимают второй аргумент, хеш с опциями:

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

Отобразит `views/index.erb`, вложенным в `views/post.erb` (по умолчанию: `views/layout.erb`, если существует).

Любые опции, не понимаемые Sinatra, будут переданы в шаблонизатор:

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

Вы также можете задавать опции для шаблонизаторов в общем:

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

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

Опции, переданные в метод, переопределяют опции, заданные с помощью `set`.

Доступные опции:

locals
: Список локальных переменных, передаваемых в документ. Например: `erb "<%= foo %>", :locals => {:foo => "bar"}`

default_encoding
: Кодировка, которую следует использовать, если не удалось определить оригинальную. По умолчанию: `settings.default_encoding`.

views
: Директория с шаблонами. По умолчанию: `settings.views`.

layout
: Использовать или нет лэйаут (`true` или `false`). Если же значение Symbol, то указывает, какой шаблон использовать в качестве лэйаута. Например: `erb :index, :layout => !request.xhr?`

content_type
: Content-Type отображенного шаблона. По умолчанию: задается шаблонизатором.

scope
: Область видимости, в которой рендерятся шаблоны. По умолчанию: экземпляр приложения. Если вы измените эту опцию, то переменные экземпляра и методы-помощники станут недоступными в ваших шаблонах.

layout_engine
: Шаблонизатор, который следует использовать для отображения лэйаута. Полезная опция для шаблонизаторов, в которых нет никакой поддержки лэйаутов. По умолчанию: тот же шаблонизатор, что используется и для самого шаблона. Пример: `set :rdoc, :layout_engine => :erb`

По умолчанию считается, что шаблоны находятся в директории `./views`. Чтобы использовать другую директорию с шаблонами:

```
set :views, settings.root + '/templates'
```

Важное замечание: вы всегда должны ссылаться на шаблоны с помощью символов (Symbol), даже когда они в поддиректории (в этом случае используйте `:'subdir/template'`). Вы должны использовать символы, потому что иначе шаблонизаторы попросту отображают любые строки, переданные им.

Буквальные шаблоны

```
get '/' do
```

```
  haml '%div.title Hello World'
end
```

Отобразит шаблон, переданный строкой.

Доступные шаблонизаторы

Некоторые языки шаблонов имеют несколько реализаций. Чтобы указать, какую реализацию использовать, вам следует просто подключить нужную библиотеку:

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

Haml шаблоны

Зависимости	haml
Расширения файлов	`.haml`
Пример	`haml :index, :format => :html5`

Erb шаблоны

Зависимости	erubis или erb (включен в Ruby)
Расширения файлов	`.erb`, `.rhtml` or `.erubis` (только Erubis)
Пример	`erb :index`

Builder шаблоны

Зависимости	builder
Расширения файлов	`.builder`
Пример	`builder { |xml| xml.em "hi" }`

Блок также используется и для встроенных шаблонов (см. пример).

Nokogiri шаблоны

Зависимости	nokogiri
Расширения файлов	`.nokogiri`
Пример	`nokogiri { |xml| xml.em "hi" }`

Блок также используется и для встроенных шаблонов (см. пример).

Sass шаблоны

Зависимости	sass
Расширения файлов	`.sass`
Пример	`sass :stylesheet, :style => :expanded`

SCSS шаблоны

Зависимости	sass
Расширения файлов	`.scss`
Пример	`scss :stylesheet, :style => :expanded`

Less шаблоны

Зависимости	less
Расширения файлов	`.less`
Пример	`less :stylesheet`

Liquid шаблоны

Зависимости	liquid
Расширения файлов	`.liquid`
Пример	`liquid :index, :locals => { :key => 'value' }`

Так как в Liquid шаблонах невозможно вызывать методы из Ruby (кроме `yield`), то вы почти всегда будете передавать в шаблон локальные переменные.

Markdown шаблоны

Зависимости	Любая из библиотек: RDiscount, RedCarpet, BlueCloth, kramdown, maruku
Расширения файлов	`.markdown`, `.mkd` and `.md`
Пример	`markdown :index, :layout_engine => :erb`

В Markdown невозможно вызывать методы или передавать локальные переменные. Следовательно, вам, скорее всего, придется использовать этот шаблон совместно с другим

шаблонизатором:

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

Заметьте, что вы можете вызывать метод `markdown` из других шаблонов:

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

Вы не можете вызывать Ruby из Markdown, соответственно, вы не можете использовать лэйауты на Markdown. Тем не менее, есть возможность использовать один шаблонизатор для отображения шаблона, а другой для лэйаута с помощью опции `:layout_engine`.

Textile шаблоны

Зависимости	RedCloth
Расширения файлов	`.textile`
Пример	`textile :index, :layout_engine => :erb`

В Textile невозможно вызывать методы или передавать локальные переменные. Следовательно, вам, скорее всего, придется использовать этот шаблон совместно с другим шаблонизатором:

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

Заметьте, что вы можете вызывать метод `textile` из других шаблонов:

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

Вы не можете вызывать Ruby из Textile, соответственно, вы не можете использовать лэйауты на Textile. Тем не менее, есть возможность использовать один шаблонизатор для отображения шаблона, а другой для лэйаута с помощью опции `:layout_engine`.

RDoc шаблоны

Зависимости	RDoc
Расширения файлов	`.rdoc`
Пример	`rdoc :README, :layout_engine => :erb`

В RDoc невозможно вызывать методы или передавать локальные переменные. Следовательно, вам, скорее всего, придется использовать этот шаблон совместно с другим шаблонизатором:

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

Заметьте, что вы можете вызывать метод `rdoc` из других шаблонов:

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

Вы не можете вызывать Ruby из RDoc, соответственно, вы не можете использовать лэйауты на RDoc. Тем не менее, есть возможность использовать один шаблонизатор для отображения шаблона, а другой для лэйаута с помощью опции `:layout_engine`.

Radius шаблоны

Зависимости	Radius
Расширения файлов	`.radius`
Пример	`radius :index, :locals => { :key => 'value' }`

Так как в Radius шаблонах невозможно вызывать методы из Ruby напрямую, то вы почти всегда будете передавать в шаблон локальные переменные.

Markaby шаблоны

Зависимости	Markaby
Расширения файлов	`.mab`
Пример	`markaby { h1 "Welcome!" }`

Блок также используется и для встроенных шаблонов (см. пример).

RABL шаблоны

Зависимости	Rabl
Расширения файлов	`.rabl`
Пример	`rabl :index`

Slim шаблоны

Зависимости	Slim Lang
Расширения файлов	`.slim`
Пример	`slim :index`

Creole шаблоны

Зависимости	Creole
Расширения файлов	`.creole`
Пример	`creole :wiki, :layout_engine => :erb`

В Creole невозможно вызывать методы или передавать локальные переменные. Следовательно, вам, скорее всего, придется использовать этот шаблон совместно с другим шаблонизатором:

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

Заметьте, что вы можете вызывать метод `creole` из других шаблонов:

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

Вы не можете вызывать Ruby из Creole, соответственно, вы не можете использовать лэйауты на Creole. Тем не менее, есть возможность использовать один шаблонизатор для отображения шаблона, а другой для лэйаута с помощью опции `:layout_engine`.

CoffeeScript шаблоны

Зависимости	CoffeeScript и способ запускать JavaScript
Расширения файлов	`.coffee`
Пример	`coffee :index`

Yajl шаблоны

Зависимости	yajl-ruby
Расширения файлов	`.yajl`
Пример	`yajl :index, :locals => { :key => 'qux' }, :callback => 'present', :variable => 'resource'`

Содержимое шаблона интерпретируется как код на Ruby, а результирующая переменная json затем конвертируется с помощью `#to_json`.

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

Опции `:callback` и `:variable` используются для “декорирования” итогового объекта.

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

WLang шаблоны

Зависимости	wlang
Расширения файлов	`.wlang`
Пример	`wlang :index, :locals => { :key => 'value' }`

Так как в WLang шаблонах невозможно вызывать методы из Ruby напрямую (за исключением `yield`), то вы почти всегда будете передавать в шаблон локальные переменные.

Доступ к переменным в шаблонах

Шаблоны интерпретируются в том же контексте, что и обработчики маршрутов. Переменные экземпляра, установленные в процессе обработки маршрутов, будут доступны напрямую в шаблонах:

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

Либо установите их через хеш локальных переменных:

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

Это обычный подход, когда шаблоны рендерятся как части других шаблонов.

Шаблоны с `yield` и вложенные раскладки (layout)

Раскладка (layout) обычно представляет собой шаблон, который исполняет `yield`. Такой шаблон может быть либо использован с помощью опции `:template`, как описано выше, либо он может быть дополнен блоком:

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

Эти инструкции в основном эквивалентны `erb :index, :layout => :post`.

Передача блоков интерпретирующим шаблоны методам наиболее полезна для создания вложенных раскладок:

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

Это же самое может быть сделано короче:

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

В настоящее время, следующие интерпретирубщие шаблоны методы принимают блок: `erb`, `haml`, `liquid`, `slim` , `wlang`. Общий метод заполнения шаблонов `render` также принимает блок.

Включённые шаблоны

Шаблоны также могут быть определены в конце исходного файла:

```
require 'sinatra'

get '/' do
  haml :index
end

__END__

@@ layout
%html
  = yield

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

Заметьте: включённые шаблоны, определенные в исходном файле, который подключил Sinatra, будут загружены автоматически. Вызовите `enable :inline_templates` напрямую, если используете включённые шаблоны в других файлах.

Именованные шаблоны

Шаблоны также могут быть определены при помощи `template` метода:

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

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

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

Если шаблон с именем “layout” существует, то он будет использоваться каждый раз при рендеринге. Вы можете отключать лэйаут в каждом конкретном случае с помощью `:layout => false` или отключить его для всего приложения: `set :haml, :layout => false`:

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

Привязка файловых расширений

Чтобы связать расширение файла с движком рендеринга, используйте `Tilt.register`. Например, если вы хотите использовать расширение `tt` для шаблонов Textile:

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

Добавление собственного движка рендеринга

Сначала зарегистрируйте свой движок в Tilt, а затем создайте метод, отвечающий за рендеринг:

```
Tilt.register :myat, MyAwesomeTemplateEngine

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

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

Отобразит `./views/index.myat`. Чтобы узнать больше о Tilt, смотрите https://github.com/rtomayko/tilt

Фильтры

`before`-фильтры выполняются перед каждым запросом в том же контексте, что и маршруты, и могут изменять как запрос, так и ответ на него. Переменные экземпляра, установленные в

фильтрах, доступны в маршрутах и шаблонах:

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

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

after-фильтры выполняются после каждого запроса в том же контексте и могут изменять как запрос, так и ответ на него. Переменные экземпляра, установленные в before-фильтрах и маршрутах, будут доступны в after-фильтрах:

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

Заметьте: если вы используете метод body, а не просто возвращаете строку из маршрута, то тело ответа не будет доступно в after-фильтрах, так как оно будет сгенерировано позднее.

Фильтры могут использовать шаблоны URL и будут интерпретированы, только если путь запроса совпадет с этим шаблоном:

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

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

Как и маршруты, фильтры могут использовать условия:

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

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

Методы-помощники

Используйте метод helpers, чтобы определить методы-помощники, которые в дальнейшем можно будет использовать в обработчиках маршрутов и шаблонах:

```
helpers do
```

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

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

Также методы-помощники могут быть заданы в отдельных модулях:

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

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

helpers FooUtils, BarUtils
```

Эффект равносилен включению модулей в класс приложения.

Использование сессий

Сессия используется, чтобы сохранять состояние между запросами. Если эта опция включена, то у вас будет один хеш сессии на одну пользовательскую сессию:

```
enable :sessions

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

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

Заметьте, что при использовании `enable :sessions` все данные сохраняются в куках (cookies). Это может быть не совсем то, что вы хотите (например, сохранение больших объемов данных увеличит ваш трафик). В таком случае вы можете использовать альтернативную Rack “прослойку” (middleware), реализующую механизм сессий. Для этого не надо вызывать `enable :sessions`, вместо этого следует подключить ее так же, как и любую другую “прослойку”:

```
use Rack::Session::Pool, :expire_after => 2592000

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

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

```
end
```

Для повышения безопасности данные сессии в куках подписываются секретным ключом. Секретный ключ генерируется Sinatra. Тем не менее, так как этот ключ будет меняться с каждым запуском приложения, вы, возможно, захотите установить ключ вручную, чтобы у всех экземпляров вашего приложения был один и тот же ключ:

```
set :session_secret, 'super secret'
```

Если вы хотите больше настроек для сессий, вы можете задать их, передав хеш опций в параметр `sessions`:

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

Прерывание

Чтобы незамедлительно прервать обработку запроса внутри фильтра или маршрута, используйте:

```
halt
```

Можно также указать статус при прерывании:

```
halt 410
```

Тело:

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

И то, и другое:

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

Можно указать заголовки:

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

И, конечно, можно использовать шаблоны с `halt`:

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

Передача

Маршрут может передать обработку запроса следующему совпадающему маршруту, используя `pass`:

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

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

Блок маршрута сразу же прерывается, и контроль переходит к следующему совпадающему маршруту. Если соответствующий маршрут не найден, то ответом на запрос будет 404.

Вызов другого маршрута

Иногда `pass` не подходит, например, если вы хотите получить результат вызова другого обработчика маршрута. В таком случае просто используйте `call`:

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

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

Заметьте, что в предыдущем примере можно облегчить тестирование и повысить производительность, перенеся `"bar"` в метод-помощник, используемый и в `/foo`, и в `/bar`.

Если вы хотите, чтобы запрос был отправлен в тот же экземпляр приложения, а не в его копию, используйте `call!` вместо `call`.

Если хотите узнать больше о `call`, смотрите спецификацию Rack.

Задание тела, кода и заголовков ответа

Хорошим тоном является установка кода состояния HTTP и тела ответа в возвращаемом значении обработчика маршрута. Тем не менее, в некоторых ситуациях вам, возможно, понадобится задать тело ответа в произвольной точке потока исполнения. Вы можете сделать это с помощью метода-помощника `body`. Если вы задействуете метод `body`, то вы можете использовать его и в дальнейшем, чтобы получить доступ к телу ответа.

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

```
end

after do
  puts body
end
```

Также можно передать блок в метод `body`, который затем будет вызван обработчиком Rack (такой подход может быть использован для реализации поточного ответа, см. “Возвращаемые значения”).

Аналогично вы можете установить код ответа и его заголовки:

```
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
```

Как и `body`, методы `headers` и `status`, вызванные без аргументов, возвращают свои текущие значения.

Стриминг ответов

Иногда требуется начать отправлять данные клиенту прямо в процессе генерирования частей этих данных. В особых случаях требуется постоянно отправлять данные до тех пор, пока клиент не закроет соединение. Вы можете использовать метод `stream` вместо написания собственных “оберток”.

