Rack Developer's Notebook

Based on the Rack Programming - A Chinese book by Liu Quiang
and the Ruby Community

Compiled By : Bala Paranj

Send feedback to rack@zepho.com

I. Introduction

1. What is Rack

Rack is a specification (and implementation) of a minimal abstract Ruby API that models HTTP.

Rack provides a minimal, modular and adaptable interface for developing web applications in Ruby. By wrapping HTTP requests and response in the simplest way possible, it unifies and distills the API for Web servers, Web frameworks, and software in between (the so-called middleware) into a single method call.

It provides the simplest possible API that represents a generic web application. Rack consists of interface, implementations and utilities. Rack distributions has the following 6 components:

1) Specification

2) Handlers

3) Adapters

4) Middlewares

5) Utilities

We will discuss them in detail in later sections.

2. Why Rack

There is a proliferation of different web application servers and web frameworks. There is a lot of code duplication among frameworks since they essentially all do the same things. And still, every Ruby web framework developer is writing his own handlers for every Web application server to be supported. Developing a Ruby web framework is not hard but it's lots of repetitive boring work. That means writing interfaces to all servers and writing decoding code or copying cgi.rb. So let's write the HTTP interfacing code once and only once.

However dealing with HTTP is easy. You get a request and return a response. The canonical format of a HTTP request is represented by a hash of CGI-like environment and a response consists of three parts: a status, a set of headers, and a body.

HTTP Request

```
GET /hello HTTP/1.1
Host: localhost:9292
Connection: keep-alive
User-Agent: Mozilla/5.0 (Macinto...
Accept: application/xml,applicati...
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: UTF-8,*;q=0.5
Cache-Control: max-age=0
Cookie: foo=bar
```

Rack "env" Hash

```
{
  "PATH_INFO"          => "/",
  "QUERY_STRING"       => "",
  "REMOTE_ADDR"        => "::1",
  "REMOTE_HOST"        => "localhost",
  "REQUEST_METHOD"     => "GET",
  "REQUEST_URI"        => "http://localhost:9292/",
  "SCRIPT_NAME"        => "",
  "SERVER_NAME"        => "localhost",
  "SERVER_PORT"        => "9292",
  "SERVER_PROTOCOL"    => "HTTP/1.1",
  "SERVER_SOFTWARE"    => "WEBrick/1.3.1 (Ruby/1.9.1/2009-12-07)",
  "HTTP_HOST"          => "localhost:9292",
  "HTTP_USER_AGENT"    => "Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-us)",
  "HTTP_ACCEPT"        => "text/html;q=0.9,text/plain;q=0.8",
  "HTTP_ACCEPT_LANGUAGE"  =>"en-us",
  "HTTP_ACCEPT_ENCODING"  =>"gzip, deflate",
  "HTTP_COOKIE"        => "",
  "HTTP_CONNECTION"    => "keep-alive",
  "rack.version"       => [1, 1],
  "rack.input"         => #<StringIO:0x00000100d87808>,
  "rack.errors"        => #<IO:<STDERR>>,
  "rack.multithread"   => true,
  "rack.multiprocess"  => false,
  "rack.run_once"      => false,
  "rack.url_scheme"    => "http",
  "HTTP_VERSION"       => "HTTP/1.1",
  "REQUEST_PATH"       => "/"
}
```

HTTP Response

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 16 Feb 2010 23:49:18 GMT
Content-Type: text/plain
Content-Length: 11
Cache-Control: max-age=60

Hello World
```

Rack Response

```
[
  200,
  {'Content-Type'   => 'text/plain',
   'Content-Length' => '11',
   'Cache-Control'  => 'max-age=60'},
  ["Hello World"]
]
```

This can be easily mapped onto a method call in Ruby like this:

```
class HelloWorld
  def call(env)
    [200, {"Content-Type" => "text/plain"}, ["Hello world!"]]
  end
end
```

This is the most simple **Rack** application. The Rack app gets call'ed with the CGI environment and retuns and Array of status, headers and body. Rack aims to provide a minimal API for connecting web servers and web frameworks. Now the developers can stop writing handlers for every webserver. Keep the request and response simple. Rack standard is very simple, the speci-

fication is only about two pages. If you want to implement a Rack compliant web server or a web framework, just meet this simple standard.[1] You can build small, simple tools that does one thing really well (like Unix)

Super simple API for writing web apps. Single API to connect to all web application servers .Based on Python's WSGI. Write once and serve it on any Rack compliant web app server. Convenient way to write micro apps.

Good like Camping - very minimal, single file applications. But better because Camping isn't multi-threaded, requests are wrapped in a mutex. Rack is small, lightweight and super easy to write and deploy. It is good for multi-threaded, non-blocking requests, so you can use Rack to write non-blocking file uploader applications to compliment Rails applications.

```
mbp2:~ bparanj$ telnet localhost http
Trying ::1...
Connected to localhost.
Escape character is '^]'.
HEAD /index.html HTTP/1.1
Host: www.zepho.com
Connection: close

HTTP/1.1 200 OK
Date: Thu, 11 Nov 2010 08:03:25 GMT
Server: Apache/2.2.14 (Unix) mod_ssl/2.2.14 OpenSSL/0.9.8l DAV/2
Content-Location: index.html.en
Vary: negotiate
TCN: choice
Last-Modified: Sat, 20 Nov 2004 20:16:24 GMT
ETag: "3b02ec-2c-3e9564c23b600;47a8458be9780"
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/html
Content-Language: en

Connection closed by foreign host.
mbp2:~ bparanj$ curl localhost
<html><body><h1>It works!</h1></body></html>mbp2:~ bparanj$
```

1. http://rack.rubyforge.org/doc/SPEC.html

The figure above shows a Telnet session where HTTP 1.1 compliant HEAD request is sent to the Apache web server. The response from the server is also shown. When the curl utility is used you can see the body which is 44 characters in length. This is seen in the response header Content-Length: 44.

When the GET request is made the output is as follows.

```
mbp2:~ bparanj$ telnet localhost http
Trying ::1...
Connected to localhost.
Escape character is '^]'.
GET /index.html HTTP/1.1
Host: www.zepho.com
Connection: close

HTTP/1.1 200 OK
Date: Thu, 11 Nov 2010 08:14:35 GMT
Server: Apache/2.2.14 (Unix) mod_ssl/2.2.14 OpenSSL/0.9.8l DAV/2
Content-Location: index.html.en
Vary: negotiate
TCN: choice
Last-Modified: Sat, 20 Nov 2004 20:16:24 GMT
ETag: "3b02ec-2c-3e9564c23b600;47a8458be9780"
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/html
Content-Language: en

<html><body><h1>It works!</h1></body></html>Connection closed by foreign host.
```

The only difference this time is that the body is included in the response.

Like most network protocols, HTTP uses the client-server model: An HTTP client opens a connection and sends a request message to an HTTP server; the server then returns a response message, usually containing the resource that was requested. After delivering the response, the server closes the connection (making HTTP a stateless protocol, i.e. not maintaining any connection information between transactions).

The format of the request and response messages are similar, and English-oriented. Both kinds of messages consist of:

* an initial line,

* zero or more header lines,
* a blank line (i.e. a CRLF by itself), and
* an optional message body (e.g. a file, or query data, or query output).

Put another way, the format of an HTTP message is:

<initial line, different for request vs. response>
Header1: value1
Header2: value2
Header3: value3

<optional message body goes here, like file contents or query data; it can be many lines long, or even binary data $&*%@!^$@>

Initial Request Line

The initial line is different for the request than for the response. A request line has three parts, separated by spaces: a method name, the local path of the requested resource, and the version of HTTP being used. A typical request line is:

GET /path/to/file/index.html HTTP/1.0

Notes:

* GET is the most common HTTP method; it says "give me this resource". Other methods include

POST and HEAD-- more on those later. Method names are always uppercase.

* The path is the part of the URL after the host name, also called the request URI (a URI is like a URL, but more general).
* The HTTP version always takes the form "HTTP/x.x", uppercase.

Initial Response Line (Status Line)

The initial response line, called the status line, also has three parts separated by spaces: the HTTP version, a response status code that gives the result of the request, and an English reason phrase describing the status code. Typical status lines are:

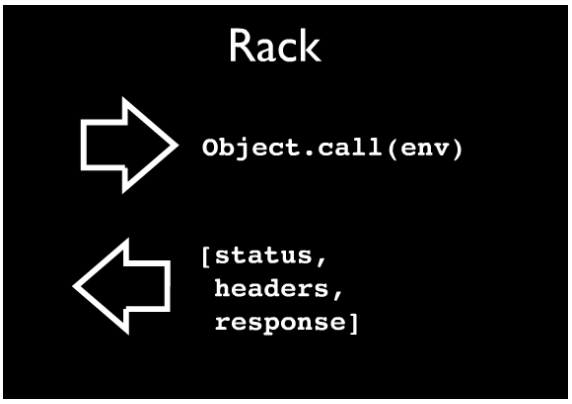
HTTP/1.0 200 OK

or

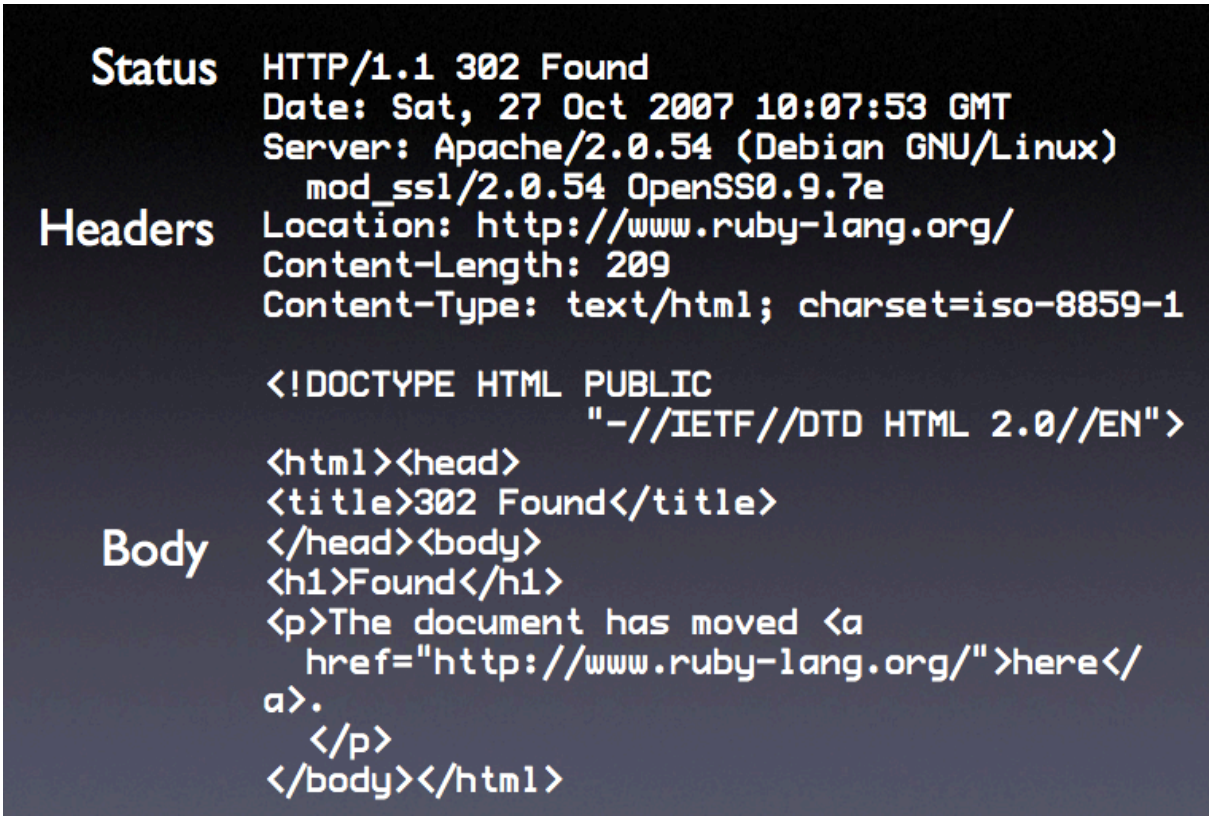
HTTP/1.0 404 Not Found

3. Rack Application

A Rack application is a Ruby object (not a class) that responds to *call*. It takes exactly one argument, the environment and returns an Array of exactly three values: The status, the headers, and the body (an object that responds to each - some random string can be the output of the call to each).

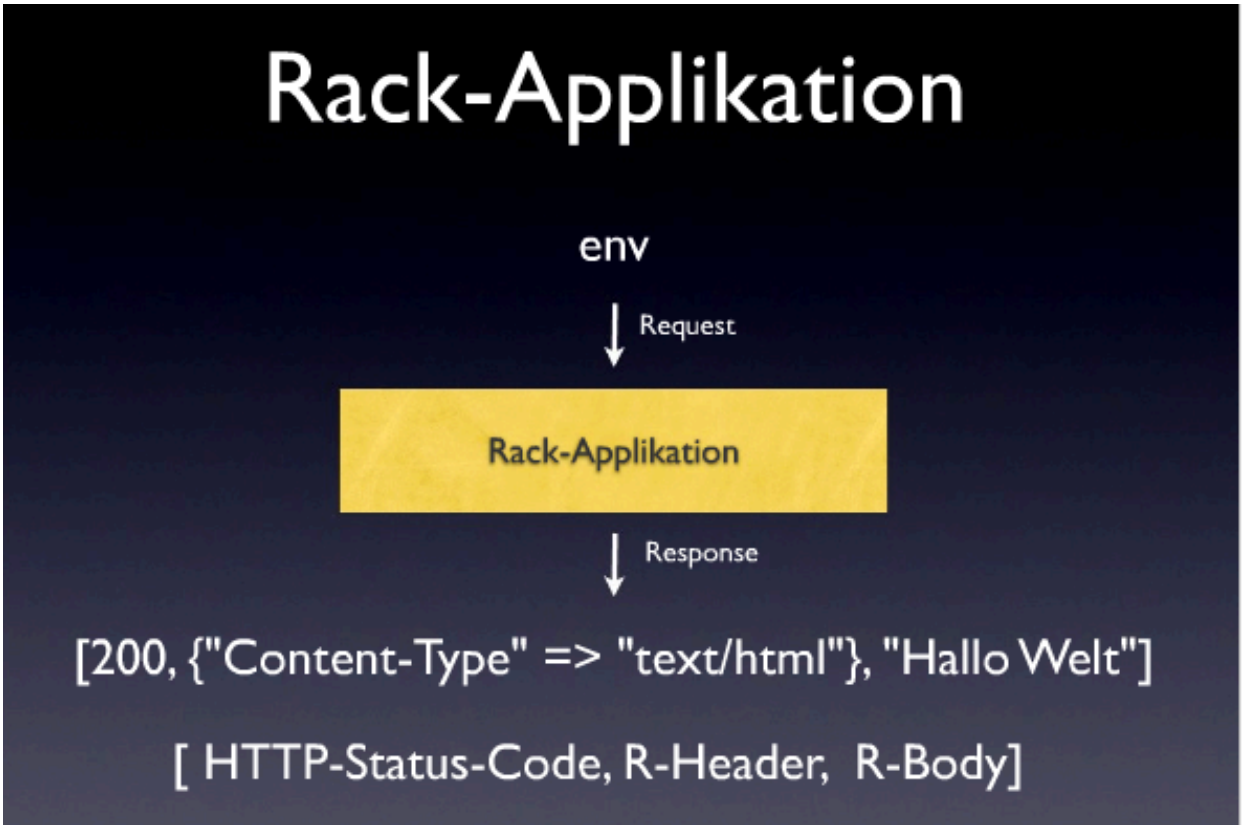


Informally, a *Rack application* is a thing that responds to *call* and takes a hash as argument, returning an array of status, headers and a body. The body needs to respond to *each* and then successively return strings that represent the response body. The hash given contains a CGI-ish set of environment variables and some special values, like the body stream of the request (env['rack.input']), or information about the run-time environment (e.g. env['rack.run_once']).



This API is mainly used by framework developers and usually will not be exposed to framework users. This API is very simple and it can be satsified by a lambda. It is not hard to adopt. The camping adapter is a mere five lines of code.

On top of this minimal API, there are libraries for commonly used things like query parsing or cookie handling and provide more convenience (Rack::Request and Rack::Response) that can be used by both framework developers and application developers.



The most exciting thing about Rack is that it provides an extremely easy way to combine these web applications. After all, they are only Ruby objects with a single method that matters. And the thing that calls you may not really be a web server, but could as well be a different application! Here is a few Rack filters (or middleware) that already exist:

- Rack::ShowExceptions catches all thrown exceptions and wraps them nicely in an helpful 500-page adapted from Django.
- Rack::CommonLogger does Apache-style logs.
- Rack::URLMap redirects to different Rack applications depending on the path and host (a very simple router).

There is another tool, Rack::Lint that checks if your applications and filters play nicely with others so everything ought to work together.

4. Advantages of Rack

What do you gain if your web framework/server/application supports Rack?

- Handlers for WEBrick, Mongrel and plain CGI, FastCGI, and every new webserver that provides a Rack handler. Let *n* and *m* be the amount of servers and frameworks, without Rack it's n*m, but with it's n+m, which means less work for everyone.
- The possibility to run several applications inside a single webserver without external configuration.
- Easier integration and functional testing, since everything can easily be mocked.
- A greater diversity among frameworks, since writers now can concentrate on the parts that make it special and stop wasting their time on boring things.
- More synergistic effects: Compare "That upload handler you wrote for Sinatra is really great, too bad I use Ramaze." with "That upload handler you wrote for Rack works great for me too!"

Rack includes Handler for:

- Mongrel
- EventedMongrel
- SwiftipliedMongrel
- WEBrick
- FCGI
- CGI
- SCGI
- LiteSpeed
- Thin
- Passenger, Unicorn etc

The following application servers include their own Rack handler:

- Ebb
- Fuzed
- Glassfish v3
- Phusion Passenger (which is mod_rack for Apache and Nginx)

- Rainbows!
- Unicorn
- Zbatery

This means that these app servers comply to the Rack interface and that any Rack application can run on them. These handlers are located in the Rack::Handler namespace. In the Rack spec, we can see:

```
Rack::Handler::CGI
Rack::Handler::EventedMongrel
Rack::Handler::FastCGI
Rack::Handler::LSWS
Rack::Handler::Mongrel
Rack::Handler::SCGI
Rack::Handler::SwiftipliedMongrel
Rack::Handler::Thin
Rack::Handler::WEBrick
```

Orginally, Rack was supported by Camping and Ramaze (adapter included with Rack). Now, almost all major web frameworks now support Rack interface. Here is the list:

- Ramaze
- Coset
- Halcyon
- Mack
- Maveric
- Merb
- Racktools::SimpleApplication
- Ruby on Rails
- Rum
- Sinatra
- Sin
- Vintage
- Waves

- Wee

All of these frameworks includes a Rack adapter. Rack was influenced by WSGI and Paste. Adapters connect Rack with third party web frameworks. All adapters are almost trivial.Here is the code for the Rack included adapter for Camping (this was included in older version of Rack where Camping was not Rack compliant and needed an adapter).

```
Rack::Adapter::Camping
# File lib/rack/adapter/camping.rb
def initialize(app)
  @app = app
end


def call(env)
  env["PATH_INFO"] ||= ""
  env["SCRIPT_NAME"] ||= ""
  controller = @app.run(env['rack.input'], env)
  h = controller.headers
  h.each_pair do |k,v|
   if v.kind_of? URI
     h[k] = v.to_s
   end
  end
  [controller.status, controller.headers, [controller.body.to_s]]
end
```

Advantages:

Simplicity (No code generation like Rails)

Speed (Low memory consumption and fast)

Power (Due to flexibility)

Disadvantage is the increased effort required to accomplish a certain task when compared to web frameworks.

Rack application is helpful in production applications with speed bottlenecks or speed-intensive end points.

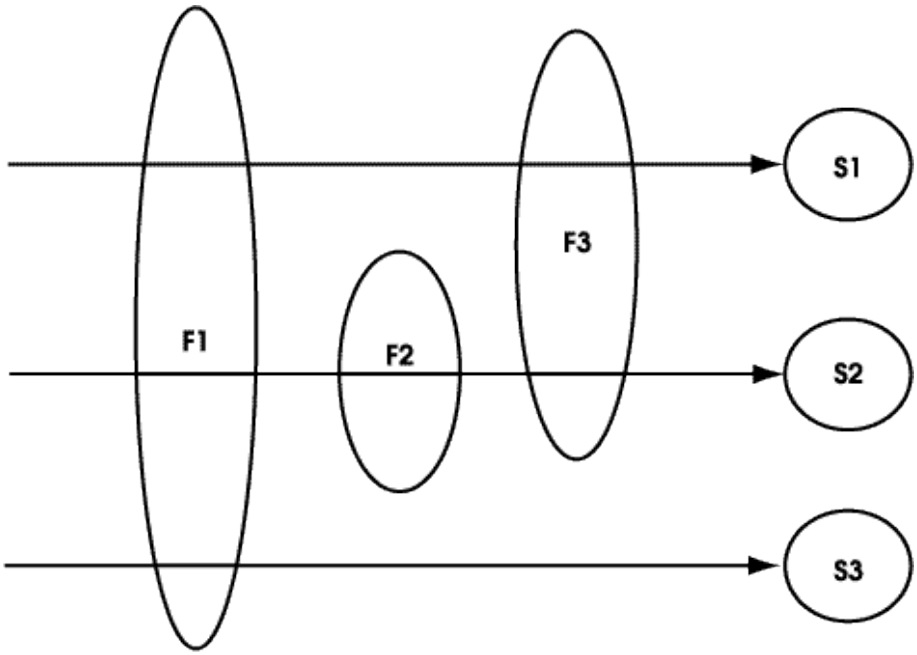
5. Middleware

Middleware is just a Rack application. Constructor takes the next app down. Can modify request or response. Can call layer below or just return.

One of the things Rack enables is Rack Middleware which is a filter that can be used to intercept a request and alter the response as a request is made to an application. Middleware acts like a filter and can pass the request on to an application, then filter the response that is returned. Rack middleware can do nothing (usually after a check). It can send response directly. It can alter environment variables. It can alter response (body, headers and status code)

Rack filters enable reuse and results in modular architecture. This improves efficiency. It has the following profound impacts:

- Web framework agnostic Rack apps can be reused in any framework.
- Can be assembled in different combinations of middleware with a web framework, differnt variants to suit different needs.
- You can combine several services provided by different web application frameworks to build a larger system.



So you can have multiple middleware (F1, F2, F3) mapped to different web frameworks Rails, Sinatra, Legacy Web app (S1, S2, S3). As you can see the request does not have to go through all the middleware and the request could also be bounced off and not reach the last layer at all.[2]

Rack Middleware is Chainable and Composable like Unix Pipelines. Rack Middleware is pipeline for HTTP request and response processing.

6. Instant Gratification

(a) Installing Rack

First install the Rack gem:

[sudo] gem install rack

(b) Rack Handler

You will now see how to use rack package, start irb:

$ irb

irb> require 'rubygems'

=> true

irb> require 'rack'

=> true

We can now check all the embedded Rack Handlers in Rack gem.

irb> Rack::Handler.constants

=> ["LSWS", "CGI", "SwiftipliedMongrel", "FastCGI", "Thin", "EventedMongrel", "WEBrick", "SCGI", "Mongrel"]

All Rack Handlers have a *run* method, so you can call *run* on any of these built-in handlers:

Rack::Handler::Mongrel.run app, :Port => 80

Rack::Handler::WEBrick.run ...

Rack::Handler::Thin.run ...

---

2. Figure from Java Servlet filter 2.3 specification

The first argument to the *run* method is Rack app and the second argument is options to run your program.

(c) Hello Rack

Rack specification defines a Rack application as: "A Rack application is a Ruby **object** (not a class) that responds to call method. It takes exactly one argument, the environment and returns an **Array** of exactly three values: the **status,** the **headers,** and the **body."**

Let's consider the first sentence: A Rack application is a Ruby **object** (not a class) that responds to call method. This implies that the object can be any of the following:

- A lambda or Proc object
- A method object
- Any object that has a call method

Lets do the bare minimum to meet this requirement: use an empty lambda. Because it can accept call.

irb > Rack::Handler::WEBrick.run lambda{}, :Port => 3002

[2010-10-03 02:58:29] INFO WEBrick 1.3.1

[2010-10-03 02:58:29] INFO ruby 1.8.7 (2010-06-23) [i686-darwin10.4.0]

[2010-10-03 02:58:29] INFO WEBrick::HTTPServer#start: pid=17111 port=3002

The second parameter is the options in this case we are specifying the port number. The log shwos that the server has started and is listening on port 3002.

Open your browser and type: http://localhost:3002

From the error message we can see that it is calling the each method when it is processing the first argument in our case it is our empty lambda Rack app.

Let's consider the second sentence of the Rack application definition: It takes exactly one argument, the environment and returns an **Array** of exactly three values: the **status,** the **headers,** and the **body.**

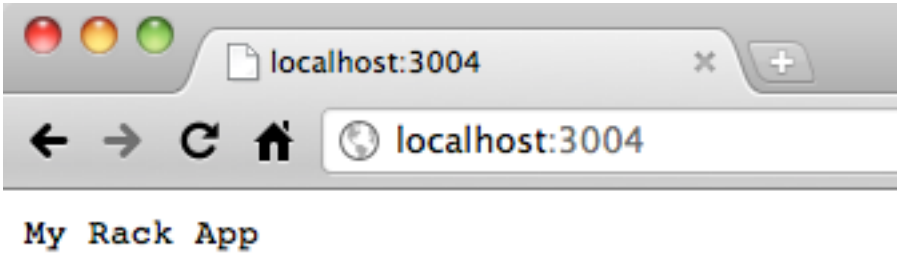
So our lambda needs to accept one parameter, namely the environment and it needs to return an Array that contains the following:

- Status - HTTP protocol defines the status code
- Headers - Hash that contains the HTTP headers
- Body - An array that returns string for every call of each method

To satisfy this requirements in the spec, let's rewrite the Rack app as follows:

```
irb > rack_app = lambda {|env| [200, {}, ["My Rack App"]] }
 => #<Proc:0x00000001012367a0@(irb):3>
irb > Rack::Handler::WEBrick.run rack_app, :Port => 3004
[2010-10-03 03:18:25] INFO  WEBrick 1.3.1
[2010-10-03 03:18:25] INFO  ruby 1.8.7 (2010-06-23) [i686-darwin10.4.0]
[2010-10-03 03:18:25] INFO  WEBrick::HTTPServer#start: pid=17234 port=3004
```

This time you see our Rack app run successfully and the output in the browser:

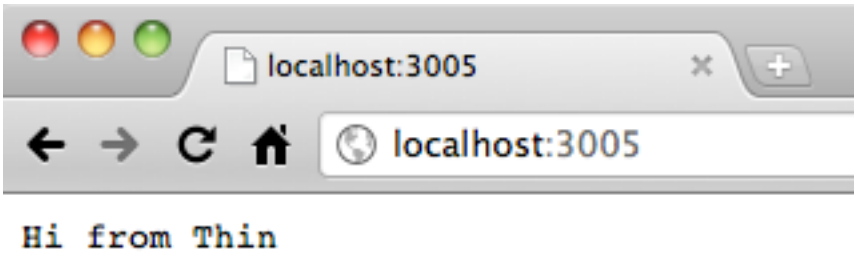


Note that we passed empty hash for the headers parameter.

If you have Thin server installed you can run it on Thin.

