Rack Bootcamp

Adam Hawkins

Table of Contents

Rack is the HTTP interface for Ruby. Rack defines a standard interface for interacting with HTTP and connecting web servers. Rack makes it easy to write HTTP facing applications in Ruby. Rack applications are shockingly simple. There is the code that accepts a request and code serves the response. Rack defines the interface between the two.

Dead Simple Rack Applications

Rack applications are objects that respond to `call`. They must return a "triplet". A triplet contains the status code, headers, and body. Here’s an example class that shows "hello world."

```
class HelloWorld
  def response
    [ 200, { }, [ 'Hello World' ] ]
  end
end
```

This class is not a Rack application. It demonstrates what a triplet looks like. The first element is the HTTP response code. The second is a hash of headers. The third is an enumerable object representing the body. We can use our `HelloWorld` class to create a simple rack application. We know that we need to create an object that responds to call. `call` takes one argument: the rack environment. We’ll come back to the `env` later.

```
class HelloWorldApp
  def self.call(env)
    HellowWorld.new.response
  end
end
```

Here is a simple class that implements `call`. It returns the response from the `HelloWorld` class. Now we need to put this online. We have implemented one side of the wall. Now we need write the other side. The rack library includes a `Server` class. This is the simplest way to serve rack applications. Let’s create a simple ruby script to serve `HelloWorldApp`.

hello_world.rb

```
require 'rack'
require 'rack/server'

class HelloWorld
  def response
    [ 200, { }, [ 'Hello World' ] ]
  end
end

class HelloWorldApp
  def self.call(env)
    HelloWorld.new.response
  end
end

Rack::Server.start :app => HelloWorldApp
```

Here’s what happens when you run this script:

```
$ ruby hello_world.rb
>> Thin web server (v1.4.1 codename Chromeo)
>> Maximum connections set to 1024
>> Listening on 0.0.0.0:8080, CTRL+C to stop
```

NOTE The output you see may be different. `Rack::Server` chooses a server based off what’s installed in order of preference. It uses Webrick if you have nothing else installed because that’s part of Ruby’s standard library. More on servers later on.

Simply open `http://localhost:8080` and you’ll see "Hello World" in the browser. It’s not fancy but you just wrote your first rack app! We didn’t write our own server and that’s ok. Matter of fact, that’s fantastic. Odds are you will never need to write your own server. There are plenty of servers to choose from: Thin, Unicorn, Rainbows, Goliath, Puma, and Passenger. You don’t want to write those. You want to write applications.

Env

We skipped over over `env` in the previous section since we didn't need it yet. `env` is a `Hash` that meets the rack spec. The spec defines **incoming** information. Outgoing data must be triplets. The `env` provides incoming headers, host info, query string and other common information. The `env` is passed to the application which decides what to do. Our `HelloWorldApp` didn't care about it. Let's update our `HelloWorldApp` to interact with incoming information.

hello_world.rb

```ruby
class HelloWorldApp
  def self.call(env)
    [ 200, { }, [ "Hello World. You said: #{env['QUERY_STRING']}" ]]
  end
end

Rack::Server.start :app => HelloWorldApp
```

Now visit `http://localhost:8080?message=foo` and you'll see "message=foo" on the page. If you're more curious about `env` you can do this:

```ruby
class EnvInspector
  def self.call(env)
    [ 200, { }, [ env.inspect ] ]
  end
end

Rack::Server.start :app => EnvInspector
```

