Introducing Rack

Christian Neukirchen
Editor in Chief of Anarchaia
chneukirchen@gmail.com

Abstract

Developing general web frameworks and frameworkless web applications is difficult because lots of different ways to connect to web servers need to be implemented. Since Ruby is popular for writing web applications, we try to address this problem by proposing a lightweight abstraction mapping HTTP requests onto a simple Ruby API.

This makes it possible to combine all kinds of web servers with different web applications without further change. Furthermore, our solution enables new functionality by combining and composing web applications, as well as better testability.

1. Introduction: The Problem

Currently, Ruby [1] has become a widespread language and is especially used a lot for developing web applications. The downside of this is that every internal web server has its own API (WEBrick [2], Mongrel [3], . . .) and there is a multitude of connection mechanisms to talk to external web servers (for example, CGI [4], FastCGI [5], SCGI [6], or proprietary interfaces like LiteSpeed).

Clearly, this screams for an abstract way of connecting web serving and content generating tiers of Ruby web applications.

In this paper, we'd like to introduce and outline a possible solution we have defined and implemented: Rack [7].

2. The Design of Rack

Rack was built on the idea that we ought to do the simplest thing possible that unifies all the different web servers and web applications. The logical conclusion therefore is to be as close to HTTP [8] as possible.

As a classical client-server protocol, HTTP is strictly based around *requests* sent from a client which then are replied to with a *response*. The simplest type therefore for a general web application is:

$$\text{Request} \rightarrow \text{Response}$$

What, however, really are the requests and responses? To investigate this, let's have a look at a sample HTTP protocol dump. Consider this request:

```
GET / HTTP/1.1
User-Agent: curl/7.12.2 ...
Host: ruby-lang.org
Pragma: no-cache
Accept: */*
```

The most common representation of this is request is its CGI environment, which would look like:

```
{"HTTP_USER_AGENT"=>"curl/7.12.2 ..."
 "REMOTE_HOST"=>"127.0.0.1",
 "PATH_INFO"=>"/",
 "HTTP_HOST"=>"ruby-lang.org",
 "SERVER_PROTOCOL"=>"HTTP/1.1",
 "SCRIPT_NAME"=>"",
 "REQUEST_PATH"=>"/",
 "REMOTE_ADDR"=>"127.0.0.1",
 "HTTP_VERSION"=>"HTTP/1.1",
 "REQUEST_URI"=>"http://ruby-lang.org/",
 "SERVER_PORT"=>"80",
 "HTTP_PRAGMA"=>"no-cache",
 "QUERY_STRING"=>"",
 "GATEWAY_INTERFACE"=>"CGI/1.1",
 "HTTP_ACCEPT"=>"*/*",
 "REQUEST_METHOD"=>"GET"}
```

As you can see, every HTTP header is mapped onto an entry `HTTP_`*`header`*, and there are additional fields for the non-header parts of the request, e.g. `REQUEST_METHOD` or `REQUEST_PATH`.

Since almost every web framework works with environments like this, we decided to keep it. The rules are comparatively simple, albeit not very well specified. Our proposed Rack specification [9] clarifies fuzzy some parts of the CGI specification [4].

Copyright © 2007 Christian Neukirchen
Verbatim copying is permitted as long as this message is preserved.
Additional material at:
http://chneukirchen.org/talks#introducingrack
Euruko '07 November 10–11, Vienna, Austria