```
irb> require 'thin'
```

```
=> true
irb> require 'rubygems'
=> false
irb> require 'rack'
=> false
irb> Rack::Handler::Thin.run lambda{|env| [200, {}, ["Hi from Thin"]]}, :Port => 3005
>> Thin web server (v1.2.7 codename No Hup)
>> Maximum connections set to 1024
>> Listening on 0.0.0.0:3005, CTRL+C to stop
```



In this demo we wrote our first Rack app to meet the requirements specified in the Rack application definition.

(d) Method Object

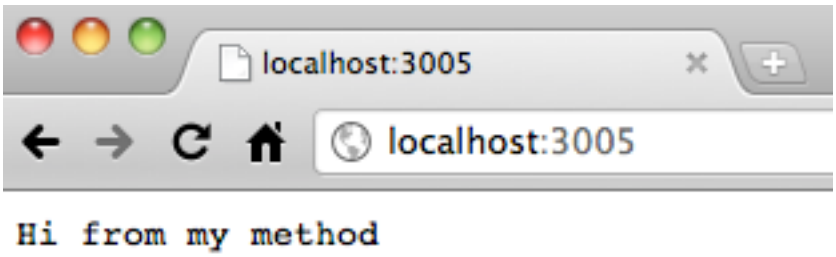
In addition to using lambda we can also use method object.

```
irb> method(:my_method)
=> #<Method: Object#my_method>
```

The function method(:any_method) returns a method Object, it can respond to call, therefore we can use it for a Rack application.

```
irb> def my_method(env)
irb> [200, {}, ["Hi from my method"]]
irb> end
=> nil
```

```
irb> method(:my_method).call({})
=> [200, {}, ["Hi from my method"]]

irb> method(:my_method).call({})
=> [200, {}, ["Hi from my method"]]
irb> method(:my_method)
=> #<Method: Object#my_method>
irb> rack_app = method(:my_method)
=> #<Method: Object#my_method>
irb> Rack::Handler::Thin.run rack_app, :Port => 3005
>> Thin web server (v1.2.7 codename No Hup)
>> Maximum connections set to 1024
>> Listening on 0.0.0.0:3005, CTRL+C to stop
```



The third case that we saw earlier was that any object that can respond to call can be used as a valid Rack app.

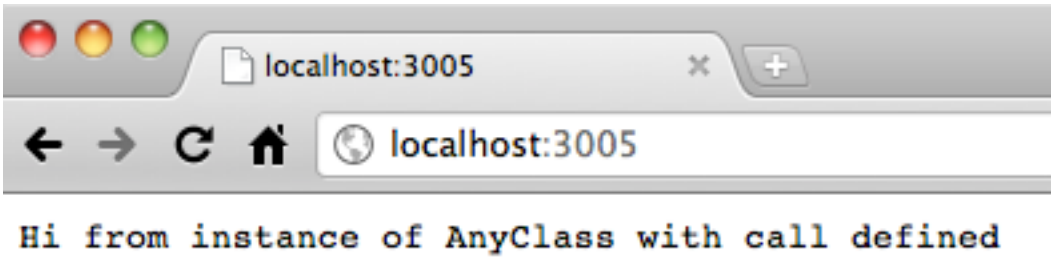
```
irb> class AnyClass
irb>   def call(env)
irb>     [200, {}, ["Hi from instance of AnyClass with call defined"]]
irb>   end
irb> end
=> nil
irb> rack_app = AnyClass.new
=> #<AnyClass:0x000001011631c0>
```

irb> Rack::Handler::Thin.run rack_app, :Port => 3005

>> Thin web server (v1.2.7 codename No Hup)

>> Maximum connections set to 1024

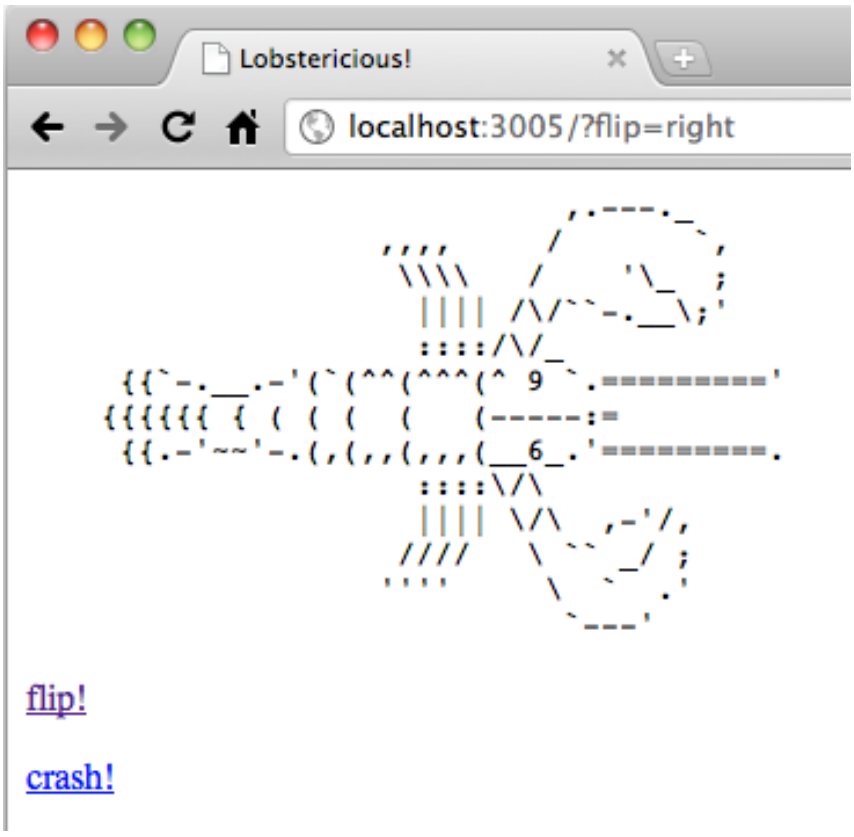
>> Listening on 0.0.0.0:3005, CTRL+C to stop



irb> require 'rack/lobster'

=> true

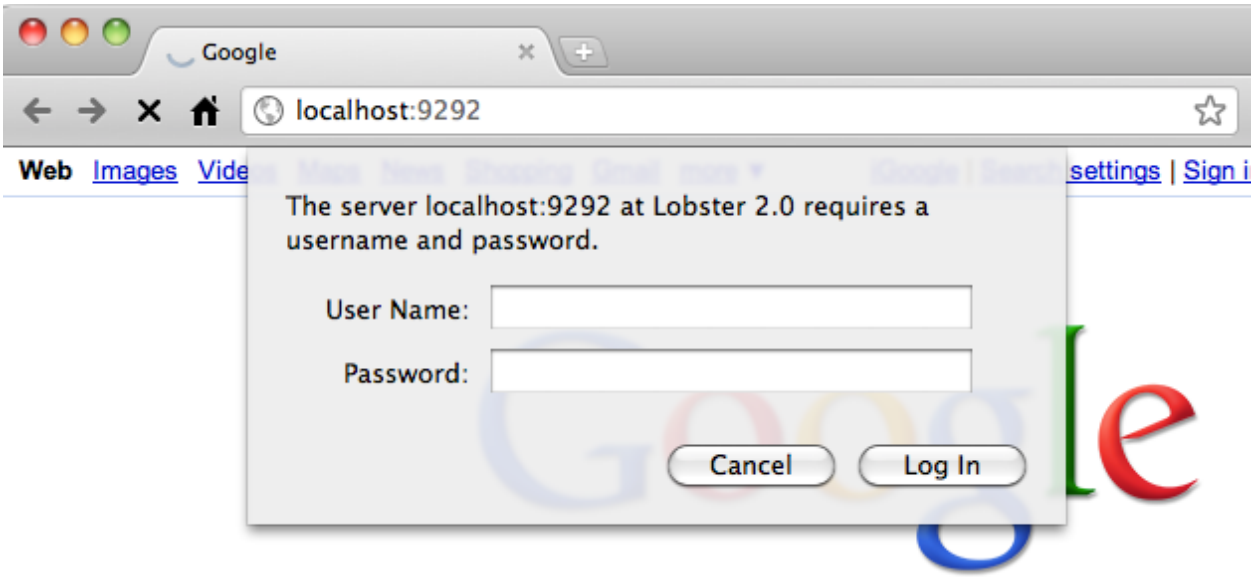
irb> Rack::Handler::Thin.run Rack::Lobster.new, :Port => 3005



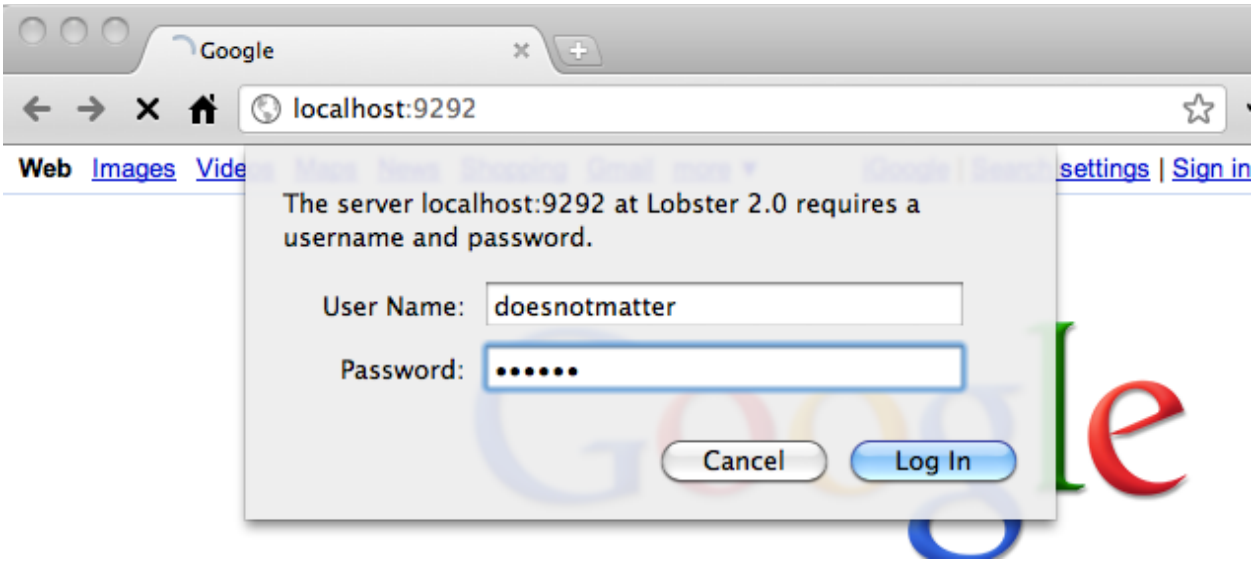
Let's now run the example that comes with Rack distribution. Protected lobster example shows how you can use http basic authentication in your Rack app.

```ruby
require 'rubygems'
require 'rack'
require 'rack/lobster'

lobster = Rack::Lobster.new

protected_lobster = Rack::Auth::Basic.new(lobster) do |username, password|
  'secret' == password
end

protected_lobster.realm = 'Lobster 2.0'

pretty_protected_lobster = Rack::ShowStatus.new(Rack::ShowExceptions.new(protected_lobster))

Rack::Handler::WEBrick.run pretty_protected_lobster, :Port => 9292
```

Run this program as : ruby protected_lobster.rb



You are prompted with http basic authentication popup.



[flip!]

[crash!]

When the login using 'secret' as the password, I can see the protected lobster.

Rack is also an implementation. The Rack gem provides helper classes for : Request / Response wrappers, logging, authentication (basic and digest), cookies, sessions, mock requests and responses.

Rack gem gives you *rackup* command which lets you start your app on any supported application server. *rackup* is a useful tool for running Rack applications, which uses the Rack::Builder DSL to configure middleware and build up applications easily. rackup automatically figures out the environment it is run in, and runs your application as FastCGI, CGI, or standalone with Mongrel or WEBrick--all from the same configuration.

Running The Application

```
use Rack::ContentLength
app = lambda { |env| [200, { 'Content-Type' => 'text/html' }, 'Hello World'] }
run app
```

Save the code above in a file named 'basic_rack.ru', the *.ru extension corresponds to Rack's rackup executable. These files are regular Ruby files.

Start the app using the following command, then visit http://0.0.0.0:3000/

```
$ rackup basic_rack.ru -p 3000
```

The Rack::ContentLength sets the content length header we will discuss them in detail chapter 5 on middleware. As you can see in the line 3, we use the *run* method to run the application. We can also use the Proc object instead of lambda and the effect is the same. In this case the line 2 would be:

app = Proc.new {|env| [200, {'Content-Type' => 'text/plain'}, 'Hello World!']}

Now let's look at an example where we can use variables.

```ruby
class HelloWorld
  def initialize(name)
    @name = name
  end

  def call(env)
    [200, {'Content-Type' => 'text/plain'}, ["Hello #{@name}"]]
  end
end
```

```
app = HelloWorld.new("Bugs")
run app
```

will produce "Hello Bugs" in the browser.

Create a config.ru with the following contents:

```
require 'rack/lobster'

run Rack::Lobster.new
```

Run:

rackup

Browse to http://localhost:9292 to see the cute lobster that knows gymnastics.

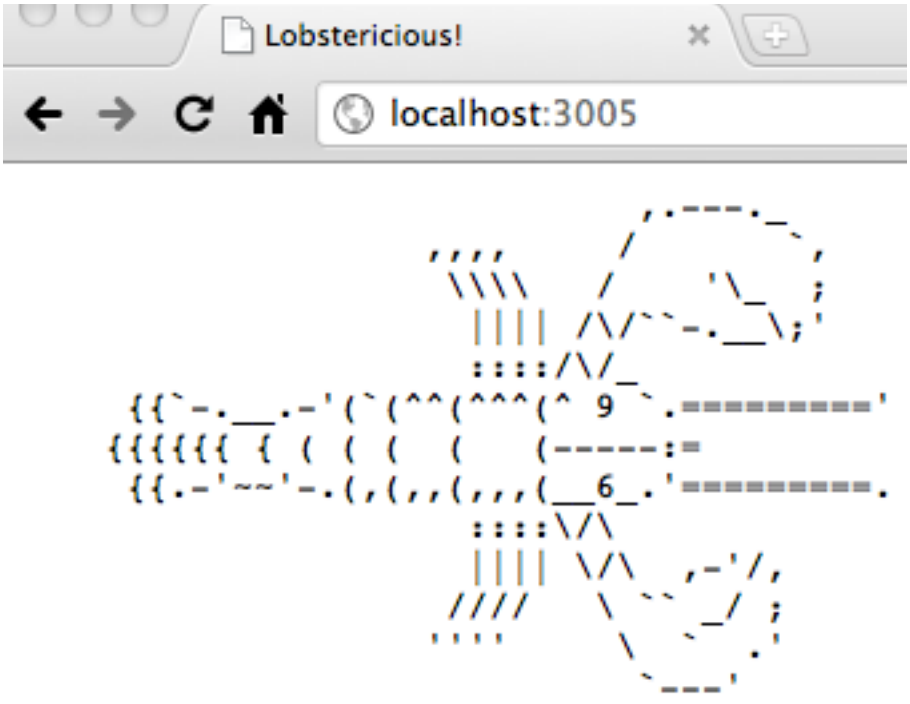
Let's run the previous http basic authentication example using rackup.

```
require 'rubygems'
require 'rack/lobster'

use Rack::ShowExceptions
use Rack::Auth::Basic, "Lobster 2.0" do |username, password|
  'test' == password
end

run Rack::Lobster.new
```

Run:

rackup protected_lobster.ru -p 3005

When you provide the 'test' as the password for the http basic authentication, you can see the lobster in the browser.



You can use middleware to filter the request. For example you can add logging and detailed exception support with just two lines in config.ru file:

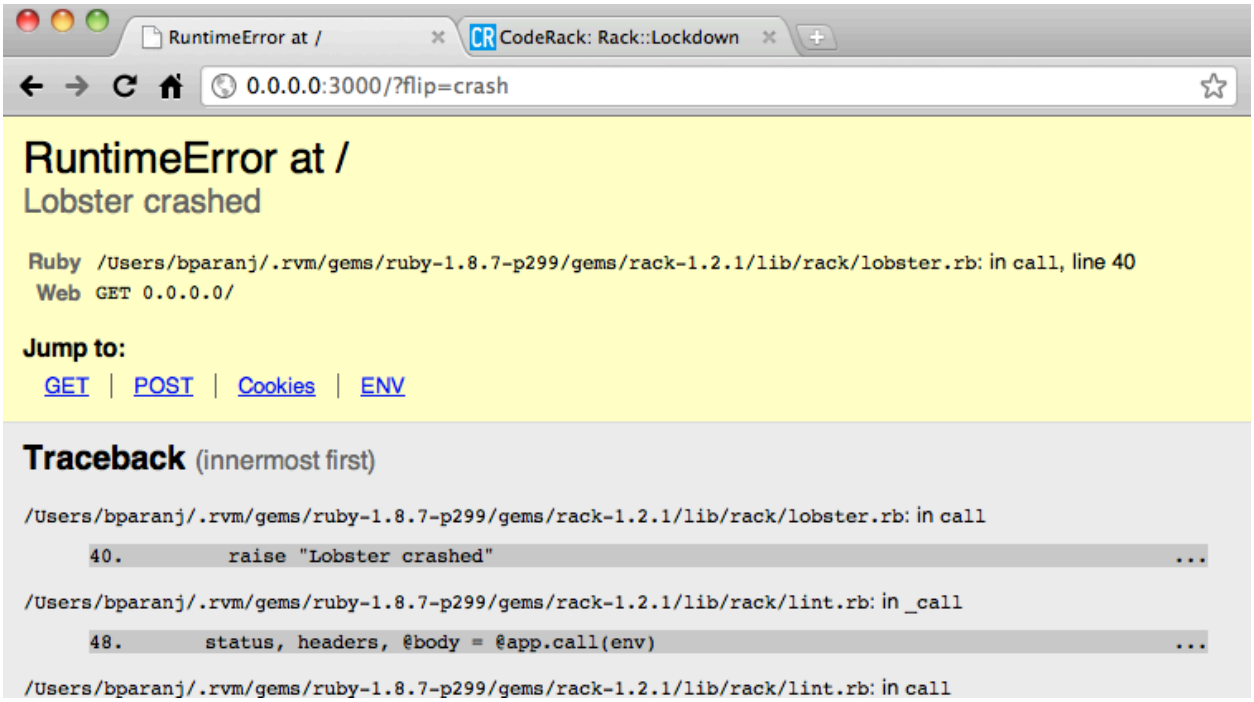
use Rack::CommonLogger, STDOUT

use Rack::ShowExceptions

You can also validate your application, the requests and responses according to the Rack spec automatically using Rack::Lint middleware.[3]

When you use Rack::ShowExceptions you get the stack trace as shown in the screenshots below:

3. Macournoyer blog post: Rack, the frameworks framework

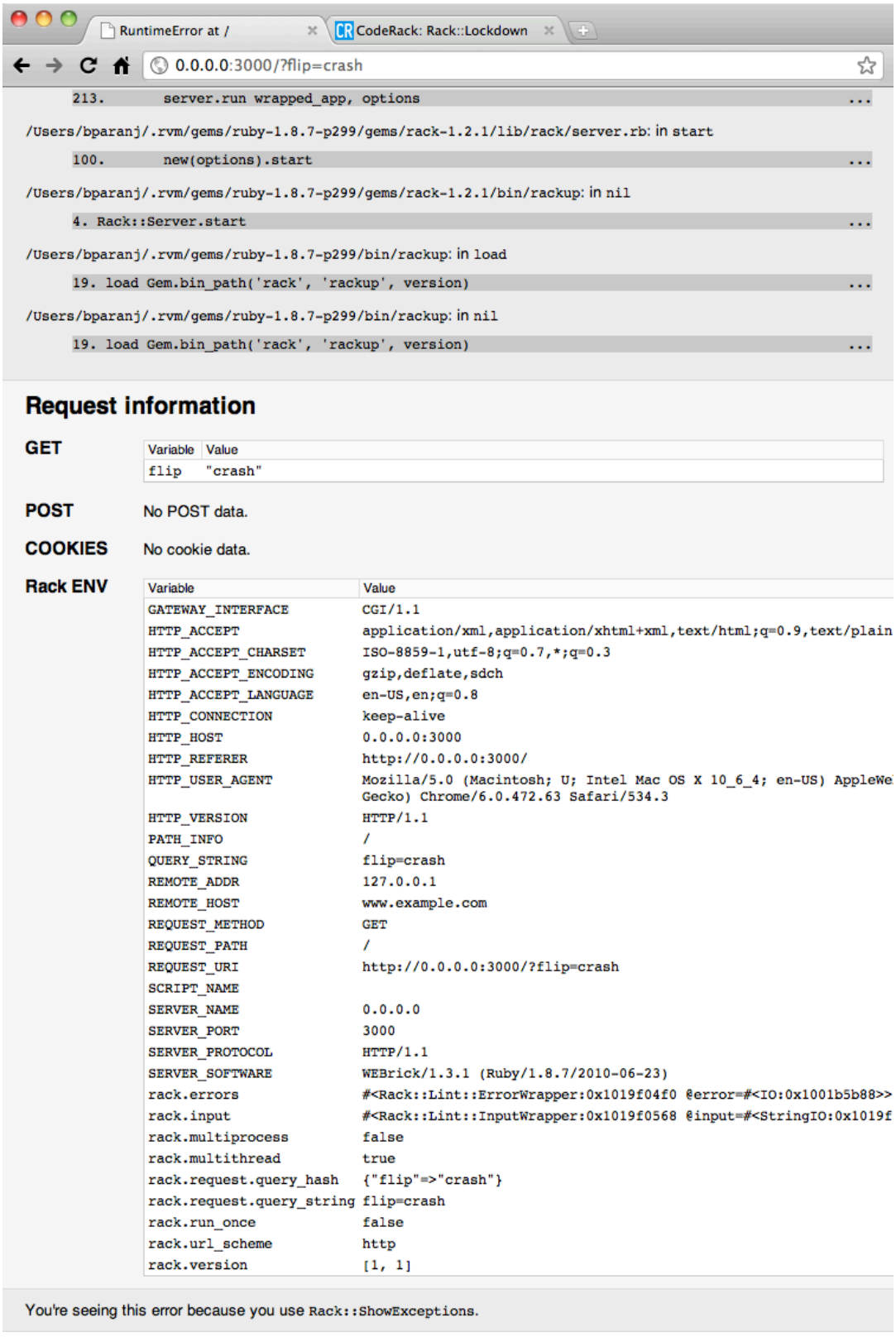


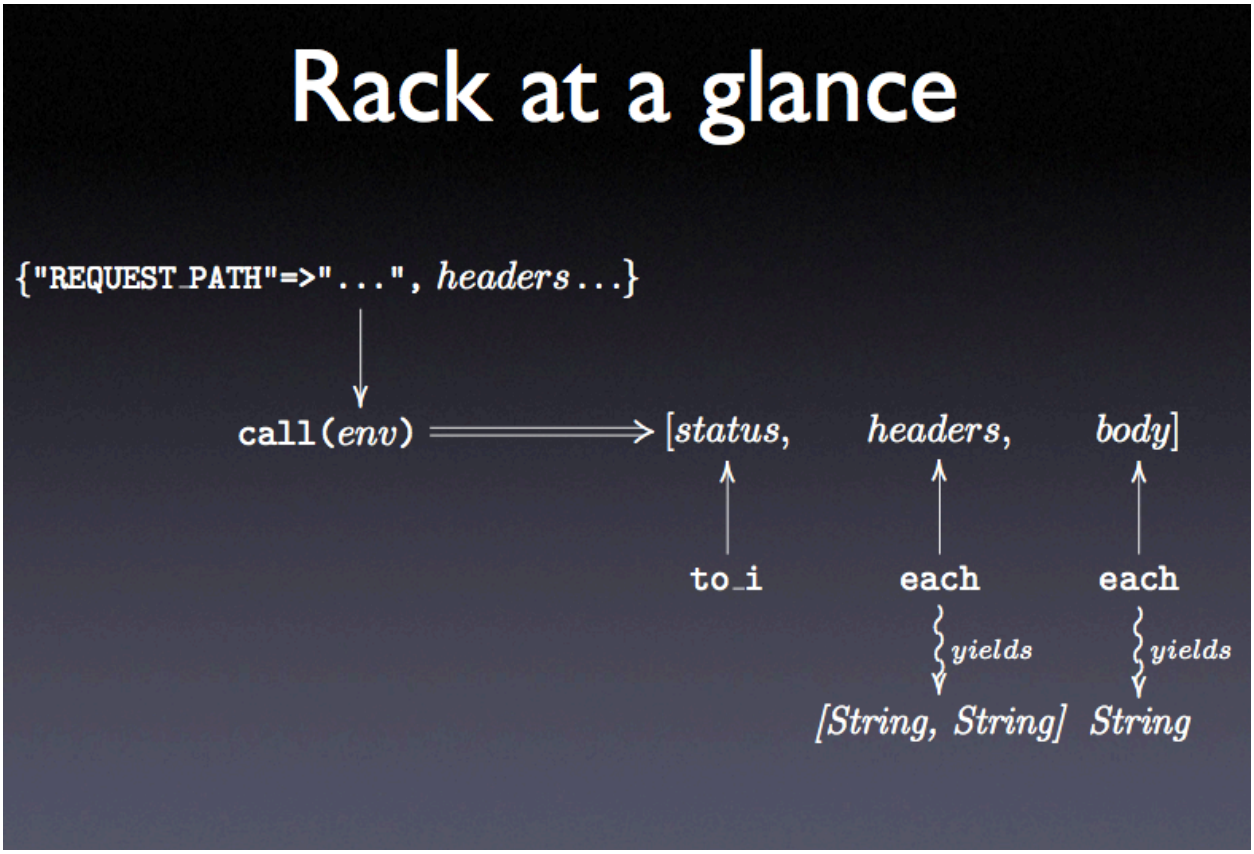
Run the code config.ru in ch1 folder by:

rackup config.ru -p 3000

and click on crash! link gives you the same result. This is because the Rack::Lobster uses Rack::ShowExceptions like this:

```
if $0 == __FILE__
  require 'rack'
  require 'rack/showexceptions'
  Rack::Handler::WEBrick.run
                        Rack::ShowExceptions.new(Rack::Lint.new(Rack::Lobster.new)),
                        :Port => 9292
end
```


RuntimeError at /
CodeRack: Rack::Lockdown
0.0.0.0:3000/?flip=crash
213. server.run wrapped_app, options ...
/Users/bparanj/.rvm/gems/ruby-1.8.7-p299/gems/rack-1.2.1/lib/rack/server.rb: in start
100. new(options).start ...
/Users/bparanj/.rvm/gems/ruby-1.8.7-p299/gems/rack-1.2.1/bin/rackup: in nil
4. Rack::Server.start ...
/Users/bparanj/.rvm/gems/ruby-1.8.7-p299/bin/rackup: in load
19. load Gem.bin_path('rack', 'rackup', version) ...
/Users/bparanj/.rvm/gems/ruby-1.8.7-p299/bin/rackup: in nil
19. load Gem.bin_path('rack', 'rackup', version) ...
Request information
GET
Variable Value
flip "crash"
POST
No POST data.
COOKIES
No cookie data.
Rack ENV
Variable Value
GATEWAY_INTERFACE CGI/1.1
HTTP_ACCEPT application/xml,application/xhtml+xml,text/html;q=0.9,text/plain
HTTP_ACCEPT_CHARSET ISO-8859-1,utf-8;q=0.7,*;q=0.3
HTTP_ACCEPT_ENCODING gzip,deflate,sdch
HTTP_ACCEPT_LANGUAGE en-US,en;q=0.8
HTTP_CONNECTION keep-alive
HTTP_HOST 0.0.0.0:3000
HTTP_REFERER http://0.0.0.0:3000/
HTTP_USER_AGENT Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_4; en-US) AppleWe
Gecko) Chrome/6.0.472.63 Safari/534.3
HTTP_VERSION HTTP/1.1
PATH_INFO /
QUERY_STRING flip=crash
REMOTE_ADDR 127.0.0.1
REMOTE_HOST www.example.com
REQUEST_METHOD GET
REQUEST_PATH /
REQUEST_URI http://0.0.0.0:3000/?flip=crash
SCRIPT_NAME
SERVER_NAME 0.0.0.0
SERVER_PORT 3000
SERVER_PROTOCOL HTTP/1.1
SERVER_SOFTWARE WEBrick/1.3.1 (Ruby/1.8.7/2010-06-23)
rack.errors #<Rack::Lint::ErrorWrapper:0x1019f04f0 @error=#<IO:0x1001b5b88>>
rack.input #<Rack::Lint::InputWrapper:0x1019f0568 @input=#<StringIO:0x1019f
rack.multiprocess false
rack.multithread true
rack.request.query_hash {"flip"=>"crash"}
rack.request.query_string flip=crash
rack.run_once false
rack.url_scheme http
rack.version [1, 1]
You're seeing this error because you use Rack::ShowExceptions.



call takes an environment Hash representing a request. It returns a three part array with each of the parts of a HTTP response.

Here is the code for Rack Handler for Thin server:

```
require "thin"
require "rack/content_length"
require "rack/chunked"

module Rack
  module Handler
    class Thin
      def self.run(app, options={})
        server = ::Thin::Server.new(options[:Host] || '0.0.0.0',
                                    options[:Port] || 8080,
                                    app)
        yield server if block_given?
        server.start
```

```
      end
    end
  end
end
```

Rack gem provides helper classes

- Request / Response wrappers
- Logging
- Authentication (basic and digest)
- Cookies and sessions
- Mock requests and responses

II. Rack Spec

In this chapter we will take a closer look at the environement variables, the request and the response as defined in the Rack specification.

1. Environment

Let's write a simple program to inspect the environment variables that is passed to the call method. Here is the code inspect the environment variable to see the key value pairs in the environment hash.

```
require "rubygems"
require "rack"
require "thin"

Rack::Handler::Thin.run lambda {|env| [200,{},[env.inspect]]} , :Port=>3005
```

Here is the output of running the code above that has been modified to see each key, value pair in its own line:

{"SERVER_SOFTWARE"=>"thin 1.2.7 codename No Hup",

"SERVER_NAME"=>"localhost",

"rack.input"=>#<StringIO:0x00000101a915d0>,

"rack.version"=>[1, 0],

"rack.errors"=>#<IO:<STDERR>>,

"rack.multithread"=>false,

"rack.multiprocess"=>false,

"rack.run_once"=>false,

"REQUEST_METHOD"=>"GET",

"REQUEST_PATH"=>"/",

"PATH_INFO"=>"/",

"REQUEST_URI"=>"/",

"HTTP_VERSION"=>"HTTP/1.1",

"HTTP_HOST"=>"localhost:3005",

"HTTP_USER_AGENT"=>"Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6; en-US; rv:1.9.2.8) Gecko/20100722 Firefox/3.6.8",

"HTTP_ACCEPT"=>"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
"HTTP_ACCEPT_LANGUAGE"=>"en-us,en;q=0.5",

"HTTP_ACCEPT_ENCODING"=>"gzip,deflate",

"HTTP_ACCEPT_CHARSET"=>"ISO-8859-1,utf-8;q=0.7,*;q=0.7",
"HTTP_KEEP_ALIVE"=>"115",

"HTTP_CONNECTION"=>"keep-alive",

"HTTP_COOKIE"=>"user_id=BAg-iCDE5OQ%3D%3D-49190645711c19ecf60df4253fe3148efdaa46b0;_blog_session=BAh7Ci-IQX2NzcmZfdG9rZW4iMWlvS3NwS1JlN2VEYTJLd0lOd012dGdFZWQrSlZrU1Ax-cXBEZ2ZwR0d2b2c9Ig9zZXNzaW9uX2lkIiViYjcyMGRmYzE4N2MxNmVhMWVjM-TUzNGJkZWViOGU0NSIZd2FyZGVuLnVzZXIudXNlci5rZXlbByIJVXNlcmkGIg1jaGV-ja19pZGlpIgpmbGFzaElDOiVBY3Rpb25EaXNwYXRjaDo6Rmxhc2g6OkZsYXNoSGFzaHs-GOgtub3RpY2UiIVRoYW5rIHlvdSBmb3IgY29udGFjdGluZyB1cy4GOgpAdXNlZG86CFNl-dAY6CkBoYXNoewY7BlQ%3D--96b3d70764bdf01d361dddf7066285689a093318",

"HTTP_CACHE_CONTROL"=>"max-age=0",

"GATEWAY_INTERFACE"=>"CGI/1.2",

"SERVER_PORT"=>"3005",

"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:0x00000101a6f228>}

From the output you can see that there are three categories, CGI-ish type variables, Rack specific variables (rack.xxx) and application specific variables (in this case Thin server related async.callback and async.close). The first two categories are specified in the Rack spec, whereas the application specific variables are used by the developers. For instance so you can add your own variables to the environment hash that is applicable to only your Rack application.

(a) CGI-like Headers

The environment is required to include the following variables. From the Rack spec:

REQUEST_METHOD:

The HTTP request method, such as "GET" or "POST". This cannot ever be an empty string, and so is always required.

SCRIPT_NAME:

The initial portion of the request URL‘s "path" that corresponds to the application object, so that the application knows its virtual "location". This may be an empty string, if the application corresponds to the "root" of the server.

PATH_INFO:

The remainder of the request URL‘s "path", designating the virtual "location" of the request‘s target within the application. This may be an empty string, if the request URL targets the application root and does not have a trailing slash. This value may be percent-encoded when I originating from a URL.

QUERY_STRING:

The portion of the request URL that follows the ?, if any. May be empty, but is always required!

SERVER_NAME, SERVER_PORT:

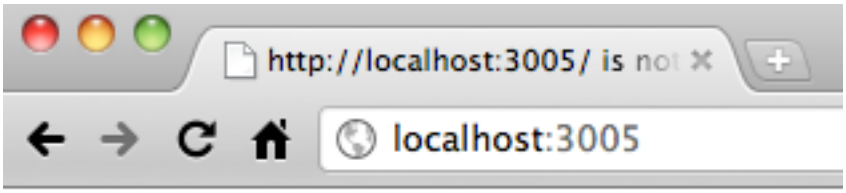
When combined with SCRIPT_NAME and PATH_INFO, these variables can be used to complete the URL. Note, however, that HTTP_HOST, if present, should be used in preference to SERVER_NAME for reconstructing the request URL. SERVER_NAME and SERVER_PORT can never be empty strings, and so are always required.

HTTP_ Variables:

Variables corresponding to the client-supplied HTTP request headers (i.e., variables whose names begin with HTTP_). The presence or absence of these variables should correspond with the presence or absence of the appropriate HTTP header in the request.

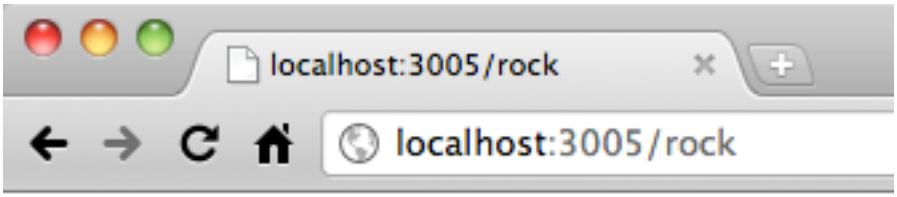
We can write a program to demonstrate these variables.

```ruby
require "rubygems"
require "rack"
require "thin"

cgi_inspector = lambda do |env|

  [200, {}, ["Your request:
          http method is #{env['REQUEST_METHOD']}
          path is #{env['PATH_INFO']}
          params is #{env['QUERY_STRING']}
         "]]
end

Rack::Handler::Thin.run cgi_inspector, :Port => 3005
```



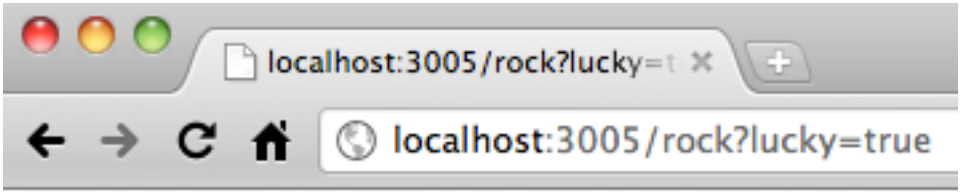

```
Your request:
http method is GET
path is /
params is
```

The default request shows that the browser made the get request to the server. Next let's add on the path to the default request like this:




```
Your request:
http method is GET
path is /rock
params is
```

Now the path info has the value of rock. Let's add parameters to the request:




```
Your request:
http method is GET
path is /rock
params is lucky=true
```

This displays the parameter that we passed in on the URL The environment variables gives you access to user requests, path names, query parameters etc. You can use them to write programs, however this low-level access makes the programming tedious even for common tasks such as parsing the query parameters, maintaining user session information, filling the appropriate HTTP response headers and so on. Rack provides a higher level convenience API to help you quickly and easily deal with application logic, later in this chapter we will look at two important classes, Request and Response.

(b) Rack Specific Variables

Rack specification states that the Rack environment must include the following Rack specific variables:

rack.version: The Array [1,1], representing this version of Rack.

rack.url_scheme: http or https, depending on the request URL.

rack.input: The input stream is an IO-like object that contains the raw HTTP POST data.

rack.errors: The error stream must respond to puts, write and flush.

rack.multithread: true if the application object may be simultaneously invoked by another thread in the same process, false otherwise.

rack.multiprocess: true if an equivalent application object may be simultaneously invoked by another process, false otherwise.

rack.run_once: true if the server expects (but does not guarantee!) that the application will only be invoked this one time during the life of its containing process. Normally, this will only be true for a server based on CGI (or something similar).

Additional environment specifications have been approved for use by standardized middleware APIs. None of these are required to be implemented by the server. These are optional:

rack.session: A hash like interface for storing request session data. The store must implement: store(key, value) (aliased as []=); fetch(key, default = nil) (aliased as []); delete(key); clear;

rack.logger: A common object interface for logging messages. The object must implement:

info(message, &block)

debug(message, &block)

warn(message, &block)

error(message, &block)

fatal(message, &block)

(c) Application Specific Variables

From the Rack spec:

The server or the application can store their own data in the environment, too. The keys must contain at least one dot, and should be prefixed uniquely. The prefix rack. is reserved for use with the Rack core distribution and other accepted specifications and must not be used otherwise. The environment must not contain the keys HTTP_CONTENT_TYPE or HTTP_CONTENT_LENGTH (use the versions without HTTP_). The CGI keys (named without a period) must have String values. There are the following restrictions:

* rack.version must be an array of Integers.
* rack.url_scheme must either be http or https.
* There must be a valid input stream in rack.input.

* There must be a valid error stream in rack.errors.

* The REQUEST_METHOD must be a valid token.

* The SCRIPT_NAME, if non-empty, must start with /

* The PATH_INFO, if non-empty, must start with /

* The CONTENT_LENGTH, if given, must consist of digits only.

* One of SCRIPT_NAME or PATH_INFO must be set. PATH_INFO should be / if SCRIPT_NAME is empty. SCRIPT_NAME never should be /, but instead be empty.

Things to do with env

Use Rack::Request to construct a request object, pull out GET and POST data. Rack borrows code from Camping, Rails and IOWA in this department.

```
request = Rack::Request.new(env)
request.GET
requeste.POST
```

Use Rack::Response to talk to clients about cookies. Additionally, you can use this instead of the [200, {}, ‘’} return value. Like this :

```
response = Rack::Response.new(“Hello World”)
Response.set_cookie(‘sess-id’, ‘abcde’)
Response.finish
```

```
%w(rubygems rack).each { |gem| require gem }
app = lambda { |evn| [200, {}, ‘Hello World”] }
Rack::Handler::Mongrel.run(app, :port => 3000)
```

The three lines of code above is a complete Rack app. Just run it.

2. Request

The request object encapsulates all information from the client request. In the HTTP protocol, this information is transmitted from the client to the server in the HTTP headers and the message body of the request.

Rack::Request provides a convenient interface to a Rack environment. It is stateless, the environment *env* passed to the constructor will be directly modified. It is your friend if you want to write Rack applications directly.

```
1    request = Rack::Request.new(env)
2    request.post?
3    request.params["data"]
```

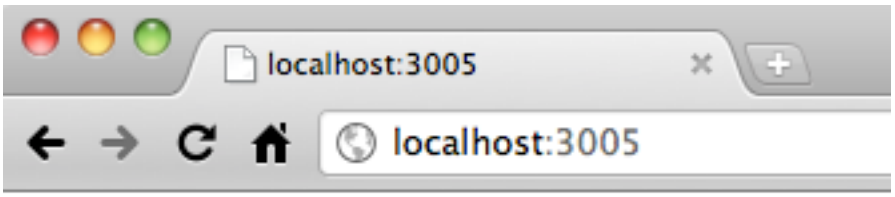
The environment hash passed will store a reference to the Request object instantiated so that it will only instantiate if an instance of the Request object doesn't already exist.

In this example you can see how you can easily access the value of request parameter data and also check if the request is a post. Instead of accessing the env variable you use API provided by the request object. Request parameters are strings sent by the client to the server as part of its request. The parameters are stored as a set of name-value pairs.

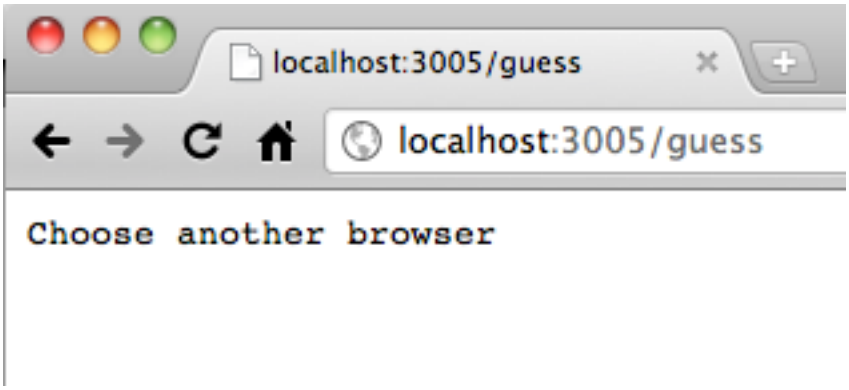
Let us write a simple program to guess our favorite browser by using the Request object.[4]

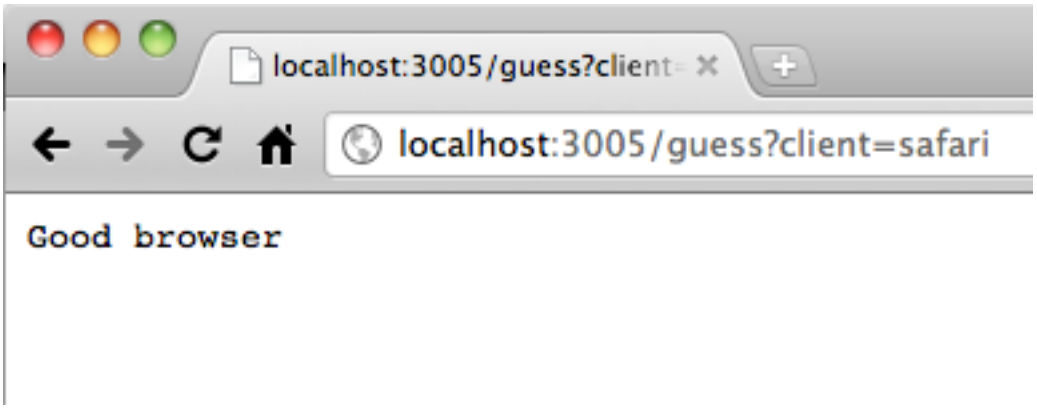
4. http://rack.rubyforge.org/doc/classes/Rack/Request.html

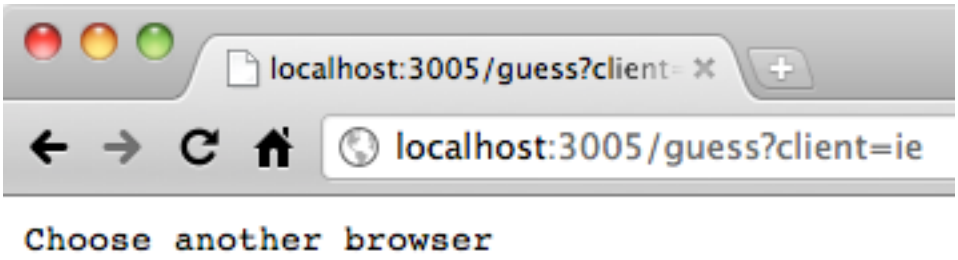
```ruby
require "rubygems"
require "rack"
require "thin"

rack_app = lambda do |env|
  request = Rack::Request.new(env)
  if request.path_info == '/guess'
     client = request['client']
     if client && client.downcase == 'safari'
       [200, {}, ["Good browser"]]
     else
       [200, {}, ["Choose another browser"]]
     end
   else
     [200, {}, ["You have to guess something"]]
   end
 end

 Rack::Handler::Thin.run rack_app, :Port => 3005
```




```
You have to guess something
```







If the user requests teh path_info that is not /guess, then we return "you have to guess something". When a user enters a query parameter that does not include client=safari, we will ask them to choose another browser.

We can query the current request object to find out the request type.

Method	Explanation
request_method	This could be GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE
get?	Whether the HTTP request is a get

post?	Whether the HTTP request is a post
put?	Whether the HTTP request is a put
delete?	Whether the HTTP request is a delete
head?	Whether the HTTP request is a head
options?	Whether the HTTP request is a options
trace?	Whether the HTTP request is a trace
xhr?	Whether the request is a XMLHttpRequest? (AJAX request)

3. Response

In the previous example we returned a hard-coded strings as the response. But in a complex Rack application, we may need to do more to control the response. For example, the need to set various HTTP response headers, handling cookies and so on. Response is your friend if you want to write Rack applications directly.

The response object encapsulates all information to be returned from the server to the client. In the HTTP protocol, this information is transmitted from the server to the client either by HTTP headers or the message body of the request.

Rack::Response provides a convenient interface to create a Rack response. It allows setting of headers and cookies, and provides useful defaults (a OK response containing HTML).

Response provides two methods to generate the response body:

- Directly set by calling response.body - In this case you have to set the response header content length
- Use response.write to write the contents of an incremental, automatic filling Content_Length header value.

You can use Response#write to iteratively generate your response, but note that this is buffered by Rack::Response until you call finish. finish however can take a block inside with calls to write that are synchronous with the Rack response.

Regardless of the method, your application's *call* should end returning Response#finish. Note that you should not mix these two methods. Browser needs Content-Length header to decide how much data to read from the server and this is a must.

Let's first look at how to directly set response.body:

```ruby
require "rubygems"
require "rack"
require "thin"

rack_app = lambda do |env|
  request = Rack::Request.new(env)
  response = Rack::Response.new
  body = "---------------Header-------------<br/>"

  if request.path_info == '/hello'
    body << "Saying hi"
    client = request['client']
    body << "from #{client}" if client
  else
    body << "You need to provide the client information"
  end
  body << "<br/>----------footer-----------------"
  response.body = [body]
  response.headers['Content-Length'] = body.bytesize
  response.to_a
end

Rack::Handler::Thin.run rack_app, :Port => 3005
```

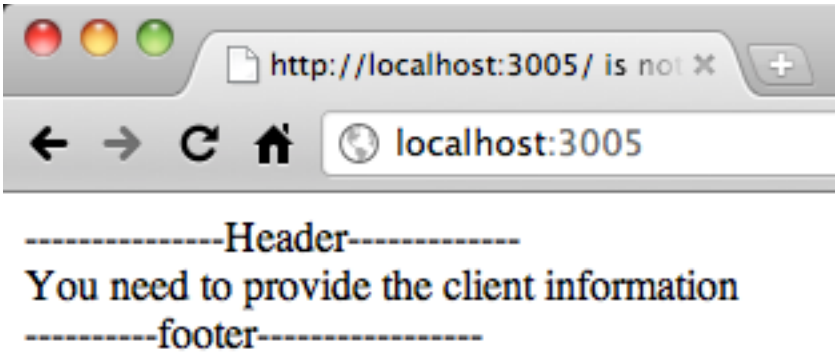
If you run the program, you will see this in the browser error:

Internal Server Error Internal Server Error

private method `split' called for 88:Fixnum private method `split 'called for 88: Fixnum

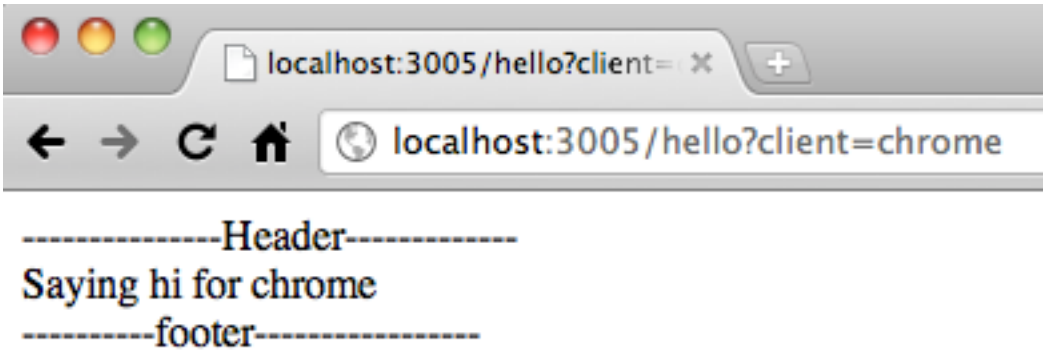
The reason is that the Content-Length value must be string, so change the line 19 body.bytesize to : body.bytesize.to_s

Now, when we run the program and make a request without /hello in the URL, we get:



This is the same output when we make the request with /hello in the URL but no client query parameter.

Here is output when we provide the value for client parameter:



Let's now write the same program using response.write. Here is the code:

```ruby
require "rubygems"
require "rack"
require "thin"

rack_app = lambda do |env|
  request = Rack::Request.new(env)
  response = Rack::Response.new
  response.write "---------------Header-------------<br/>"

  if request.path_info == '/hello'
    response.write "Saying hi "
    client = request['client']
    response.write "for #{client}" if client
  else
    response.write "You need to provide the client information"
  end
  response.write "<br/>----------footer-----------------"
  response.finish
end

Rack::Handler::Thin.run rack_app, :Port => 3005
```

The behavior is still the same. Notice that in the previous example we generated the body string and we set it as the response body. We also had to set the content-length header value. In this example we directly write the body by using the response.write method. We also used finish instead of to_a method, to_a is an alias for finish method.

Rack::Request & Rack::Response is used to write Rack apps directly. Here is an example:

```
def call(env)
  req = Rack::Request.new(env)
  res = Rack::Response.new

  if req.get?
    res.write "Hi #{req.GET["name"]}"
  elsif req.post?
    file = req.POST["file"]
```

```
    FileUtils.cp file[:tempfile].path, File.join(UPLOADS, file[:filename])
    res.write "Uploaded."
  end
  res.finish
end
```

4. Status Code

We can directly access the Response object to change the status code like this:

response.status = 200

If it is not set, then the status code defaults to 200. Response provides a *redirect* method to redirect directly:

redirect(target, status=302)

Let' us now run a program to illustrate the redirect method. This program will redirect your browser to Google.