Now `curl` the URL and to see everything included in `env`. It's a standard `Hash` instance.

```
{
  "SERVER_SOFTWARE"=>"thin 1.4.1 codename Chromeo",
  "SERVER_NAME"=>"localhost",
  "rack.input"=>#<StringIO:0x007fa1bce039f8>,
  "rack.version"=>[1, 0],
  "rack.errors"=>#<IO:<STDERR>>,
  "rack.multithread"=>false,
  "rack.multiprocess"=>false,
  "rack.run_once"=>false,
  "REQUEST_METHOD"=>"GET",
  "REQUEST_PATH"=>"/favicon.ico",
  "PATH_INFO"=>"/favicon.ico",
  "REQUEST_URI"=>"/favicon.ico",
  "HTTP_VERSION"=>"HTTP/1.1",
  "HTTP_HOST"=>"localhost:8080",
  "HTTP_CONNECTION"=>"keep-alive",
  "HTTP_ACCEPT"=>"*/*",
  "HTTP_USER_AGENT"=>"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4)
AppleWebKit/536.11 (KHTML, like Gecko) Chrome/20.0.1132.47 Safari/536.11",
  "HTTP_ACCEPT_ENCODING"=>"gzip,deflate,sdch",
  "HTTP_ACCEPT_LANGUAGE"=>"en-US,en;q=0.8",
  "HTTP_ACCEPT_CHARSET"=>"ISO-8859-1,utf-8;q=0.7,*;q=0.3",
  "HTTP_COOKIE"=> "_gauges_unique_year=1;  _gauges_unique_month=1",
  "GATEWAY_INTERFACE"=>"CGI/1.2",
  "SERVER_PORT"=>"8080",
  "QUERY_STRING"=>"",
  "SERVER_PROTOCOL"=>"HTTP/1.1",
  "rack.url_scheme"=>"http",
  "SCRIPT_NAME"=>"",
  "REMOTE_ADDR"=>"127.0.0.1",
  "async.callback"=>#<Method: Thin::Connection#post_process>,
  "async.close"=>#<EventMachine::DefaultDeferrable:0x007fa1bce35b88
}
```

You may have noticed that the `env` doesn't do any fancy parsing. The query string wasn't a hash. It was the string. It is raw data. Rack is simple to understand and use. You could only work with hashes and triplets. However that's tedious and doesn't scale. Complex applications need abstractions. Enter `Rack::Request` and `Rack::Response`.

TIP

Further Reading

- The Rack Spec

Abstractions

`Rack::Request` is an abstraction around the `env` hash. It provides abstractions for things like cookies and POST body paramters. Here's an example.

hello_world.rb

```ruby
class HelloWorldApp
  def self.call(env)
    request = Rack::Request.new env
    request.params # contains the union of GET and POST params
    request.xhr?   # requested with AJAX
    require.body   # the incoming request IO stream

    if request.params['message']
      [ 200, { }, [ request.params['message' ] ]
    else
      [ 400, { }, [ 'Say something to me!' ] ]
    end
  end
end
```

`Rack::Request` is simply a proxy for the `env` hash. The underlying `env` hash is modified so keep that in mind.

`Rack::Response` is an abstraction around response triplets. It simplifies access to headers, cookies, and the body. Here's an example:

```ruby
class HelloWorldApp
  def self.call(env)
    response = Rack::Response.new
    response.write 'Hello World'      # write some content to the body
    response.body = [ 'Hello World' ] # or set it directly
    response['X-Custom-Header'] = 'foo'
    response.set_cookie 'bar', 'baz'
    response.status = 202

    response.finish # return the generated triplet
  end
end
```

These are basic abstractions. They don't require much explanation. You can learn more about them by reading the documentation.

Now we build more complex applications now on these abstracations. It's hard to make an application when all the logic is in one class. Applications are composed of different classes with different responsiblities. These discrete chunks are called "middleware".

TIP

Further Reading

- `Rack::Request` docs
- `Rack::Response` docs

Middleware

Rack applications are objects that respond to `call`. We can do whatever we want inside `call`, for instance we can delegate to another class. Here's an example:

```ruby
class ParamsParser
  def initialize(app)
    @app = app
  end

  def call(env)
    request = Rack::Request.new env
    env['params'] = request.params
    app.call env
  end
end

class HelloWorldApp
  def self.call(env)
    parser = ParamsParser.new self
    env = parser.call env
    # env['params'] is now set to a hash for all the input paramters

    [ 200, { }, [ env['params'].inspect ] ]
  end
end
```