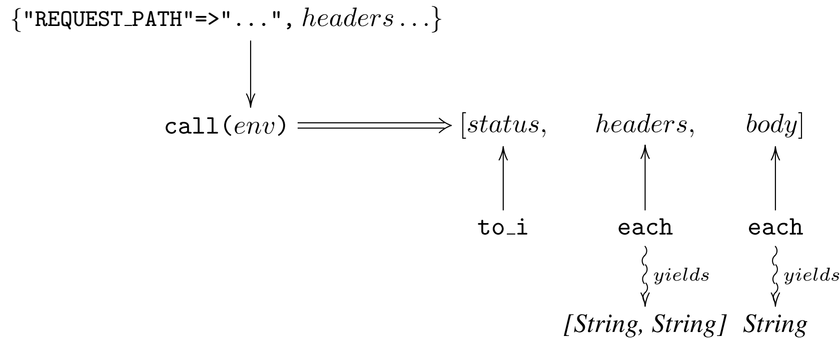

Figure 1. Rack's abstract interface at a glance

After processing a request like this, the server replies with:

```
HTTP/1.1 302 Found
Date: Sat, 27 Oct 2007 10:07:53 GMT
Server: Apache/2.0.54 (Debian GNU/Linux)
  mod_ssl/2.0.54 OpenSSL/0.9.7e
Location: http://www.ruby-lang.org/
Content-Length: 209
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC
               "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a
  href="http://www.ruby-lang.org/">here</a>.
  </p>
</body></html>
```

Here, CGI does not help us anymore. In plain CGI, everything is just sent to the standard output, which we obviously cannot do in an abstract API. Given a closer look however, we can see that every HTTP reply consists of three parts:

- A HTTP status code, here 302 (Found).
- A set of HTTP headers
- The response body

Each status code is represented by an integer, and every header maps a string to another string (taken weird `Cookie`-headers aside). It would be naive, however, to make the response body a plain string: it should be possible to *stream* the body, which is natural to do in Ruby by yielding values from a method.

We decided the yielded values have to be Strings, which are binary-safe and encoding-free in Ruby 1.8.

According to Jon Postel's law [10], Rack is conservative in what it provides (the request) and liberal in what it accepts (the response). Therefore, the reply is only specified as duck-types [11]:

The reply is an Array with three items:

The status code, which responds to `to_i`.

The header, which responds to `each` and yields pairs of keys (strings) and values (yielding strings on `each`; this is because of Cookie headers, which possibly are multiline—Ruby 1.8 returns each line on `each`).

The body, which responds to `each` and yields strings, which are the chunks of the reply.

After defining the representations of requests and responses, we now need to define the actual API the Ruby side will use. Here, we decided to use a little trick: wouldn't it be totally cool to have lambdas as web applications? In Ruby, `Procs` respond to `call` with arguments for the lambda, and Rack does the same. A simple "Hello, world"-application doesn't need more than this:

```
lambda { |env|
  [200,                              # Status
   {"Content-Type"=>"text/plain"}, # Headers
   ["Hello, world!"]]               # Body
}
```

The whole API at a glance is shown in Figure 1.

3. Parts and Pieces

Now, let's have a closer look at the particular parts of the Rack distribution.

3.1 The specification

The specification is the heart of Rack, a document that defines how valid Rack applications should behave, what's okay and what is not.

The specification is comparatively short (less than 1000 words), but still in progress: not every detail is precisely defined yet.

The specification is generated from the sources of a piece of middleware (see below), Rack::Lint, that checks for conformation at runtime. Every statement of the specification is accompanied with code testing it; an example:

```
## * The <tt>CONTENT_LENGTH</tt>,
##   if given, must consist of digits only.
assert("Invalid CONTENT_LENGTH: " +
       "#{env["CONTENT_LENGTH"]}") {
  !env.include?("CONTENT_LENGTH") ||
    env["CONTENT_LENGTH"] =~ /\A\d+\z/
}
```

In case the application behaves in invalid ways and Rack::Lint is used, Rack::Lint will generate a HTTP status 500 (Internal Server Error) and abort the request with an explanation of which assertion failed (using it is highly recommended during development).

3.2 Handlers: Making web servers talk to Rack

Since all of this wouldn't be very useful if nothing supported Rack, Rack includes several *handlers*, which connect Rack to the most common web servers used with Ruby. As of version Rack 0.2, this is:

- CGI
- FastCGI
- WEBrick
- Mongrel (Swiftcore's [12] *evented_mongrel* is supported as well)

The next release of Rack will also include support for the proprietary LiteSpeed protocol, kindly contributed by Adrian Madrid.