```ruby
require "rubygems"
require "rack"
require "thin"

rack_app = lambda do |env|
  request = Rack::Request.new(env)
  response = Rack::Response.new

  if request.path_info == '/redirect'
    response.redirect("http://google.com")
  else
    response.write "You did not get redirected"
  end
  response.finish
end

Rack::Handler::Thin.run rack_app, :Port => 3005

```

5. Response Header

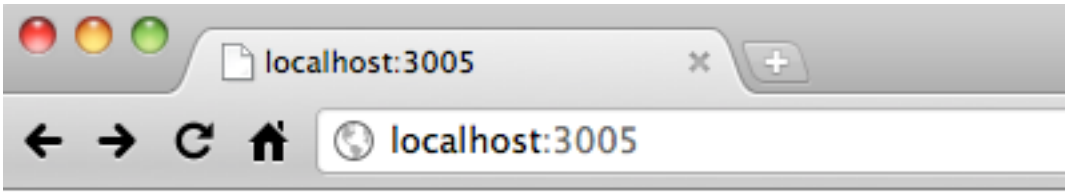
You can also set the response header directly on the response object. It is a hash, so you can set it like this:

response.headers['Content-Type'] = 'text/plain'

Let's modify the previous example to return plain text instead of html to the browser.

```ruby
require "rubygems"
require "rack"
require "thin"

rack_app = lambda do |env|
  request = Rack::Request.new(env)
  response = Rack::Response.new

  if request.path_info == '/redirect'
    response.redirect("http://google.com")
  else
    response.headers['Content-Type'] = "text/plain"
    response.write "<h1>You did not get redirected</h1>"
  end
  response.finish
end

Rack::Handler::Thin.run rack_app, :Port => 3005
```

This will display the output as plain text on the browser. When you don't set the Content-Type, it defaults to html and you get the following htmlized output instead of plain text output:



You did not get redirected

Now we are ready to see the basic structure of a Rack application.

```
class BasicRack
  def initialize(app)
    @app = app
  end

  def call(env)
    # Can modify request here (env)

    # Can call layer below like this:
    status, headers, response = @app.call(env)

    # Can modify response here

    [status, headers, response]
  end
end
```

Here is an example that modifies the response:

```
class Downcase
  def initialize(app)
```

```
    @app = app
  end
  def call(env)
    status, headers, response = @app.call(env)
    [status, headers, response.body.downcase]
  end
end
```

III. Middleware

1. Introduction

Middleware is a general term for any program that serves to "glue together" two separate programs. It is the software that connects two separate applications and passes data between them. Alternatively, software that allows more than one application to share information seamlessly.

Rack filters allow on the fly transformations of payload and header information in both the request into a resource and the response from a resource.

A Rack application can transform the content of HTTP requests, responses and header information. They can modify or adapt the requests for a resource and modify and adapt responses from a resource.

Rack filters can intercept :

- The accessing of a resource before a request to it is invoked.
- The processing of the request for a resource before it is invoked.
- The modification of request headers and data by wrapping the request in customized versions of the request object.
- The modification of response headers and response data by providing customized versions of the response object.
- The interception of an invocation of a resource after its call.

Here is some examples on how Rack filters could be used:

- Authentication
- Logging and auditing
- Image conversion
- Data compression
- Encryption
- Tokenizing
- Filters that trigger resource access events
- XSL/T filters that transform XML content
- MIME-type chain filters
- Caching
- Exception Handling

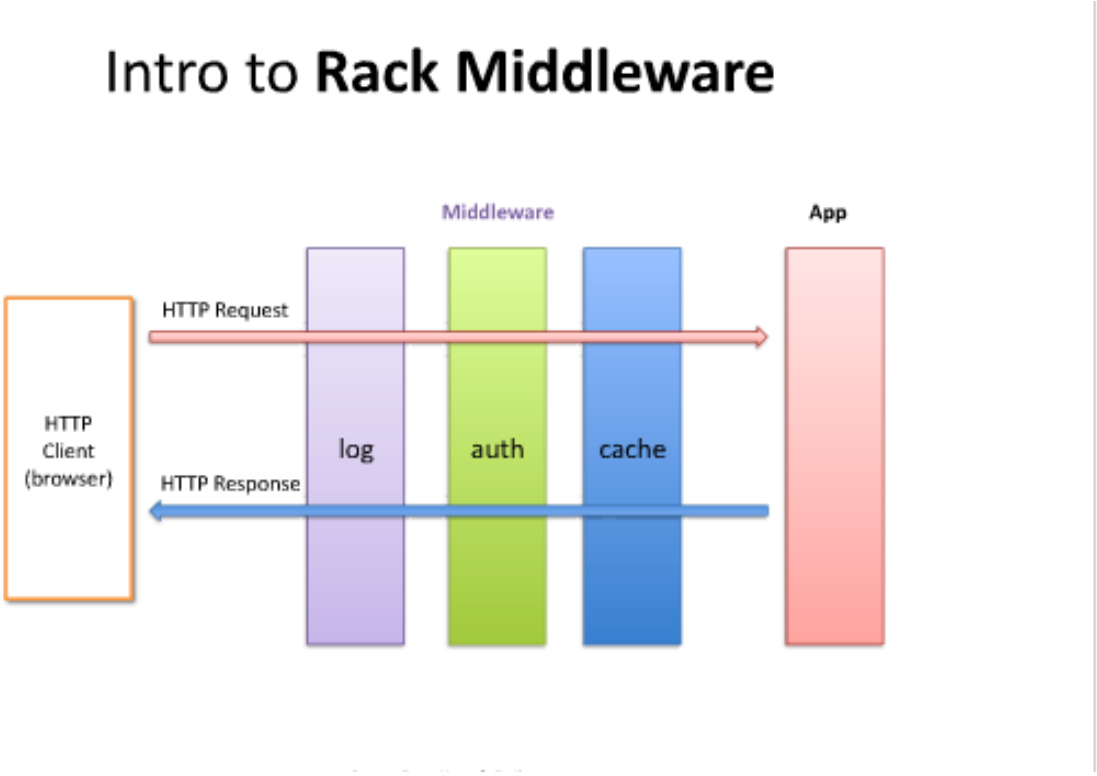
Examples of Rack Middleware that comes with Rack distribution:

Rack::Static - Handles fetching and returning static assets or caches

Rack::Session::Cookie - Handles your sessions via cookies

Rack::ShowExceptions - Catches any exceptions thrown in your app and returns a useful back-trace on the exception.

and many more.



In chapter 2, figure 3.4, the code outputs header and footer for the content generated by the Rack app. Here it is reproduced again for your convenience:

```ruby
require "rubygems"
require "rack"
require "thin"

rack_app = lambda do |env|
  request = Rack::Request.new(env)
  response = Rack::Response.new
  response.write "---------------Header-------------<br/>"

  if request.path_info == '/hello'
    response.write "Saying hi "
    client = request['client']
    response.write "for #{client}" if client
  else
    response.write "You need to provide the client information"
  end
  response.write "<br/>----------footer-----------------"
  response.finish
end

Rack::Handler::Thin.run rack_app, :Port => 3005
```

We are going to split this into two Rack apps that will co-operate to produce the same output as the above program. First step is to remove the footer and header generation out of the above program so that it now becomes:

```ruby
require 'rubygems'
require 'rack'
require 'thin'
require 'decorator'

rack_app = lambda do |env|
  request = Rack::Request.new(env)
  response = Rack::Response.new

  if request.path_info == '/hello'
    response.write("You said hello ")
    client = request['client']
    response.write("from #{client}") if client
  else
    response.write("You need to provide some client information")
  end
  response.finish
end

Rack::Handler::Thin.run Decorator.new(rack_app), :Port => 3005

```

Compare this code to the program in the previous page, you see that the last line that invokes the run is now using Decorator class. This class will generate the header and footer. Let's define the Decorator class which will take the original Rack app as an argument to its constructor. The class will look something like:

```ruby
class Decorator
  def initialize(app)
   ...
  end
  def call(env)
   ...
  end
end
```

Now look at the implementation of Decorator below. Notice how it wraps the response body from the original response body from the Rack app that was passed into its constructor.

```ruby
class Decorator
  def initialize(app)
    @next_in_chain = app
  end
  def call(env)
    status, headers, body = @next_in_chain.call(env)
    new_body = "------------Header---------------<br/>"
    body.each{ |s| new_body << s }
    new_body << "<br/>------------Footer------------"
    headers['Content-Length'] = new_body.bytesize.to_s
    [status, headers, [new_body]]
  end
end

require "rubygems"
require "rack"
require "thin"

rack_app = lambda do |env|
  request = Rack::Request.new(env)
  response = Rack::Response.new

  if request.path_info == '/hello'
    response.write("You said hello ")
    client = request['client']
    response.write("from #{client}") if client
  else
    response.write "You did not provide client information"
  end
  response.finish

end

Rack::Handler::Thin.run Decorator.new(rack_app), :Port => 3005
```

The line number 34 now passes an instance of Decorator, another Rack app as the first argument. The constructor of Decorator takes a Rack app and it modifies the body of the response to print the header and the footer thereby wrapping the output of the Rack app that was passed to it in the constructor. This produces the same output as the example in chapter 2 which prints "You said hello" enclosed by the header and footer.

Decorator is a valid Rack application since it meets the Rack spec. As a general rule, any middleware must be a valid Rack application.

When the Thin server is run it calls the *call* method of the Decorator. In line 6 you see that our original Rack app gets called by the Decorator. This *call* to original Rack app returns the status, headers and body which gets initialized in the local variables.

The status is an HTTP status. When parsed as integer (to_id), it must be greater than or equal to 100.

The header must respond to *each* and yield values of key and value. The header keys must be Strings. The header must not contain a *Status* key, contain keys with : or newlines in their name, contain key names that end in hyphen or underscore, but only contain keys that consist of letters, digits, underscore or hyphen and start with a letter. The values of the header must be Strings, consisting of lines (for multiple header values, e.g. multiple *Set-Cookie* values) separated by "\n". The lines must not contain characters below 037.

There must be a *Content-Type*, except when the *Status* is 1xx, 204 or 304, in which case there must be none given.

There must not be a *Content-Length* header when the *Status* is 1xx, 204 or 304.

The Body must respond to *each* and must only yield String values. The Body itself should not be an instance of String, as this will break in Ruby 1.9. If the Body responds to *close*, it will be called after iteration. If the Body responds to *to_path*, it must return a String identifying the location of a file whose contents are identical to that produced by calling *each*; this may be used by the server as an alternative, possibly more efficient way to transport the response. The Body commonly is an Array of Strings, teh application instance itself, or File-like object.

Coming back to our example, we only know that the original body can respond to *each,* which returns a String. Therefore we call *each* on the body and append the old string values from the original Rack app body to the header and append the footer to the end to create a new_body. This happens in 7, 8 and 9 in the code shown above.

Let's insert puts body.inspect before the line 8 and run the app. In the console we get the output of inspecting body:

#<Rack::Response:0x00000100bae1d0 @status=200, @header={"Content-Type"=>"text/html", "Content-Length"=>"38"}, @writer=#<Proc:0x00000100badf00@/usr/local/lib/ruby/gems/1.9.1/gems/rack-1.2.1/lib/rack/response.rb:27 (lambda)>, @block=nil, @length=38, @body=["You did not provide client information"]>

Well, this is a surprise. You might have expected the type to be String but it is an instance of Rack::Request. If you refer the documentation for Rack::Response you will see that it responds to each method and returns String thereby satisfying the Rack specification that the Body must respond to *each* and only yield String values. It also says that the Body commonly is an Array of String, the application instance itself or a File-like object.

Let's take a closer look at line number 30:

response.finish is implemented as shown below:

```
1   def finish(&block)
2     @block = block
3
4     if [204, 304].include?(status.to_i)
5       header.delete "Content-Type"
6       [status.to_i, header, []]
7     else
8       [status.to_i, header, self]
9     end
10  end
```

As you can see the body value is an instance of Response object itself. Let's look at the implementation of *each* for Response:

```
1   def each(&callback)
2     @body.each(&callback)
3     @writer = callback
4     @block.call(self) if @block
5   end
```

The *each* method delegates the work to the body variable of the Request object.

In line number 10 we make sure that we set the value for *Content-Length.*

Middleware is framework-independent components that process requests independently or in concert with other middleware.

Between the server and the framework, Rack can be customized to your applications needs using middleware, for example:

Rack::URLMap to route to multiple applications inside the same process.

Rack::CommonLogger for creating Apache-style log files.

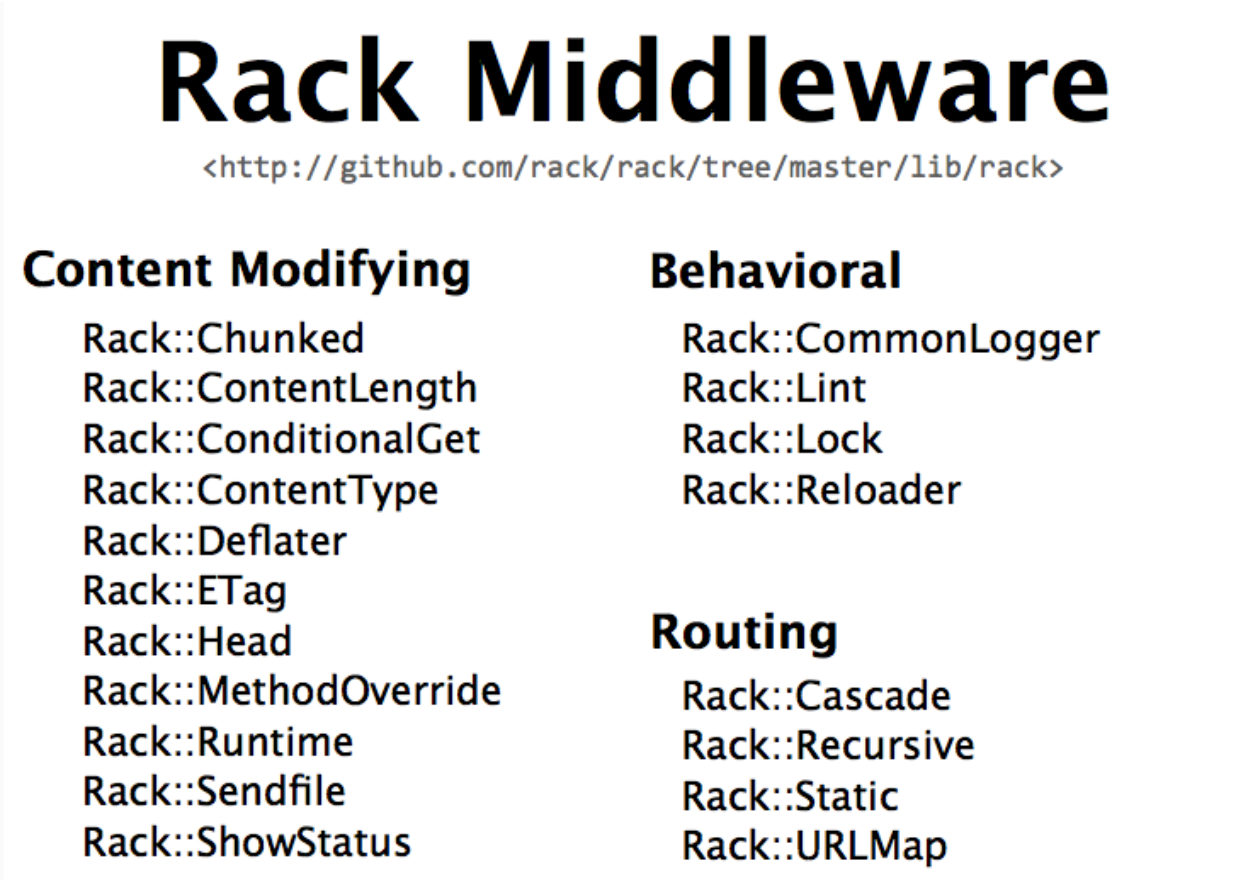
Rack::ShowException, for catching unhandled exceptions and presenting them in a nice and helpful way with clickable backtrace.

Rack::File, for serving static files.

Rack::Cascade - Try a request with several apps, and return the first non-404 result

…many others!

All these components use the same interface, which is described in detail in the Rack specification. These optional components can be used in any way you wish. Here is the list of Rack built-in middleware:



2. Why Middleware

Separation of Concerns is the process of separating a program into distinct features that overlap in functionality as little as possible. A concern is any piece of interest or focus in a program. It is achieved through modularity of programming and encapsulation with the help of information hiding. Layered designs in information systems are also based on separation of concerns(presentation, business logic, data access, database etc).[5]

If we are attempting to separate concern A from concern B, then we are seeking a design that provides that variations in A do not induce or require a corresponding change in B (and the converse). If we manage to successfully separate those concerns then we say that they are decoupled.

---

5. Separation of Concerns Wikipedia Entry

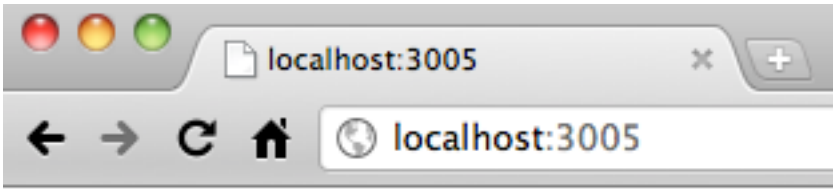
Most programs require some form of security and logging. Security and logging are often secondary concerns, whereas the primary concern is often on accomplishing business goals. The core concern is the business logic. Security and logging are cross-cutting concerns.

The middleware can achieve separation of business logic from other comon logic. These common logic can be used in a different business context. For instance the same security and logging functionality can be applied to a financial domain as well as medical domain. This enables the core domain model to evolve independent of other models such as security and logging resulting in reuse and ease of maintenance.

Decorator we wrote earlier can be used with any Rack application.

```
1   class Decorator
2     def initialize(app)
3       @next_in_chain = app
4     end
5     def call(env)
6       status, headers, body = @next_in_chain.call(env)
7       new_body = "-------------Header---------------<br/>"
8       puts body.inspect
9       body.each{ |s| new_body << s }
10      new_body << "<br/>------------Footer------------"
11      headers['Content-Length'] = new_body.bytesize.to_s
12      [status, headers, [new_body]]
13    end
14  end
15
16  rack_app = lambda do |env|
17    [200, {'Content-Type' => 'text/html'}, ["whatever rack app"]]
18  end
19
20  require "rubygems"
21  require "rack"
22  require "thin"
23
24  Rack::Handler::Thin.run Decorator.new(rack_app), :Port => 3005
```

The output is shown below:



----------Header----------
whatever rack app
----------Footer----------

This is a simple example. Suppose we had implemented a middleware for user authentication, then the middleware can be used with any Rack application without any modification. Develop middleware that is focused on doing one thing really well. The advantages are :

a) Each middleware can be developed independently and used in a plug-n-play manner.

b) We can mix and match middleware to combine them in several ways to meet the needs of different applications and can be dynamically configured based on application needs.

3. Middleware Assembly

(a) How to Assemble

We often need to stack multiple Rack applications to meet the application requirements. If we have a Rack app and two middlewares called Middleware1 and Middleware2 then we can use the two middlewares like this:

Rack::Handler::Thin.run Middleware1.new(Middleware2.new(rack_app))

When we need to pass options in the second argument the syntax would be as follows:

```
1   Rack::Handler::Thin.run(
2     Middleware1.new(
3        Middleware2.new(rack_app, options2),
4        options1)
```

If we need to use more than two middleware this code will become even more verbose. If you want to modify the order of the middleware, it becomes complex and error-prone.

We can define a class that has methods to use and run a given middleware.

```ruby
class Builder
  def use
    ...
  end
  def run
    ...
  end
end
```

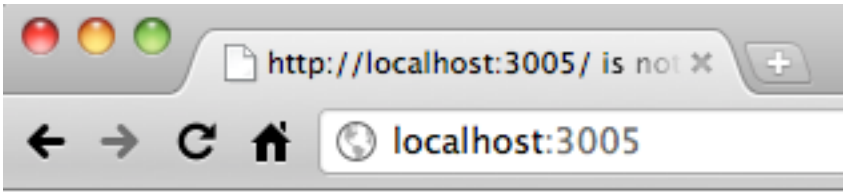
Now we need to be able to specify the order of the middleware that we would like to chain and run a given Rack app in the end. We want to specify this in code, with something like this:

```ruby
Builder.new do
  use Middleware1
  use Middleware2
  run MyRackApplication
end
```

Rack::Builder implements a small DSL to iteratively construct Rack applications. Rack implementation comes with lot of middleware. One of the middleware, Rack::ContentLength can automatically set the response header "Content-Length". So let's delete the line 11 which sets the Content-Length : headers['Content-Length'] = new_body.bytesize.to_s

from Decorator class.

```ruby
class Decorator
  def initialize(app)
    @next_in_chain = app
  end
  def call(env)
    status, headers, body = @next_in_chain.call(env)
    new_body = "------------Header--------------<br/>"
    body.each{ |s| new_body << s }
    new_body << "<br/>------------Footer------------"
    [status, headers, [new_body]]
  end
end

require "rubygems"
require "rack"
require "thin"

builder = Rack::Builder.new do
  use Rack::ContentLength
  use Decorator
  run lambda do|env|
    [200, {'Content-Type' => 'text/html'}, ["Hello from Builder"]]
  end
end

Rack::Handler::Thin.run Decorator.new(builder), :Port => 3005
```



This cuts-off the body. If you add text to the response body without updating the Content-Length, the HTML may get truncated (because the browser is only displaying the original number of characters, before you added more text to the response body). Rack::ContentLength only works under certain conditions. So we need to include the line on setting the content-length before the

line 10 headers['Content-Length'] = new_body.bytesize.to_s as shown below to fix this problem.[6]

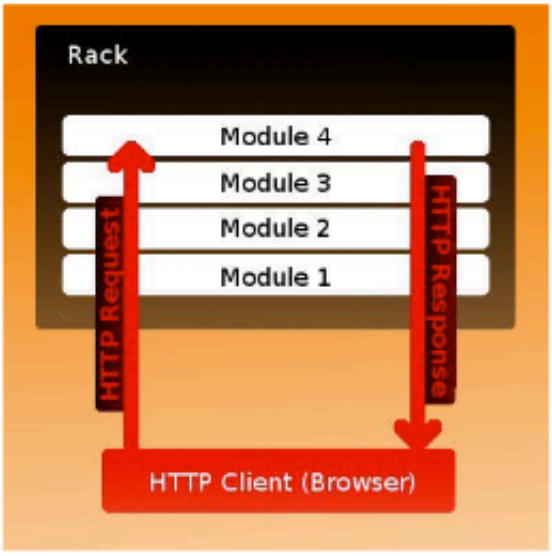
```
1   class Decorator
2     def initialize(app)
3       @next_in_chain = app
4     end
5     def call(env)
6       status, headers, body = @next_in_chain.call(env)
7       new_body = "-------------Header---------------<br/>"
8       body.each{ |s| new_body << s }
9       new_body << "<br/>------------Footer------------"
10      headers['Content-Length'] = new_body.bytesize.to_s
11      [status, headers, [new_body]]
12    end
13  end
```

Let's write a simple example that works properly to illustrate the use of Builder.

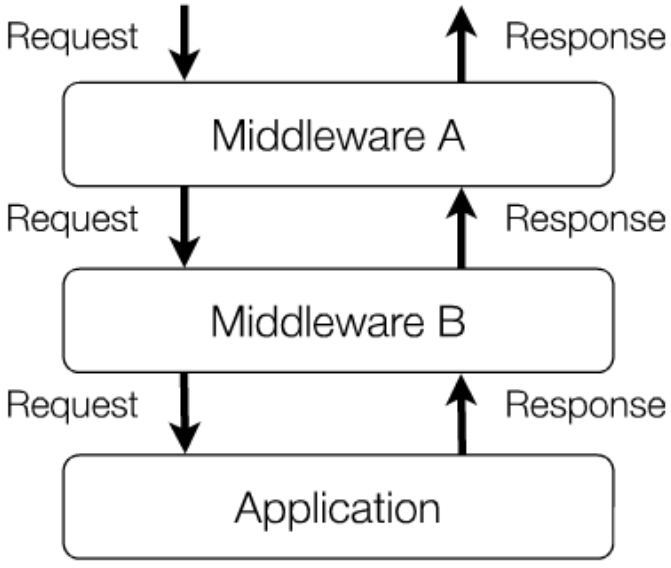
```
1   require "rubygems"
2   require "rack"
3   require "thin"
4
5   inspector = lambda do |env|
6     [200, {'Content-Type' => 'text/html'}, env.inspect]
7   end
8
9   builder = Rack::Builder.new
10  builder.run inspector
11
12  Rack::Handler::Thin.run builder, :Port => 3005
```

6. Based on Remi's feedback on Railscasts episode 151 on Rack Middleware

Stack



Middleware



Stack basically is the equivalent of pipes in Unix. Middleware is utilities that combine, compose, aggregate or modify Rack applications. Middleware is stackable, it's just a Rack application itself.

Since Rack applications are just Ruby objects they are easy to write

Rack builtin middleware:

Rack::Static & Rack::File

Rack::CommonLogger

Rack::Reloader

Rack::ShowExceptions

Rack Utilities

Rackup is a useful tool for running Rack applications. Builder is a DSL to configure middleware and build up applications easily. The rackup file is a configuration DSL for a Rack app. It is Server-independent. It allows stacking of middleware and provides simple route mapping. Rack uses rackup file to load and configure rack applications and middleware.

The config file is config.ru if none is specified. Handling of config files depends on whether it's .ru, or something else. It is important to define the application that rackup will run correctly, failure to do so will result in mysterious runtime errors!

Your config.ru file:

```
class RackupDemo
  def call(env)
    [200, {'Content-Type' => 'text/html'}, ["Rackup Demo"]]
  end
end

run RackupDemo.new
```

Rack loads it like this:

```
config_file = File.read(config)
rack_application = eval("Rack::Builder.new {#{config_file}}")
server.run rack_application, options
```

.ru:

The config file is treated as if it is the body of app = Rack::Builder.new { ... config ... }.to_app

Also, the first line starting with #\ is treated as if it was options, allowing rackup arguments to be specified in the config file. For example:

```
#\ -w -p 8765
use Rack::Reloader, 0
use Rack::ContentLength

app = proc do |env|
  [ 200, {'Content-Type' => 'text/plain'}, "a" ]
end

run app
```

Would run with Ruby warnings enabled, and request port 8765 (which will be ignored unless the server supports the :Port option).

.rb, etc:

The config file is required. It must assign the app to a global constant so rackup can find it. The name of the constant should be config file's base name, stripped of a trailing .rb (if present), and captitalized. The following config files all look for Config: ~/bin/config, config.rb, /usr/bin/config, example/config.rb.

This will work if the file name is octet.rb:

```
octet = Rack::Builder.new do
  use Rack::Reloader, 0
  use Rack::ContentLength
  app = proc do |env|
    [ 200, {'Content-Type' => 'text/plain'}, "b" ]
  end
  run app
end.to_app
```

Auto-Selection of a Server

The specified server (from Handler.get) is used, or the first of these to match is selected:

PHP_FCGI_CHILDREN is in the process environment, use FastCGI

REQUEST_METHOD is in the process environment, use CGI

Mongrel is installed, use it

Otherwise, use Webrick

Overriding basic configuration:

$ myrackapp.ru -s Thin -p 8080

Automatic Middleware

rackup will automatically use some middleware, depending on the environment you select, the -E switch, with development being the default:

development: CommonLogger, ShowExceptions, Lint

deployment: CommonLogger

none: none

CommonLogger isn't used with the CGI server, because it writes to stderr, which doesn't interact so well with CGI.

Debugging Libraries

Rack::Lint and Rack::ShowExceptions

racksh

Rack::Bug (mostly for Rails apps, can be used with any Rack app, http://github.com/brynary/rack-bug)

1) script/plugin install git://github.com/brynary/rack-bug.git

2) # config/environments/development.rb

config.middleware.use "Rack::Bug"

3) # add bookmarklet to browser

open http://RAILS_APP/__rack_bug__/bookmarklet.html

Rack::Debug (not 1.9 ready yet)

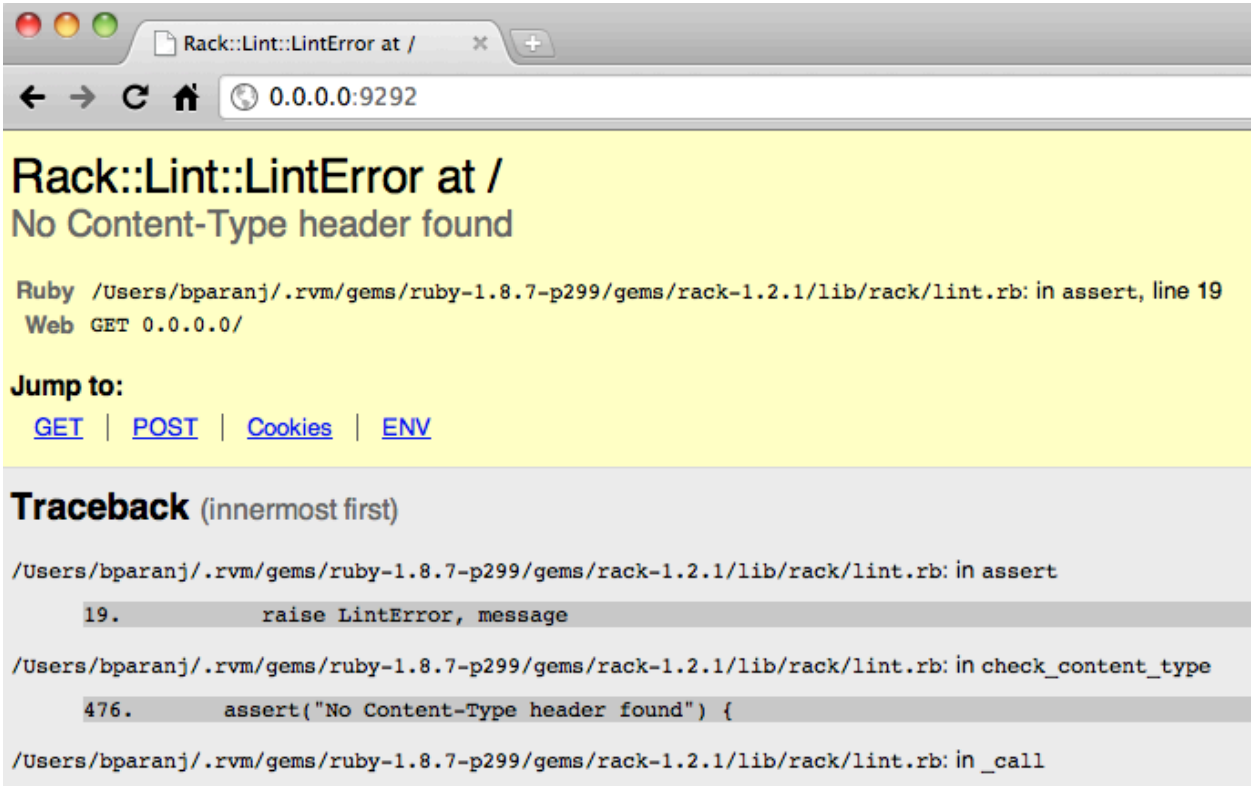
Most app servers will automatically use Rack::ShowExceptions and Rack::Lint when running in the development environment, which is the default.

```
1  use Rack::ShowExceptions
2  use Rack::Lint
3
4  run lambda {|env| [200, {}, ["Some string"]]}
```

When the code in ch3/5 is run:

$ rackup exceptions_demo.ru

You get the following output:



As you can see even if you don't explicitly use Rack::Lint by default it is used in the development environment. Replace the run line with :

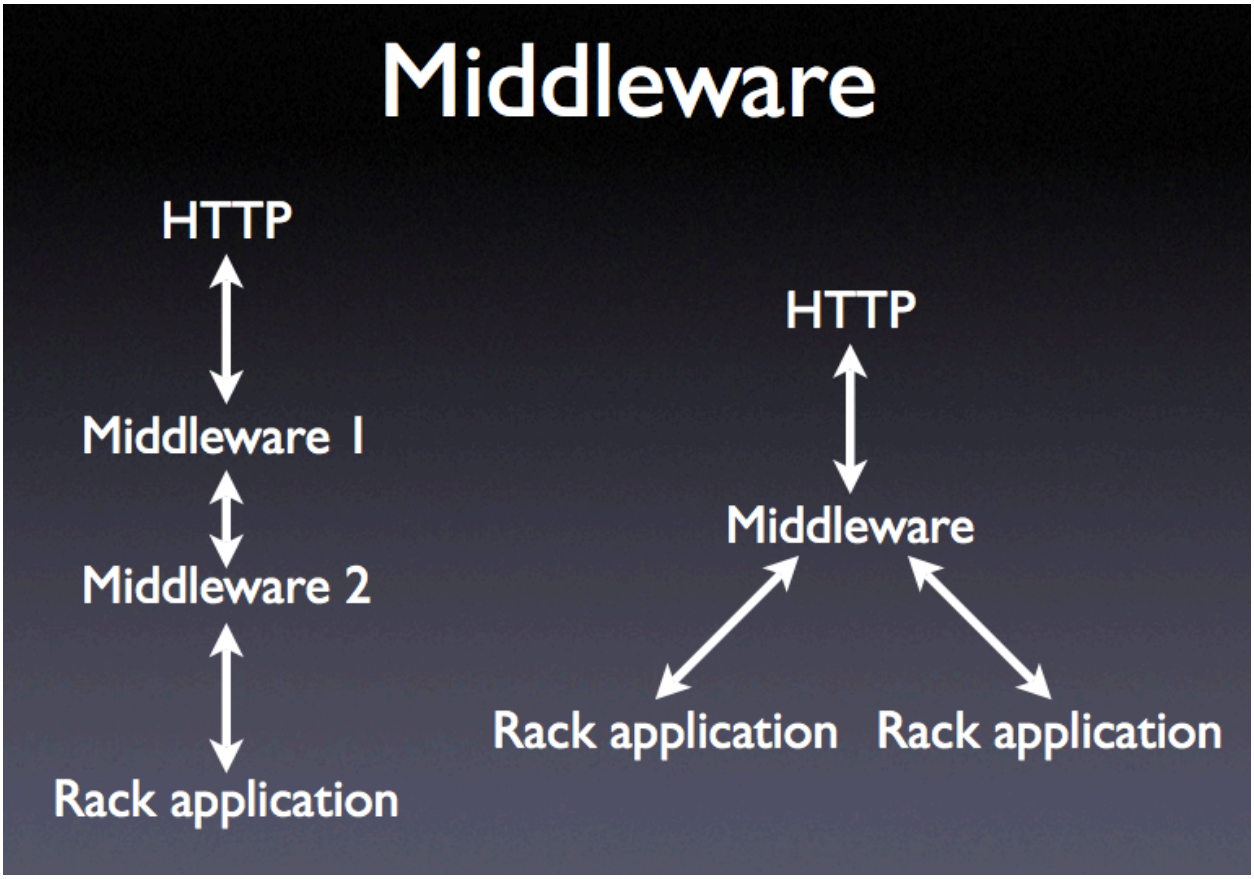
run lambda {|env| [200, {"Content-Type" => "text/html"}, ["some string"]]}

to fix the error. The output shows "some string" in the browser.

Testing Libraries

Rack::MockRequest and Rack::MockResponse

rack-test



(b) Designing Builder

use adds a middleware to the stack, *run* dispatches to an application. The *initialize* method of builder should have the signature *initialize(&block)* in order to enable use and run methods as DSL language verbs, *initialize* should instance_eval the block in the context of the current Builder instance. The signature of use method should be *use(middlewareclass, options)*, it should save the middleware so that it can create an instance of the middleware and track its order in the stack. The signature of the *run* method should be *run(rack_app)*, it must stash away the Rack app and when call is invoked it must invoke *to_app* on the Rack app and run according to the saved information to create the final application.

There are two ways to implement these methods

Traditional Methods

Use an array to store the middlewares and the order of the middleware and to_app method to create the middleware.

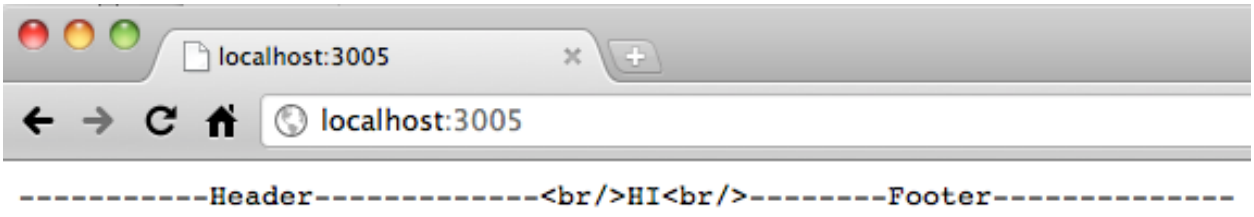
```ruby
class Builder
  def initialize(&block)
    @middleware_list = []
    self.instance_eval(&block)
  end
  def use(middleware)
    @middleware_list << middleware
  end
  def run(app)
    @app = app
  end
  def to_app
    app = @app
    @middleware_list.reverse.each do |middleware|
      app = middleware.new(app)
    end
    app
  end
end
```

You can use inject method to simplify the to_app method:

```ruby
def to_app
  @middleware_list.reverse.inject(@app) do |app, middleware|
    middleware.new(app)
  end
end
```

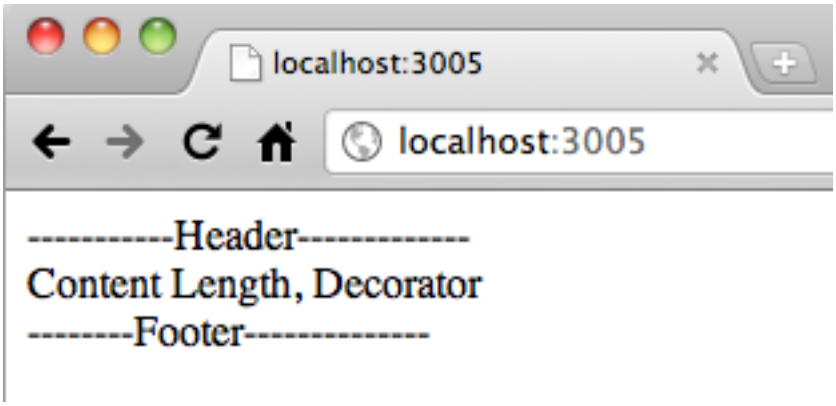
```ruby
require 'rubygems'
require 'rack'
require 'builder'
require 'decorator'
require 'thin'

app = Builder.new do
  use Decorator
  run lambda{|env| [200, {}, ["Hi"]]}
end.to_app

Rack::Hanlder::Thin.run app, :Port => 3000
```

We get the following output when we run the code in ch3/3/ex1.rb shown above:



Now let's run the code shown below (ch3/3/ex2.rb) :

```ruby
require 'rubygems'
require 'rack'
require 'builder'
require 'decorator'
require 'thin'

app = Builder.new do
  use Rack::ContentLength
  use Decorator
  run lambda{|env|[200, {'Content-Type' => 'text/html'}, ["Content Length and then Decorator"]]}
end.to_app

Rack::Handler::Thin.run app, :Port => 3000
```



When we change the lines :

use Rack::ContentLength

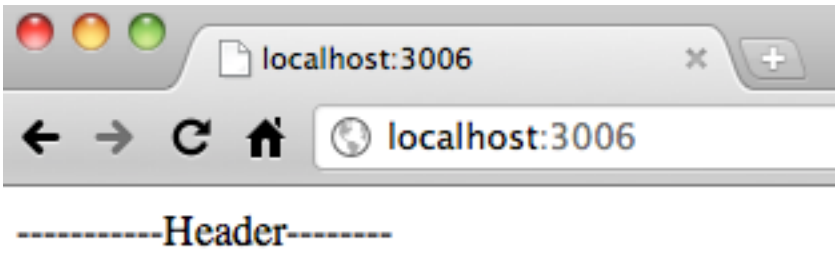
use Decorator

to:

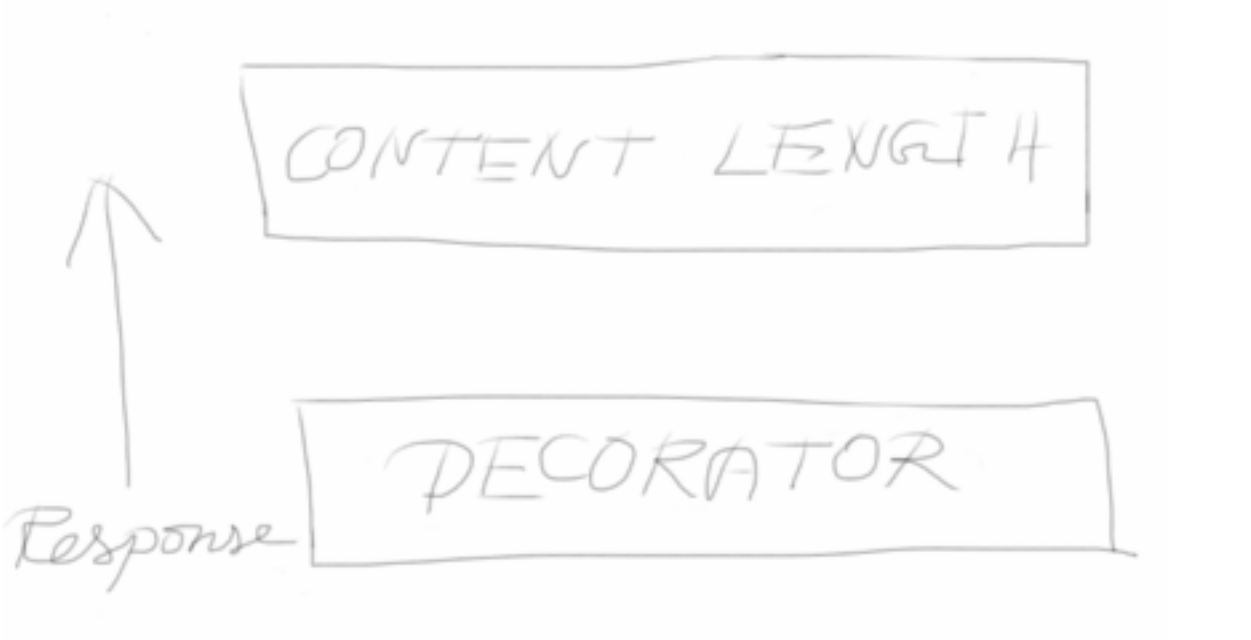
use Decorator

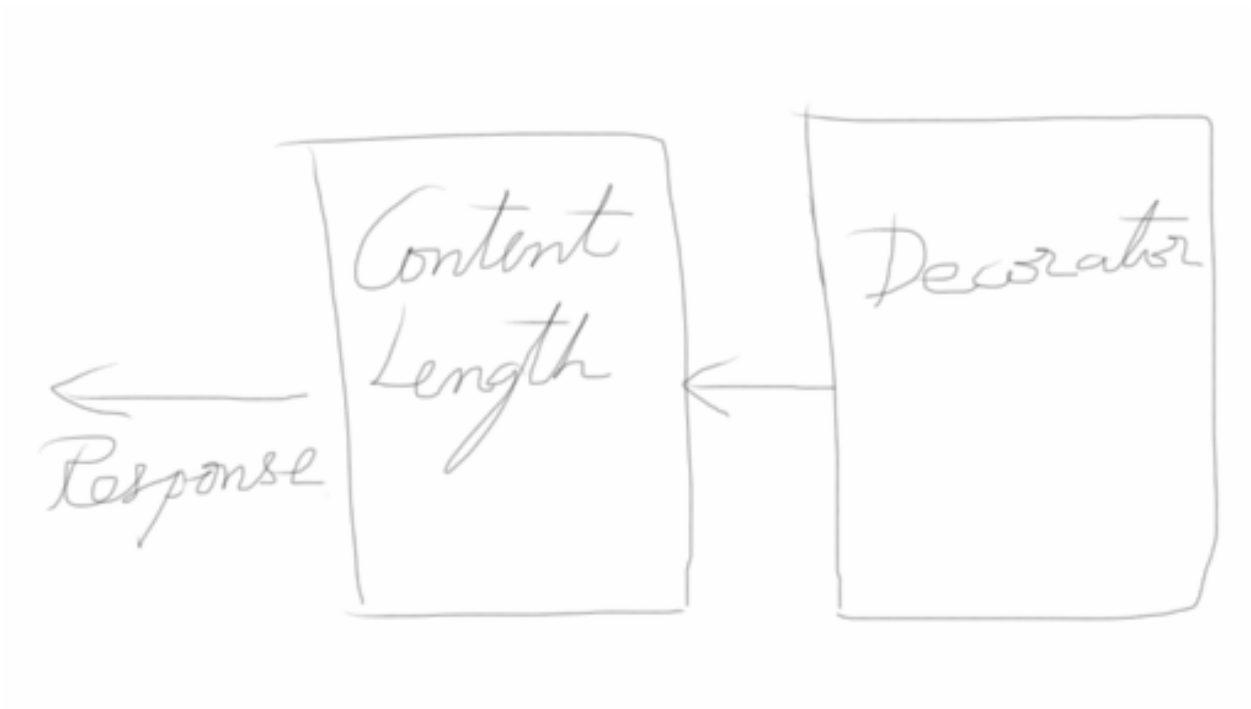
use Rack::ContentLength

in the Builder, we get the following output:



You see the output is truncated. This is because the Rack::ContentLength sets the content-length before the Decorator adds more content to the body of the response.





Later we will see that *use* method maintains a list of array and adds the middleware that gets passed in as the parameter to it. When to_app method is called it processes the array in reverse

order. We will revisit this later, for now just remember that order does matter and it can have an impact on the generated response in some cases.

Delayed Execution Method

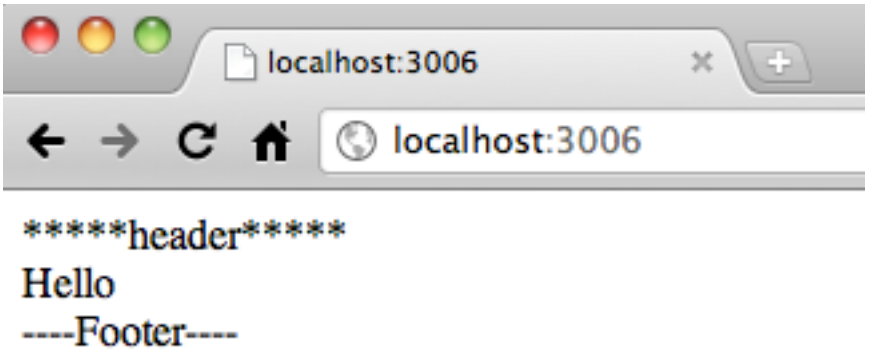
The Traditional approach implementation has its own limitations. For example, if we need to pass options and block as parameters to the use method but it is not necessary to create the middleware inside the use method. We need to create an instance of middleware in the to_app method. For situations where we need to execute code later, Ruby gives us lambda.

```
1   class Builder
2     def initialize(&block)
3       @middleware_list = []
4       self.instance_eval(&block)
5     end
6     def use(middleware_class, *options, &block)
7       @middleware_list << lambda{|app| middleware_class.new(app, *options, &block)}
8     end
9     def run(app)
10      @app = app
11    end
12    def to_app
13      @middleware_list.reverse.inject(@app){ |app, middleware| middleware.call(app) }
14    end
15  end
```

In line 7 the *use* method now stores proc objects in the middleware list array by using lambda, and the creation of middleware is delayed till to_app method is called, only at this time the app variable is passed to create a new middleware instance. Lets modify the previous program to take parameters for our middleware.

```ruby
class Builder
  def initialize(&block)
    @middleware_list = []
    self.instance_eval(&block)
  end
  def use(middleware_class, *options, &block)
    @middleware_list << lambda{|app| middleware_class.new(app, *options, &block)}
  end
  def run(app)
    @app = app
  end
  def to_app
    @middleware_list.reverse.inject(@app){ |app, middleware| middleware.call(app) }
  end
end

class Decorator
  def initialize(app, *options, &block)
    @app = app
    @options = (options[0] || {})
  end
  def call(env)
    status, headers, body = @app.call(env)
    new_body = ""
    new_body << (@options[:header] || "----Header----<br/>")
    body.each {|str| new_body << str}
    new_body << (@options[:footer] || "<br/>----Footer----")
    [status, headers, [new_body]]
  end
end

require 'rubygems'
require 'rack'
require 'thin'
require 'rack/content_length'

app = Builder.new do
  use Rack::ContentLength
  use Decorator, :header => "*****header*****<br/>"
  run lambda {|env| [200, {'Content-Type' => 'text/html'}, ["Hello"]]}
end.to_app

Rack::Handler::Thin.run app, :Port => 3006
```

From the program output you can see we now have a custom header that was passed in as the options to the constructor of Builder.



Here is the actual implementation of Builder in Rack gem:

```ruby
class Builder
  def self.parse_file(config, opts = Server::Options.new)
    options = {}
    if config =~ /\.ru$/
      cfgfile = ::File.read(config)
      if cfgfile[/^#\\(.*)/] && opts
        options = opts.parse! $1.split(/\s+/)
      end
      cfgfile.sub!(/^__END__\n.*/, '')
      app = eval "Rack::Builder.new {( " + cfgfile + "\n )}.to_app",
        TOPLEVEL_BINDING, config
    else
      require config
      app = Object.const_get(::File.basename(config, '.rb').capitalize)
    end
    return app, options
  end

  def initialize(&block)
    @ins = []
    instance_eval(&block) if block_given?
  end

  def self.app(&block)
    self.new(&block).to_app
  end

  def use(middleware, *args, &block)
    @ins << lambda { |app| middleware.new(app, *args, &block) }
  end

  def run(app)
    @ins << app #lambda { |nothing| app }
  end

  def map(path, &block)
    if @ins.last.kind_of? Hash
      @ins.last[path] = self.class.new(&block).to_app
    else
      @ins << {}
      map(path, &block)
    end
  end

  def to_app
    @ins[-1] = Rack::URLMap.new(@ins.last) if Hash === @ins.last
    inner_app = @ins.last
    @ins[0...-1].reverse.inject(inner_app) { |a, e| e.call(a) }
  end

  def call(env)
    to_app.call(env)
  end
end
```

(c) Middleware Ordering

Your application is called, and then middleware is invoked in the order that you specify. These middlewares call each other, acting as a set of 'filters' for the response, so it is important to note that the ordering to which you 'use' them can be important, and have an effect on the results.

The following example illustrates that the body contents can be altered by several middleware's:

```
module Rack
  class Upcase
    def initialize app
      @app = app
    end

    def call env
      puts 'upcase'
      p @app
      puts
      status, headers, body = @app.call env
      [status, headers, [body.first.upcase]]
    end
  end
end

module Rack
  class Reverse
    def initialize app
      @app = app
    end

    def call env
      puts 'reverse'
      p @app
      puts
```

```
      status, headers, body = @app.call env
      [status, headers, [body.first.reverse]]
    end
  end
end

use Rack::Upcase
use Rack::Reverse
use Rack::ContentLength

app = lambda { |env| [200, { 'Content-Type' => 'text/html' }, 'Hello World'] }
run app
```

As you can see in the terminal output, our Upcase object's application is actually the Reverse middleware, and Reverse's application is the next one in line which happens to be ContentLength.

```
upcase
#<Rack::Reverse:0x5574e0@app=#<Rack::ContentLength:0x557620
@app=#<Proc:0x00561210@rack.ru:38>>>

reverse
#<Rack::ContentLength:0x557620 @app=#<Proc:0x00561210@rack.ru:38>>
```

Realworld Example of JSON Middleware

Below is a very simple realworld example which translates the body results to json when the response Content-Type header is that of json (application/json). This sort of middleware can really clean up your app, help others, and well really this is where this sort of functionality belongs!

```
require 'json'

module Rack
  class JSON
    def initialize app
```

```
    @app = app
  end

  def call env
    @status, @headers, @body = @app.call env
    @body = ::JSON.generate @body if json_response?
    [@status, @headers, @body]
  end

  def json_response?
    @headers['Content-Type'] == 'application/json'
  end
 end
end
```

Middleware can be utilized via the *use* method which, in turn autoload's a Ruby file containing the middleware. Autoloading of middleware is simply a Rack convention, but helps to keep things clean. Configured with 'use' in rackup file. Rack applications can be stacked together to form a larger app. Middleware respond to *call*, accept an environment hash and return a response array.