This example is contrived. You want not do this in practice. The point is to illustrate that you can manipulate env (or response). You can create a middleware stack as deep as you like. Each middleware simply calls the next one and returns its value. Before we move to the next step, let's define what a middleware looks like:

middleware_template.rb

```ruby
class Middleware
  def initialize(app)
    @app = app
  end

  # This is a "null" middlware because it simply calls the next one.
  # We can manipulate the input before calling the next middleware
  # or manipulate the response before returning up the chain.
  def call(env)
    @app.call env
  end
end
```

request_modification_middleware.rb

```ruby
class Middleware
  def initialize(app)
    @app = app
  end

  # the My-Custom-Header on every request before sending to the
  # application before processing.
  def call(env)
    @app.call(env.merge({
      'HTTP_MY_CUSTOM_HEADER' => 'foo'
    })
  end
end
```

response_modification_middleware.rb

```ruby
class Middleware
  def initialize(app)
    @app = app
  end

  # Set My-Custom-Header on every response before sending to the
  # web server
  def call(env)
    status, headers, body = @app.call env
    [ status, headers.merge({ 'My-Custom-Header' => 'foo' }), body ]
  end
end
```

The middleware stack exmplifies the builder pattern. Composing Rack applications is so common (and required) that Rack includes a class to make this easy.

Middleware Stacks

`Rack::Builder` creates a middleware stack. Each object calls the next one and returns its return value. Rack contains a bunch of handy middlewares. They have one for caching and encodings. Let's increase our test applications performance.

```ruby
# this returns an app that responds to call cascading down the list of
# middlewares. Technically there is no difference between "use" and
# "run". "run" is just there to illustrate that it's the end of the
# chain.
app = Rack::Builder.new do
  use Rack::Etag            # Add an ETag
  use Rack::ConditionalGet  # Support Caching
  use Rack::Deflator        # GZip
  run HelloWorldApp         # Say Hello
end

Rack::Server.start :app => app
```

`app` has a `call` method that generates this call tree:

```
Rack::Etag
  Rack::ConditionalGet
    Rack::Deflator
      HelloWorldApp
```

We'll skip the functionality of each middleware because that's not important. This is an example of how you can build up functionality in applications. Middlewares are powerful. You can add manipulate incoming data before hitting the next one or modify the response from an existing one. Let's create some for practice.

```ruby
class EnsureJsonResponse
  def initialize(app)
    @app = app
  end

  # Set the 'Accept' header to 'application/json' no matter what.
  # Hopefully the next middleware respects the accept header :)
  def call(env)
    env['HTTP_ACCEPT'] = 'application/json'
    @app.call env
  end
end
```

```ruby
class Timer
  def initialize(app)
    @app = app
  end

  def call(env)
    before = Time.now
    status, headers, body = @app.call env

    headers['X-Timing'] = (Time.now - before).to_i.to_s

    [ status, headers, body ]
  end
end
```

Now we can use those middlewares in our app.

```ruby
app = Rack::Builder.new do
  use Timer # put the timer at the top so it captures everything below it
  use EnsureJsonResponse
  run HelloWorldApp
end

Rack::Server.start :app => app
```

We’ve just written our own middeware and learned how to generate a runnable application with a middleware stack. This is how rack apps are written in practice. Now onto the final piece of the puzzle: `config.ru`

TIP

Further Reading

rack-contrib

A grab bag of handy middleware such as `Rack::PostBodyContentTypeParser` for handling `application/json` request bodies.

manifold

Dead simple CORS support for APIs

`Rack::URLMap`

Map different sub-paths to different rack applications. Example: `/foo ⇒ FooApp` or `/bar ⇒ BarApp`.

Rackup

Rack includes the `rackup` comamnds. It starts a web server process using one of the rack servers installed on the system. It reads `config.ru` (by default) to build the rack application (and associated middleware stack). It's evaluated inside (via `instance_eval`) a `Rack::Builder` as shown before. Here's all the work we've done up to now in `config.ru`

config.ru

```
# HelloWorldApp defintion
# EnsureJsonResponse defintion
# Timer definition

use Timer
use EnsureJsonResponse
run HelloWorldApp
```

Now navigate into the correct directory and run: `rackup` and you'll see:

```
$ rackup
>> Thin web server (v1.4.1 codename Chromeo)
>> Maximum connections set to 1024
>> Listening on 0.0.0.0:8080, CTRL+C to stop
```

NOTE The exact output may vary system to system depending on which gems are installed and their versions.

