Sinatra Recipes

Community contributed recipes and techniques

Fork me on GitHub

Select a topic

Middleware ¶ ↑

Rack is the interface behind Sinatra, and many other Ruby web frameworks. This section will cover middleware supported by Rack and Sinatra. Take a look at the Sinatra README and Sinatra Book for more on Rack middleware.

- Rack Auth Basic And Digest
- Rack Commonlogger
- Rack Parser
- Twitter Authentication With Omniauth

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the book.

See the README for more details.

Top

Sinatra Recipes

Community contributed recipes and techniques

Fork me on GitHub

Select a topic

Chapters

Rack::Auth Basic and Digest

You can easily protect your Sinatra application using HTTP Basic and Digest Authentication with the help of Rack middlewares.

Protect the whole application

These examples show how to protect the whole application (all routes).

HTTP Basic Authentication

For classic applications:

```
#main.rb

require 'sinatra'

use Rack::Auth::Basic, "Protected Area" do |username, password|
  username == 'foo' && password == 'bar'
```

```
end

get '/' do
  "secret"
end

get '/another' do
  "another secret"
end
```

For modular applications:

```
#main.rb

require 'sinatra/base'

class Protected < Sinatra::Base

  use Rack::Auth::Basic, "Protected Area" do |username, password|
    username == 'foo' && password == 'bar'
  end

  get '/' do
    "secret"
  end

end

Protected.run!
```

To try these examples just run `ruby main.rb -p 4567` and visit localhost:4567

HTTP Digest Authentication

To use digest authentication with current versions of Rack a `config.ru` file is needed.

For classic applications:

```
#main.rb

require 'sinatra'

get '/' do
  "secret"
end
```

```
#config.ru

require File.expand_path '../main.rb', __FILE__
```

```
app = Rack::Auth::Digest::MD5.new(Sinatra::Application) do |username|
  {'foo' => 'bar'}[username]
end

app.realm = 'Protected Area'
app.opaque = 'secretkey'

run app
```

For modular applications:

```
#main.rb

require 'sinatra/base'

class Protected < Sinatra::Base

  get '/' do
    "secret"
  end

  def self.new(*)
    app = Rack::Auth::Digest::MD5.new(super) do |username|
      {'foo' => 'bar'}[username]
    end
    app.realm = 'Protected Area'
    app.opaque = 'secretkey'
    app
  end
end
```

```
#config.ru

require File.expand_path '../main.rb', __FILE__

run Protected
```

To try these examples just run `rackup -p 4567` and visit localhost:4567

Protect specific routes

If you want to protect just specific routes things get a bit complicated. There are many ways to do it. The one I show here uses modular applications and a `config.ru` file.

HTTP Basic Authentication

First the `main.rb`

```
#main.rb

require 'sinatra/base'

class Protected < Sinatra::Base

  use Rack::Auth::Basic, "Protected Area" do |username, password|
    username == 'foo' && password == 'bar'
  end

  get '/' do
    "secret"
  end

  get '/another' do
    "another secret"
  end

end

class Public < Sinatra::Base

  get '/' do
    "public"
  end

end
```

And the `config.ru`

```
#config.ru

require File.expand_path '../main.rb', __FILE__

run Rack::URLMap.new({
  "/" => Public,
  "/protected" => Protected
})
```

To try these examples just run `rackup -p 4567` and visit localhost:4567

The resulting routes are explained at the bottom of this page.

HTTP Digest Authentication

First the `main.rb`

```
#main.rb

require 'sinatra/base'

class Protected < Sinatra::Base

  get '/' do
    "secret"
  end

  get '/another' do
    "another secret"
  end

  def self.new(*)
    app = Rack::Auth::Digest::MD5.new(super) do |username|
      {'foo' => 'bar'}[username]
    end
    app.realm = 'Protected Area'
    app.opaque = 'secretkey'
    app
  end
end

class Public < Sinatra::Base

  get '/' do
    "public"
  end

end
```

And the `config.ru`

```
#config.ru

require File.expand_path '../main.rb', __FILE__

run Rack::URLMap.new({
  "/" => Public,
  "/protected" => Protected
})
```

To try these examples just run `rackup -p 4567` and visit localhost:4567

The resulting routes

The routes display the following:

- `/` displays "public"
- `/protected` displays "secret"
- `/protected/another` displays "another secret"

All the protected routes are mounted at `/protected` so if you add another route to the Protected class like for example `get '/foo' do...` it can be reached at `/protected/foo`. To change it just modify the call to `Rack::URLMap.new...` to your likings.

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the book.

See the README for more details.

Top

Sinatra Recipes

Community contributed recipes and techniques

Select a topic

Chapters

Rack::CommonLogger ¶ ↑

Sinatra has logging support, but it's nearly impossible to log to a file and to the stdout (like Rails does).

However, there is a little trick you can use to log to stdout *and* to a file:

```
require 'sinatra'

configure do
  # logging is enabled by default in classic style applications,
  # so `enable :logging` is not needed
  file = File.new("#{settings.root}/log/#{settings.environment}.log", 'a+')
  file.sync = true
  use Rack::CommonLogger, file
end

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

You can use the same configuration for modular style applications, but you have to `enable :logging` first:

```
require 'sinatra/base'

class SomeApp < Sinatra::Base
  configure do
    enable :logging
    file = File.new("#{settings.root}/log/#{settings.environment}.log", 'a+')
    file.sync = true
```

```
    use Rack::CommonLogger, file
  end

  get '/' do
    'Hello World'
  end

  run!
end
```

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the book.

See the README for more details.

Top

Sinatra Recipes

Community contributed recipes and techniques

Select a topic

Chapters

Using Rack::Parser to encapsulate parsing logi web service applications

Sinatra is often used for web service applications. One common need these applications have is to parse incoming messages, typically JSON or XML.

Often this is a two step process – first parsing the message string into predictable data structures, then loading that into models. So a typical route might look like