```
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
```

Что позволяет вам реализовать стриминговые API, Server Sent Events, и может служить основой для WebSockets. Также такой подход можно использовать для увеличения производительности в случае, когда какая-то часть контента зависит от медленного ресурса.

Заметьте, что возможности стриминга, особенно количество одновременно обслуживаемых запросов, очень сильно зависят от используемого веб-сервера. Некоторые серверы, например, WEBRick, могут и вовсе не поддерживать стриминг. Если сервер не поддерживает стриминг, то все данные будут отправлены за один раз сразу после того, как блок, переданный в `stream`, завершится. Стриминг вообще не работает при использовании Shotgun.

Если метод используется с параметром `keep_open`, то он не будет вызывать `close` у объекта потока, что позволит вам закрыть его позже в любом другом месте. Это работает только с событийными серверами, например, с Thin и Rainbows. Другие же серверы все равно будут закрывать поток:

```
# long polling

set :server, :thin
connections = []

get '/subscribe' do
  # регистрация клиента
  stream(:keep_open) { |out| connections << out }

  # удаление "мертвых клиентов"
  connections.reject!(&:closed?)

  # допуск
  "subscribed"
end

post '/message' do
  connections.each do |out|
    # уведомить клиента о новом сообщении
    out << params[:message] << "\n"

    # указать клиенту на необходимость снова соединиться
    out.close
  end

  # допуск
  "message received"
end
```

Логирование

В области видимости запроса метод `logger` предоставляет доступ к экземпляру `Logger`:

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

Этот логер автоматически учитывает ваши настройки логирования в Rack. Если логирование выключено, то этот метод вернет пустой (dummy) объект, поэтому вы можете смело использовать его в маршрутах и фильтрах.

Заметьте, что логирование включено по умолчанию только для `Sinatra::Application`, а если ваше приложение — подкласс `Sinatra::Base`, то вы, наверное, захотите включить его вручную:

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

```
    enable :logging
  end
end
```

Чтобы избежать использования любой логирующей “прослойки”, задайте опции `logging` значение `nil`. Тем не менее, не забывайте, что в такой ситуации `logger` вернет `nil`. Чаще всего так делают, когда задают свой собственный логер. Sinatra будет использовать то, что находится в `env['rack.logger']`.

Mime-типы

Когда вы используете `send_file` или статические файлы, у вас могут быть mime-типы, которые Sinatra не понимает по умолчанию. Используйте `mime_type` для их регистрации по расширению файла:

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

Вы также можете использовать это в `content_type` методе-помощнике:

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

Генерирование URL

Чтобы сформировать URL, вам следует использовать метод `url`, например, в Haml:

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

Этот метод учитывает обратные прокси и маршрутизаторы Rack, если они присутствуют.

Наряду с `url` вы можете использовать `to` (смотрите пример ниже).

Перенаправление (редирект)

Вы можете перенаправить браузер пользователя с помощью метода `redirect`:

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

Любые дополнительные параметры используются по аналогии с аргументами метода `halt`:

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

Вы также можете перенаправить пользователя обратно, на страницу, с которой он пришел, с помощью `redirect back`:

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

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

Чтобы передать какие-либо параметры вместе с перенаправлением, либо добавьте их в строку запроса:

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

либо используйте сессию:

```
enable :sessions

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

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

Управление кэшированием

Установка корректных заголовков — основа правильного HTTP кэширования.

Вы можете легко выставить заголовок Cache-Control таким образом:

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

Совет: задавайте кэширование в `before`-фильтре:

```
before do
```

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

Если вы используете метод `expires` для задания соответствующего заголовка, то `Cache-Control` будет выставлен автоматически:

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

Чтобы как следует использовать кэширование, вам следует подумать об использовании `etag` или `last_modified`. Рекомендуется использовать эти методы-помощники до выполнения ресурсоемких вычислений, так как они немедленно отправят ответ клиенту, если текущая версия уже есть в их кэше:

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

Также вы можете использовать weak ETag:

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

Эти методы-помощники не станут ничего кэшировать для вас, но они дадут необходимую информацию для вашего кэша. Если вы ищете легкое решение для кэширования, попробуйте rack-cache:

```
require 'rack/cache'
require 'sinatra'

use Rack::Cache

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

Используйте опцию `:static_cache_control` (см. ниже), чтобы добавить заголовок `Cache-Control` к статическим файлам.

В соответствии с RFC 2616 ваше приложение должно вести себя по-разному, когда заголовки If-Match или If-None-Match имеют значение `*`, в зависимости от того, существует или нет запрашиваемый ресурс. Sinatra предполагает, что ресурсы, к которым обращаются с помощью безопасных (GET) и идемпотентных (PUT) методов, уже существуют, а остальные ресурсы (к которым обращаются, например, с помощью POST) считает новыми. Вы можете изменить данное поведение с помощью опции `:new_resource`:

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

Если вы хотите использовать weak ETag, задайте опцию `:kind`:

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

Отправка файлов

Для отправки файлов пользователю вы можете использовать метод `send_file`:

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

Этот метод имеет несколько опций:

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

Возможные опции:

filename
: имя файла, по умолчанию: реальное имя файла.

last_modified
: значение для заголовка Last-Modified, по умолчанию: mtime (время изменения) файла.

type
: тип файла, по умолчанию: определяется по расширению файла.

disposition
: используется для заголовка Content-Disposition, возможные значения: `nil` (по умолчанию), `:attachment` и `:inline`.

length
: значения для заголовка Content-Length, по умолчанию: размер файла.

status
: Код ответа. Полезно, когда отдается статический файл в качестве страницы с сообщением об ошибке.

Этот метод будет использовать возможности Rack сервера для отправки файлов, если они доступны, в противном случае будет напрямую отдавать файл из Ruby процесса. Метод `send_file` также обеспечивает автоматическую обработку частичных (range) запросов с помощью Sinatra.

Доступ к объекту запроса

Объект входящего запроса доступен на уровне обработки запроса (в фильтрах, маршрутах, обработчиках ошибок) с помощью `request` метода:

```
# приложение запущено на 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                # тело запроса, посланное клиентом (см. ниже)
  request.scheme              # "http"
  request.script_name         # "/example"
  request.path_info           # "/foo"
  request.port                # 80
  request.request_method      # "GET"
  request.query_string        # ""
  request.content_length      # длина тела запроса
  request.media_type          # медиатип тела запроса
  request.host                # "example.com"
  request.get?                # true (есть аналоги для других методов HTTP)
  request.form_data?          # false
  request["some_param"]       # значение параметра some_param. Шорткат для хеша params
  request.referrer            # источник запроса клиента либо '/'
  request.user_agent          # user agent (используется для :agent условия)
  request.cookies             # хеш, содержащий cookies браузера
  request.xhr?                # является ли запрос ajax запросом?
  request.url                 # "http://example.com/example/foo"
  request.path                # "/example/foo"
  request.ip                  # IP-адрес клиента
  request.secure?             # false (true, если запрос сделан через SSL)
  request.forwarded?          # true (если сервер работает за обратным прокси)
  request.env                 # "сырой" env хеш, полученный Rack
end
```

Некоторые опции, такие как `script_name` или `path_info`, доступны для изменения:

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

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

`request.body` является IO или StringIO объектом:

```
post "/api" do
  request.body.rewind  # в случае, если кто-то уже прочитал тело запроса
  data = JSON.parse request.body.read
  "Hello #{data['name']}"
end
```

Вложения

Вы можете использовать метод `attachment`, чтобы сказать браузеру, что ответ сервера должен быть сохранен на диск, а не отображен:

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

Вы также можете указать имя файла:

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

Работа со временем и датами

Sinatra предлагает метод-помощник `time_for`, который из заданного значения создает объект Time. Он также может конвертировать `DateTime`, `Date` и подобные классы:

```
get '/' do
  pass if Time.now > time_for('Dec 23, 2012')
  "still time"
end
```

Этот метод используется внутри Sinatra методами `expires`, `last_modified` и им подобными. Поэтому вы легко можете расширить функционал этих методов, переопределив `time_for` в своем приложении:

```
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
```

Поиск шаблонов

Для поиска шаблонов и их последующего рендеринга используется метод `find_template`:

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

Это не слишком полезный пример. Зато полезен тот факт, что вы можете переопределить этот метод, чтобы использовать свой собственный механизм поиска. Например, если вы хотите, чтобы можно было использовать несколько директорий с шаблонами:

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

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

Другой пример, в котором используются разные директории для движков рендеринга:

```
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
```

Вы можете легко вынести этот код в расширение и поделиться им с остальными!

Заметьте, что `find_template` не проверяет, существует ли файл на самом деле, а вызывает заданный блок для всех возможных путей. Дело тут не в производительности, дело в том, что `render` вызовет `break`, как только файл не будет найден. Содержимое и местонахождение шаблонов будет закэшировано, если приложение запущено не в режиме разработки (`set :environment, :development`). Вы должны помнить об этих нюансах, если пишите по-настоящему “сумасшедший” метод.

Конфигурация

Этот блок исполняется один раз при старте в любом окружении, режиме (environment):

```
configure do
  # задание одной опции
  set :option, 'value'

  # устанавливаем несколько опций
  set :a => 1, :b => 2

  # то же самое, что и `set :option, true`
  enable :option
```

```
  # то же самое, что и `set :option, false`
  disable :option

  # у вас могут быть "динамические" опции с блоками
  set(:css_dir) { File.join(views, 'css') }
end
```

Будет запущено, когда окружение (RACK_ENV переменная) `:production`:

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

Будет запущено, когда окружение `:production` или `:test`:

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

Вы можете получить доступ к этим опциям с помощью `settings`:

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

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

Настройка защиты от атак

Sinatra использует Rack::Protection для защиты приложения от простых атак. Вы можете легко выключить эту защиту (что сделает ваше приложение чрезвычайно уязвимым):

```
disable :protection
```

Чтобы пропустить какой-либо уровень защиты, передайте хеш опций в параметр `protection`:

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

Вы также можете отключить сразу несколько уровней защиты:

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

Доступные настройки

absolute_redirects
: если отключено, то Sinatra будет позволять использование относительных перенаправлений, но при этом перестанет соответствовать RFC 2616 (HTTP 1.1), который разрешает только абсолютные перенаправления.
Включайте эту опцию, если ваше приложение работает за обратным прокси, который настроен не совсем корректно. Обратите внимание, метод `url` все равно будет генерировать абсолютные URL, если вы не передадите `false` вторым аргументом.
Отключено по умолчанию.

add_charsets
: mime-типы, к которым метод `content_type` будет автоматически добавлять информацию о кодировке. Вам следует добавлять значения к этой опции вместо ее переопределения: `settings.add_charsets << "application/foobar"`

app_file
: путь к главному файлу приложения, используется для нахождения корневой директории проекта, директорий с шаблонами и статическими файлами, вложенных шаблонов.

bind
: используемый IP-адрес (по умолчанию: 0.0.0.0). Используется только встроенным сервером.

default_encoding
: кодировка, если неизвестна (по умолчанию: `"utf-8"`).

dump_errors
: отображать ошибки в логе.

environment
: текущее окружение, по умолчанию, значение `ENV['RACK_ENV']` или `"development"`, если `ENV['RACK_ENV']` недоступна.

logging
: использовать логер.

lock
: создает блокировку для каждого запроса, которая гарантирует обработку только одного запроса в текущий момент времени в Ruby процессе.
Включайте, если ваше приложение не потоко-безопасно (thread-safe). Отключено по умолчанию.

method_override
: использовать "магический" параметр `_method`, для поддержки PUT/DELETE форм в браузерах, которые не поддерживают эти методы.

port
: порт, на котором будет работать сервер. Используется только встроенным сервером.

prefixed_redirects
: добавлять или нет параметр `request.script_name` к редиректам, если не задан абсолютный путь. Таким образом, `redirect '/foo'` будет вести себя как `redirect to('/foo')`. Отключено по умолчанию.

protection
: включена или нет защита от атак. Смотрите секцию выше.

public_dir
: Алиас для `public_folder`.

public_folder
: путь к директории, откуда будут раздаваться статические файлы. Используется, только если

включена раздача статических файлов (см. опцию `static` ниже).

reload_templates
: перезагружать или нет шаблоны на каждый запрос. Включено в режиме разработки.

root
: путь к корневой директории проекта.

raise_errors
: выбрасывать исключения (будет останавливать приложение). По умолчанию включено только в окружении `test`.

run
: если включено, Sinatra будет самостоятельно запускать веб-сервер. Не включайте, если используете rackup или аналогичные средства.

running
: работает ли сейчас встроенный сервер? Не меняйте эту опцию!

server
: сервер или список серверов, которые следует использовать в качестве встроенного сервера. По умолчанию: `['thin', 'mongrel', 'webrick']`, порядок задает приоритет.

sessions
: включить сессии на основе кук (cookie) на базе `Rack::Session::Cookie`. Смотрите секцию "Использование сессий" выше.

show_exceptions
: показывать исключения/стек вызовов (stack trace) в браузере. По умолчанию включено только в окружении `development`.
: Может быть установлено в `:after_handler` для запуска специфичной для приложения обработки ошибок, перед показом трассировки стека в браузере.

static
: должна ли Sinatra осуществлять раздачу статических файлов.
: Отключите, когда используете какой-либо веб-сервер для этой цели.
: Отключение значительно улучшит производительность приложения.
: По умолчанию включено в классических и отключено в модульных приложениях.

static_cache_control
: когда Sinatra отдает статические файлы, используйте эту опцию, чтобы добавить им заголовок `Cache-Control`. Для этого используется метод-помощник `cache_control`. По умолчанию отключено.
: Используйте массив, когда надо задать несколько значений: `set :static_cache_control, [:public, :max_age => 300]`

threaded
: если включено, то Thin будет использовать `EventMachine.defer` для обработки запросов.

views
: путь к директории с шаблонами.

Режим, окружение

Есть 3 предопределенных режима, окружения: `"development"`, `"production"` и `"test"`. Режим может быть задан через переменную окружения `RACK_ENV`. Значение по умолчанию — `"development"`. В этом режиме работы все шаблоны перезагружаются между запросами. А также задаются специальные обработчики `not_found` и `error`, чтобы вы могли увидеть стек вызовов. В

окружениях `"production"` и `"test"` шаблоны по умолчанию кэшируются.

Для запуска приложения в определенном окружении используйте ключ `-e`

```
ruby my_app.rb -e [ENVIRONMENT]
```

Вы можете использовать предопределенные методы `development?`, `test?` и +production?, чтобы определить текущее окружение.

Обработка ошибок

Обработчики ошибок исполняются в том же контексте, что и маршруты, и `before`-фильтры, а это означает, что всякие прелести вроде `haml`, `erb`, `halt` и т.д. доступны и им.

Not Found

Когда выброшено исключение `Sinatra::NotFound`, или кодом ответа является 404, то будет вызван `not_found` обработчик:

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

Ошибки

Обработчик ошибок `error` будет вызван, когда исключение выброшено из блока маршрута, либо из фильтра. Объект-исключение доступен как переменная `sinatra.error` в Rack:

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

Конкретные ошибки:

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

Тогда, если это произошло:

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

То вы получите:

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

Также вы можете установить обработчик ошибок для кода состояния HTTP:

```
error 403 do
  'Access forbidden'
end

get '/secret' do
  403
end
```

Либо набора кодов:

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

Sinatra устанавливает специальные `not_found` и `error` обработчики, когда приложение запущено в режиме разработки (окружение `:development`).

Rack “прослойки”

Sinatra использует Rack, минимальный стандартный интерфейс для веб-фреймворков на Ruby. Одной из самых интересных для разработчиков возможностей Rack является поддержка “прослоек” (“middleware”) — компонентов, находящихся “между” сервером и вашим приложением, которые отслеживают и/или манипулируют HTTP запросами/ответами для предоставления различной функциональности.

В Sinatra очень просто использовать такие “прослойки” с помощью метода `use`:

```
require 'sinatra'
require 'my_custom_middleware'

use Rack::Lint
use MyCustomMiddleware

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

Семантика `use` идентична той, что определена для Rack::Builder DSL (чаще всего используется в rackup файлах). Например, метод `use` принимает как множественные переменные, так и блоки:

```
use Rack::Auth::Basic do |username, password|
  username == 'admin' && password == 'secret'
```

```
end
```

Rack распространяется с различными стандартными “прослойками” для логирования, отладки, маршрутизации URL, аутентификации, обработки сессий. Sinatra использует многие из этих компонентов автоматически, основываясь на конфигурации, чтобы вам не приходилось подключать (`use`) их вручную.