`rackup` prefers better Servers like Thin over WeBrick. The code inside `config.ru` is evaluated and built using a `Rack::Builder` which generates an rack API compliant object. The object is passed to the rack server (Thin). The rack server puts the application online.

TIP Your webserver (e.g. `puma`, `thin`) usually provides their own CLI with server specific options. `rackup` provides a way to forward options to the underlying application server but this may not work in all cases. The underlying application server may also have a separate configuration file for its specific configuration (e.g. a control port/socket). You're advised to understand that you can launch your webserver in multiple ways. Investigate and learn which fits your use case.

Rails & Rack

Rails 3+ is fully Rack compliant. A Rails 3 application is more complex Rack application. It uses a complex middleware stack. The dispatcher is the final middlware. The dispatcher reads the routing table and calls the correct controller and method. Here's the stock middleware stack used in production:

```
use Rack::Cache
use ActionDispatch::Static
use Rack::Lock
use #<ActiveSupport::Cache::Strategy::LocalCache::Middleware:0x007fce77f21690>
use Rack::Runtime
use Rack::MethodOverride
use ActionDispatch::RequestId
use Rails::Rack::Logger
use ActionDispatch::ShowExceptions
use ActionDispatch::DebugExceptions
use ActionDispatch::RemoteIp
use ActionDispatch::Callbacks
use ActiveRecord::ConnectionAdapters::ConnectionManagement
use ActiveRecord::QueryCache
use ActionDispatch::Cookies
use ActionDispatch::Session::CookieStore
use ActionDispatch::Flash
use ActionDispatch::ParamsParser
use ActionDispatch::Head
use Rack::ConditionalGet
use Rack::ETag
use ActionDispatch::BestStandardsSupport
run YourApp::Application.routes
```

The middleware stack is not declared explicitly in `config.ru`. Rails uses it's own middleware chains build from different configuration files. The application instance delegates `call` to the middleware chain. Here's an example `config.ru` for a rails app:

config.ru

```
# This file is used by Rack-based servers to start the application.

require ::File.expand_path('../config/environment',  __FILE__)
run Example::Application
```

You know that `Example::Application` must have a `call` method. Here's the implementation of this method from 3.2 stable:

```ruby
# Rails::Application, Rails::Application < Rails::Engine
def call(env)
  env["ORIGINAL_FULLPATH"] = build_original_fullpath(env)
  super(env)
end

# Rails::Engine
# the super class method
def call(env)
  app.call(env.merge!(env_config))
end

# app method in the super class
def app
  @app ||= begin
    config.middleware = config.middleware.merge_into(default_middleware_stack)
    config.middleware.build(endpoint)
  end
end
```

Testing

We’ve discussed how rack applications respond to `call`. The `call` method takes a `Hash` with expected keys that return triplets. The request and responses are plain ol' ruby objects. Thus it’s easy enough to test our web applications by making method calls and asserting on the return values. It’s suprisingly functional. The rack-test library wraps this process by providing high level methods for sending "requests" and getting "responses" back from rack compatible objects. Let’s walk through a few examples of the most common uses. We’ll use minitest since it’s build into the standard library. `rack-test` is a module so you can include it in any Ruby object (or use with any testing framework). First up is a "hello world" type test.

hello_world_test.rb

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

class HelloWorldApp
  def self.call(env)
    [ 200, { 'Content-Type' => 'text/plain' }, [ 'Hello World' ] ]
  end
end

class HelloWorldAppTest < MiniTest::Test
  include Rack::Test::Methods ①

  def app
    HelloWorldApp ②
  end

  def test_returns_hello_world
    get '/' ③

    assert last_response.ok?
    assert_equal 'Hello World', last_response.body
    assert_equal 'text/plain', last_response.content_type
  end
end
```

① Include methods to use in the test case

② Define the rack compatible object to make "requests" too

③ Make a get request to `/`, response is stored in `last_response`

This example demonstrates the most simple `GET` request. `Rack::Test` also includes methods for `POST`, `GET`, `PUT`, `DELETE`, and `PATCH`. All these methods accept three arguments. First is the path. Second is either a data hash (which is form encoded) or a raw string. Third is a hash to merge into the `env`.

The requests are `stored` in `last_response`. The library gives you access to headers via `[]`, `status`, `body`, and some helpers like `content_type` for common headers. Let’s demonstrate more complex functionality.

hello_world_test.rb