3.3 Adapters: Making Rack talk to Web frameworks

Currently, Rack supports only one framework out of the box, which is why the lucky stiff's Camping [13]. However, several external frameworks already use Rack and ship with their own adapters, the more stable of which are:

- Ramaze [14]
- Merb [15]

There also is an ongoing effort by the Fuzed [16] hackers to write a Rack adapter for Rails [17].

All these adapters show that adapting Rack for existing web frameworks is easy: they all are shorter than two pages of code. This indicates that supporting Rack is simple and quickly achievable.

Furthermore, the author of this paper wrote a framework directly based directly on Rack, dubbed *Coset*. It is described below.

3.4 Rack middleware

One obvious benefit of having an abstract way to interact with web applications is that the actual calls can happen without an actual HTTP connection, that is, the web application can be used like any ordinary Ruby object.

Rack middleware uses this to enable many advanced facilities web developers ask for. Middleware operates on Rack applications and combines, composes, aggregates or modifies it in useful ways.

These pieces of middleware are currently included in the Rack distribution:

Rack::Cascade takes an array of Rack applications and tries the incoming requests on all of them until one returns a non-404 result. This is useful for recreating the common "application-as-file-not-found-handler" pattern.

Rack::CommonLogger intercepts each request to create an Apache-like logfile.

Rack::Lint ensures all applications behave validly.

Rack::Recursive enables Rack applications to make requests of itself and its sub-applications.

Rack::Reloader loads changed code on each request, which is very useful for development.

Rack::ShowExceptions catches unhandled exceptions and displays a browsable backtrace with source snippets.

Rack::ShowStatus creates default pages for common HTTP error codes.

Rack::Static intercepts requests for static files (javascript files, images, stylesheets, etc) based on the URL prefixes passed in the options, and serves them using a Rack::File.

Rack::URLMap routes request based on (virtual) hosts and paths to different applications.

3.5 Utilities

Rack also includes several utilities that simplify Rack and Rack application development:

Rack::Request wraps the Rack request environment with a nice and convenient API providing easy access, and can parse GET and POST arguments.

Rack::Response simplifies generating Rack responses by ensuring their validity and providing a CGI-like output object. It also can set cookies and has methods to simplify testing.

Rack::Utils contains a grab bag of standard methods needed for Web development, for example request parsing, a MIME-Multipart parser, URL escaping and a list of HTTP status codes.

Rack::File can be used to directly serve a file from disk, allowing for sendfile(2)-like optimizations in the respective handlers.

Rack::MockRequest and Rack::MockResponse simplify Test- and Behavior Driven Development by allowing to call Rack applications conveniently without real HTTP roundtrips.

For example, a specification of Rack::Recursive looks like this:

```
app = Rack::Recursive.new \
        Rack::URLMap.new("/app1" => @app1,
                         "/app3" => @app3,
                         "/app4" => @app4)

res = Rack::MockRequest.new(app).get("/app3")
res.should.be.ok
res.body.should.equal "App1"
```

Rack has an extensive test suite and largely was developed in a largely behavior-driven style.

4. Rack::Builder and `rackup`

Real world Rack applications tend to use multiple utilities wrapping the actual application. In pure Ruby, this often results in code like:

```
app = MyRackApp.new
app = Rack::Lint.new(app)
app = Rack::ShowStatus.new(app)
app = Rack::ShowExceptions.new(app)
app = Rack::CommonLogger.new(app)

Rack::Handler::Mongrel.run app, :Port => 8080
```

Or worse:

```
app = Rack::CommonLogger.new(
        Rack::ShowExceptions.new(
          Rack::ShowStatus.new(
            Rack::Lint.new(MyRackApp.new))))
```

Rack proposes a DSL for easy combination of apps and middleware, implemented by Rack::Builder:

```
app = Rack::Builder.new do
  use Rack::CommonLogger
  use Rack::ShowExceptions
  use Rack::ShowStatus
  use Rack::Lint
  run MyRackApp.new
end

Rack::Handler::Mongrel.run app, :Port => 8080
```

Note that the `use`-statements are written in "reverse" order, the outermost first.