```
use Rack::ContentLength
use Rack::JSON

app = lambda { |env| [200, { 'Content-Type' => 'application/json' }, { :some => 'json', :stuff => ['here'] } ] }
run app
```

IV. Simple Web Framework

Rack comes with builtin utilities that is useful for web app development. It provides:

- Access to Request and Response
- Routing that provides URL mapping
- Handling cookies
- Access to the session
- Logging

1. Rack::Builder

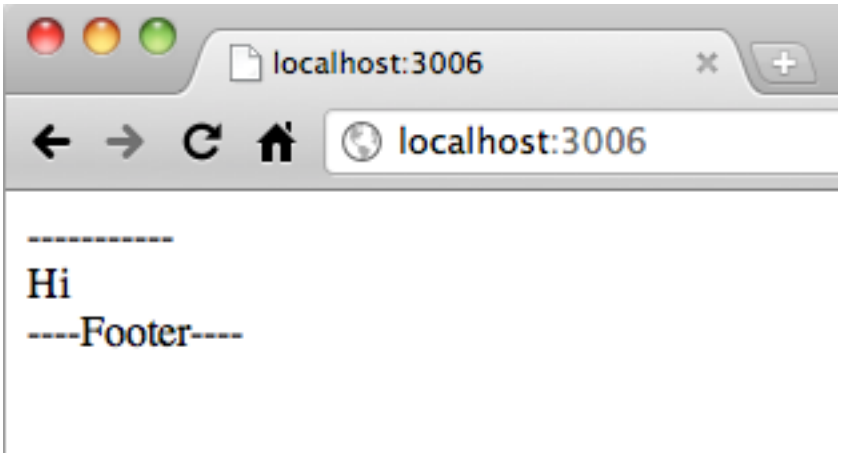
Previously we developed our own Builder to construct a stack of middleware. Rack comes with its own Rack::Builder which besides providing us the *use* and *run* methods also uses Rack::URLMap to handle the routing.

(a) Using Rack::Builder

Instead of using our own builder we are going to modify the previous example to use the Rack builtin Rack::Builder as shown below: ($: trick was required for Ruby 1.9.2 version)

```ruby
#!/usr/bin/env ruby

# Put the following in a startup file so that require 'decorator' will work.
$: << File.expand_path(File.dirname(__FILE__))
# Above line avoids doing this
# require File.expand_path(File.dirname(__FILE__))

require 'rubygems'
require 'rack'
require 'decorator'
require 'thin'
require 'rack/handler'

app = Rack::Builder.new do
  use Rack::ContentLength
  use Decorator, :header => "-----------<br/>"
  run lambda {|env| [200, {'Content-Type' => 'text/html'}, ["Hi"]]}
end.to_app

Rack::Handler::Thin.run app, :Port => 3006
```

which gives us the following output:



So we are able to customize the header by providing the a custom header in the use method.

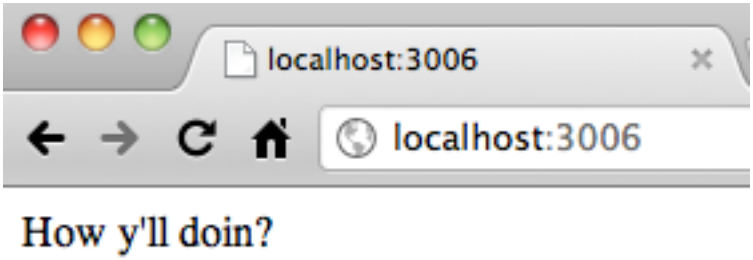
2. Routing

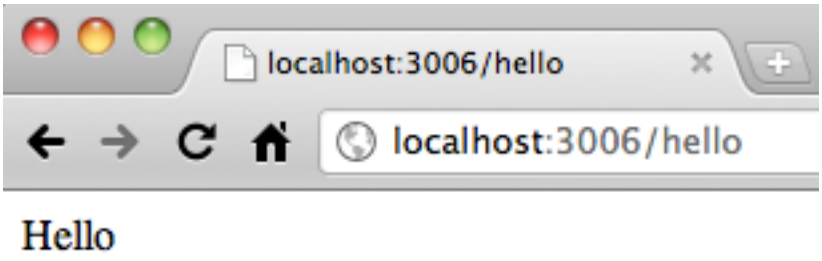
Web programs usually have different handlers to handle different URL. This mapping of URL to its corresponding handler is called routing. The simplest route is a path which maps to a code block.

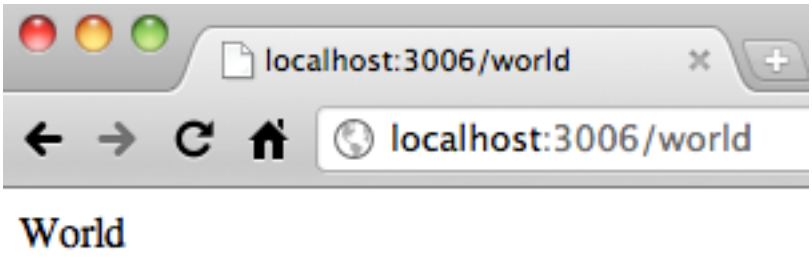
(a) Simple Route

```ruby
#!/usr/bin/env ruby

require 'rubygems'
require 'rack'
require 'thin'

app = Rack::Builder.new do
  map '/hello' do
    run lambda {|env| [200, {'Content-Type' => 'text/html'}, ["Hello "]]}
  end
  map '/world' do
    run lambda {|env| [200, {'Content-Type' => 'text/html'}, ["World "]]}
  end
  map '/' do
    run lambda {|env| [200, {'Content-Type' => 'text/html'}, ["How y'll doin?"]]}
  end
end.to_app

Rack::Handler::Thin.run app, :Port => 3006
```

Here we are using the map method of Rack::Builder class. Make this script executable by doing: chmod +x router_demo.rb and run the script.







These output illustrates how the URL gets mapped to the code block.

(b) Implementing Route#map

Using *use* and *run*

Rack::Builder implementation is essentially the same as our Builder implementation. The Rack::Builder implementation looks like:

```ruby
def initialize(&block)
  @ins = []
  instance_eval(&block) if block_given?
end

def self.app(&block)
  self.new(&block).to_app
end

def use(middleware, *args, &block)
  @ins << lambda { |app| middleware.new(app, *args, &block) }
end

def run(app)
  @ins << app #lambda { |nothing| app }
end

def to_app
  @ins[-1] = Rack::URLMap.new(@ins.last) if Hash === @ins.last
  inner_app = @ins.last
  @ins[0...-1].reverse.inject(inner_app) { |a, e| e.call(a) }
end
```

Our implementation of builder uses a separate @app instance variable to hold the original Rack app whereas Rack's builder implementation stores the app instance to the end of the array. Therefore the members of the array in turn contains all the middleware. to_app method above skips the last app stored while iterating using inject method and accomplishes the same functionality as our implementation of to_app method. At each step of the inject block variable ***a*** is set to the value returned by the block.

This basically results in the inner_app being passed in the first run to the call method of the last element of the @ins variable, the output of this is sent in as the input parameter to the call method that is invoked on the next element in the @ins list. So the call cascades through the elements in the @ins list beginning from the next to last element.

Here is our implementation reproduced here for comparison.

```
def initialize(&block)
  @middleware_list = []
  self.instance_eval(&block)
end
def use(middleware_class, *options, &block)
```

```
  @middleware_list << lambda{|app| middleware_class.new(app, *options, &block)}
end
def run(app)
  @app = app
end
def to_app
  @middleware_list.reverse.inject(@app){ |app, middleware| middleware.call(app) }
end
```

(c) Map Method

The difference in to_app method implementation is if the last member is a Hash then it will create an instance of Rack::URLMap and replace the last element in the array with this instance.

```
1       def map(path, &block)
2         if @ins.last.kind_of? Hash
3           @ins.last[path] = self.class.new(&block).to_app
4         else
5           @ins << {}
6           map(path, &block)
7         end
8       end
```

Then the map method creates an instance of Builder by passing a block if the element is a Hash and stores it as the value of the path string key. When @ins is not a member of Hash, it adds an empty hash to the end of the @ins. As to_app method is always called at the end of the Builder.new block. If there is a map method at the same level as run, that is, fhe following situation is not valid:

```
1   Rack::Builder.new {
2     use ...
3     use ...
4     run ...
5     map ... do
6     .....
7   end
```

The else condition calls the map method recursively. Since the last element in the array @ins is a Hash, so the statement:

@ins.last[path] = self.class.new(&block).to_app

creates a correspondence between the path string and the response object that results from calling all the middleware in the list.

Let's consider a example that uses Rack::Builder to create a Rack application:

```
1   #!/usr/bin/env ruby
2   require "rubygems"
3   require 'rack'
4   require 'thin'
5
6   app = Rack::Builder.new do
7     use Rack::ContentLength
8     map '/hello' do
9       run lambda {|env| [200, {"Content-Type" => "text/html"}, ["hello"]] } }
10    end
11  end.to_app
12
13  Rack::Handler::Thin.run app, :Port => 3006
```

Now @ins array will consist of two elements: one to create the middleware Rack::ContentLength object and the second is the Hash which contains the path "/hello" as the key and the value is a Rack application that can be passed into run method. So the value is this:

lambda {|env| [200, {Content-Type => text/html}, [hello]] }

Let's see another example to illustrate the *map* method:

```ruby
#!/usr/bin/env ruby
require 'rubygems'
require 'rack'
require 'thin'

app = Rack::Builder.new do
  use Rack::ContentLength
  map '/hello' do
    run lambda {|env| [200, {'Content-Type' => 'text/html'}, ['hi']]}
  end
  map '/world' do
    run lambda {|env| [200, {'Content-Type' => 'text/html'}, ['world']]}
  end
end.to_app

Rack::Handler::Thin.run app, :Port => 3006
```

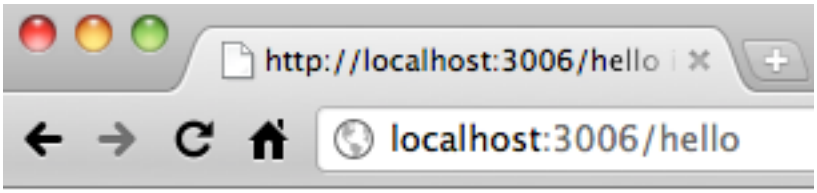
Now @ins has two elements in the array: the first middleware is still Rack::ConentLength the last one is Hash, with two elements in it. There are:
'hello' => lambda {|env| [200, {Content-Type => text/html}, [hi]] }

'world' => lambda {|env| [200, {Content-Type => text/html}, [world]] }

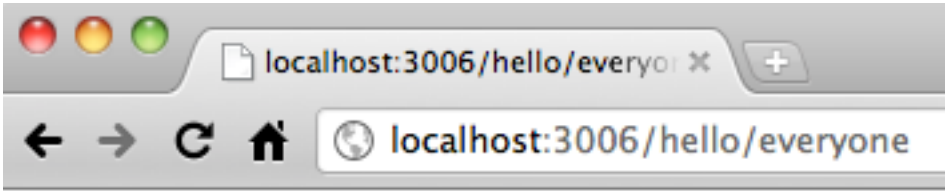
If the last member is a Hash, the member will be used to create a new Rack::URLMap instance. Rack::URLMap saves the URL and the corresponding relationship between the Rack procedures, if the user entered in the url : http://localhost:3006/hello , it will call the first application.

The matching path '/hello' will become the environment inside the SCRIPT_NAME and the rest of the URL goes into PATH_INFO, if we modify the program as follows.

```ruby
#!/usr/bin/env ruby

require 'rubygems'
require 'rack'
require 'thin'

app = Rack::Builder.new do
  map '/hello' do
    run lambda { |env|
      [200, {'Content-Type' => 'text/html'},
        ["SCRIPT_NAME=#{env['SCRIPT_NAME']} ", "PATH_INFO=#{env['PATH_INFO']}"]]
    }
  end
  map '/world' do
    run lambda {|env| [200, {'Content-Type' => 'text/html'}, ["World "]]}
  end
end.to_app

Rack::Handler::Thin.run app, :Port => 3006
```



SCRIPT_NAME=/hello PATH_INFO=



SCRIPT_NAME=/hello PATH_INFO=/everyone

3. Nested Map

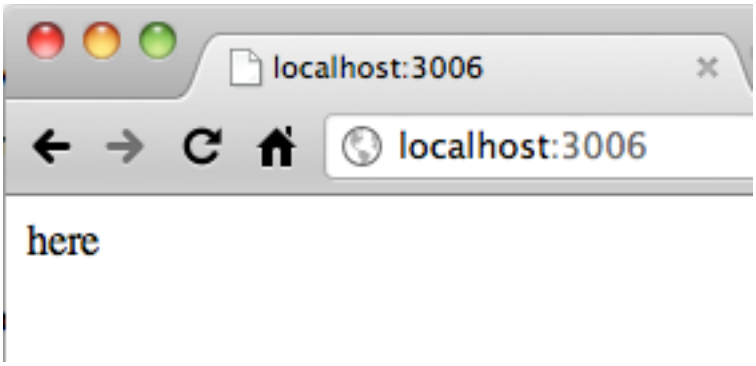
Recall that map method looks like this:

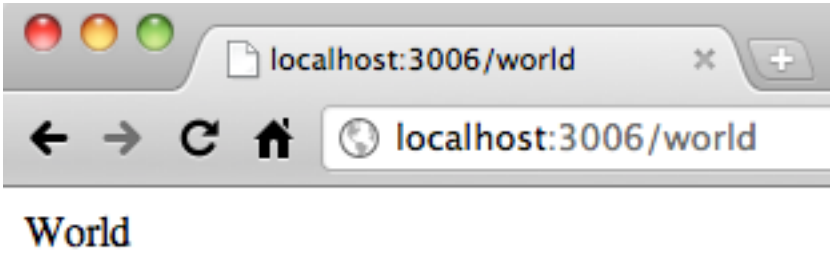
```
 def map(path, &block)
    if @ins.last.kind_of? Hash
      @ins.last[path] = self.class.new(&block).to_app
    else
      @ins << {}
      map(path, &block)
    end
  end
```

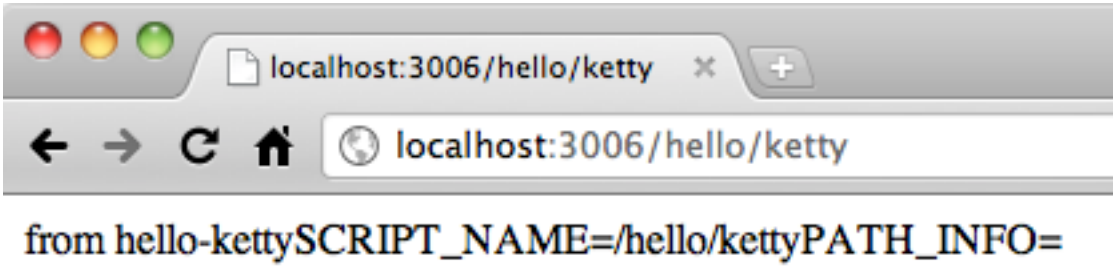
We recursively create a stored hash with Builder.new instances for the needed Rack applications. This means we can also have nested map.

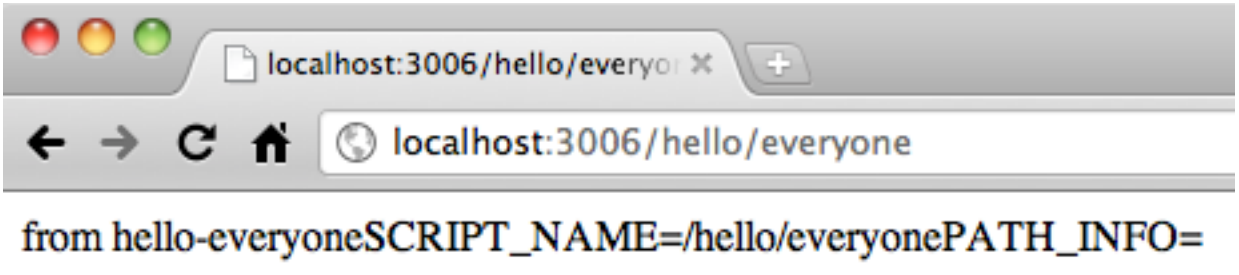
(a) Nested Map Demo

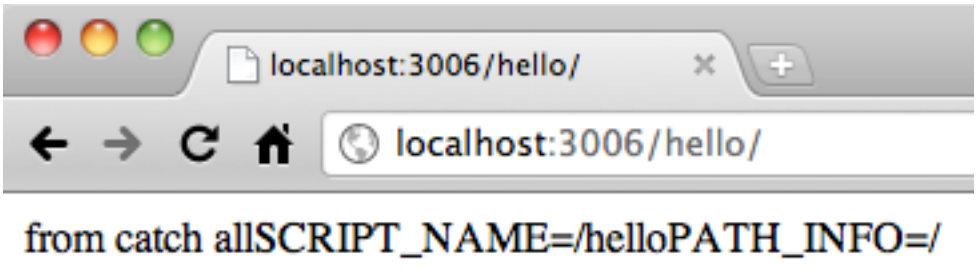
```ruby
#!/usr/bin/env ruby

require 'rubygems'
require 'rack'
require 'thin'

app = Rack::Builder.new do
  use Rack::ContentLength
  map '/hello' do
    use Rack::CommonLogger
    map '/ketty' do
      run lambda { |env|
        [200, {'Content-Type' => 'text/html'},
          ["from hello-ketty",
            "SCRIPT_NAME=#{env['SCRIPT_NAME']}",
            "PATH_INFO=#{env['PATH_INFO']}"]]
      }
    end
    map '/everyone' do
      run lambda {|env| [200, {"Content-Type" => 'text/html'},
        ["from hello-everyone",
          "SCRIPT_NAME=#{env['SCRIPT_NAME']}",
          "PATH_INFO=#{env['PATH_INFO']}"]]}
    end
    map '/' do
      run lambda {|env| [200, {'Content-Type' => 'text/html'},
        ['from catch all',
          "SCRIPT_NAME=#{env['SCRIPT_NAME']}",
          "PATH_INFO=#{env['PATH_INFO']}"]]
          }
    end
  end
  map '/world' do
    run lambda {|env| [200, {'Content-Type' => 'text/html'}, ["World "]]}
  end
  map '/' do
    run lambda {|env|[200, {'Content-Type' => 'text/html'},['here']]}
  end

end.to_app

Rack::Handler::Thin.run app, :Port => 3006
```

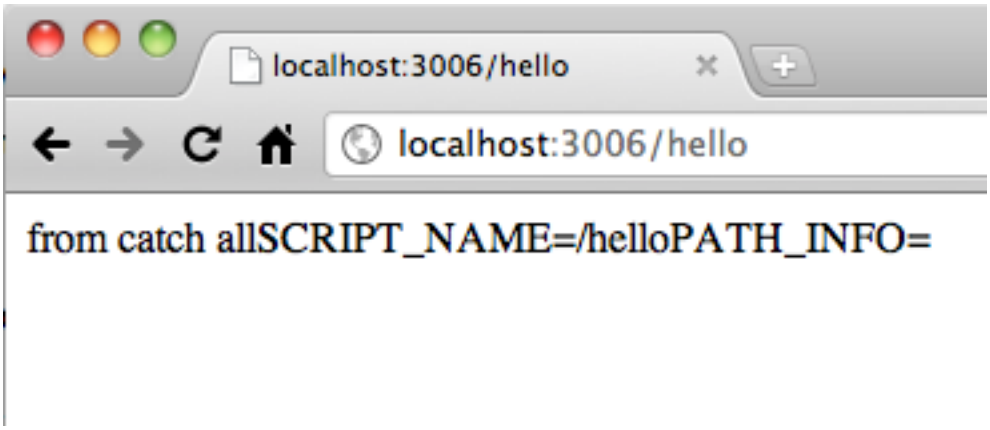

localhost:3006
localhost:3006
here


localhost:3006/world
localhost:3006/world
World


localhost:3006/hello/ketty
localhost:3006/hello/ketty
from hello-kettySCRIPT_NAME=/hello/kettyPATH_INFO=


localhost:3006/hello/everyo
localhost:3006/hello/everyone
from hello-everyoneSCRIPT_NAME=/hello/everyonePATH_INFO=





4. Rackup

Rackup Profile

Rack offers a simple rackup command that allows a configuration file to run our application. If you provide a configuration file config.rb (you can name it anything but the extension must be ru) then run

rackup config.ru

What this command does is equivalent to this:

app = Rack::Builder.new { ... profile goes here... }.to_app

and then running this app via run app.

Let's change the previous program to run it using rackup.

```ruby
# Save the following code in config.ru

map '/hello' do
  run lambda {|env| [200, {"Content-Type" => "text/html"},
    ["SCRIPT_NAME=#{env['SCRIPT_NAME']}", "PATH_INFO=#{env['PATH_INFO']}"]]
    }
end

map '/world' do
  run lambda {|env| [200, {"Content-Type" => "text/html"}, ["world"]]}
end

# Run the above file as :
$ rackup -s thin -p 3006
```

Save the lines from 3 to 11 in a config.ru, run config.ru in the directory where configu.ru is saved. You can see that we removed the require statements and we no longer have hard-coded port numbers in the code.

Besides port number rackup provides other command line parameters.

$ rackup --help

Usage: rackup [ruby options] [rack options] [rackup config]

Ruby options:

-e, --eval LINE evaluate a LINE of code

-d, --debug set debugging flags (set $DEBUG to true)

-w, --warn turn warnings on for your script

-I, --include PATH specify $LOAD_PATH (may be used more than once)

-r, --require LIBRARY require the library, before executing your script

Rack options:

-s, --server SERVER serve using SERVER (webrick/mongrel)

-o, --host HOST listen on HOST (default: 0.0.0.0)

-p, --port PORT use PORT (default: 9292)

-E, --env ENVIRONMENT use ENVIRONMENT for defaults (default: development)

-D, --daemonize run daemonized in the background

-P, --pid FILE file to store PID (default: rack.pid)

Common options:

-h, --help Show this message

--version Show version

We can specify the rackup web application server and the port number. For example:

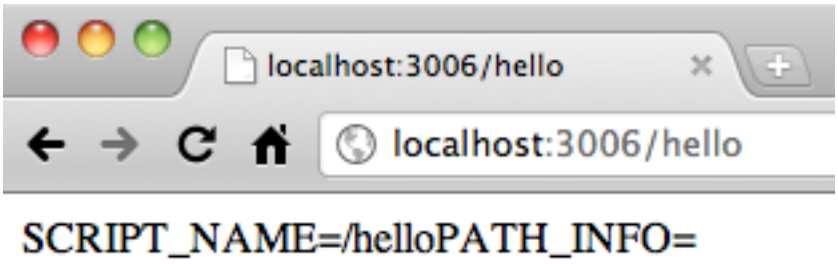
$ rackup -s thin -p 3006

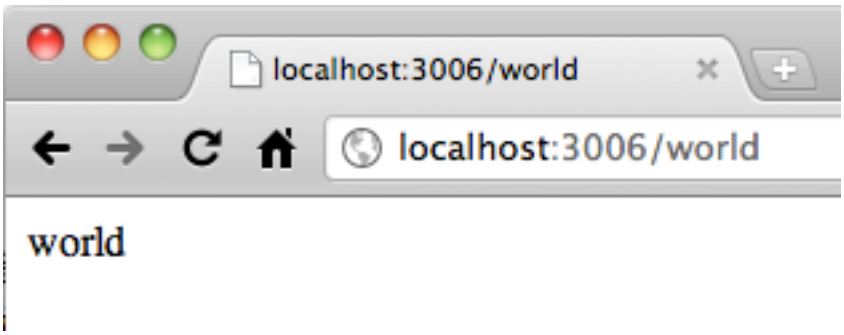
>> Thin web server (v1.2.7 codename No Hup)

>> Maximum connections set to 1024

>> Listening on 0.0.0.0:3006, CTRL+C to stop

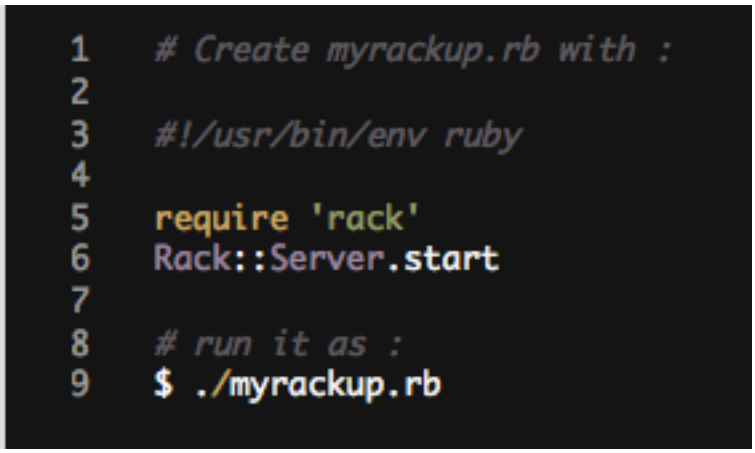
Here is the output of running the program.





5. Closer Look at rackup

Let's take a closer look at how rackup is implemented to get a deeper understanding of how it works behind the scenes.



```
1   # Create myrackup.rb with :
2
3   #!/usr/bin/env ruby
4
5   require 'rack'
6   Rack::Server.start
7
8   # run it as :
9   $ ./myrackup.rb
```

The implementation is just one line of code that involves starting the Rack::Server. If you run this script:

$ ./myrackup.rb

you get the error message: configuration config.ru not found.

Create an empty config.ru file in the same directory and run the script again. This time the server boots up properly.

Rack::Server Interface

Rack:: Server's interface is very simple, it has two class methods, a constructor and five instance methods as shown below.

```ruby
module Rack
  class Server
    def self.start
    def self.middleware

    def initialize(options = nil)
    def options
    def app
    def middleware
    def start
    def server
  end
end
```

Class Methods

Class method *start* is an entry point to Rack::Server, it just creates a new server instance, and it calls the *start* instance method.

```ruby
def self.start
  new.start
end
```

Class method *middleware* assembles some of the default middleware.