```ruby
require 'minitest/autorun'
require 'rack/request'
require 'rack/test'

class HelloWorldApp
  def self.call(env)
    request = Rack::Request.new env

    [
      200,
      { 'Content-Type' => 'text/plain' },
      [ "Hello #{request.params['name']}" ]
    ]
  end
end

class HelloWorldAppTest < MiniTest::Test
  include Rack::Test::Methods

  def app
    HelloWorldApp
  end

  def test_returns_hello_world
    get '/', name: 'Adam'

    assert last_response.ok?
    assert_equal 'Hello Adam', last_response.body
    assert_equal 'text/plain', last_response.content_type
  end
end
```

Now let’s send some custom headers.

hello_world_test.rb

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

class HelloWorldApp
  def self.call(env)
    if env['HTTP_APPLICATION_VERSION'].to_i < 4 ①
      [
        200,
        { 'Content-Type' => 'text/plain' },
        [ "OK" ]
      ]
    else
      [
        400,
        { 'Content-Type' => 'text/plain' },
        [ "Version out of date" ]
      ]
    end
  end
end

class HelloWorldAppTest < MiniTest::Test
  include Rack::Test::Methods

  def app
    HelloWorldApp
  end

  def test_returns_200_when_version_after_4
    get '/', { }, { 'HTTP_APPLICATION_VERSION' => '4' } ②

    assert last_response.ok?
    assert_equal 'OK', last_response.body
  end

  def test_returns_200_when_version_before_4
    get '/', { }, { 'HTTP_APPLICATION_VERSION' => '3' } ③

    assert_equal 400,  last_response.status
    assert_equal 'Version out of date', last_response.body
  end
end
```

① Convert value to number. Header values are always strings because HTTP has no notion of types.

② Headers are represented as `HTTP_` keys in `env`. All characters are in uppercase and non-alphanumeric characters are replaced with `_`. The value should also be s string as noted in the previous point.

③ Use `{ }` as the second argument. This request does not include any POST data. Omitting the second argument is a common mistake when you want to only specify headers.

Servers

We’ve covered how `rackup` and `Rack::Server` use available server libraries to start the actual HTTP server. The Ruby community offers many options for various use cases. Let’s review them so you can make the best choice.

WeBrick

This server is included in the standard library. It’s meant **only** for development. Webrick is a toy implementation. It should never be used for anything remotely close to production.

Thin

This server uses reactor pattern provided Eventmachine. It used to be popular but has been superseeded by better alternatives. If you cannot use anything better, then go for thin. It’s better than Webrick.

Puma

Puma is multi-threaded and multi-process. This the most flexible server out there for pure Ruby application. You can use multi-process (a.k.a. clustered mode) if your Ruby platform has a GIL (read: MRI) and your application is CPU bound. IO bound applications (most web applications) can go ahead with the mutli-threaded mode (default mode) since the GIL blocks true concurrency. Puma also includes a HTTP control interface for programmatical manipulation or getting stats. Check out puma before going for anything else—especially if you’re using JRuby or Rubinius.

Unicorn

Unicorn uses a pre-forking model. This means it will start up multiple processes and use system calls to "load balance" requests across child processes. Users must note that connections to external resources (such as a database) must happen in child proess. Unicorn provides an "after fork" for this. Users locked on MRI should investigate this option.

Passenger

Passenger is a bit different than the others in the list. Passenger is is an nginx or apache module. It does also have a standalone mode also. It supports multi-threaded and multi-process models. You should consider passenger if you need to deploy the application to an existing apache or nginx server.