Вы можете найти полезные прослойки в rack, rack-contrib, CodeRack или в Rack wiki.

Тестирование

Тесты для Sinatra приложений могут быть написаны с помощью библиотек, фреймворков, поддерживающих тестирование Rack. Rack::Test рекомендован:

```
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
```

Sinatra::Base — “прослойки”, библиотеки и модульные приложения

Описание своего приложения самым простейшим способом (с помощью DSL верхнего уровня, классический стиль) отлично работает для крохотных приложений. В таких случаях используется конфигурация, рассчитанная на микро-приложения (единственный файл приложения, `./public` и `./views` директории, логирование, страница информации об

исключении и т.д.). Тем не менее, такой метод имеет множество недостатков при создании компонентов, таких как Rack middleware (“прослоек”), Rails metal, простых библиотек с серверными компонентами, расширений Sinatra. И тут на помощь приходит `Sinatra::Base`:

```
require 'sinatra/base'

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

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

Методы, доступные `Sinatra::Base` подклассам идентичны тем, что доступны приложениям в DSL верхнего уровня. Большинство таких приложений могут быть конвертированы в `Sinatra::Base` компоненты с помощью двух модификаций:

- Вы должны подключать `sinatra/base` вместо `sinatra`, иначе все методы, предоставляемые Sinatra, будут импортированы в глобальное пространство имен.
- Поместите все маршруты, обработчики ошибок, фильтры и опции в подкласс `Sinatra::Base`.

`Sinatra::Base` — это чистый лист. Большинство опций, включая встроенный сервер, по умолчанию отключены. Смотрите Опции и конфигурация для детальной информации об опциях и их поведении.

Модульные приложения против классических

Вопреки всеобщему убеждению, в классическом стиле (самом простом) нет ничего плохого. Если этот стиль подходит вашему приложению, вы не обязаны переписывать его в модульное приложение.

Основным недостатком классического стиля является тот факт, что у вас может быть только одно приложение Sinatra на один процесс Ruby. Если вы планируете использовать больше, переключайтесь на модульный стиль. Вы можете смело смешивать модульный и классический стили.

Переходя с одного стиля на другой, примите во внимание следующие изменения в настройках:

```
Опция              Классический          Модульный

app_file           файл с приложением    файл с подклассом Sinatra::Base
run                $0 == app_file        false
logging            true                  false
method_override    true                  false
inline_templates   true                  false
static             true                  false
```

Запуск модульных приложений

Есть два общепринятых способа запускать модульные приложения: запуск напрямую с помощью `run!`:

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

class MyApp < Sinatra::Base
  # ... здесь код приложения ...

  # запускаем сервер, если исполняется текущий файл
  run! if app_file == $0
end
```

Затем:

```
ruby my_app.rb
```

Или с помощью конфигурационного файла `config.ru`, который позволяет использовать любой Rack-совместимый сервер приложений.

```
# config.ru
require './my_app'
run MyApp
```

Запускаем:

```
rackup -p 4567
```

Запуск классических приложений с config.ru

Файл приложения:

```
# app.rb
require 'sinatra'

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

И соответствующий `config.ru`:

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

Когда использовать config.ru?

Вот несколько причин, по которым вы, возможно, захотите использовать `config.ru`:

- вы хотите разворачивать свое приложение на различных Rack-совместимых серверах (Passenger, Unicorn, Heroku, ...);
- вы хотите использовать более одного подкласса `Sinatra::Base`;
- вы хотите использовать Sinatra только в качестве “прослойки” Rack.

Совсем необязательно переходить на использование `config.ru` лишь потому, что вы стали использовать модульный стиль приложения. И необязательно использовать модульный стиль, чтобы запускать приложение с помощью `config.ru`.

Использование Sinatra в качестве “прослойки”

Не только сама Sinatra может использовать “прослойки” Rack, но и любое Sinatra приложение само может быть добавлено к любому Rack endpoint в качестве “прослойки”. Этим endpoint (конечной точкой) может быть другое Sinatra приложение, или приложение, основанное на Rack (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
  # "прослойка" будет запущена перед фильтрами
  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
```

Создание приложений “на лету”

Иногда требуется создавать Sinatra приложения “на лету” (например, из другого приложения). Это возможно с помощью `Sinatra.new`:

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

Этот метод может принимать аргументом приложение, от которого следует наследоваться:

```
# config.ru
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
```

Это особенно полезно для тестирования расширений Sinatra и при использовании Sinatra внутри вашей библиотеки.

Благодаря этому, использовать Sinatra как “прослойку” очень просто:

```
require 'sinatra/base'

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

run RailsProject::Application
```

Области видимости и привязка

Текущая область видимости определяет методы и переменные, доступные в данный момент.

Область видимости приложения / класса

Любое Sinatra приложение соответствует подклассу `Sinatra::Base`. Если вы используете DSL верхнего уровня (`require 'sinatra'`), то этим классом будет `Sinatra::Application`, иначе это

будет подкласс, который вы создали вручную. На уровне класса вам будут доступны такие методы, как `get` или `before`, но вы не сможете получить доступ к объектам `request` или `session`, так как существует только один класс приложения для всех запросов.

Опции, созданные с помощью `set`, являются методами уровня класса:

```
class MyApp < Sinatra::Base
  # Я в области видимости приложения!
  set :foo, 42
  foo # => 42

  get '/foo' do
    # Я больше не в области видимости приложения!
  end
end
```

У вас будет область видимости приложения внутри:

- тела вашего класса приложения;
- методов, определенных расширениями;
- блока, переданного в `helpers`;
- блоков, использованных как значения для `set`;
- блока, переданного в `Sinatra.new`.

Вы можете получить доступ к объекту области видимости (классу приложения) следующими способами:

- через объект, переданный блокам конфигурации (`configure { |c| ... }`);
- `settings` внутри области видимости запроса.

Область видимости запроса/экземпляра

Для каждого входящего запроса будет создан новый экземпляр вашего приложения, и все блоки обработчика будут запущены в этом контексте. В этой области видимости вам доступны `request` и `session` объекты, вызовы методов рендеринга, такие как `erb` или `haml`. Вы можете получить доступ к области видимости приложения из контекста запроса, используя метод-помощник `settings`:

```
class MyApp < Sinatra::Base
  # Я в области видимости приложения!
  get '/define_route/:name' do
    # Область видимости запроса '/define_route/:name'
    @value = 42

    settings.get("/#{params[:name]}") do
      # Область видимости запроса "/#{params[:name]}"
      @value # => nil (другой запрос)
    end

    "Route defined!"
  end
```

```
end
```

У вас будет область видимости запроса в:

- get/head/post/put/delete/options блоках;
- before/after фильтрах;
- методах-помощниках;
- шаблонах/отображениях.

Область видимости делегирования

Область видимости делегирования просто перенаправляет методы в область видимости класса. Однако, она не полностью ведет себя как область видимости класса, так как у вас нет привязки к классу. Только методы, явно помеченные для делегирования, будут доступны, а переменных/состояний области видимости класса не будет (иначе говоря, у вас будет другой `self` объект). Вы можете непосредственно добавить методы делегирования, используя `Sinatra::Delegator.delegate :method_name`.

У вас будет контекст делегирования внутри:

- привязки верхнего уровня, если вы сделали `require 'sinatra'`;
- объекта, расширенного с помощью `Sinatra::Delegator`.

Посмотрите сами в код: вот примесь Sinatra::Delegator расширяет главный объект.

Командная строка

Sinatra приложения могут быть запущены напрямую:

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

Опции включают:

```
-h # раздел помощи
-p # указание порта (по умолчанию 4567)
-o # указание хоста (по умолчанию 0.0.0.0)
-e # указание окружения, режима (по умолчанию development)
-s # указание rack сервера/обработчика (по умолчанию thin)
-x # включить мьютекс-блокировку (по умолчанию выключена)
```

Системные требования

Следующие версии Ruby официально поддерживаются:

Ruby 1.8.7
: 1.8.7 полностью поддерживается, тем не менее, если вас ничто не держит на этой версии, рекомендуем обновиться до 1.9.2 или перейти на JRuby или Rubinius. Поддержка 1.8.7 не будет прекращена до выхода Sinatra 2.0 и Ruby 2.0, разве что в случае релиза 1.8.8 (что маловероятно). Но даже тогда, возможно, поддержка не будет прекращена. **Ruby 1.8.6 больше не поддерживается.** Если вы хотите использовать 1.8.6, откатитесь до Sinatra 1.2, которая будет получать все исправления ошибок до тех пор, пока не будет выпущена Sinatra 1.4.0.

Ruby 1.9.2
: 1.9.2 полностью поддерживается и рекомендована к использованию. Не используйте 1.9.2p0, известно, что эта версия очень нестабильна при использовании Sinatra. Эта версия будет поддерживаться по крайней мере до выхода Ruby 1.9.4/2.0, а поддержка последней версии 1.9 будет осуществляться до тех пор, пока она поддерживается командой разработчиков Ruby.

Ruby 1.9.3
: 1.9.3 полностью поддерживается. Заметьте, что переход на 1.9.3 с ранних версий сделает недействительными все сессии.

Rubinius
: Rubinius официально поддерживается (Rubinius >= 1.2.4), всё, включая все языки шаблонов, работает. Предстоящий релиз 2.0 также поддерживается.

JRuby
: JRuby официально поддерживается (JRuby >= 1.6.5). Нет никаких проблем с использованием альтернативных шаблонов. Тем не менее, если вы выбираете JRuby, то, пожалуйста, посмотрите на JRuby Rack-серверы, так как Thin не поддерживается полностью на JRuby. Поддержка расширений на C в JRuby все еще экспериментальная, что на данный момент затрагивает только RDiscount, Redcarpet и RedCloth.

Мы также следим за предстоящими к выходу версиями Ruby.

Следующие реализации Ruby не поддерживаются официально, но известно, что на них запускается Sinatra:

- старые версии JRuby и Rubinius;
- Ruby Enterprise Edition;
- MacRuby, Maglev, IronRuby;
- Ruby 1.9.0 и 1.9.1 (настоятельно не рекомендуются к использованию).

То, что версия официально не поддерживается, означает, что, если что-то не работает на этой версии, а на поддерживаемой работает — это не наша проблема, а их.

Мы также запускаем наши CI-тесты на версии Ruby, находящейся в разработке (предстоящей 2.0.0), и на 1.9.4, но мы не можем ничего гарантировать, так как они находятся в разработке. Предполагается, что 1.9.4p0 и 2.0.0p0 будут поддерживаться.

Sinatra должна работать на любой операционной системе, в которой есть одна из указанных выше версий Ruby.

Пока невозможно запустить Sinatra на Cardinal, SmallRuby, BlueRuby и на любой версии Ruby до 1.8.7.

На острие

Если вы хотите использовать самый последний код Sinatra, не бойтесь запускать свое приложение вместе с кодом из master ветки Sinatra, она весьма стабильна.

Мы также время от времени выпускаем предварительные версии, так что вы можете делать так:

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

Чтобы воспользоваться некоторыми самыми последними возможностями.

С помощью Bundler

Если вы хотите запускать свое приложение с последней версией Sinatra, то рекомендуем использовать Bundler.

Сначала установите Bundler, если у вас его еще нет:

```
gem install bundler
```

Затем создайте файл `Gemfile` в директории вашего проекта:

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

# другие зависимости
gem 'haml'                    # например, если используете haml
gem 'activerecord', '~> 3.0'  # может быть, вам нужен и ActiveRecord 3.x
```

Обратите внимание, вам нужно будет указывать все зависимости вашего приложения в этом файле. Однако, непосредственные зависимости Sinatra (Rack и Tilt) Bundler автоматически скачает и добавит.

Теперь вы можете запускать свое приложение так:

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

Вручную

Создайте локальный клон репозитория и запускайте свое приложение с `sinatra/lib` директорией в `$LOAD_PATH`:

```
cd myapp
git clone git://github.com/sinatra/sinatra.git
ruby -Isinatra/lib myapp.rb
```

Чтобы обновить исходники Sinatra:

```
cd myapp/sinatra
git pull
```

Установка глобально

Вы можете самостоятельно собрать gem:

```
git clone git://github.com/sinatra/sinatra.git
cd sinatra
rake sinatra.gemspec
rake install
```

Если вы устанавливаете пакеты (gem) от пользователя root, то вашим последним шагом должна быть команда

```
sudo rake install
```

Версии

Sinatra использует Semantic Versioning, SemVer и SemVerTag.

Дальнейшее чтение

- Веб-сайт проекта — Дополнительная документация, новости и ссылки на другие ресурсы.
- Участие в проекте — Обнаружили баг? Нужна помощь? Написали патч?
- Слежение за проблемами/ошибками
- Twitter
- Группы рассылки
- #sinatra на http://freenode.net
- Sinatra Book учебник и сборник рецептов
- Sinatra Recipes сборник рецептов
- API документация к последнему релизу или текущему HEAD на http://rubydoc.info
- Сервер непрерывной интеграции

Sinatra

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

Introducción

Atención: Este documento es una traducción de la versión en inglés y puede estar desactualizado.

Sinatra es un DSL para crear aplicaciones web rápidamente en Ruby con un mínimo esfuerzo:

```
# miapp.rb
require 'sinatra'

get '/' do
  'Hola mundo!'
end
```

Instalá el gem y corré la aplicación con:

```
gem install sinatra
ruby miapp.rb
```

Podés verla en: http://localhost:4567

Es recomendable además ejecutar `gem install thin`, ya que Sinatra lo va a utilizar cuando esté disponible.

Rutas

En Sinatra, una ruta está compuesta por un método HTTP y un patrón de una URL. Cada ruta se asocia con un bloque:

```
get '/' do
  .. mostrar algo ..
end
```

```
post '/' do
  .. crear algo ..
end

put '/' do
  .. reemplazar algo ..
end

patch '/' do
  .. modificar algo ..
end

delete '/' do
  .. aniquilar algo ..
end

options '/' do
  .. informar algo ..
end
```

Las rutas son comparadas en el orden en el que son definidas. La primera ruta que coincide con la petición es invocada.

Los patrones de las rutas pueden incluir parámetros nombrados, accesibles a través del hash `params`:

```
get '/hola/:nombre' do
  # coincide con "GET /hola/foo" y "GET /hola/bar"
  # params[:nombre] es 'foo' o 'bar'
  "Hola #{params[:nombre]}"
end
```

También podés acceder a los parámetros nombrados usando parámetros de bloque:

```
get '/hola/:nombre' do |n|
  # coincide con "GET /hola/foo" y "GET /hola/bar"
  # params[:nombre] es 'foo' o 'bar'
  # n almacena params[:nombre]
  "Hola #{n}"
end
```

Los patrones de ruta también pueden incluir parámetros splat (o wildcard), accesibles a través del arreglo `params[:splat]`:

```
get '/decir/*/al/*' do
  # coincide con /decir/hola/al/mundo
  params[:splat] # => ["hola", "mundo"]
end

get '/descargar/*.*' do
  # coincide con /descargar/path/al/archivo.xml
  params[:splat] # => ["path/al/archivo", "xml"]
end
```

O, con parámetros de bloque:

```
get '/descargar/*.*' do |path, ext|
  [path, ext] # => ["path/al/archivo", "xml"]
end
```

Rutas con Expresiones Regulares:

```
get %r{/hola/([\w]+)} do
  "Hola, #{params[:captures].first}"
end
```

O con un parámetro de bloque:

```
get %r{/hola/([\w]+)} do |c|
  "Hola, #{c}"
end
```

Los patrones de ruta pueden contener parámetros opcionales:

```
get '/posts.?:formato?' do
  # coincide con "GET /posts" y además admite cualquier extensión, por
  # ejemplo, "GET /posts.json", "GET /posts.xml", etc.
end
```

A propósito, a menos que desactives la protección para el ataque *path traversal* (ver más abajo), el path de la petición puede ser modificado antes de que se compare con los de tus rutas.

Condiciones

Las rutas pueden incluir una variedad de condiciones de selección, como por ejemplo el user agent:

```
get '/foo', :agent => /Songbird (\d\.\d)[\d\/]*?/ do
  "Estás usando la versión de Songbird #{params[:agent][0]}"
end

get '/foo' do
  # Coincide con browsers que no sean songbird
end
```

Otras condiciones disponibles son `host_name` y `provides`:

```
get '/', :host_name => /^admin\./ do
  "Área de Administración, Acceso denegado!"
end

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

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

Podés definir tus propias condiciones fácilmente:

```
set(:probabilidad) { |valor| condition { rand <= valor } }

get '/gana_un_auto', :probabilidad => 0.1 do
  "Ganaste!"
end

get '/gana_un_auto' do
  "Lo siento, perdiste."
end
```

Si tu condición acepta más de un argumento, podés pasarle un arreglo. Al definir la condición puede resultarte conveniente utilizar el operador splat en la lista de parámetros:

```
set(:autorizar) do |*roles|   # <- mirá el splat
  condition do
    unless sesion_iniciada? && roles.any? {|rol| usuario_actual.tiene_rol? rol }
      redirect "/iniciar_sesion/", 303
    end
  end
end

get "/mi/cuenta/", :autorizar => [:usuario, :administrador] do
  "Detalles de mi cuenta"
end

get "/solo/administradores/", :autorizar => :administrador do
  "Únicamente para administradores!"
end
```

Valores de Retorno

El valor de retorno de un bloque de ruta que determina al menos el cuerpo de la respuesta que se le pasa al cliente HTTP o al siguiente middleware en la pila de Rack. Lo más común es que sea un string, como en los ejemplos anteriores. Sin embargo, otros valores también son aceptados.

Podés devolver cualquier objeto que sea una respuesta Rack válida, un objeto que represente el cuerpo de una respuesta Rack o un código de estado HTTP:

De esa manera podemos, por ejemplo, implementar fácilmente un streaming:

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

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

Comparadores de Rutas Personalizados

Como se mostró anteriormente, Sinatra permite utilizar Strings y expresiones regulares para definir las rutas. Sin embargo, la cosa no termina ahí. Podés definir tus propios comparadores muy fácilmente:

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

  def initialize(excepto)
    @excepto  = excepto
    @capturas = Match.new([])
  end

  def match(str)
    @capturas unless @excepto === str
  end
end

def cualquiera_menos(patron)
  PatronCualquieraMenos.new(patron)
end

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

Tené en cuenta que el ejemplo anterior es un poco rebuscado. Un resultado similar puede conseguirse más sencillamente:

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

O, usando un lookahead negativo:

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

Archivos Estáticos

Los archivos estáticos son servidos desde el directorio público `./public`. Podés especificar una ubicación diferente ajustando la opción `:public_folder`:

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

Notá que el nombre del directorio público no está incluido en la URL. Por ejemplo, el archivo `./public/css/style.css` se accede a través de `http://ejemplo.com/css/style.css`.

Usá la configuración `:static_cache_control` para agregar el encabezado `Cache-Control` (ver la sección de configuración para más detalles).

Vistas / Plantillas

Cada lenguaje de plantilla se expone a través de un método de renderizado que lleva su nombre. Estos métodos simplemente devuelven un string:

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

Renderiza `views/index.erb`.

En lugar del nombre de la plantilla podés proporcionar directamente el contenido de la misma:

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

Los métodos de renderizado, aceptan además un segundo argumento, el hash de opciones:

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

Renderiza `views/index.erb` embebido en `views/post.erb` (por defecto, la plantilla `:index` es embebida en `views/layout.erb` siempre y cuando este último archivo exista).

Cualquier opción que Sinatra no entienda le será pasada al motor de renderizado de la plantilla:

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

Además podés definir las opciones para un lenguaje de plantillas de forma general:

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

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

Las opciones pasadas al método de renderizado tienen precedencia sobre las definidas mediante `set`.

Opciones disponibles:

locals
: Lista de variables locales pasadas al documento. Resultan muy útiles cuando se combinan con parciales. Ejemplo: `erb "<%= foo %>", :locals => {:foo => "bar"}`

default_encoding
: Encoding utilizado cuando el de un string es dudoso. Por defecto toma el valor de `settings.default_encoding`.

views
: Directorio desde donde se cargan las vistas. Por defecto toma el valor de `settings.views`.

layout
: Si es `true` o `false` indica que se debe usar, o no, un layout, respectivamente. También puede ser un símbolo que especifique qué plantilla usar. Ejemplo: `erb :index, :layout => !request.xhr?`

content_type
: Content-Type que produce la plantilla. El valor por defecto depende de cada lenguaje de plantillas.

scope
: Ámbito en el que se renderiza la plantilla. Por defecto utiliza la instancia de la aplicación. Tené en cuenta que si cambiás esta opción las variables de instancia y los helpers van a dejar de estar disponibles.

layout_engine
: Motor de renderizado de plantillas que usa para el layout. Resulta conveniente para lenguajes que no soportan layouts. Por defecto toma el valor del motor usado para renderizar la plantilla. Ejemplo: `set :rdoc, :layout_engine => :erb`
Se asume que las plantillas están ubicadas directamente bajo el directorio `./views`. Para usar un directorio de vistas diferente: `set :views, settings.root + '/plantillas'`
Es importante acordarse que siempre tenés que referenciar a las plantillas con símbolos, incluso cuando se encuentran en un subdirectorio (en este caso tenés que usar: `:'subdir/plantilla'` o `'subdir/plantilla'.to_sym`). Tenés que usar un símbolo porque los métodos de renderización van a renderizar directamente cualquier string que se les pase como argumento.

Lenguajes de Plantillas Disponibles

Algunos lenguajes tienen varias implementaciones. Para especificar que implementación usar (y para ser thread-safe), deberías requerirla antes de usarla:

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

Plantillas Haml

Dependencias	haml
Expresiones de Archivo	`.haml`
Ejemplo	`haml :index, :format => :html5`

Plantillas Erb

Dependencias	erubis o erb (incluida en Ruby)
Extensiones de Archivo	`.erb`, `.rhtml` o `.erubis` (solamente con Erubis)
Ejemplo	`erb :index`

Plantillas Builder

Dependencias	builder
Extensiones de Archivo	`.builder`
Ejemplo	`builder { |xml| xml.em "hola" }`

Además, acepta un bloque con la definición de la plantilla (ver el ejemplo).

Plantillas Nokogiri

Dependencias	nokogiri
Extensiones de Archivo	`.nokogiri`
Ejemplo	`nokogiri { |xml| xml.em "hola" }`

Además, acepta un bloque con la definición de la plantilla (ver el ejemplo).

Plantillas Sass

Dependencias	sass
Extensiones de Archivo	`.sass`
Ejemplo	`sass :stylesheet, :style => :expanded`

Plantillas SCSS

Dependencias	sass
Extensiones de Archivo	`.scss`
Ejemplo	`scss :stylesheet, :style => :expanded`

Plantillas Less

Dependencias	less
Extensiones de Archivo	`.less`
Ejemplo	`less :stylesheet`

Plantillas Liquid

Dependencias	liquid
Extensiones de Archivo	`.liquid`
Ejemplo	`liquid :index, :locals => { :clave => 'valor' }`

Como no vas a poder llamar a métodos de Ruby (excepto por `yield`) desde una plantilla Liquid, casi siempre vas a querer pasarle locales.

Plantillas Markdown

Dependencias	RDiscount, RedCarpet, BlueCloth, kramdown o maruku
Extensiones de Archivo	`.markdown`, `.mkd` y `.md`
Ejemplo	`markdown :index, :layout_engine => :erb`

No es posible llamar métodos desde markdown, ni pasarle locales. Por lo tanto, generalmente vas a usarlo en combinación con otro motor de renderizado:

```
erb :resumen, :locals => { :texto => markdown(:introduccion) }
```

Tené en cuenta que también podés llamar al método `markdown` desde otras plantillas:

```
%h1 Hola Desde Haml!
%p= markdown(:saludos)
```

Como no podés utilizar Ruby desde Markdown, no podés usar layouts escritos en Markdown. De todos modos, es posible usar un motor de renderizado para el layout distinto al de la plantilla pasando la opción `:layout_engine`.

Plantillas Textile

Dependencias	RedCloth
Extensiones de Archivo	`.textile`
Ejemplo	`textile :index, :layout_engine => :erb`

No es posible llamar métodos desde textile, ni pasarle locales. Por lo tanto, generalmente vas a usarlo en combinación con otro motor de renderizado:

```
erb :resumen, :locals => { :texto => textile(:introduccion) }
```

Tené en cuenta que también podés llamar al método `textile` desde otras plantillas:

```
%h1 Hola Desde Haml!
%p= textile(:saludos)
```

Como no podés utilizar Ruby desde Textile, no podés usar layouts escritos en Textile. De todos modos, es posible usar un motor de renderizado para el layout distinto al de la plantilla pasando la opción `:layout_engine`.

Plantillas RDoc

Dependencias	RDoc
Extensiones de Archivo	`.rdoc`
Ejemplo	`rdoc :README, :layout_engine => :erb`

No es posible llamar métodos desde rdoc, ni pasarle locales. Por lo tanto, generalmente vas a usarlo en combinación con otro motor de renderizado:

```
erb :resumen, :locals => { :texto => rdoc(:introduccion) }
```

Tené en cuenta que también podés llamar al método `rdoc` desde otras plantillas:

```
%h1 Hola Desde Haml!
%p= rdoc(:saludos)
```

Como no podés utilizar Ruby desde RDoc, no podés usar layouts escritos en RDoc. De todos modos, es posible usar un motor de renderizado para el layout distinto al de la plantilla pasando la opción `:layout_engine`.

Plantillas Radius

Dependencias	Radius
Extensiones de Archivo	`.radius`
Ejemplo	`radius :index, :locals => { :clave => 'valor' }`

Como no vas a poder llamar a métodos de Ruby (excepto por `yield`) desde una plantilla Radius, casi siempre vas a querer pasarle locales.

Plantillas Markaby

Dependencias	Markaby

Extensiones de Archivo	`.mab`
Ejemplo	`markaby { h1 "Bienvenido!" }`

Además, acepta un bloque con la definición de la plantilla (ver el ejemplo).

Plantillas RABL

Dependencias	Rabl
Extensiones de Archivo	`.rabl`
Ejemplo	`rabl :index`

Plantillas Slim

Dependencias	Slim Lang
Extensiones de Archivo	`.slim`
Ejemplo	`slim :index`

Plantillas Creole

Dependencias	Creole
Extensiones de Archivo	`.creole`
Ejemplo	`creole :wiki, :layout_engine => :erb`

No es posible llamar métodos desde creole, ni pasarle locales. Por lo tanto, generalmente vas a usarlo en combinación con otro motor de renderizado:

```
erb :resumen, :locals => { :texto => cerole(:introduccion) }
```

Tené en cuenta que también podés llamar al método `creole` desde otras plantillas:

```
%h1 Hola Desde Haml!
%p= creole(:saludos)
```

Como no podés utilizar Ruby desde Creole, no podés usar layouts escritos en Creloe. De todos modos, es posible usar un motor de renderizado para el layout distinto al de la plantilla pasando la opción `:layout_engine`.

Plantillas CoffeeScript

Dependencias	CoffeeScript y un mecanismo para ejecutar javascript
Extensiones de Archivo	`.coffee`

Ejemplo	`coffee :index`

Plantillas Stylus

Dependencias	Stylus y un mecanismo para ejecutar javascript
Extensiones de Archivo	`.styl`
Ejemplo	`stylus :index`

Plantillas Yajl

Dependencias	yajl-ruby
Extensiones de Archivo	`.yajl`
Ejemplo	`yajl :index, :locals => { :key => 'qux' }, :callback => 'present', :variable => 'resource'`

El contenido de La plantilla se evalúa como código Ruby, y la variable `json` es convertida a JSON mediante `#to_json`.

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

Las opciones `:callback` y `:variable` se pueden utilizar para decorar el objeto renderizado:

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

Plantillas WLang

Dependencias	wlang
Extensiones de Archivo	`.wlang`
Ejemplo	`wlang :index, :locals => { :clave => 'valor' }`

Como no vas a poder llamar a métodos de Ruby (excepto por `yield`) desde una plantilla WLang, casi siempre vas a querer pasarle locales.

Plantillas Embebidas

```
get '/' do
  haml '%h1 Título Hola Mundo'
end
```

Renderiza el template embebido en el string.

Accediendo a Variables en Plantillas

Las plantillas son evaluadas dentro del mismo contexto que los manejadores de ruta. Las variables de instancia asignadas en los manejadores de ruta son accesibles directamente por las plantillas:

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

O es posible especificar un Hash de variables locales explícitamente:

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

Esto es usado típicamente cuando se renderizan plantillas como parciales desde adentro de otras plantillas.

Plantillas Inline

Las plantillas pueden ser definidas al final del archivo fuente:

```
require 'rubygems'
require 'sinatra'

get '/' do
  haml :index
end

__END__

@@ layout
%html
  = yield

@@ index
%div.titulo Hola mundo!!!!!
```

NOTA: únicamente las plantillas inline definidas en el archivo fuente que requiere sinatra son cargadas automáticamente. Llamá `enable :inline_templates` explícitamente si tenés plantillas inline en otros archivos fuente.