```ruby
def self.middleware
  @middleware ||= begin
    m = Hash.new {|h,k| h[k] = []}
    m["deployment"].concat[
      lambda do |server|
        server.server =~ /CGI/ ? nil :[Rack::CommonLogger, $stderr]
      end
    ]
    m["development"].concat
    m["deployment"] + [[Rack::ShowExceptions], [Rack::Lint]]
    m
  end
end
```

Depending on the environment (we can use -e switch to select the environment) we can load different middleware.

* For the default development environment, it will load ShowExceptions and Lint middleware.

* For the deployment environment, it will load ShowExceptions, Lint and CommonLogger middeware.

As the CommonLogger writes to $stderr, and it is not possible on a server with CGI, so on the CGI server, CommonLogger will not be loaded.

@middleware is a Hash, it's key is the name of the environment, its value is an array, the first element of the array contains the name of the middleware class that needs to be pre-loaded for this particular environment and the second element of the array contains the parameters that is needed to instantiate the middleware. For example: Rack::CommonLogger.new($stderr) instantiates the CommonLogger middleware.

Instance Methods

start depends on other method to fulfill it's funtion, namely options:

```
1   def options
2     @options ||= parse_options(ARGV)
3   end
```

The *parse_options* resolves parameters passed at the command line. It takes the default parameters and command line parameters passed at run-time to merge, and finally return a Hash. For example, if the command line entered is:

```
rackup -s Thin config.ru
```

Then the options will include: server => 'Thin',: config => 'config.ru' two keywords, the value of app

```
1   def app
2     @app ||= begin
3       if !::File.exist? options[:config]
4         abort "configuration #{options[:config]} not found"
5       end
6       app, options = Rack::Builder.parse_file(self.options[:config],opt_parser)
7       self.options.merge! options
8       app
9     end
10  end
```

We know the options [: config] contains a configuration file name. So lines 3-5 checks if the file exists. Most important line is 7, which uses Rack::Builder to read the configuration file, and create a app. If you check Rack::Builder files, you can see

```
1   class Builder
2     def self.parse_file(config, opts = Server::Options.new)
3       ...
4       app = eval "Rack::Builder.new {( " + cfgfile + "\n )}.to_app",
5       ...
6   end
```

Why does *parse_file* return options? It is because the Rack::Builder also allows you to start with the configuration file options. If the first line in a config.ru starts with: # \ , it means this line is the option line. For example, you can specify the server running on port 8765 and turn on warnings:

```
1   #\ -w -p 8765
2   run lambda {|env| [200, {"Content-Type" => "text/html"}, ['hello'] ]}
```

server

```
1   def server
2    @_server ||= Rack::Handler.get(options[:server]) || Rack::Handler.default
3   end
```

This method depends on the configuration provided on the command line for the server as -s option to obtain the corresponding Rack::Handler, it this is not specified then it defaults to Rack::Handler::WEBrick.

middleware

```
1   def middleware
2     self.class.middleware
3   end
```

You can either use this instance method or directly call the class method for middleware.

build_app

Finally, we also need to know how the build_app method works

```
1   def build_app(app)
2     middleware[options[:environment]].reverse_each do |middleware|
3       middleware = middleware.call(self) if middleware.respond_to?(:call)
4       next unless middleware
5       klass = middleware.shift
6       app = klass.new(app, *middleware)
7     end
8     app
9   end
10
11  def wrapped_app
12    @wrapped_app ||= build_app app
13  end
```

app is the parameter passed Rack::Server when using Rack::Builder to construct applications. The value of middleware [options [: environment]] determines what need to be preloaded as discussed in an earlier section. Remember we are talking about class method middleware, all the needs of a particular environment is a pre-loaded with the required middleware.

Each member of each array represents a middleware class - there are two possibilities:

1) lambda {|server| server.server =~ /CGI/ ? nil :[Rack::CommonLogger, $stderr] }. The result of calling this lambda is that you may get a nil or get an array of middleware.

2) An array, one or more elements, the first element contains the class name of the middleware and the other elements in that array contains the required parameters to instantiate that middleware.

build_app method handles the processing of the first case on lines 3-4. The second case is handled by lines 5-6 :

start

start method is very easy to understand.

```ruby
def start
  if options[:debug]
    $DEBUG = true
    require 'pp'
    p options[:server]
    pp wrapped_app
    pp app
  end

  if options[:warn]
    $-w = true
  end

  if includes = options[:include]
    $LOAD_PATH.unshift *includes
  end

  if library = options[:require]
    require library
  end

  daemonize_app if options[:daemonize]
  write_pid if options[:pid]
  server.run wrapped_app, options
end
```

In addition to handling some of the parameters, the most important part is the last statement

server.run wrapped_app, options

This is like:

Rack::Handler:XXX.run app, options

The sharp reader will find rackup did not involve dealing with session, logging, etc. The next chapter will cover those topics.

V. Middleware - Indepth Look

1. Response Body Revisit

In chapter two we saw how to set the response body, and then in chapter 3 we said that the response body must be able to respond to *each* method, and now we will take an in-depth look at response body.

Rack specification states:

The Body must respond to *each* and must only yield String values.

The Body itself should not be an instance of String, as this will break in Ruby 1.9. If the Body responds to *close*, it will be called after iteration. If the Body responds to *to_path*, it must return a String identifying the location of a file whose contents are identical to that produced by calling *each*; this may be used by the server as an alternative, possibly more efficient way to transport the response. The Body commonly is an array of Strings, the application instance itself, or a File-like object.

Here are some observations:

1) The response body must be able to respond to *each* method, *each* returns only string values.

2) String#each method has been removed in Ruby 1.9, so response body should not be a string.

3) If the Body responds to *to_path*, then this method should return the path name of a file, which can be handled more efficiently.

4) Response body is usually an array of strings, the application instance itself or File-like object

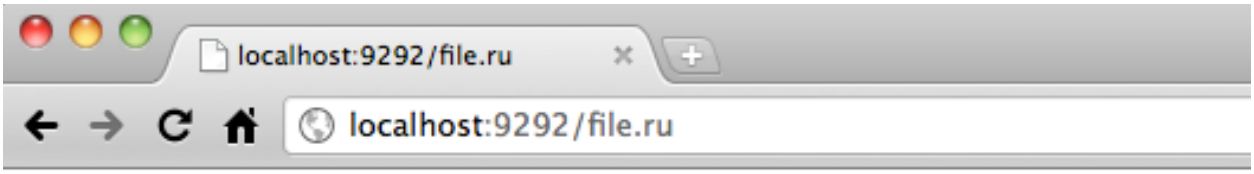
5) If the body responds to the *close* method, it will be called after iteration, where you can implement cleanup work, such as closing files, etc.

6) *close* will be called if present

Here is an example of the response body that uses File object

```
1   use Rack::ContentLength
2   use Rack::ContentType, "text/plain"
3   run lambda {|env| [200, {}, File.new(env['PATH_INFO'][1..-1])]}
```

Save this program as file.ru and run it as rackup file.ru. Open a browser and enter: http://localhost:9292/file.ru, we can see file.ru file in the browser. This is so because our response body a File object, so it can respond to *each* method, Handler will continue to call *each* method, and for each time we get a string output. Also note that the default port is 9292.




```
use Rack::ContentLength
use Rack::ContentType, "text/plain"
run lambda {|env| [200, {}, File.new(env['PATH_INFO'][1..-1])]}
```

So *each* method has many advantages

1) Response body can be any class of any object, as long as it is able to respond to *each* method, each time producing a string value, this gives us a lot of flexibility.

2) Reduce resource consumption by using the File object. We do not need to read the entire file content at once. Otherwise, if the file is large enough, we could run out of memory while processing.

3) Web servers can have more opportunities and choices to optimize, for example, depending on the type of content, each output can be selected each time, one output or the output buffer to a certain size to generate the response content.

Now let's consider why the response body is usually the application itself. Let's revisit the example from chapter 3 reproduced here for reference.

```
1   class Decorator
2     def initialize(app)
3       @app = app
4     end
5     def call(env)
6       status, headers, body = @app.call(env)
7       new_body = "==========header==========<br/>"
8       body.each {|str| new_body << str}
9       new_body << "<br/>==========footer==========="
10      [ status, headers, [new_body]]
11    end
12  end
```

We removed setting the 'Content-Length' related code - this can be done to other middleware. The middleware does not seem a problem. But suppose our original Rack application is a response body that is a document, in front of this we use the Decorator as in the following config.ru:

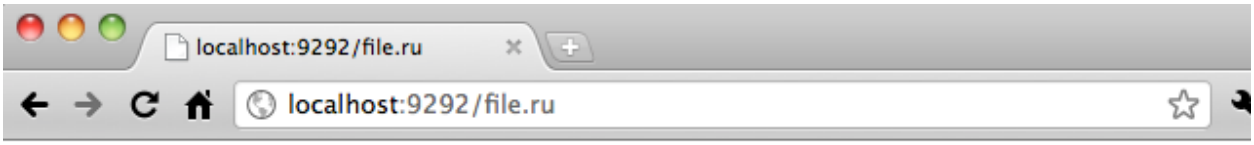
```ruby
require 'decorator'

use Rack::ContentLength
use Rack::ContentType, "text/plain"
use Decorator
run lambda {|env| [200, {}, File.new(env['PATH_INFO'][1..-1])] }
```

Decorator's line number 8 appends all the contents of the file to a new_body string, when the files are very large, it takes too much time to process. To solve this problem, we only need to modify Decorator middleware. Add an *each* method, and delegate the *each* method call to the body as shown below.

```ruby
class Decorator2
  def initialize(app, *options, &block)
    @app = app
    @options = (options[0] || {})
  end
  def call(env)
    status, headers, @body = @app.call(env)
    @header = (@options[:header] || '-----Header-----')
    @footer = (@options[:footer] || '-----Footer-----')
    [status, headers, self]
  end

  def each(&block)
    block.call @header
    @body.each(&block)
    block.call @footer
  end
end
```

First, *each* method calls the block with @header, then it delegates the each method call to the @body and passes the block as the input parameter, the final output is generated by calling the block with @footer.

Here is the ouput that is wrapped by header and footer:




```
-----Header-----use Rack::ContentLength
use Rack::ContentType, "text/plain"
run lambda {|env| [200, {}, File.new(env['PATH_INFO'][1..-1])]}-----Footer-----
```

Body as Array

```ruby
require 'rubygems'
require 'rack'

class HelloWorld
  def call(env)
    [200, {"Content-Type" => "text/html"}, ["Hello ", "World!"]]
  end
end

Rack::Handler::Mongrel.run HelloWorld.new, :Port => 9292
```

Body as IO Object

```ruby
require 'rubygems'
require 'rack'

class HelloWorld
  def call(env)
    [200, {"Content-Type" => "text/html"},
      StringIO.new("Hello World!")]
  end
end

Rack::Handler::Mongrel.run HelloWorld.new, :Port => 9292
```

Body as self

```ruby
require 'rubygems'
require 'rack'

class HelloWorld
  def call(env)
    [200, {"Content-Type" => "text/html"}, self]
  end

  def each
    yield "Hello "
    yield "World!"
  end
end

Rack::Handler::Mongrel.run HelloWorld.new, :Port => 9292
```

2. Rack Middleware

Rack itself provides a lot of middleware, involving all aspects of Web development, including

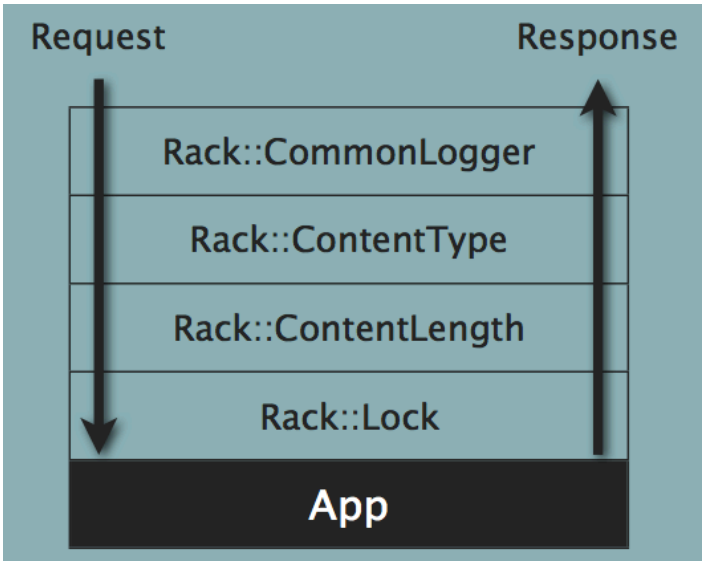
1) HTTP protocol-related processing middleware, the middleware can be the cornerstone of Web frameworks

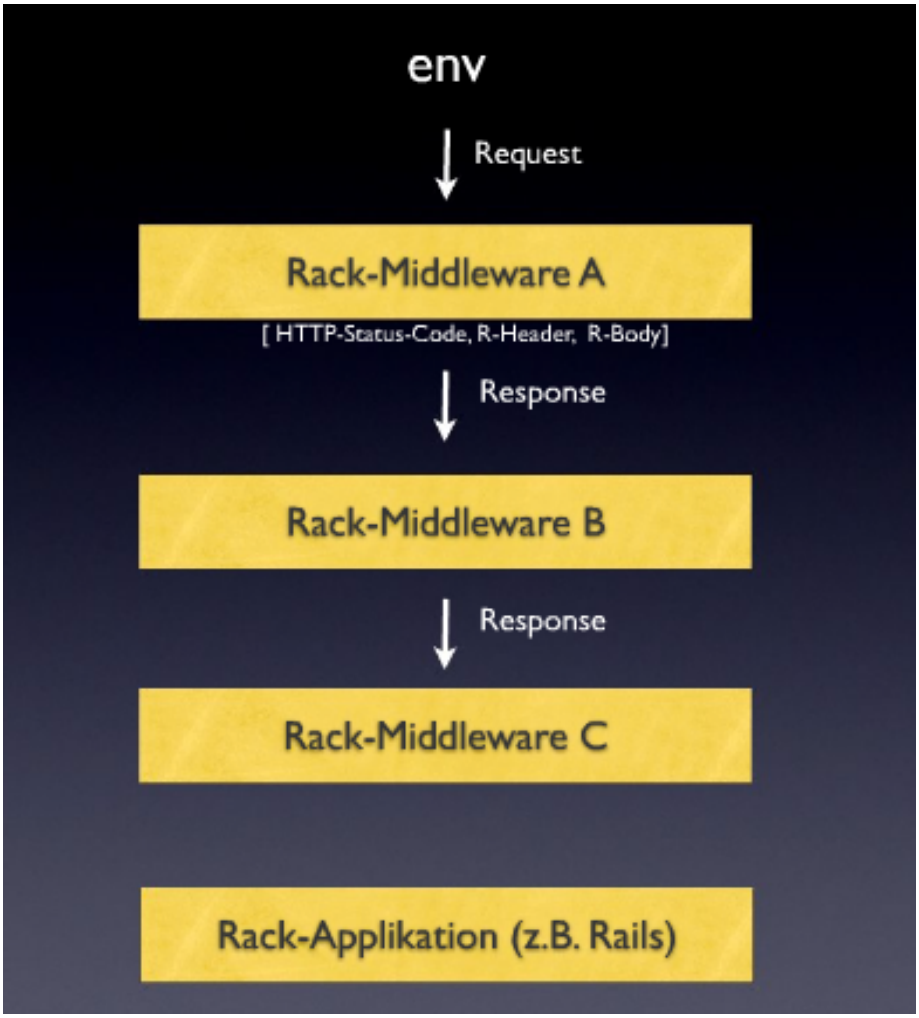
2) Utilities related to the development, such as code reload, logs, compliance checking, etc

3) Web applications often deal with common problems, including the session, cookies, etc.

Some middleware can reduce the development time of our application - they are frequently encountered problems in Web application development and their common solution - such as static file processing. And familiarity with the other middleware allows us to quickly understand any Web framework that complies with Rack spec. Of course, if you want to develop a new Web framework, as long as it is Rack compliant your middleware can be easily reused in any other Rack compliant Web framework.

Rack distribution includes utilities that combine, compose, aggregate or modify Rack applications. Middleware is stackable, it's just a Rack application itself. Since Rack applications are just Ruby objects they are easy to write.


Request
Response
Rack::CommonLogger
Rack::ContentType
Rack::ContentLength
Rack::Lock
App


env
Request
Rack-Middleware A
[HTTP-Status-Code, R-Header, R-Body]
Response
Rack-Middleware B
Response
Rack-Middleware C
Rack-Applikation (z.B. Rails)

(a) HTTP Protocol Middleware

Rack::Chunked

HTTP protocol has a mechanism to block transfer encoding (Chunked Transfer Encoding), that is, a HTTP consumer information can be divided into multiple parts for transmission.

HTTP request and HTTP response it is applicable. We are here mainly considering the transfer to the client from the server response

In general, in a HTTP service, the content is transferred to the client all at once, the length of this content In the 'Content-Length' header field in the statement. This field is needed because the client needs to know when the Response ends. However, in some cases, the server may not know in advance size of the content to be transferred or because of performance reasons do not want a one-time generation and transfer all of the response (compression is such an example)

Then it can use this mechanism to block transmission to transfer data piece by piece. In general, the "block" size is the same, but this is not a mandatory requirement. To take advantage of block transfer mechanism, the server has to first write a Transfer-Encoding header field and so its value is "'chunked"', the contents of each piece consists of two parts (CRLF followed by linefeed):

1. A 16-hexadecimal value that the block size, followed by a CRLF

2. Data itself followed by a CRLF

One line of the last block, which block size is 0, and finally the entire HTTP message with a CRLF in the end. The following is an example of the HTTP message.

HTTP/1.1 200 OK

Content-Type: text/plain

Transfer-Encoding: chunked

25

This is the data in the first chunk

1C

and this is the second one

0

Note that after the last one and a blank line 0

Now Rack::Chunked code is very easy to understand

```ruby
def call(env)
  status, headers, body = @app.call(env)
  headers = HeaderHash.new(headers)

  if env['HTTP_VERSION'] == 'HTTP/1.0' ||
    STATUS_WITH_NO_ENTITY_BODY.include?(status) ||
    headers['Content-Length'] ||
    headers['Transfer-Encoding']
    [status, headers, body]
  else
    dup.chunk(status, headers, body)
  end
end
```

HeaderHash is a subclass of Hash found in Rack utils library. Its key is not case sensitive, but preserves the original case of a header when set.[7]

require 'rubygems'

require 'rack'

h = Rack::Utils::HeaderHash.new

h["abc"] = "234" #=> "234"

h["ABC"] #=> "234"

h.keys #=> ["abc"]

It can be used to easily access the HTTP header information call method first to determine whether the current HTTP_VERSION is 1.0, or status is STATUS_WITH_NO_ENTITY_BODY, whether the headers contains the Content-Length and Transfer-Encoding header field set. If so, the Rack::Chunked does nothing, otherwise it calls:

dup.chunk(status, headers, body)

chunked method is a typical response that returns self as the response body.

---

7. http://github.com/rack/rack/blob/master/lib/rack/utils.rb

```ruby
def chunk(status, headers, body)
  @body = body
  headers.delete('Content-Length')
  headers['Transfer-Encoding'] = 'chunked'
  [status, headers, self]
end
```

This means that the Rack::Chunked must have an each method implemented as:

```ruby
def each
  TERM = "\r\n"
  @body.each do |chunk|
    size = bytesize(chunk)
    next if size == 0
    yield [size.to_s(16), TERM, chunk, TERM].join
  end
  yield "0#{TERM}#{TERM}"
end
```

The each method above encodes the output of block according to HTTP protocol standard: First, the size of the output block is calculated to add a CRLF, then the specific content to add a CRLF, and finally a 0 and a A CRLF.

Finally Rack::Chunked also defines a *close* method to comply with Rack Specification, so if the body responds to *close* then the *close* method is invoked on the body.

```ruby
def close
  @body.close if @body.respond_to?(:close)
end
```

Many Web servers' Rack Handler will automatically load the Rack::Chunked middleware, we will discuss this later.

Rack::ConditionalGet

The semantics of the GET method change to a "conditional GET" if the request message includes an If-Modified-Since, If-Unmodified-Since, If-Match, If-None-Match, or If-Range header field. A conditional GET method requests that the entity be transferred only under the circumstances described by the conditional header field(s). The conditional GET method is intended to reduce

unnecessary network usage by allowing cached entities to be refreshed without requiring multiple requests or transferring data already held by the client. [8]

HTTP protocol defines a caching mechanism, when a client requests a resource, if the server found that the resources has not been modified, it can tell the client that the content is not modified, so the client can use it's own cached content. This has many advantages, for example, a server does not have to re-generate content, thereby reducing both the CPU and network resources.

When a browser requests a certain url for the first time, because there is no cache data, so it makes a direct request like this:

GET /example.html HTTP/1.1

Host: www.mydomain.com

Cache Control

The best outcome is that the client does not send the request to the server and it directly serves up the content from its local cache.

The Cache-Control general-header field is used to specify directives that must be obeyed by all caching mechanisms along the request/response chain. The directives specify behavior intended to prevent caches from adversely interfering with the request or response. These directives typically override the default caching algorithms. Cache directives are unidirectional in that the presence of a directive in a request does not imply that the same directive is to be given in the response.

Cache directives must be passed through by a proxy or gateway application, regardless of their significance to that application, since the directives might be applicable to all recipients along the request/response chain. It is not possible to specify a cache- directive for a specific cache. [9]

If you want the cache server to control the server response, the header will contain a directive called the Cache-Control. For example:

Cache-Control: max-age=21600

8. http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

9. http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

In this example, Cache-Control contains a max-age value, this value indicates the number of seconds expired. Clients save the contents of the document in its own cache, and record the document cache expiration time. Content is cached till the max-age specified number of seconds is reached, the client knows that the document is fresh (ie., the server document has not changed), so within the max-age duration the client does not interact with the server in order to serve the same document. In this case the client can obtain the document directly from the cache. Therefore, for static content, the server should send a :

Cache-Control: maxage = ...,

and set the max-age to a very large value. We cannot determine the proper value for the max-age, therefore, the usual practice is to set the value to Max.

(b) HTTP Protocol Middleware

But doing this will lead to another dilemma, that if we really took place static content changes, the browser will almost always get the latest content.

There is a small trick that we can use, the so-called "smart URL", the server generating the static files of any reference, in the actual URL followed by a value of the last modification time, for example:

http://www.mydomain.com/javascripts/jquery.js?1263983825

Clients get the file, its cache is set to max-age to the maximum. If the file is indeed never been modified, then any time the client makes a request they get this document. The URL is above URL. If this file is changed, then the server re-generates the document and it will add a different time value. For example:

http://www.mydomain.com/javascripts/jquery.js?1373486878

This client sees a new URL, so it will go back to the server to obtain the new document.

This method is particularly suitable for resource files, including images, CSS, JavaScript, and static pages etc.

There are more options that are available and we will not be covering it here. You can refer http://www.w3.org/Protocols/rfc2616 for more details.

If the document cache has expired (more than the max-age range of the specified time), then the server must revalidate.[10]

---

10. http://rtomayko.github.com/rack-cache/

1) If the verification result is "document is not changed," you can continue to use the cached document.

2) If the verification result is "document changed", then get the new content of the document, used to update the cache. In addition to the cache expires, there are many reasons to re-confirm the cause, including

a) Force a refresh of the user when the client, regardless of whether the new cached document will re-confirm.

b) Other Cache-Control directive. We only discussed the max-age, Cache-Control has many options. For example, using must-revalidate directive at any time to tell the client to re-confirm the document, the dynamic content in general will use this command. Revalidation is to be a condition of access requests. There are two ways to achieve the conditions for the request, one based on Last-Modified and Etag. Determine the different methods, the server may return a different response.

Last-Modified Based

The first time the server's HTTP response message, in addition to the message body, Cache-Control header field and other content, but also includes a Last-Modified header field, its value is the time the document was last modified, for example,

Last-Modified: 20 Sep 2008 18:23:00 GMT

Conditions for the client sends the request will include the following request header field

If-Modified-Since: 20 Sep 2008 18:23:00 GMT

After the server accepts this request, you will have the requested document was last modified time, and If-Modified-Since the specified time. If you have changed, the latest documents are normally sent. If there is no change, then the server will return a 304 Not Modified header.

If we can easily calculate the time of last modification of a document, then the Last-Modified is appropriate. Note however that if page content is a combination of many fragments generated, then we need calculation for all segments of the last modification time and date of the last modification time to take the entire document as the last modification time. If you have multiple Web servers at the same time service, we also must ensure that the time between these servers is synchronized.

ETag based

Sometimes the contents of the document is a combination of many pieces (such as different database records), it is difficult to calculate modified, or last modified time. In these cases we can use Etag. Etag usually relies on the value of the real content of the document, we must ensure different content produce different Etag and the same content have the same Etag.

In this way, the server's first response contains a Etag value, for example:

Etag: 4135cda4de5f

Conditions for the client when it sends the request will include the following request header field

If-None-Match: 4135cda4de5f

Combination of Etag and Last-Modified

Server can mix Etag and Last-Modified. If the client receives from the server for the first time responses Header field that contains both the directives, then validate the document in the back when the cache request header is in the same package with If-Modified-Since and If-None-Match header field.

How to determine whether there was a change to the document on the server depends on implementation. On the Rack::ConditionalGet, we only need to match any one of them, but for other middleware, frameworks, or Web servers, it may need to match both of these two conditions.

RFC2616 provides the message body 304 response, after all the header information followed by a blank line to indicate ring should end.

Rack::ConditionalGet Implementation

Middleware that enables conditional GET using If-None-Match and If-Modified-Since. The application should set either or both of the Last-Modified or Etag response headers according to RFC 2616. When either of the conditions is met, the response body is set to be zero length and the response status is set to 304 Not Modified.

Applications that defer response body generation until the body's each message is received will avoid response body generation completely when a conditional GET matches. [11]

If any of the following two conditions are met, then it does not change the contents of the request:

* Etag Match: request header HTTP_IF_NONE_MATCH response header value is equal to Etag

* Last-Modified Match: request header HTTP_IF_MODIFIED_SINCE response header value is equal to Last-Modified

The following code in the headers parameter is the response header

11. http://rack.rubyforge.org/doc/Rack/ConditionalGet.html

```ruby
def etag_matches?(env, headers)
  etag = headers['Etag'] and etag == env['HTTP_IF_NONE_MATCH']
end

def modified_since?(env, headers)
  last_modified = headers['Last-Modified'] and
    last_modified == env['HTTP_IF_MODIFIED_SINCE']
end
```

To make sure that this is a document that did not change, ConditionalGet will do three things

* Set the response status code to 304

* Clear the response headers Content-Type and Content-Length

* Clear the response body

Now, the following code should be easy to understand:

```ruby
def call(env)
  return @app.call(env) unless %w[GET HEAD].include?(env['REQUEST_METHOD'])

  status, headers, body = @app.call(env)
  headers = Utils::HeaderHash.new(headers)
  if etag_matches?(env, headers) || modified_since?(env, headers)
    status = 304
    headers.delete('Content-Type')
    headers.delete('Content-Length')
    body = []
  end
  [status, headers, body]
end
```

We can now see how to use Etag:

Please note that the actual procedure should have the appropriate algorithm (such as Digest provide methods) to generate Etag. In a terminal run the program with rackup, and in another terminal open telnet

```
mbp2:book bparanj$ telnet 127.0.0.1 3001
Trying 127.0.0.1...
Connected to app.test.
Escape character is '^]'.
GET / HTTP/1.1
Host: localhost

HTTP/1.1 200 OK
Etag: 12345678
Connection: Keep-Alive
Content-Type: text/html
Date: Wed, 13 Oct 2010 03:56:06 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.7/2010-06-23)
Content-Length: 11

hello worldConnection closed by foreign host.
mbp2:book bparanj$
```

After *Host: localhost* line you must hit <*enter*> twice to get the server response. Our request did not include any Etag information, so the program returns 200 OK

The response contains this document Etag, Content-Length and Content-Type, as well as actual content. We now know that this document Etag is 12345678, you can use a If-None-Match specified Etag:

```
mbp2:book bparanj$ telnet 127.0.0.1 3001
Trying 127.0.0.1...
Connected to app.test.
Escape character is '^]'.
GET / HTTP/1.1
Host: localhost
If-None-Match: 12345678

HTTP/1.1 304 Not Modified
Etag: 12345678
Date: Wed, 13 Oct 2010 04:03:59 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.7/2010-06-23)
```

This time, the program returns a 304 Not Modified, which means that status code has been set correctly. This time there is no Content-Length and Content-Type, or the actual contents of the body.

You can write a simple program to test Last-Modified header field. Note you can use httpdate method to convert any time object to a legitimate http date format object.

require 'time'

Time.now.httpdate

This further demonstrates why we should not directly assemble a string when called, but delay the implementation to *each* method where it is generated, because in some cases we may not need to generate the complete body.

Rack::ContentLength

Except for some special cases, HTTP protocol requires the server response header, including the ContentLength to be set. It must be equal to the length of the response body.

A program may use a number of Rack middleware, the middleware may modify the response of each body of content, response body will require each layer to calculate and set the correct ContentLength field.