For easier deployment, the Rack distribution includes a standalone Rack::Builder runner called `rackup`. It can directly run the Rack::Builder DSL and uses heuristics on the environment to decide whether to run as CGI, FastCGI or standalone server.

```
#!/usr/bin/env rackup

use Rack::CommonLogger
use Rack::ShowExceptions
use Rack::ShowStatus
use Rack::Lint
run MyRackApp.new
```

Port, server and host parameters can easily be overridden with commandline parameters:

```
$ myrackapp.ru -s mongrel -p 8080
```

5. Comparisons

5.1 Rack compared to CGI/FastCGI

Compared to classical web server interface protocols, Rack has the advantage of being more flexible since it also can be used without change for internal web servers. Furthermore, the design makes the creation of new middleware trivial (which theoretically could be done with several processes and the classical protocols, but nobody seems to do that.)

5.2 Rack compared to WSGI

As the reader with open source web development background will have noticed quickly, Rack takes a lot of inspiration from Python's WSGI [18], and in fact the specification is largely based on PEP333 [19], however it was simplified further. (For example, the stdout-like object was moved to utilities instead of forcing every handler/middleware to implement it.)

WSGI on the other hand has become very popular within the Python community and every serious Python web framework supports it.

Rack is still young, but in the opinion of the author, the Ruby web development community could benefit a lot from adapting and making Rack a pseudo-standard for web development just as WSGI has become.

6. Real World Results

Since Rack is only a very thin layer, it does not have a big performance impact compared to the full application stack most Ruby web applications use. Experiments with Mongrel show for that simple servers pushing data from MemCache have less than 3% impact on the total performance, i.e. Rack slows down the application by less than 3% compared to a pure Mongrel-only implementation. This is a cheap price to pay given the flexibility and ease of developing using Rack.

At the time of writing, there is at least one mission-critical high-performance web application running on Rack directly which successfully delivers 30 billions of requests per month, using Mongrel and Swiftcore.

Rack also has been used for high-speed development of small, but urgently needed web applications. Due to the simplicity and convenient APIs and helpers provided, we had working results quicker than with any other current Ruby web framework.

7. A short excursion: Coset

Although Rack is pretty young, several Ruby web frameworks already depend on Rack and use it as their main interface (Section 3.3).

We also wrote a framework specifically for implementing RESTful [20] Rack applications and services, called *Coset*[1].

Coset's API is inspired by Camping [13], web.py [21] and RESTlet [22] and features dispatching on URI templates [23] as well as support for dealing with multiple content-types. A simple time server could be written like this, to give an example:

```
require 'coset'

class TimeServer < Coset
  GET "/time{EXT}" do
    now = Time.now
    wants "text/html" do
      res.write "<title>Current time</title>\n"
      res.write "It's now #{now}.\n"
    end
    wants "text/plain" do
      res["Content-Type"] = "text/plain"
      res.write now.to_s + "\n"
    end
    wants "application/json" do
      res["Content-Type"] = "application/json"
      res.write "{\"current_time\": \"#{now}\"}\n"
    end
  end
end
```

The special template {EXT} matches a file extension and "fakes" the according content-type (useful for browsers or in cases in which you cannot pass your own headers).