Plantillas Nombradas

Las plantillas también pueden ser definidas usando el método top-level `template`:

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

template :index do
  '%div.titulo Hola Mundo!'
end

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

Si existe una plantilla con el nombre “layout”, va a ser usada cada vez que una plantilla es renderizada. Podés desactivar los layouts individualmente pasando `:layout => false` o globalmente con `set :haml, :layout => false`:

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

Asociando Extensiones de Archivo

Para asociar una extensión de archivo con un motor de renderizado, usá `Tilt.register`. Por ejemplo, si querés usar la extensión `tt` para las plantillas Textile, podés hacer lo siguiente:

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

Agregando Tu Propio Motor de Renderizado

Primero, registrá tu motor con Tilt, y después, creá tu método de renderizado:

```
Tilt.register :mipg, MiMotorParaPlantillaGenial

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

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

Renderiza `./views/index.mypg`. Mirá https://github.com/rtomayko/tilt para aprender más de Tilt.

Filtros

Los filtros `before` son evaluados antes de cada petición dentro del mismo contexto que las rutas. Pueden modificar la petición y la respuesta. Las variables de instancia asignadas en los filtros son accesibles por las rutas y las plantillas:

```
before do
  @nota = 'Hey!'
  request.path_info = '/foo/bar/baz'
end

get '/foo/*' do
  @nota #=> 'Hey!'
  params[:splat] #=> 'bar/baz'
end
```

Los filtros `after` son evaluados después de cada petición dentro del mismo contexto y también pueden modificar la petición y la respuesta. Las variables de instancia asignadas en los filtros `before` y en las rutas son accesibles por los filtros `after`:

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

Nota: A menos que uses el método `body` en lugar de simplemente devolver un string desde una ruta, el cuerpo de la respuesta no va a estar disponible en un filtro after, debido a que todavía no se ha generado.

Los filtros aceptan un patrón opcional, que cuando está presente causa que los mismos sean evaluados únicamente si el path de la petición coincide con ese patrón:

```
before '/protegido/*' do
  autenticar!
end

after '/crear/:slug' do |slug|
  session[:ultimo_slug] = slug
end
```

Al igual que las rutas, los filtros también pueden aceptar condiciones:

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

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

Ayudantes

Usá el método top-level *helpers* para definir métodos ayudantes que pueden ser utilizados dentro de los manejadores de rutas y las plantillas:

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

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

Por cuestiones organizativas, puede resultar conveniente organizar los métodos ayudantes en distintos módulos:

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

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

helpers FooUtils, BarUtils
```

El efecto de utilizar *helpers* de esta manera es el mismo que resulta de incluir los módulos en la clase de la aplicación.

Usando Sesiones

Una sesión es usada para mantener el estado a través de distintas peticiones. Cuando están activadas, proporciona un hash de sesión para cada sesión de usuario:

```
enable :sessions

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

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

Tené en cuenta que `enable :sessions` guarda todos los datos en una cookie, lo cual no es siempre deseable (guardar muchos datos va a incrementar el tráfico, por citar un ejemplo). Podés usar cualquier middleware Rack para manejar sesiones, de la misma manera que usarías cualquier otro middleware, pero con la salvedad de que *no* tenés que llamar a `enable :sessions`:

```
use Rack::Session::Pool, :expire_after => 2592000

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

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

Para incrementar la seguridad, los datos de la sesión almacenados en la cookie son firmados con un secreto de sesión. Este secreto, es generado aleatoriamente por Sinatra. De cualquier manera, hay que tener en cuenta que cada vez que inicies la aplicación se va a generar uno nuevo. Así, si querés que todas las instancias de tu aplicación compartan un único secreto, tenés que definirlo vos:

```
set :session_secret, 'super secreto'
```

Si necesitás una configuración más específica, `sessions` acepta un Hash con opciones:

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

Interrupción

Para detener inmediatamente una petición dentro de un filtro o una ruta usá:

```
halt
```

También podés especificar el estado:

```
halt 410
```

O el cuerpo:

```
halt 'esto va a ser el cuerpo'
```

O los dos:

```
halt 401, 'salí de acá!'
```

Con cabeceras:

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

Obviamente, es posible utilizar `halt` con una plantilla:

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

Paso

Una ruta puede pasarle el procesamiento a la siguiente ruta que coincida con la petición usando `pass`:

```
get '/adivina/:quien' do
  pass unless params[:quien] == 'Franco'
  'Adivinaste!'
end

get '/adivina/*' do
  'Erraste!'
end
```

Se sale inmediatamente del bloque de la ruta y se le pasa el control a la siguiente ruta que coincida. Si no coincide ninguna ruta, se devuelve 404.

Ejecutando Otra Ruta

Cuando querés obtener el resultado de la llamada a una ruta, `pass` no te va a servir. Para lograr esto, podés usar `call`:

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

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

Notá que en el ejemplo anterior, es conveniente mover `"bar"` a un helper, y llamarlo desde `/foo` y `/bar`. Así, vas a simplificar las pruebas y a mejorar el rendimiento.

Si querés que la petición se envíe a la misma instancia de la aplicación en lugar de a otra, usá `call!` en lugar de `call`.

En la especificación de Rack podés encontrar más información sobre `call`.

Asignando el Código de Estado, los Encabezados y el Cuerpo de una Respuesta

Es posible, y se recomienda, asignar el código de estado y el cuerpo de una respuesta con el valor de retorno de una ruta. De cualquier manera, en varios escenarios, puede que sea conveniente asignar

el cuerpo en un punto arbitrario del flujo de ejecución con el método `body`. A partir de ahí, podés usar ese mismo método para acceder al cuerpo de la respuesta:

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

after do
  puts body
end
```

También es posible pasarle un bloque a `body`, que será ejecutado por el Rack handler (podés usar esto para implementar streaming, mirá “Valores de retorno”).

De manera similar, también podés asignar el código de estado y encabezados:

```
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
```

También, al igual que `body`, tanto `status` como `headers` pueden utilizarse para obtener sus valores cuando no se les pasa argumentos.

Streaming De Respuestas

A veces vas a querer empezar a enviar la respuesta a pesar de que todavía no terminaste de generar su cuerpo. También es posible que, en algunos casos, quieras seguir enviando información hasta que el cliente cierre la conexión. Cuando esto ocurra, el `stream` helper te va a ser de gran ayuda:

```
get '/' do
  stream do |out|
    out << "esto va a ser legen -\n"
    sleep 0.5
    out << " (esperalo) \n"
    sleep 1
    out << "- dario!\n"
  end
end
```

Podés implementar APIs de streaming, Server-Sent Events y puede ser usado como base para WebSockets. También puede ser usado para incrementar el throughput si solo una parte del contenido depende de un recurso lento.

Hay que tener en cuenta que el comportamiento del streaming, especialmente el número de peticiones concurrentes, depende del servidor web utilizado para servir la aplicación. Puede que algunos servidores, como es el caso de WEBRick, no soporten streaming directamente, así el cuerpo

de la respuesta será enviado completamente de una vez cuando el bloque pasado a `stream` finalice su ejecución. Si estás usando Shotgun, el streaming no va a funcionar.

Cuando se pasa `keep_open` como parámetro, no se va a enviar el mensaje `close` al objeto de stream. Queda en vos cerrarlo en el punto de ejecución que quieras. Nuevamente, hay que tener en cuenta que este comportamiento es posible solo en servidores que soporten eventos, como Thin o Rainbows. El resto de los servidores van a cerrar el stream de todos modos:

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

get '/' do
  # mantenemos abierto el stream
  stream(:keep_open) { |salida| conexiones << salida }
end

post '/' do
  # escribimos a todos los streams abiertos
  conexiones.each { |salida| salida << params[:mensaje] << "\n" }
  "mensaje enviado"
end
```

Log (Registro)

En el ámbito de la petición, el helper `logger` (registrador) expone una instancia de `Logger`:

```
get '/' do
  logger.info "cargando datos"
  # ...
end
```

Este logger tiene en cuenta la configuración de logueo de tu Rack handler. Si el logueo está desactivado, este método va a devolver un objeto que se comporta como un logger pero que en realidad no hace nada. Así, no vas a tener que preocuparte por esta situación.

Tené en cuenta que el logueo está habilitado por defecto únicamente para `Sinatra::Application`. Si heredaste de `Sinatra::Base`, probablemente quieras habilitarlo manualmente:

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

Para evitar que se inicialice cualquier middleware de logging, configurá `logging` a `nil`. Tené en cuenta que, cuando hagas esto, `logger` va a devolver `nil`. Un caso común es cuando querés usar tu propio logger. Sinatra va a usar lo que encuentre en `env['rack.logger']`.

Tipos Mime

Cuando usás `send_file` o archivos estáticos tal vez tengas tipos mime que Sinatra no entiende. Usá `mime_type` para registrarlos a través de la extensión de archivo:

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

También lo podés usar con el ayudante `content_type`:

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

Generando URLs

Para generar URLs deberías usar el método `url`. Por ejemplo, en Haml:

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

Tiene en cuenta proxies inversos y encaminadores de Rack, si están presentes.

Este método también puede invocarse mediante su alias `to` (mirá un ejemplo a continuación).

Redirección del Navegador

Podés redireccionar al navegador con el método `redirect`:

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

Cualquier parámetro adicional se utiliza de la misma manera que los argumentos pasados a `halt`:

```
redirect to('/bar'), 303
redirect 'http://google.com', 'te confundiste de lugar, compañero'
```

También podés redireccionar fácilmente de vuelta hacia la página desde donde vino el usuario con `redirect back`:

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

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

Para pasar argumentos con una redirección, podés agregarlos a la cadena de búsqueda:

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

O usar una sesión:

```
enable :sessions

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

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

Cache Control

Asignar tus encabezados correctamente es el cimiento para realizar un cacheo HTTP correcto.

Podés asignar el encabezado Cache-Control fácilmente:

```
get '/' do
  cache_control :public
  "cachealo"
end
```

Pro tip: configurar el cacheo en un filtro `before`:

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

Si estás usando el helper `expires` para definir el encabezado correspondiente, `Cache-Control` se va a definir automáticamente:

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

Para usar cachés adecuadamente, deberías considerar usar `etag` o `last_modified`. Es recomendable que llames a estos helpers *antes* de hacer cualquier trabajo pesado, ya que van a enviar la respuesta inmediatamente si el cliente ya tiene la versión actual en su caché:

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

También es posible usar una weak ETag:

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

Estos helpers no van a cachear nada por vos, sino que van a facilitar la información necesaria para poder hacerlo. Si estás buscando soluciones rápidas de cacheo con proxys reversos, mirá rack-cache:

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

use Rack::Cache

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

Usá la configuración `:static_cache_control` para agregar el encabezado `Cache-Control` a archivos estáticos (ver la sección de configuración para más detalles).

De acuerdo con la RFC 2616 tu aplicación debería comportarse diferente si a las cabeceras If-Match o If-None-Match se le asigna el valor `*` cuando el recurso solicitado ya existe. Sinatra asume para peticiones seguras (como get) e idempotentes (como put) que el recurso existe, mientras que para el resto (como post) asume que no. Podes cambiar este comportamiento con la opción `:new_resource`:

```
get '/crear' do
  etag '', :new_resource => true
  Articulo.create
  erb :nuevo_articulo
end
```

Si querés seguir usando una weak ETag, indicalo con la opción `:kind`:

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

Enviando Archivos

Para enviar archivos, podés usar el método `send_file`:

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

Además acepta un par de opciones:

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

Estas opciones son:

[filename] nombre del archivo devuelto, por defecto es el nombre real del archivo.

[last_modified] valor para el encabezado Last-Modified, por defecto toma el mtime del archivo.

[type] el content type que se va a utilizar, si no está presente se intenta adivinar a partir de la extensión del archivo.

[disposition] se utiliza para el encabezado Content-Disposition, y puede tomar alguno de los siguientes valores: `nil` (por defecto), `:attachment` e `:inline`

[length] encabezado Content-Length, por defecto toma el tamaño del archivo.

[status] código de estado devuelto. Resulta útil al enviar un archivo estático como una página de error.

Si el Rack handler lo soporta, se intentará no transmitir directamente desde el proceso de Ruby. Si usás este método, Sinatra se va a encargar automáticamente peticiones de rango.

Accediendo al objeto de la petición

El objeto de la petición entrante puede ser accedido desde el nivel de la petición (filtros, rutas y manejadores de errores) a través del método `request`:

```
# app corriendo en http://ejemplo.com/ejemplo
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                # cuerpo de la petición enviado por el cliente (ver más
abajo)
  request.scheme              # "http"
  request.script_name         # "/ejemplo"
  request.path_info           # "/foo"
  request.port                # 80
  request.request_method      # "GET"
  request.query_string        # ""
  request.content_length      # longitud de request.body
  request.media_type          # tipo de medio de request.body
  request.host                # "ejemplo.com"
  request.get?                # true (hay métodos análogos para los otros verbos)
```

```
  request.form_data?          # false
  request["UNA_CABECERA"]     # valor de la cabecera UNA_CABECERA
  request.referrer            # la referencia del cliente o '/'
  request.user_agent          # user agent (usado por la condición :agent)
  request.cookies             # hash de las cookies del browser
  request.xhr?                # es una petición ajax?
  request.url                 # "http://ejemplo.com/ejemplo/foo"
  request.path                # "/ejemplo/foo"
  request.ip                  # dirección IP del cliente
  request.secure?             # false (sería true sobre ssl)
  request.forwarded?          # true (si se está corriendo atrás de un proxy reverso)
  requuest.env                # hash de entorno directamente entregado por Rack
end
```

Algunas opciones, como `script_name` o `path_info` pueden también ser escritas:

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

get "/" do
  "todas las peticiones llegan acá"
end
```

El objeto `request.body` es una instancia de IO o StringIO:

```
post "/api" do
  request.body.rewind  # en caso de que alguien ya lo haya leído
  datos = JSON.parse request.body.read
  "Hola #{ datos['nombre'] }"
end
```

Archivos Adjuntos

Podés usar el método helper `attachment` para indicarle al navegador que almacene la respuesta en el disco en lugar de mostrarla en pantalla:

```
get '/' do
  attachment
  "guardalo"
end
```

También podés pasarle un nombre de archivo:

```
get '/' do
  attachment "info.txt"
  "guardalo"
end
```

Fecha y Hora

Sinatra pone a tu disposición el helper `time_for`, que genera un objeto `Time` a partir del valor que recibe como argumento. Este valor puede ser un `String`, pero también es capaz de convertir objetos `DateTime`, `Date` y de otras clases similares:

```
get '/' do
  pass if Time.now > time_for('Dec 23, 2012')
  "todavía hay tiempo"
end
```

Este método es usado internamente por métodos como `expires` y `last_modified`, entre otros. Por lo tanto, es posible extender el comportamiento de estos métodos sobreescribiendo `time_for` en tu aplicación:

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

get '/' do
  last_modified :ayer
  expires :mañana
  "hola"
end
```

Buscando los Archivos de las Plantillas

El helper `find_template` se utiliza para encontrar los archivos de las plantillas que se van a renderizar:

```
find_template settings.views, 'foo', Tilt[:haml] do |archivo|
  puts "podría ser #{       }"
end
```

Si bien esto no es muy útil, lo interesante es que podés sobreescribir este método, y así enganchar tu propio mecanismo de búsqueda. Por ejemplo, para poder utilizar más de un directorio de vistas:

```
set :views, ['vistas', 'plantillas']

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

Otro ejemplo consiste en usar directorios diferentes para los distintos motores de renderizado:

```
set :views, :sass => 'vistas/sass', :haml => 'plantillas', :defecto => 'vistas'

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

¡Es muy fácil convertir estos ejemplos en una extensión y compartirla!

Notá que `find_template` no verifica si un archivo existe realmente, sino que llama al bloque que recibe para cada path posible. Esto no representa un problema de rendimiento debido a que `render` va a usar `break` ni bien encuentre un archivo que exista. Además, las ubicaciones de las plantillas (y su contenido) se cachean cuando no estás en el modo de desarrollo. Es bueno tener en cuenta lo anteiror si escribís un método medio loco.

Configuración

Ejecutar una vez, en el inicio, en cualquier entorno:

```
configure do
  # asignando una opción
  set :opcion, 'valor'