```
post '/messages'
  message = Message.from_hash( ::MultiJson.decode(request.body) )
  message.save
  halt 202, {'Location' => "/messages/#{message.id}"}, ''
end
```

And your Message model would have an appropriate `#from_hash` method that grabs what it needs from the parsed message and throws it into a new instance.

If your application has several different endpoints, all using the same content-type, you could save some repetition by moving it to a helper:

```
helpers do
  def parsed_body
    ::MultiJson.decode(request.body)
  end
end
```

```
post '/orders'
  order = Order.from_hash( parsed_body )
  order.process
  # ....
end
```

This works fine for simple scenarios. But if you have multiple content-types, need to validate the message format, or do other pre-processing on it, moving the parsing logic out of your application itself starts to become attractive.

One option is to move it into a module that your app extends.

But another option is to do the parsing in a middleware. That way, your app is not responsible for doing the basic parsing at all -- it's done and exposed to your app by the time the request arrives.

The idea is *to mimic what Rack does to process form data*, which is to populate the `env[rack.request.form_hash]` with a hash, and then expose that through the `params` hash. But instead of parsing url parameters or multipart form data, it will parse the XML or JSON body.

`Rack::Parser` is a Rack middleware that does just that, and lets you configure custom parsing routines for any given content-type.

So using Rack::Parser, the example above would be changed to

```
# in config.ru
use Rack::Parser, :content_types => {
  'application/json'  => Proc.new { |body| ::MultiJson.decode body }
}

# in application
post '/orders'
  order = Order.from_hash( params['order'] )
  order.process
  # ....
end
```

This is quite convenient if your app accepts either form data or JSON for a given endpoint -- since the code is then identical whether a user submits a form directly or some web-service client submits a JSON request.

Custom parsing

Note that the content-type handlers are just procs (or anything that responds to `#call`) that take a single parameter -- the request body.

This makes it quite easy to set up custom parsing, validation, or other pre- processing.

Rack::Parser also has recently added support for error handling, so that errors in parsing can be mapped into a suitable http response, and logged. This is quite useful for handling validation errors.

For instance, say you have set up vendor-specific json message formats, which you want to validate using a `validate_input` class method on your models.

```
class MyJsonParser
```

```
  def initialize(validator_class)
    @klass = validator_class
  end

  def call(body)
    json = ::MultiJson.decode(body)
    if @klass.respond_to?(:validate_input)
      @klass.validate_input(json)
    end
    json
  end

end

use Rack::Parser, :content_types => {
  'application/vnd-my-message+json' => MyJsonParser.new(Message),
  'application/vnd-my-order+json'   => MyJsonParser.new(Order)
}
```

Then any parsing error raised in `MulitJson.decode` or validation error in `validate_input` will result in a 400 (bad request) response in the same content-type as the request -- like

```
{"errors": "Validation error in input - cannot set 'private_info'"}
```

This is the default; you can also completely customize how errors are converted into responses, per content-type.

For more info

Rack::Parser

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the book.

See the README for more details.

Top

Sinatra Recipes

Community contributed recipes and techniques

Select a topic

Chapters

Twitter authentication with OmniAuth

OmniAuth provides several strategies (released as gems) that provides authentication for a lot of systems, such as Facebook, Google, GitHub, and many more.

Each strategy is a Rack middleware, so it's very easy to integrate with Sinatra. This recipe will show you how to add user authentication to your Sinatra application using Twitter as your authentication provider.

First, you have to create a new application at Twitter Developers. It's very important to set the `Callback url` to `http://example.com/auth/twitter/callback`. This url is where twitter will redirect the client when the user is successfully authenticated. Once you created your application, you have to remember it's `Consumer key` and `Consumer secret`. You will need them when you configure Omniauth, like this:

```
require 'sinatra'
require 'omniauth-twitter'

configure do
  enable :sessions

  use OmniAuth::Builder do
    provider :twitter, ENV['CONSUMER_KEY'], ENV['CONSUMER_SECRET']
  end
end
```

Note that we used the CONSUMER_KEY and CONSUMER_SECRET environment variables. This is because it's bad to store this information on your code, so each time you run your app do it like this:

```
$ CONSUMER_KEY=<your consumer key> CONSUMER_SECRET=<your consumer secret> ruby app.rb
```

If you are using rackup, the same rule applies.

Then, you have to secure your application by redirecting non-authenticated users to twitter, so they can sign in:

```
helpers do
  # define a current_user method, so we can be sure if an user is authenticated
  def current_user
    !session[:uid].nil?
  end
end

before do
  # we do not want to redirect to twitter when the path info starts
  # with /auth/
  pass if request.path_info =~ /^\/auth\//

  # /auth/twitter is captured by omniauth:
  # when the path info matches /auth/twitter, omniauth will redirect to twitter
  redirect to('/auth/twitter') unless current_user
end
```

Lastly, you have create a new user session when the authentication was successful:

```
get '/auth/twitter/callback' do
  # probably you will need to create a user in the database too...
  session[:uid] = env['omniauth.auth']['uid']
  # this is the main endpoint to your application
  redirect to('/')
end

get '/auth/failure' do
  # omniauth redirects to /auth/failure when it encounters a problem
  # so you can implement this as you please
end

get '/' do
  'Hello omniauth-twitter!'
end
```

Needless to say that this approach is useful for other omniauth strategies.

Did we miss something?

It's very possible we've left something out, that's why we need your help!This is a community driven project after all. Feel free to fork the project and send us a pull request to get your recipe or tutorial included in the

book.

See the README for more details.

Top