Now, we can test the time server (superfluous headers are suppressed for reasons of brevity):

```
$ curl -i localhost:3333/time
HTTP/1.1 200 OK
Content-Type: text/html

<title>Current time</title>
It's now Sun Nov 04 12:03:18 CET 2007.

$ curl -i localhost:3333/time.txt
HTTP/1.1 200 OK
Content-Type: text/plain

Sun Nov 04 12:03:56 CET 2007

$ curl -i -H "Accept: application/json" \
                          localhost:3333/time
HTTP/1.1 200 OK
Content-Type: application/json

"current_time": "Sun Nov 04 12:05:14 CET 2007"
```

[1] Which is a really bad German mathematical pun: Coset can be translated as *Restklasse*.

While Coset is not yet officially released due to limited developer time, the Darcs head version [24] already runs several small sites successfully.

Coset is by no means finished yet, and discussion about future features as well as design and implementation details is welcome.

8. Summary

We have shown how a minimal interface abstracting HTTP simplifies web development by allowing for improved testability and giving more flexible options for code reuse.

The Rack specification, in spite of a low version number, already satisfies the current needs of web developers and specifies a suitable way to run web applications.

Abstracting HTTP into what essentially is a method application enables us to use decades of functional programming for building and reusing code for the web.

Our implementation proves that this approach is fast and stable enough to run business-critical web services without restrictions in expressivity or performance, but with even quicker turnaround times than with traditional frameworks, since the convenient APIs built upon the core Rack specification save developer's valuable time.

Acknowledgments

Thanks go to all developers contributing to the Rack code base and the project itself—an always current list is found in the Rack source distribution.

I would like to especially thank Michael Fellinger with whom I talked for hours pondering and improving the design of Rack.

Thanks also go to Personifi for allowing me to test and benchmark Rack on powerful servers with real world applications. Our results and development times speak for themselves.

Last, but not the least, I'd like to thank the WSGI team for paving the way—they did the major work in solving the fiddly design issues which appear now and then—and especially the Paste [25] developers, whose project greatly influenced the design of the Rack utilities.

Finally, thanks to Horacio López a.k.a. vruz, Johan Sørensen, Alexander Kellett and Aria Stewart for reviewing this paper.

References

[1] http://www.ruby-lang.org/en/

[2] http://raa.ruby-lang.org/project/webrick/

[3] http://mongrel.rubyforge.org/

[4] The CGI Specification, version 1.1, http://hoohoo.ncsa.uiuc.edu/cgi/interface.html

[5] M. R. Brown, "FastCGI Specification 1.0", http://www.fastcgi.com/devkit/doc/fcgi-spec.html

[6] N. Schemenauer, "SCGI: A Simple Common Gateway Interface alternative", http://www.fastcgi.com/devkit/doc/fcgi-spec.html

[7] http://rack.rubyforge.org/

[8] R. Fielding et al., "Hypertext Transfer Protocol – HTTP/1.1", RFC 2616, Internet Engineering Task Force, June 1999, http://www.ietf.org/rfc/rfc2616.txt

[9] C. Neukirchen, "Rack Specification 0.1", http://rack.rubyforge.org/doc/files/SPEC.html

[10] J. Postel, "Transmission Control Protocol", section "Robustness Principle", RFC 793, Internet Engineering Task Force, Sep. 1999, http://www.ietf.org/rfc/rfc793.txt

[11] http://c2.com/cgi/wiki?DuckTyping

[12] http://swiftiply.swiftcore.org/

[13] http://code.whytheluckystiff.net/camping/

[14] http://ramaze.rubyforge.org/

[15] http://merb.rubyforge.org/

[16] http://fuzed.rubyforge.org/

[17] http://www.rubyonrails.org/

[18] http://www.wsgi.org/

[19] P. J. Eby, "Python Web Server Gateway Interface v1.0", PEP 333, Apr. 2006, http://www.python.org/dev/peps/pep-0333/

[20] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures." Doctoral dissertation, University of California, Irvine, 2000.

[21] http://webpy.org/

[22] http://www.restlet.org/

[23] J. Gregorio et al., "URI Template", Internet-Draft, work in progress, July 2007.

[24] http://chneukirchen.org/repos/coset/

[25] http://pythonpaste.org/