  # asignando varias opciones
  set :a => 1, :b => 2

  # atajo para `set :opcion, true`
  enable :opcion

  # atajo para `set :opcion, false`
  disable :opcion

  # también podés tener configuraciones dinámicas usando bloques
  set(:css_dir) { File.join(views, 'css') }
end
```

Ejecutar únicamente cuando el entorno (la variable de entorno RACK_ENV) es `:production`:

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

Ejecutar cuando el entorno es `:production` o `:test`:

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

Podés acceder a estas opciones utilizando el método `settings`:

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

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

Configurando la Protección de Ataques

Sinatra usa Rack::Protection para defender a tu aplicación de los ataques más comunes. Si por algún motivo, querés desactivar esta funcionalidad, podés hacerlo como se indica a continuación (tené en cuenta que tu aplicación va a quedar expuesta a un montón de vulnerabilidades bien conocidas):

```
disable :protection
```

También es posible desactivar una única capa de defensa:

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

O varias:

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

Configuraciones Disponibles

absolute_redirects
: Si está deshabilitada, Sinatra va a permitir redirecciones relativas, sin embargo, como consecuencia de esto, va a dejar de cumplir con el RFC 2616 (HTTP 1.1), que solamente permite redirecciones absolutas. Activalo si tu apliación está corriendo atrás de un proxy reverso que no se ha configurado adecuadamente. Notá que el helper `url` va a seguir produciendo URLs absolutas, a menos que le pasés `false` como segundo parámetro. Deshabilitada por defecto.

add_charsets
: Tipos mime a los que el helper `content_type` les añade automáticamente el charset. En general, no deberías asignar directamente esta opción, sino añadirle los charsets que quieras: `settings.add_charsets << "application/foobar"`

app_file
: Path del archivo principal de la aplicación, se utiliza para detectar la raíz del proyecto, el directorio de las vistas y el público, así como las plantillas inline.

bind

Dirección IP que utilizará el servidor integrado (por defecto: 0.0.0.0).

default_encoding
: Encoding utilizado cuando el mismo se desconoce (por defecto `"utf-8"`).

dump_errors
: Mostrar errores en el log.

environment
: Entorno actual, por defecto toma el valor de `ENV['RACK_ENV']`, o `"development"` si no está disponible.

logging
: Define si se utiliza el logger.

lock
: Coloca un lock alrededor de cada petición, procesando solamente una por proceso. Habilitá esta opción si tu aplicación no es thread-safe. Se encuentra deshabilitada por defecto.

method_override
: Utiliza el parámetro `_method` para permtir formularios put/delete en navegadores que no los soportan.

port
: Puerto en el que escuchará el servidor integrado.

prefixed_redirects
: Define si inserta `request.script_name` en las redirecciones cuando no se proporciona un path absoluto. De esta manera, cuando está habilitada, `redirect '/foo'` se comporta de la misma manera que `redirect to('/foo')`. Se encuentra deshabilitada por defecto.

protection
: Define si deben activarse las protecciones para los ataques web más comunes. Para más detalles mirá la sección sobre la configuración de protección de ataques más arriba.

public_dir
: Alias para `public_folder`, que se encuentra a continuación.

public_folder
: Path del directorio desde donde se sirven los archivos públicos. Solo se utiliza cuando se sirven archivos estáticos (ver la opción `static`). Si no está presente, se infiere del valor de la opción `app_file`.

reload_templates
: Define si se recargan las plantillas entre peticiones. Se encuentra activado en el entorno de desarrollo.

root
: Path del directorio raíz del proyecto. Si no está presente, se infiere del valor de la opción `app_file`.

raise_errors
: Elevar excepciones (detiene la aplicación). Se encuentra activada por defecto cuando el valor de `environment` es `"test"`. En caso contrario estará desactivada.

run
: Cuando está habilitada, Sinatra se va a encargar de iniciar el servidor web, no la habilites cuando estés usando rackup o algún otro medio.

running
: Indica si el servidor integrado está ejecutandose, ¡no cambiés esta configuración!.

server
: Servidor, o lista de servidores, para usar como servidor integrado. Por defecto: `['thin', 'mongrel', 'webrick']`, el orden establece la prioridad.

sessions
: Habilita el soporte de sesiones basadas en cookies a través de `Rack::Session::Cookie`. Ver la sección 'Usando Sesiones' para más información.

show_exceptions
: Muestra un stack trace en el navegador cuando ocurre una excepción. Se encuentra activada por defecto cuando el valor de `environment` es `"development"`. En caso contrario estará desactivada.

static
: Define si Sinatra debe encargarse de servir archivos estáticos. Deshabilitala cuando uses un servidor capaz de hacerlo por sí solo, porque mejorará el rendimiento. Se encuentra habilitada por defecto en el estilo clásico y desactivado en el el modular.

static_cache_control
: Cuando Sinatra está sirviendo archivos estáticos, y está opción está habilitada, les va a agregar encabezados `Cache-Control` a las respuestas. Para esto utiliza el helper `cache_control`. Se encuentra deshabilitada por defecto. Notar que es necesario utilizar un array cuando se asignan múltiples valores: `set :static_cache_control, [:public, :max_age => 300]`.

views
: Path del directorio de las vistas. Si no está presente, se infiere del valor de la opción `app_file`.

Entornos

Existen tres entornos (`environments`) predefinidos: `development`, `production` y `test`. El entorno por defecto es `development` y tiene algunas particularidades:

- Se recargan las plantillas entre una petición y la siguiente, a diferencia de `production` y `test`, donde se cachean.
- Se instalan manejadores de errores `not_found` y `error` especiales que muestran un stack trace en el navegador cuando son disparados.

Para utilizar alguno de los otros entornos puede asignarse el valor correspondiente a la variable de entorno `RACK_ENV`, o bien utilizar la opción `-e` al ejecutar la aplicación:

```
ruby mi_app.rb -e <ENTORNO>
```

Los métodos `development?`, `test?` y `production?` te permiten conocer el entorno actual.

Manejo de Errores

Los manejadores de errores se ejecutan dentro del mismo contexto que las rutas y los filtros `before`, lo que significa que podés usar, por ejemplo, `haml`, `erb`, `halt`, etc.

No encontrado *(Not Found)*

Cuando se eleva una excepción `Sinatra::NotFound`, o el código de estado de la respuesta es 404, el

manejador `not_found` es invocado:

```
not_found do
  'No existo'
end
```

Error

El manejador `error` es invocado cada vez que una excepción es elevada desde un bloque de ruta o un filtro. El objeto de la excepción se puede obtener de la variable Rack `sinatra.error`:

```
error do
  'Disculpá, ocurrió un error horrible - ' + env['sinatra.error'].name
end
```

Errores personalizados:

```
error MiErrorPersonalizado do
  'Lo que pasó fue...' + env['sinatra.error'].message
end
```

Entonces, si pasa esto:

```
get '/' do
  raise MiErrorPersonalizado, 'algo malo'
end
```

Obtenés esto:

Lo que pasó fue... algo malo

También, podés instalar un manejador de errores para un código de estado:

```
error 403 do
  'Acceso prohibido'
end

get '/secreto' do
  403
end
```

O un rango:

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

Sinatra instala manejadores `not_found` y `error` especiales cuando se ejecuta dentro del entorno de desarrollo “development”.

Rack Middleware

Sinatra corre sobre Rack[http://rack.rubyforge.org/], una interfaz minimalista que es un estándar para frameworks webs escritos en Ruby. Una de las características más interesantes de Rack para los desarrolladores de aplicaciones es el soporte de "middleware" – componentes que se ubican entre el servidor y tu aplicación, supervisando y/o manipulando la petición/respuesta HTTP para proporcionar varios tipos de funcionalidades comunes.

Sinatra hace muy sencillo construir tuberías de Rack middleware a través del método top-level `use`:

```
require 'sinatra'
require 'mi_middleware_personalizado'

use Rack::Lint
use MiMiddlewarePersonalizado

get '/hola' do
  'Hola Mundo'
end
```

La semántica de `use` es idéntica a la definida para el DSL Rack::Builder[http://rack.rubyforge.org/doc/classes/Rack/Builder.html] (más frecuentemente usado en archivos rackup). Por ejemplo, el método `use` acepta argumentos múltiples/variables así como bloques:

```
use Rack::Auth::Basic do |nombre_de_usuario, password|
  nombre_de_usuario == 'admin' && password == 'secreto'
end
```

Rack es distribuido con una variedad de middleware estándar para logging, debugging, enrutamiento URL, autenticación, y manejo de sesiones. Sinatra usa muchos de estos componentes automáticamente de acuerdo a su configuración para que típicamente no tengas que usarlas (con `use`) explícitamente.

Podés encontrar middleware útil en rack, rack-contrib, con CodeRack o en la Rack wiki.

Pruebas

Las pruebas para las aplicaciones Sinatra pueden ser escritas utilizando cualquier framework o librería de pruebas basada en Rack. Se recomienda usar Rack::Test:

```
require 'mi_app_sinatra'
require 'test/unit'
require 'rack/test'
```

```
class MiAppTest < Test::Unit::TestCase
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_mi_defecto
    get '/'
    assert_equal 'Hola Mundo!', last_response.body
  end

  def test_con_parametros
    get '/saludar', :name => 'Franco'
    assert_equal 'Hola Frank!', last_response.body
  end

  def test_con_entorno_rack
    get '/', {}, 'HTTP_USER_AGENT' => 'Songbird'
    assert_equal "Estás usando Songbird!", last_response.body
  end
end
```

Sinatra::Base - Middleware, Librerías, y Aplicaciones Modulares

Definir tu aplicación en el top-level funciona bien para micro-aplicaciones pero trae inconvenientes considerables a la hora de construir componentes reutilizables como Rack middleware, Rails metal, librerías simples con un componente de servidor, o incluso extensiones de Sinatra. El DSL de top-level asume una configuración apropiada para micro-aplicaciones (por ejemplo, un único archivo de aplicación, los directorios `./public` y `./views`, logging, página con detalles de excepción, etc.). Ahí es donde `Sinatra::Base` entra en el juego:

```
require 'sinatra/base'

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

  get '/' do
    'Hola Mundo!'
  end
end
```

Las subclases de `Sinatra::Base` tienen disponibles exactamente los mismos métodos que los provistos por el DSL de top-level. La mayoría de las aplicaciones top-level se pueden convertir en componentes `Sinatra::Base` con dos modificaciones:

- Tu archivo debe requerir `sinatra/base` en lugar de `sinatra`; de otra manera, todos los métodos del DSL de sinatra son importados dentro del espacio de nombres principal.
- Poné las rutas, manejadores de errores, filtros y opciones de tu aplicación en una subclase de `Sinatra::Base`.

`Sinatra::Base` es una pizarra en blanco. La mayoría de las opciones están desactivadas por defecto, incluyendo el servidor incorporado. Mirá Opciones y Configuraciones para detalles sobre las opciones disponibles y su comportamiento.

Estilo Modular vs. Clásico

Contrariamente a la creencia popular, no hay nada de malo con el estilo clásico. Si se ajusta a tu aplicación, no es necesario que la cambies a una modular.

Las desventaja de usar el estilo clásico en lugar del modular consiste en que solamente podés tener una aplicación Sinatra por proceso Ruby. Si tenés planificado usar más, cambiá al estilo modular. Al mismo tiempo, tené en cuenta que no hay ninguna razón por la cuál no puedas mezclar los estilos clásico y modular.

A continuación se detallan las diferencias (sutiles) entre las configuraciones de ambos estilos:

Configuración	Clásica	Modular
app_file	archivo que carga sinatra	archivo con la subclase de Sinatra::Base
run	$0 == app_file	false
logging	true	false
method_override	true	false
inline_templates	true	false
static	true	false

Sirviendo una Aplicación Modular

Las dos opciones más comunes para iniciar una aplicación modular son, iniciarla activamente con `run!`:

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

class MiApp < Sinatra::Base
  # ... código de la app  ...

  # iniciar el servidor si el archivo fue ejecutado directamente
  run! if app_file == $0
end
```

Iniciar con:

```
ruby mi_app.rb
```

O, con un archivo `config.ru`, que permite usar cualquier handler Rack:

```
# config.ru
require './mi_app'
run MiApp
```

Después ejecutar:

```
rackup -p 4567
```

Usando una Aplicación Clásica con un Archivo config.ru

Escribí el archivo de tu aplicación:

```
# app.rb
require 'sinatra'

get '/' do
  'Hola mundo!'
end
```

Y el `config.ru` correspondiente:

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

¿Cuándo Usar config.ru?

Indicadores de que probablemente querés usar `config.ru`:

- Querés realizar el deploy con un hanlder Rack distinto (Passenger, Unicorn, Heroku, ...).
- Querés usar más de una subclase de `Sinatra::Base`.
- Querés usar Sinatra únicamente para middleware, pero no como un endpoint.

No hay necesidad de utilizar un archivo `config.ru` exclusivamente porque tenés una aplicación modular, y no necesitás una aplicación modular para iniciarla con `config.ru`.

Utilizando Sinatra como Middleware

Sinatra no solo es capaz de usar otro Rack middleware, sino que a su vez, cualquier aplicación Sinatra puede ser agregada delante de un endpoint Rack como middleware. Este endpoint puede ser otra aplicación Sinatra, o cualquier aplicación basada en Rack (Rails/Ramaze/Camping/...):

```
require 'sinatra/base'

class PantallaDeLogin < Sinatra::Base
```

```
  enable :sessions

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

  post('/login') do
    if params[:nombre] == 'admin' && params[:password] == 'admin'
      session['nombre_de_usuario'] = params[:nombre]
    else
      redirect '/login'
    end
  end
end

class MiApp < Sinatra::Base
  # el middleware se ejecutará antes que los filtros
  use PantallaDeLogin

  before do
    unless session['nombre_de_usuario']
      halt "Acceso denegado, por favor <a href='/login'>inicia sesión</a>."
    end
  end

  get('/') { "Hola #{session['nombre_de_usuario']}" }
end
```

Creación Dinámica de Aplicaciones

Puede que en algunas ocasiones quieras crear nuevas aplicaciones en tiempo de ejecución sin tener que asignarlas a una constante. Para esto tenés `Sinatra.new`:

```
require 'sinatra/base'
mi_app = Sinatra.new { get('/') { "hola" } }
mi_app.run!
```

Acepta como argumento opcional una aplicación desde la que se heredará:

```
# config.ru
require 'sinatra/base'

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

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

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

Construir aplicaciones de esta forma resulta especialmente útil para testear extensiones Sinatra o

para usar Sinatra en tus librerías.

Por otro lado, hace extremadamente sencillo usar Sinatra como middleware:

```
require 'sinatra/base'

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

run ProyectoRails::Application
```

Ámbitos y Ligaduras

El ámbito en el que te encontrás determina que métodos y variables están disponibles.

Ámbito de Aplicación/Clase

Cada aplicación Sinatra es una subclase de `Sinatra::Base`. Si estás usando el DSL de top-level (`require 'sinatra'`), entonces esta clase es `Sinatra::Application`, de otra manera es la subclase que creaste explícitamente. Al nivel de la clase tenés métodos como `get` o `before`, pero no podés acceder a los objetos `request` o `session`, ya que hay una única clase de la aplicación para todas las peticiones.

Las opciones creadas utilizando `set` son métodos al nivel de la clase:

```
class MiApp < Sinatra::Base
  # Ey, estoy en el ámbito de la aplicación!
  set :foo, 42
  foo # => 42

  get '/foo' do
    # Hey, ya no estoy en el ámbito de la aplicación!
  end
end
```

Tenés la ligadura al ámbito de la aplicación dentro de:

- El cuerpo de la clase de tu aplicación
- Métodos definidos por extensiones
- El bloque pasado a `helpers`
- Procs/bloques usados como el valor para `set`

Este ámbito puede alcanzarse de las siguientes maneras:

- A través del objeto pasado a los bloques de configuración (`configure { |c| ...}`)
- Llamando a `settings` desde dentro del ámbito de la petición

Ámbito de Petición/Instancia

Para cada petición entrante, una nueva instancia de la clase de tu aplicación es creada y todos los bloques de rutas son ejecutados en ese ámbito. Desde este ámbito podés acceder a los objetos `request` y `session` o llamar a los métodos de renderización como `erb` o `haml`. Podés acceder al ámbito de la aplicación desde el ámbito de la petición utilizando `settings`:

```
class MiApp < Sinatra::Base
  # Ey, estoy en el ámbito de la aplicación!
  get '/definir_ruta/:nombre' do
    # Ámbito de petición para '/definir_ruta/:nombre'
    @valor = 42

    settings.get("/#{params[:nombre]}") do
      # Ámbito de petición para "/#{params[:nombre]}"
      @valor # => nil (no es la misma petición)
    end

    "Ruta definida!"
  end
end
```

Tenés la ligadura al ámbito de la petición dentro de:

- bloques pasados a get/head/post/put/delete/options
- filtros before/after
- métodos ayudantes
- plantillas/vistas

Ámbito de Delegación

El ámbito de delegación solo reenvía métodos al ámbito de clase. De cualquier manera, no se comporta 100% como el ámbito de clase porque no tenés la ligadura de la clase: únicamente métodos marcados explícitamente para delegación están disponibles y no compartís variables/estado con el ámbito de clase (léase: tenés un `self` diferente). Podés agregar delegaciones de método llamando a `Sinatra::Delegator.delegate :nombre_del_metodo`.

Tenés la ligadura al ámbito de delegación dentro de:

- La ligadura del top-level, si hiciste `require "sinatra"`
- Un objeto extendido con el mixin `Sinatra::Delegator`

Hechale un vistazo al código: acá está el Sinatra::Delegator mixin que extiende el objeto main.

Línea de Comandos

Las aplicaciones Sinatra pueden ser ejecutadas directamente:

```
ruby miapp.rb [-h] [-x] [-e ENTORNO] [-p PUERTO] [-o HOST] [-s MANEJADOR]
```

Las opciones son:

```
-h # ayuda
-p # asigna el puerto (4567 es usado por defecto)
-o # asigna el host (0.0.0.0 es usado por defecto)
-e # asigna el entorno (development es usado por defecto)
-s # especifica el servidor/manejador rack (thin es usado por defecto)
-x # activa el mutex lock (está desactivado por defecto)
```

Versiones de Ruby Soportadas

Las siguientes versiones de Ruby son soportadas oficialmente:

Ruby 1.8.7
: 1.8.7 es soportado completamente. Sin embargo, si no hay nada que te lo prohiba, te recomendamos que uses 1.9.2 o cambies a JRuby o Rubinius. No se dejará de dar soporte a 1.8.7 hasta Sinatra 2.0 y Ruby 2.0, aunque si se libera la versión 1.8.8 de Ruby las cosas podrían llegar a cambiar. Sin embargo, que eso ocurra es muy poco probable, e incluso el caso de que lo haga, puede que se siga dando soporte a 1.8.7. **Hemos dejado de soportar Ruby 1.8.6.** Si querés ejecutar Sinatra sobre 1.8.6, podés utilizar la versión 1.2, pero tené en cuenta que una vez que Sinatra 1.4.0 sea liberado, ya no se corregirán errores por más que se reciban reportes de los mismos.

Ruby 1.9.2
: 1.9.2 es soportado y recomendado. No uses 1.9.2p0, porque se producen fallos de segmentación cuando se ejecuta Sinatra. El soporte se mantendrá al menos hasta que se libere la versión 1.9.4/2.0 de Ruby. El soporte para la última versión de la serie 1.9 se mantendrá mientras lo haga el core team de Ruby.

Ruby 1.9.3
: 1.9.3 es soportado y recomendado. Tené en cuenta que el cambio a 1.9.3 desde una versión anterior va a invalidar todas las sesiones.

Rubinius
: Rubinius es soportado oficialmente (Rubinius >= 1.2.4). Todo funciona correctamente, incluyendo los lenguajes de plantillas. La próxima versión, 2.0, también es soportada, incluyendo el modo 1.9.

JRuby
: JRuby es soportado oficialmente (JRuby >= 1.6.7). No se conocen problemas con librerías de plantillas de terceras partes. Sin embargo, si elegís usar JRuby, deberías examinar sus Rack handlers porque el servidor web Thin no es soportado completamente. El soporte de JRuby para extensiones C se encuentra en una etapa experimental, sin embargo, de momento solamente RDiscount, Redcarpet, RedCloth y Yajl, así como Thin y Mongrel se ven afectadas.

Siempre le prestamos atención a las nuevas versiones de Ruby.

Las siguientes implementaciones de Ruby no se encuentran soportadas oficialmente. De cualquier manera, pueden ejecutar Sinatra:

- Versiones anteriores de JRuby y Rubinius
- Ruby Enterprise Edition
- MacRuby, Maglev e IronRuby
- Ruby 1.9.0 y 1.9.1 (pero no te recomendamos que los uses)

No estar soportada oficialmente, significa que si las cosas solamente se rompen ahí y no en una plataforma soportada, asumimos que no es nuestro problema sino el suyo.

Nuestro servidor CI también se ejecuta sobre ruby-head (que será la próxima versión 2.1.0) y la rama 1.9.4. Como están en movimiento constante, no podemos garantizar nada. De todas formas, podés contar con que tanto 1.9.4-p0 como 2.1.0-p0 sea soportadas.

Sinatra debería funcionar en cualquier sistema operativo soportado por la implementación de Ruby elegida.

En este momento, no vas a poder ejecutar Sinatra en Cardinal, SmallRuby, BlueRuby o cualquier versión de Ruby anterior a 1.8.7.

A la Vanguardia

Si querés usar el código de Sinatra más reciente, sentite libre de ejecutar tu aplicación sobre la rama master, en general es bastante estable.

También liberamos prereleases de vez en cuando, así, podés hacer:

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

Para obtener algunas de las últimas características.

Con Bundler

Esta es la manera recomendada para ejecutar tu aplicación sobre la última versión de Sinatra usando Bundler.

Primero, instalá bundler si no lo hiciste todavía:

```
gem install bundler
```

Después, en el directorio de tu proyecto, creá un archivo `Gemfile`:

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

# otras dependencias
gem 'haml'                    # por ejemplo, si usás haml
gem 'activerecord', '~> 3.0'  # quizás también necesités ActiveRecord 3.x
```

Tené en cuenta que tenés que listar todas las dependencias directas de tu aplicación. No es necesario listar las dependencias de Sinatra (Rack y Tilt) porque Bundler las agrega directamente.

Ahora podés arrancar tu aplicación así:

```
bundle exec ruby miapp.rb
```

Con Git

Cloná el repositorio localmente y ejecutá tu aplicación, asegurándote que el directorio `sinatra/lib` esté en el `$LOAD_PATH`:

```
cd miapp
git clone git://github.com/sinatra/sinatra.git
ruby -Isinatra/lib miapp.rb
```

Para actualizar el código fuente de Sinatra en el futuro:

```
cd miapp/sinatra
git pull
```

Instalación Global

Podés construir la gem vos mismo:

```
git clone git://github.com/sinatra/sinatra.git
cd sinatra
rake sinatra.gemspec
rake install
```

Si instalás tus gems como root, el último paso debería ser

```
sudo rake install
```

Versionado

Sinatra utiliza el Versionado Semántico, siguiendo las especificaciones SemVer y SemVerTag.

Lecturas Recomendadas

- Sito web del proyecto - Documentación adicional, noticias, y enlaces a otros recursos.
- Contribuyendo - ¿Encontraste un error?. ¿Necesitás ayuda?. ¿Tenés un parche?.
- Seguimiento de problemas
- Twitter
- Lista de Correo
- IRC: #sinatra en http://freenode.net
- Sinatra Book Tutorial (en inglés).
- Sinatra Recipes Recetas contribuidas por la comunidad (en inglés).
- Documentación de la API para la última versión liberada o para la rama de desarrollo actual en http://rubydoc.info/
- Servidor de CI

Sinatra

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

始めよう

注） 本文書は英語から翻訳したものであり、その内容が最新でない場合もあります。最新の情報はオリジナルの英語版を参照して下さい。

DSLです。

```
# myapp.rb
require 'sinatra'
get '/' do
  'Hello world!'
end
```

gemをインストールして動かしてみる。

```
gem install sinatra
ruby myapp.rb
```

localhost:4567 を見る。

ルート

Sinatraでは、ルートはHTTPメソッドとURLマッチングパターンがペアになっています。 ルートはブロックに結び付けられています。

```
get '/' do
  .. 何か見せる ..
end

post '/' do
  .. 何か生成する ..
end
```

```
put '/' do
  .. 何か更新する ..
end

delete '/' do
  .. 何か削除する ..
end
```

ルートは定義された順番にマッチします。 リクエストに最初にマッチしたルートが呼び出されます。

ルートのパターンは名前付きパラメータを含むことができ、 `params`ハッシュで取得できます。

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

また、ブロックパラメータで名前付きパラメータにアクセスすることもできます。

```
get '/hello/:name' do |n|
  "Hello #{n}!"
end
```

ルートパターンはsplat(またはワイルドカード)を含むこともでき、 `params[:splat]` で取得できます。

```
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
```

ブロックパラーメータを使用した場合:

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

正規表現を使ったルート:

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

ブロックパラーメータを使用した場合:

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

条件

ルートにはユーザエージェントのようなさまざまな条件を含めることができます。

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

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

ほかに`host_name`と`provides`条件が利用可能です:

```
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
```

独自の条件を定義することも簡単にできます:

```
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
```

戻り値

ルートブロックの戻り値は、HTTPクライアントまたはRackスタックでの次のミドルウェアに渡されるレスポンスボディを決定します。

これは大抵の場合、上の例のように文字列ですが、それ以外の値も使用することができます。

Rackレスポンス、Rackボディオブジェクト、HTTPステータスコードのいずれかとして 妥当なオブジェクトであればどのようなオブジェクトでも返すことができます:

そのように、例えばストリーミングの例を簡単に実装することができます:

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

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

静的ファイル

静的ファイルは`./public`ディレクトリから配信されます。 `:public_folder`オプションを指定することで別の場所を指定することができます。

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

注意: この静的ファイル用のディレクトリ名はURL中に含まれません。 例えば、`./public/css/style.css`は`http://example.com/css/style.css`でアクセスできます。

ビュー / テンプレート

テンプレートは`./views`ディレクトリ下に配置されています。 他のディレクトリを使用する場合の例:

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

テンプレートはシンボルを使用して参照させることを覚えておいて下さい。 サブデレクトリでもこの場合は`:'subdir/template'`のようにします。 レンダリングメソッドは文字列が渡されると、そのまま文字列を出力します。

Haml テンプレート

hamlを使うにはhamlライブラリが必要です:

```
# hamlを読み込みます
require 'haml'

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

`./views/index.haml`を表示します。

Haml's options はSinatraの設定でグローバルに設定することができます。 Options and Configurations, を参照してそれぞれ設定を上書きして下さい。

```
set :haml, {:format => :html5 } # デフォルトのフォーマットは:xhtml

get '/' do
  haml :index, :haml_options => {:format => :html4 } # 上書き
end
```

Erb テンプレート

```
# erbを読み込みます
require 'erb'

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

`./views/index.erb`を表示します。

Erubis

erubisテンプレートを表示するには、erubisライブラリが必要です:

```
# erubisを読み込みます
require 'erubis'

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

`./views/index.erubis`を表示します。

Builder テンプレート

builderを使うにはbuilderライブラリが必要です:

```
# builderを読み込みます
require 'builder'

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

`./views/index.builder`を表示します。

鋸 テンプレート

鋸を使うには鋸ライブラリが必要です:

```
# 鋸を読み込みます
require 'nokogiri'

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

./views/index.nokogiriを表示します。

Sass テンプレート

Sassテンプレートを使うにはsassライブラリが必要です:

```
# hamlかsassを読み込みます
require 'sass'

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

./views/stylesheet.sassを表示します。

Sass' options はSinatraの設定でグローバルに設定することができます。 see Options and Configurations, を参照してそれぞれ設定を上書きして下さい。

```
set :sass, {:style => :compact } # デフォルトのSass styleは :nested

get '/stylesheet.css' do
  sass :stylesheet, :sass_options => {:style => :expanded } # 上書き
end
```

Scss テンプレート

Scssテンプレートを使うにはsassライブラリが必要です:

```
# hamlかsassを読み込みます
require 'sass'

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

`./views/stylesheet.scss`を表示します。

Sass' options はSinatraの設定でグローバルに設定することができます。 see Options and Configurations, を参照してそれぞれ設定を上書きして下さい。

```
set :scss, :style => :compact # デフォルトのScss styleは:nested

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

Less テンプレート

Lessテンプレートを使うにはlessライブラリが必要です:

```
# lessを読み込みます
require 'less'

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

`./views/stylesheet.less`を表示します。

Liquid テンプレート

Liquidテンプレートを使うにはliquidライブラリが必要です:

```
# liquidを読み込みます
require 'liquid'

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

`./views/index.liquid`を表示します。

LiquidテンプレートからRubyのメソッド(`yield`を除く)を呼び出すことができないため、 ほぼ全ての場合にlocalsを指定する必要があるでしょう:

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

Markdown テンプレート

Markdownテンプレートを使うにはrdiscountライブラリが必要です:

```
# rdiscountを読み込みます
require "rdiscount"

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

`./views/index.markdown`を表示します。(`md`と`mkd`も妥当な拡張子です)

markdownからメソッドを呼び出すことも、localsに変数を渡すこともできません。 それゆえ、他のレンダリングエンジンとの組み合わせで使うのが普通です:

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

他のテンプレートからmarkdownメソッドを呼び出してもよいことに注意してください:

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

Textile テンプレート

Textileテンプレートを使うにはRedClothライブラリが必要です:

```
# redclothを読み込みます
require "redcloth"

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

`./views/index.textile`を表示します。

textileからメソッドを呼び出すことも、localsに変数を渡すこともできません。 それゆえ、他のレンダリングエンジンとの組み合わせで使うのが普通です:

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

他のテンプレートからtextileメソッドを呼び出してもよいことに注意してください:

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

RDoc テンプレート

RDocテンプレートを使うにはRDocライブラリが必要です:

```
# rdoc/markup/to_htmlを読み込みます
require "rdoc"
require "rdoc/markup/to_html"

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

`./views/index.rdoc`を表示します。

rdocからメソッドを呼び出すことも、localsに変数を渡すこともできません。 それゆえ、他のレンダリングエンジンとの組み合わせで使うのが普通です:

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

他のテンプレートからrdocメソッドを呼び出してもよいことに注意してください:

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

Radius テンプレート

Radiusテンプレートを使うにはradiusライブラリが必要です:

```
# radiusを読み込みます
require 'radius'

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

`./views/index.radius`を表示します。

RadiusテンプレートからRubyのメソッド(`yield`を除く)を呼び出すことができないため、 ほぼ全ての場合にlocalsを指定する必要があるでしょう:

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

Markaby テンプレート

Markabyテンプレートを使うにはmarkabyライブラリが必要です:

```
# markabyを読み込みます
require 'markaby'
```

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

./views/index.mabを表示します。

RABL テンプレート

RABLテンプレートを使うにはrablライブラリが必要です:

```
# rablを読み込みます
require 'rabl'

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

./views/index.rablを表示します。

Slim テンプレート

Slimテンプレートを使うにはslimライブラリが必要です:

```
# slimを読み込みます
require 'slim'

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

./views/index.slimを表示します。

Creole テンプレート

Creoleテンプレートを使うにはcreoleライブラリが必要です:

```
# creoleを読み込みます
require 'creole'

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

./views/index.creoleを表示します。

CoffeeScript テンプレート

CoffeeScriptテンプレートを表示するにはcoffee-scriptライブラリと`coffee`バイナリが必要です:

```
# coffee-scriptを読み込みます
require 'coffee-script'

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

`./views/application.coffee`を表示します。

インラインテンプレート

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

文字列をテンプレートとして表示します。

テンプレート内で変数にアクセスする

テンプレートはルートハンドラと同じコンテキストの中で評価されます。. ルートハンドラでセットされたインスタンス変数は テンプレート内で直接使うことができます。

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

ローカル変数を明示的に定義することもできます。

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

このやり方は他のテンプレート内で部分テンプレートとして表示する時に典型的に使用されます。

ファイル内テンプレート

テンプレートはソースファイルの最後で定義することもできます。

```
require 'rubygems'
require 'sinatra'

get '/' do
  haml :index
end

__END__

@@ layout
%html
  = yield

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

注意: sinatraをrequireするファイル内で定義されたファイル内テンプレートは自動的に読み込まれます。 他のファイルで定義されているテンプレートを使うには `enable :inline_templates`を明示的に呼んでください。

名前付きテンプレート

テンプレートはトップレベルの`template`メソッドで定義することができます。

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

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

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

「layout」というテンプレートが存在する場合、そのテンプレートファイルは他のテンプレートが 表示される度に使用されます。`:layout => false`することでlayoutsを無効にできます。

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

ヘルパー

トップレベルの`helpers`を使用してルートハンドラやテンプレートで使うヘルパメソッドを 定義できます。

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

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

フィルタ

beforeフィルタはリクエストされたコンテキストを実行する前に評価され、 リクエストとレスポンスを変更することができます。フィルタ内でセットされた インスタンス変数はルーティングとテンプレートで使用できます。

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

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

afterフィルタは同じコンテキストにあるリクエストの後に評価され、 同じくリクエストとレスポンスを変更することができます。 beforeフィルタとルートで設定されたインスタンス変数は、 afterフィルタからアクセスすることができます:

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

フィルタにはオプションとしてパターンを渡すことができ、 この場合はリクエストのパスがパターンにマッチした場合のみフィルタが評価されます:

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

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

強制終了

before :

ルートか　　フィルタ内で直ちに実行を終了する方法

```
halt
```

ステータスを指定することができます:

```
halt 410
```

body部を指定することもできます …

```
halt 'ここにbodyを書く'
```

ステータスとbody部を指定する …

```
halt 401, '立ち去れ!'
```

ヘッダを指定:

```
halt 402, {'Content-Type' => 'text/plain'}, 'リベンジ'
```

パッシング(Passing)

ルートはpassを使って次のルートに飛ばすことができます:

```
get '/guess/:who' do
  pass unless params[:who] == 'Frank'
  "見つかっちゃった!"
end

get '/guess/*' do
  "はずれです!"
end
```

ルートブロックからすぐに抜け出し、次にマッチするルートを実行します。 マッチするルートが見当たらない場合は404が返されます。

リクエストオブジェクトへのアクセス

受信するリクエストオブジェクトは、`request`メソッドを通じてリクエストレベル(フィルタ、ルート、エラーハンドラ)からアクセスすることができます:

```
# アプリケーションが http://example.com/example で動作している場合
get '/foo' do
  request.body              # クライアントによって送信されたリクエストボディ(下記参照)
```

```
  request.scheme           # "http"
  request.script_name      # "/example"
  request.path_info        # "/foo"
  request.port             # 80
  request.request_method   # "GET"
  request.query_string     # ""
  request.content_length   # request.bodyの長さ
  request.media_type       # request.bodyのメディアタイプ
  request.host             # "example.com"
  request.get?             # true (他の動詞についても同様のメソッドあり)
  request.form_data?       # false
  request["SOME_HEADER"]   # SOME_HEADERヘッダの値
  request.referer          # クライアントのリファラまたは'/'
  request.user_agent       # ユーザエージェント (:agent 条件によって使用される)
  request.cookies          # ブラウザクッキーのハッシュ
  request.xhr?             # Ajaxリクエストかどうか
  request.url              # "http://example.com/example/foo"
  request.path             # "/example/foo"
  request.ip               # クライアントのIPアドレス
  request.secure?          # false
  request.env              # Rackによって渡された生のenvハッシュ
end
```

`script_name`や`path_info`などのオプションは次のように利用することもできます:

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

get "/" do
  "全てのリクエストはここに来る"
end
```

`request.body`はIOまたはStringIOのオブジェクトです:

```
post "/api" do
  request.body.rewind  # 既に読まれているときのため
  data = JSON.parse request.body.read
  "Hello #{data['name']}!"
end
```

設定

どの環境でも起動時に 1 回だけ実行されます。

```
configure do
  ...
end
```

環境(RACK_ENV環境変数)が:`production`に設定されている時だけ実行する方法:

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

環境が:productionか:testの場合に設定する方法:

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

エラーハンドリング

エラーハンドラーはルートコンテキストとbeforeフィルタ内で実行します。 haml、erb、haltなどを使うこともできます。

Not Found

Sinatra::NotFoundが起きた時か レスポンスのステータスコードが 404の時にnot_foundハンドラーが発動します。

```
not_found do
  'ファイルが存在しません'
end
```

エラー

error ハンドラーはルートブロックかbeforeフィルタ内で例外が発生した時はいつでも発動します。 例外オブジェクトはRack変数sinatra.errorから取得できます。

```
error do
  'エラーが発生しました。 - ' + env['sinatra.error'].name
end
```

エラーをカスタマイズする場合は、

```
error MyCustomError do
  'エラーメッセージ...' + env['sinatra.error'].message
end
```

と書いておいて,下記のように呼び出します。

```
get '/' do
  raise MyCustomError, '何かがまずかったようです'
end
```

そうするとこうなります:

```
エラーメッセージ... 何かがまずかったようです
```

あるいは、ステータスコードに対応するエラーハンドラを設定することもできます:

```
error 403 do
  'Access forbidden'
end

get '/secret' do
  403
end
```

範囲指定もできます:

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

開発環境として実行している場合、Sinatraは特別な`not_found`と`error`ハンドラーを インストールしています。

MIMEタイプ

`send_file`か静的ファイルを使う時、Sinatraが理解でいないMIMEタイプがある場合があります。 その時は `mime_type` を使ってファイル拡張子毎に登録して下さい。

```
mime_type :foo, 'text/foo'
```

これはcontent_typeヘルパで利用することができます:

```
content_type :foo
```

Rackミドルウェア

SinatraはRackフレームワーク用の 最小限の標準インターフェース 上で動作しています。Rack中でもアプリケーションデベロッパー 向けに一番興味深い機能はミドルウェア(サーバとアプリケーション間に介在し、モニタリング、HTTPリクエストとレスポンス の手動操作ができるなど、一般的な機能のいろいろなことを提供するもの)をサポートすることです。

Sinatraではトップレベルの`use` メソッドを使ってRackにパイプラインを構築します。

```
require 'sinatra'
require 'my_custom_middleware'
```

```
use Rack::Lint
use MyCustomMiddleware

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

use Rack::Builder DSLで定義されていることと全て一致します。 例えば use メソッドはブロック構文のように複数の引数を受け取ることができます。

```
use Rack::Auth::Basic do |username, password|
  username == 'admin' && password == 'secret'
end
```

Rackはログ、デバッギング、URLルーティング、認証、セッションなどいろいろな機能を備えた標準的ミドルウェアです。 Sinatraはその多くのコンポーネントを自動で使うよう基本設定されているため、useで明示的に指定する必要はありません。

テスト

SinatraでのテストはRack-basedのテストライブラリかフレームワークを使って書くことができます。 Rack::Test をおすすめします。やり方:

```
require 'my_sinatra_app'
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 "あなたはSongbirdを使ってますね!", last_response.body
  end
end
```

注意: ビルトインのSinatra::TestモジュールとSinatra::TestHarnessクラスは 0.9.2リリース以降、廃止予定になっています。

Sinatra::Base – ミドルウェア、ライブラリ、 モジュラーアプリ

トップレベル(グローバル領域)上でいろいろ定義していくのは軽量アプリならうまくいきますが、 Rackミドルウェアや Rails metal、サーバのコンポーネントを含んだシンプルな ライブラリやSinatraの拡張プログラムを考慮するような場合はそうとは限りません。 トップレベルのDSLがネームスペースを汚染したり、設定を変えてしまうこと(例:./publicや./view)がありえます。 そこでSinatra::Baseの出番です。

```
require 'sinatra/base'

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

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

このMyAppは独立したRackコンポーネントで、Rackミドルウェアや Rackアプリケーション Rails metalとして使用することができます。`config.ru`ファイル内で `use` か、または `run` でこのクラスを指定するか、ライブラリとしてサーバコンポーネントをコントロールします。

```
MyApp.run! :host => 'localhost', :port => 9090
```

Sinatra::Baseのサブクラスで使えるメソッドはトップレベルのDSLを経由して確実に使うことができます。 ほとんどのトップレベルで記述されたアプリは、以下の２点を修正することでSinatra::Baseコンポーネントに変えることができます。

- `sinatra`の代わりに`sinatra/base`を読み込む

(そうしない場合、SinatraのDSLメソッドの全てがメインネームスペースにインポートされます)

- ルート、エラーハンドラー、フィルター、オプションをSinatra::Baseのサブクラスに書く

`Sinatra::Base` はまっさらです。ビルトインサーバを含む、ほとんどのオプションがデフォルト で無効になっています。オプション詳細については Options and Configuration をご覧下さい。

補足: Sinatraのトップレベル DSLはシンプルな委譲(delgation)システムで実装されています。 `Sinatra::Application`クラス(Sinatra::Baseの特別なサブクラス)は、トップレベルに送られる :get、 :put、 :post、:delete、 :before、:error、:not_found、 :configure、:set messagesのこれら 全てを受け取ります。 詳細を閲覧されたい方はこちら(英語): Sinatra::Delegator mixin included into the main namespace.

Sinatraをミドルウェアとして利用する

Sinatraは他のRackミドルウェアを利用することができるだけでなく、 全てのSinatraアプリケーションは、それ自体ミドルウェアとして別のRackエンドポイントの前に追加することが可能です。

このエンドポイントには、別のSinatraアプリケーションまたは他のRackベースのアプリケーション(Rails/Ramaze/Camping/...)が用いられるでしょう。

```
require 'sinatra/base'

class LoginScreen < Sinatra::Base
  enable :sessions

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

  post('/login') do
    if params[:name] = 'admin' and 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
```

スコープとバインディング

現在のスコープはどのメソッドや変数が利用可能かを決定します。

アプリケーション/クラスのスコープ

全てのSinatraアプリケーションはSinatra::Baseのサブクラスに相当します。 もしトップレベルDSLを利用しているならば(`require 'sinatra'`)このクラスはSinatra::Applicationであり、 そうでなければ、あなたが明示的に作成したサブクラスです。 クラスレベルでは`get`や`before`のようなメソッドを持っています。 しかし`request`オブジェクトや`session`には、全てのリクエストのために1つのアプリケーションクラスが存在するためアクセスできません。

`set`によって作られたオプションはクラスレベルのメソッドです:

```
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
```

次の場所ではアプリケーションスコープバインディングを持ちます:

- アプリケーションのクラス本体
- 拡張によって定義されたメソッド
- `helpers`に渡されたブロック
- `set`の値として使われるProcまたはブロック

このスコープオブジェクト(クラス)は次のように利用できます:

- configureブロックに渡されたオブジェクト経由(`configure { |c| ... }`)
- リクエストスコープの中での`settings`

リクエスト/インスタンスのスコープ

やってくるリクエストごとに、あなたのアプリケーションクラスの新しいインスタンスが作成され、全てのハンドラブロックがそのスコープで実行されます。 このスコープの内側からは`request`や`session`オブジェクトにアクセスすることができ、`erb`や`haml`のような表示メソッドを呼び出すことができます。 リクエストスコープの内側からは、`settings`ヘルパによってアプリケーションスコープにアクセスすることができます。

```
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
```

次の場所ではリクエストスコープバインディングを持ちます:

- get/head/post/put/delete ブロック
- before/after フィルタ
- helper メソッド

- テンプレート/ビュー

デリゲートスコープ

デリゲートスコープは、単にクラススコープにメソッドを転送します。 しかしながら、クラスのバインディングを持っていないため、クラススコープと全く同じふるまいをするわけではありません: 委譲すると明示的に示されたメソッドのみが利用可能であり、またクラススコープと変数/状態を共有することはできません(注: 異なった`self`を持っています)。 `Sinatra::Delegator.delegate :method_name`を呼び出すことによってデリゲートするメソッドを明示的に追加することができます。

次の場所ではデリゲートスコープを持ちます:

- もし`require "sinatra"`しているならば、トップレベルバインディング
- `Sinatra::Delegator` mixinでextendされたオブジェクト

コードをご覧ください: ここでは Sinatra::Delegator mixin はmain 名前空間にincludeされています.

コマンドライン

Sinatraアプリケーションは直接実行できます。

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

オプション:

```
-h # ヘルプ
-p # ポート指定(デフォルトは4567)
-o # ホスト指定(デフォルトは0.0.0.0)
-e # 環境を指定 (デフォルトはdevelopment)
-s # rackserver/handlerを指定 (デフォルトはthin)
-x # mutex lockを付ける (デフォルトはoff)
```

最新開発版について

Sinatraの開発版を使いたい場合は、ローカルに開発版を落として、 `LOAD_PATH`の`sinatra/lib`ディレクトリを指定して実行して下さい。

```
cd myapp
git clone git://github.com/sinatra/sinatra.git
ruby -Isinatra/lib myapp.rb
```

```
sinatra/lib                                LOAD_PATH
```

ディレクトリをアプリケーションの　　に追加する方法もあります。

```
$LOAD_PATH.unshift File.dirname(__FILE__) + '/sinatra/lib'
require 'rubygems'
require 'sinatra'

get '/about' do
  "今使ってるバージョンは" + Sinatra::VERSION
end
```

Sinatraのソースを更新する方法:

```
cd myproject/sinatra
git pull
```

その他

日本語サイト

- Greenbear Laboratory Rack日本語マニュアル
  - Rackの日本語マニュアル

英語サイト

- プロジェクトサイト – ドキュメント、 ニュース、他のリソースへのリンクがあります。
- プロジェクトに参加(貢献)する
  - バグレポート パッチの送信、サポートなど
- Issue tracker – チケット管理とリリース計画
- Twitter
- メーリングリスト
- IRC: #sinatra on freenode.net