```
1   def call(env)
2     status, headers, body = @app.call(env)
3     headers = HeaderHash.new(headers)
4
5     if !STATUS_WITH_NO_ENTITY_BODY.include?(status.to_i) &&
6        !headers['Content-Length'] &&
7        !headers['Transfer-Encoding'] &&
8        (body.respond_to?(:to_ary) || body.respond_to?(:to_str))
9
10      body = [body] if body.respond_to?(:to_str) # rack 0.4 compat
11      length = body.to_ary.inject(0) { |len, part| len + bytesize(part) }
12      headers['Content-Length'] = length.to_s
13    end
14
15    [status, headers, body]
16  end
```

The bytesize used in line 11 is defined in Rack::Utils module, it is mainly used to deal with the Ruby 1.9 and Ruby 1.8 differences. String#size Ruby 1.9 will return the number of bytes rather than number of characters (for example, the code for UTF-8 code, each character takes more than one byte), only String.bytesize the number of bytes before returning true - this is the HTTP protocol requirements.

```
1   # Return the bytesize of String; uses String#length under Ruby 1.8 and
2   # String#bytesize under 1.9.
3   if ''.respond_to?(:bytesize)
4     def bytesize(string)
5       string.bytesize
6     end
7   else
8     def bytesize(string)
9       string.size
10    end
11  end
```

Middleware to make this set Content-Length value must meet the four conditions:

* Response body with content does not contain a response header Content-Length value.

* Not set Transfer-Encoding header field. Transfer-Encoding header field contains the response.

* Must respond to to_ary or to_str

* to_str mainly for code written in Ruby 1.8, the body may be used directly in response to a string

* to_ary that can be converted to an array, this number is not fixed because of the length of the message

(Eg, block transfer) does not want to set the Content-Length, or is unable to calculate the length of the Response body

Transfer Length

For completeness sake, we are here to discuss the length of the transmission of messages, you do not have to understand all the details.

Server and client according to the length of the message to determine when to transmit this request / response content has been full Department has been received.

Standard argument with the agreement, the transmission of a message refers to the length of the length of the message body, that is, all the transmission Coding is applied after transmission length. The length of the message body can follow the following rules to determine (ranked by priority Order):

1. Should not be any response message body contains the message - a message that includes 1xx, 204,304, and any of HEAD Requests

Demand response, all header fields are always in the back of the end of the first blank line. Rack::Utils define these status codes:

Responses with HTTP status codes that should not have an entity body

STATUS_WITH_NO_ENTITY_BODY = Set.new((100..199).to_a << 204 << 304)

2 If a Transfer-Encoding header, and its value is not identity, then there are two ways to

To determine the transfer length:

* Use the "chunked" transfer encoding to determine (see earlier discussion); or

* Close the connection that the message has ended

3 If you have a Content-Length header field, it represents the number of physical constants and transmission length Degrees. If the entity's length and the transmission of different length, must not set the Content-Length header word segment. (Examples include Transfer-Encoding and Content-Encoding is set Situation.) If the received message includes both Transfer-Encoding header field and Content-Length header field, then the Content-Length should be ignored

4.If a message using the media type "multipart /byteranges", and does not refer to the previous method used Given transmission length, then the media type itself with rules to determine the boundaries of the transmission length

5.Last resort is the server closes the connection (this method can only be used by the server, the-client off. If the connection can not accept the response of the) Conversely, if you send a message, we can know the message length, you should set the combined appropriate Content-Length header field. And if you can not know in advance the length, or very difficult to calculate the length of, We can tell each other in three ways:

1) Using chunked transfer encoding. HTTP/1.1 applications that will require all HTTP to be able to handle block transfers.

2) Close the connection (can only be used by theserver)

3) Using multipart / byteranges, but it is not suitable for all messages. It should be noted that in the not confirm the length of the case, must not set the Content-Length header field. If set to an incorrect Content-Length, it will have the following consequences:

a) Content-Length value is greater than the actual transfer length. This will lead to the receiver (such as browser) Waiting to receive the message can not reach, the browser has been suspended. In the case of persistent connections, the recipient will read the next response (or request) of the part, the entire communication process is destroyed.

b) Content-Length value is less than the actual transfer length. In the case of persistent connections, the message of a partly as a response to the next (or request) to read the entire communication process is destroyed.

Rack::ContentType

Sets the Content-Type header on responses which don't have one.
Builder usage:
use Rack:: ContentType, "text/html"
When no content type argument is provided, "text/html" is assumed.

```
require 'rack/utils'

module Rack

  # Sets the Content-Type header on responses which don't have one.
  #
  # Builder Usage:
  # use Rack::ContentType, "text/plain"
  #
  # When no content type argument is provided, "text/html" is assumed.
  class ContentType
    def initialize(app, content_type = "text/html")
      @app, @content_type = app, content_type
    end

    def call(env)
      status, headers, body = @app.call(env)
      headers = Utils::HeaderHash.new(headers)
      headers['Content-Type'] ||= @content_type
      [status, headers, body]
    end
  end
end
```

Rack::Deflater

HTTP supports the transmission of compressed content. HTTP compression, otherwise known as content encoding, is a publicly defined way to compress textual content transferred from web servers to browsers. HTTP compression uses public domain compression algorithms, like gzip and compress, to compress XHTML, JavaScript, CSS, and other text files at the server. This standards-based method of delivering compressed content is built into HTTP 1.1, and most modern browsers that support HTTP 1.1 support ZLIB inflation of deflated documents. In other words, they can decompress compressed files automatically, which saves time and bandwidth.

Stephen Pierzchala, Senior Technical Performance Analyst with Gomez, said this about HTTP compression:

"When tied to other methods, such as proper caching configurations and the use of persistent connections, HTTP compression can greatly improve Web performance. In most cases, the total cost of ownership of implementing HTTP compression (which for users of some Web platforms is nothing!) is extremely low, and it will pay for itself in reduced bandwidth usage and improved customer satisfaction."[12]

Refer appendix to read http specfication to understand the basics.

Rack::Deflater Implementation

Here is the specs from the Rack gem:

should "be able to deflate bodies that respond to each"

TODO: This is really just a special case of the above...

should "be able to deflate String bodies"

should "be able to gzip bodies that respond to each"

should "be able to fallback to no deflation"

should "be able to skip when there is no response entity body"

should "handle the lack of an acceptable encoding"

should "handle gzip response with Last-Modified header"

should "do nothing when no-transform Cache-Control directive present"

---

12. http://www.websiteoptimization.com/speed/tweak/compress/

Now we are ready to look at the implementation.

```
def call(env)
  status, headers, body = @app.call(env)
  headers = Utils::HeaderHash.new(headers)
  # Skip compressing empty entity body responses and responses with no-transform set.
  if Utils::STATUS_WITH_NO_ENTITY_BODY.include?(status) ||
       headers['Cache-Control'].to_s =~ /\bno-transform\b/
    return [status, headers, body]
  end
```

If the response body of Cache-Control header contains no-transform directive then return without modifying anything.

```
request = Request.new(env)
encoding = Utils.select_best_encoding(%w(gzip deflate identity), request.accept_encoding)
```

First with env create a new request object, so that we can directly access the request.accept_encoding. The line Utils.select_best_encoding(%w(gzip deflate identity), request.accept_encoding) chooses the best encoding from the available choices we have already discussed in detail before. The first parameter: %w(gzip deflate identity) is the list of available encoding.

```
# Set the Vary HTTP header.
vary = headers["Vary"].to_s.split(",").map { |v| v.strip }
unless vary.include?("*") || vary.include?("Accept-Encoding")
  headers["Vary"] = vary.push("Accept-Encoding").join(",")
end
```

This snippet of code handles the Vary header, if one already contains the * or Accept-Encoding, you do not need to do anything. Otherwise, the content will be encoded.

The following code to encode the content, is a case statement. There are three encoding Deflater middleware available, gzip, deflate, and identity. If the code is not selected any of them (se-

lect_best_encoding returns nil), in accordance with protocol requirements discussed earlier, we must return a 406 response. Indentity code that is without any coding, can also be returned intact.

```
case encoding
when "gzip"
  .....
when "deflate"
  .....
when "identity"
  [status, headers, body]
when nil
  message = "An acceptable encoding for the requested resource #{request.fullpath} could not be found."
  [406, {"Content-Type" => "text/plain", "Content-Length" => message.length.to_s}, [message]]
end
```

gzip encoding

Gzip branch of the case statement, the code is as follows:

```
when "gzip"
  headers['Content-Encoding'] = "gzip"
  headers.delete('Content-Length')
  mtime = headers.key?("Last-Modified") ? Time.httpdate(headers["Last-Modified"]) : Time.now
  [status, headers, GzipStream.new(body, mtime)]
```

This sets Content-Encoding header to gzip, removes the Content-Length header field. Gets mtime for the document that was last modified. Finally, returns the response [Status, headers, GzipStream.new (body, mtime)], the Response body is a new GzipStream object.

Why do we have to remove the Content-Length header? Because we use a certain compression algorithm, if you want to set the correct Content-Length, it must be the compressed length of the content. To get the length of compressed content, we may need to read the memory of all the original content, which is clearly not acceptable for a big document. Then how do we determine the encoded transmission length? Recall from our previous discussion that we can use chunked transfer encoding. In accordance with RFC2616, we should remove Content-Length header field.

Another point, once you delete the Content-Length header field, the application itself, Web Framework or Web server can use Rack::Chunked middleware to use chunked transfer transmission mechanism. If there is Content-Length header field, Rack::Chunked will have no effect.

The response body based on the requirements, GzipStream must be able to respond to *each*, *each* produces a string:

```
class GzipStream
  def initialize(body, mtime)
    @body = body
    @mtime = mtime
  end
  def each(&block)
    @writer = block
    gzip =::Zlib::GzipWriter.new(self)
    gzip.mtime = @mtime
    @body.each { |part| gzip.write(part) }
    @body.close if @body.respond_to?(:close)
    gzip.close
    @writer = nil
  end
  def write(data)
    @writer.call(data)
  end
end
```

The constructor for Zlib::GzipWriter takes an object, that object is the output of compressed data objects, in general, this object is an output IO object (such as File). When you call the write method on GzipWriter instance, GzipWriter will be used to compress by Gzip content compression algorithm.

In the code above, GzipStream#each method first creates a gzip object by passing itself as the parameter to the Zlib::GzipWriter constructor. So when we call write on gzip object, gzip will call an instance of the current GzipStream. The *each* method on the body instance iterates for each part of the original content and compresses the data.

Let's now look at deflate encoding case statement, the branch of the code is as follows:

```
when "deflate"
  headers['Content-Encoding'] = "deflate"
  headers.delete('Content-Length')
  [status, headers, DeflateStream.new(body)]
```

The gzip encoding related code is very similiar to the previous discussion. DeflateStream is fairly simple.

```
class DeflateStream
    DEFLATE_ARGS = [
      Zlib::DEFAULT_COMPRESSION,
      # drop the zlib header which causes both Safari and IE to choke
      -Zlib::MAX_WBITS,
      Zlib::DEF_MEM_LEVEL,
      Zlib::DEFAULT_STRATEGY
    ]

    def initialize(body)
      @body = body
    end

    def each
      deflater = ::Zlib::Deflate.new(*DEFLATE_ARGS)
      @body.each { |part| yield deflater.deflate(part) }
      @body.close if @body.respond_to?(:close)
      yield deflater.finish
      nil
    end
  end
```

The constructor takes the body that needs to be deflated, and the each method delegates the task of deflating to the instance of Zlib::Deflate class which takes some default arguments for handling the compression.

You can either use the following code to test, using telnet, or observe the returned response header in firebug. Here we will use Telnet for demonstration. Save the following file in a deflater.ru.

```
1   use Rack::Chunked
2   use Rack::Deflater
3   run lambda {|env| [200, {'Content-Type'=>"text/html"},["abcde"*1000]]}
```

Run rackup:

$ rackup deflater.ru -p 3010

Telnet into the server to see the header from the server.

```
mbp2:~ bparanj$ telnet 127.0.0.1 3010
Trying 127.0.0.1...
Connected to app.test.
Escape character is '^]'.
GET / HTTP/1.1
Host: localhost

HTTP/1.1 200 OK
Connection: Keep-Alive
Content-Type: text/html
Date: Wed, 13 Oct 2010 22:50:51 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.7/2010-06-23)
Content-Length: 5013
Vary: Accept-Encoding
Transfer-Encoding: chunked

1388
abcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdea
abcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdea
abcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdea
abcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdea
```

Web server to handle the different ways different, in automatically in the necessity to use Rack::Chunked in between the pieces, so you do not have their own use. The WEBrick improperly set Content-Length.

Rack::Etag

In previous section, we discussed how to use HTTP caching by using Etag. Rack::ConditionalGet middleware need to program their own calculations Etag, what it does is to compare the request and response should be in the Etag, and set the appropriate response headers and status. The advantage is that you can in the application code according to some algorithm Etag, once Etag match, middleware ConditionalGet can completely do not call the health into the specific content of each methods, thereby saving the server's CPU resources, network transmission is also greatly reduced.

In some cases, you may not be calculated within the program need to calculate the ETag Etag-fragment are many, many fragments are likely to change. A typical example is a Rails program, a change has occurred a page change depends not only on the content model, but also on the content of external Layout, and all such information may be supplied with the users vary. This time you may choose a program: each time the server still generates capacity, but before the output of the entire response body to do the calculation a Etag-if the content did not change, you should not then transfer the content to the client. Here is the implementation of Rack::ETag:

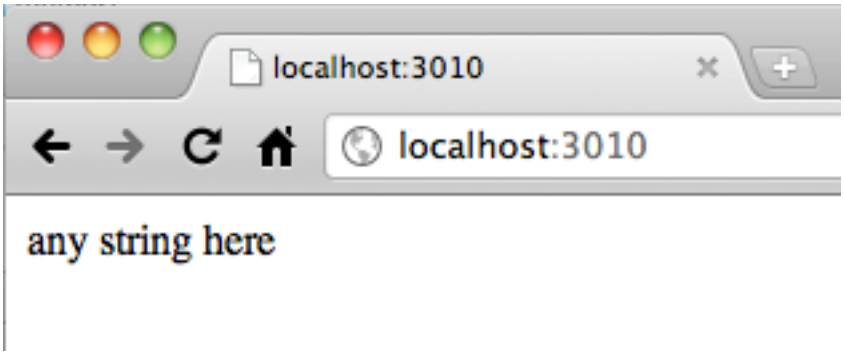
```
require 'digest/md5'

module Rack
  # Automatically sets the ETag header on all String bodies
  class ETag
    def initialize(app)
      @app = app
    end

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

      if !headers.has_key?('ETag')
        digest, body = digest_body(body)
```

```
      headers['ETag'] = %("#{digest}")
    end

    [status, headers, body]
  end

  private
    def digest_body(body)
      digest = Digest::MD5.new
      parts = []
      body.each do |part|
        digest << part
        parts << part
      end
      [digest.hexdigest, parts]
    end
  end
end
```

Etag middleware handles only the first field, not including "Etag" situation. It reads the contents of the Body, and transferred replaced by an array of strings, and finally with Digest::MD5.hexdigest calculate the Etag. Clearly Etag needs and Rack::ConditionalGet to work with:

```
1   use Rack::ConditionalGet
2   use Rack::ETag
3   run lambda{|env| [200, {'Content-Type'=>'text/html'},["any string here"]]}
```

Run this file with rackup.



The first response is 200, the second is 304 as shown in the figure below:

```
rackup etag.ru -p 3010
[2010-10-13 16:12:17] INFO  WEBrick 1.3.1
[2010-10-13 16:12:17] INFO  ruby 1.8.7 (2010-06-23) [i686-darwin10.4.0]
[2010-10-13 16:12:17] INFO  WEBrick::HTTPServer#start: pid=5523 port=3010
127.0.0.1 - - [13/Oct/2010 16:12:33] "GET / HTTP/1.1" 200 - 0.0008
127.0.0.1 - - [13/Oct/2010 16:12:33] "GET /favicon.ico HTTP/1.1" 200 - 0.0007


127.0.0.1 - - [13/Oct/2010 16:13:01] "GET / HTTP/1.1" 304 - 0.0009
127.0.0.1 - - [13/Oct/2010 16:13:01] "GET /favicon.ico HTTP/1.1" 304 - 0.0136
```

Not Modified 304

If the client has done a conditional GET and access is allowed, but the document has not been modified since the date and time specified in If-Modified-Since field, the server responds with a 304 status code and does not send the document body to the client.

Response headers are as if the client had sent a HEAD request, but limited to only those headers which make sense in this context. This means only headers that are relevant to cache managers and which may have changed independently of the document's Last-Modified date. Examples include Date , Server and Expires.

The purpose of this feature is to allow efficient updates of local cache information (including relevant meta information) without requiring the overhead of multiple HTTP requests (e.g. a HEAD followed by a GET) and minimizing the transmittal of information already known by the requesting client (usually a caching proxy).[13]

If the response body content is not large (eg Content-Type to text / html) situation, all content groups synthesis of a string and calculate the cost of MD5 should be able to bear. However, if the

13. http://www.w3.org/Protocols/HTTP/HTRESP.html

response is a large body of File, then this approach is clearly not feasible. So please use caution, at least to determine what type of response body.

Rack::Head

RFC2616 requires HEAD request response body must be empty, which is done by Rack::HEAD:

```
def call(env)
  status, headers, body = @app.call(env)
  if env["REQUEST_METHOD"] == "HEAD"
    [status, headers, []]
  else
    [status, headers, body]
  end
end
```

Rack::MethodOverride

Browser and Web servers generally do not directly support the PUT and DELETE methods, and only support POST method. The Rest-style programming for the PUT, DELETE, and POST have more strict distinction. So we need POST method to simulate using PUT and DELETE methods. There are two methods generally used to simulate POST:

1. In the form POST data submitted, embed a hidden field like this:

```
<form action="....." method="post">
<input name="_method" type="hidden" value="put" />
....
</form>
```

There is a hidden form "_method" field, its value is "put", says that the request is actually a A PUT instead of POST.

2. In the HTTP request add an extended X_HTTP_METHOD_OVERRIDE request header, this Header is in the Rack environment HTTP_X_HTTP_METHOD_OVERRIDE.

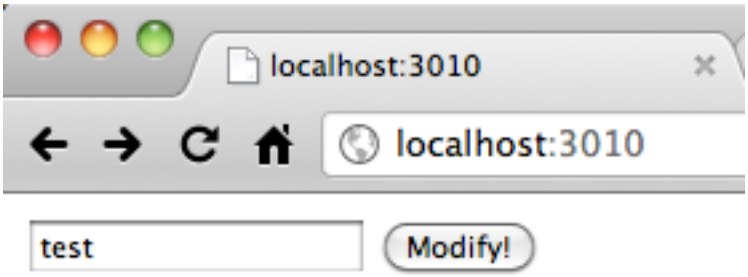
Here's Rack::MethodOverride implementation of how these two cases are treated.

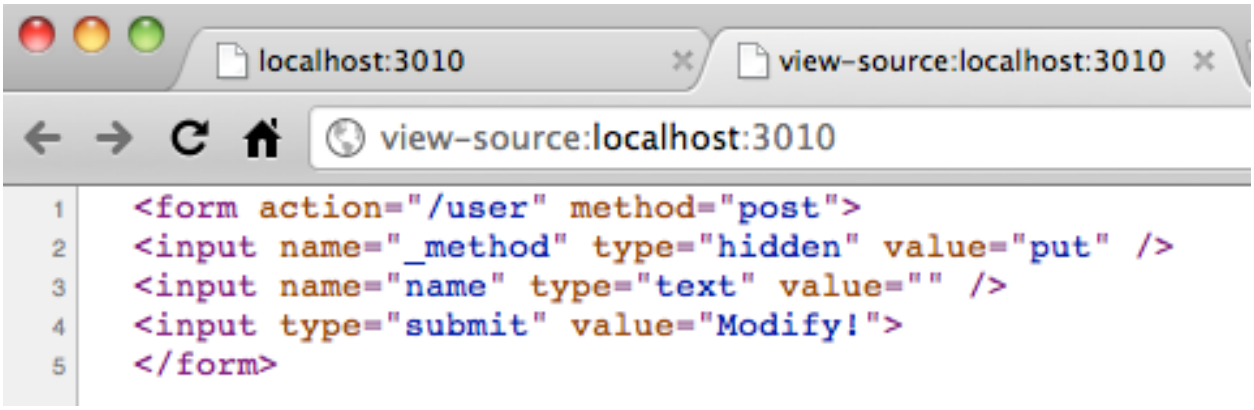
```
HTTP_METHODS = %w(GET HEAD PUT POST DELETE OPTIONS)
METHOD_OVERRIDE_PARAM_KEY = "_method".freeze
HTTP_METHOD_OVERRIDE_HEADER ="HTTP_X_HTTP_METHOD_OVERRIDE".freeze

def call(env)
  if env["REQUEST_METHOD"] == "POST"
    req = Request.new(env)
    method = req.POST[METHOD_OVERRIDE_PARAM_KEY] ||
      env[HTTP_METHOD_OVERRIDE_HEADER]
    method = method.to_s.upcase
    if HTTP_METHODS.include?(method)
      env["rack.methodoverride.original_method"] = env["REQUEST_METHOD"]
      env["REQUEST_METHOD"] = method
    end
  end
  @app.call(env)
end
```

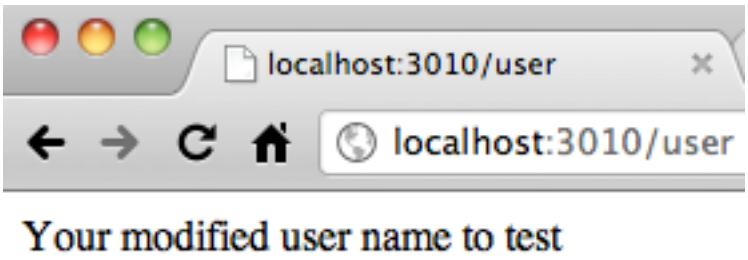
If the request is a POST request, and the data contained in the request "_method" value of the package or in the environment with HTTP_ X_HTTP_METHOD_OVERRIDE values, and their values are legitimate HTTP method, then the middleware will save the original POST and set the new REQUEST_METHOD value. The following program tests whether Rack::MethodOverride works:

```ruby
use Rack::MethodOverride
map '/' do
  form = <<-HERE
    <form action="/user" method="post">
    <input name="_method" type="hidden" value="put" />
    <input name="name" type="text" value="" />
    <input type="submit" value="Modify!">
    </form>
  HERE
  run lambda {|env| [200, {"Content-Type" => "text/html"}, [form]] }
end

map '/user' do
  run lambda {|env|
    req = Rack::Request.new env
    res = Rack::Response.new
    if(req.put?)
      res.write("you modify user name to#{req.params['name']}")
    else
      res.write("we only support put method to modify user,yours is #{req.request_method}")
  end
  res.finish
}
end
```

When a user requests http://localhost:9292, the program returns a form, with hidden field "_method", which sets its value to "put". After the user submits the form, the program receives a PUT request. If you remove the use of MethodOverride, then the method is still the request for the POST, and not PUT.







The output below shows the PUT request sent by the client:

```
mbp2:ch5 bparanj$ rackup method_overide.ru -p 3010
[2010-10-13 16:49:08] INFO  WEBrick 1.3.1
[2010-10-13 16:49:08] INFO  ruby 1.8.7 (2010-06-23) [i686-darwin10.4.0]
[2010-10-13 16:49:08] INFO  WEBrick::HTTPServer#start: pid=5699 port=3010


127.0.0.1 - - [13/Oct/2010 16:49:14] "PUT /user HTTP/1.1" 200 31 0.0020
127.0.0.1 - - [13/Oct/2010 16:49:14] "GET /favicon.ico HTTP/1.1" 200 - 0.0012
```

Now we are ready to look at the code for Rack handler for Thin server:

```
require "thin"
require "rack/content_length"
require "rack/chunked"

module Rack
  module Handler
    class Thin
```

```
    def self.run(app, options={})
      app = Rack::Chunked.new(Rack::ContentLength.new(app))
      server = ::Thin::Server.new(options[:Host] || '0.0.0.0',
                                  options[:Port] || 8080,
                                  app)
      yield server if block_given?
      server.start
    end
  end
 end
end
```

3. Behavioral Middleware

All of our application development follows a similar pattern, we need to read the log, measure performance, check whether a user is logged in and so on. Rack provides a lot of programs related to the development of middleware.

(a) Rack::CommonLogger

Any Web application must log information, Rack::CommonLogger forwards every request to a given app and logs a line in the Apache common log format to the logger or rack.errors by default. You can refer to the specific format of the log at: http: //httpd.apache.org/docs/1.3/logs.html#common

The logger must be a legal error stream, it must:

• respond to the puts, write and flush method

• We should be able to use a parameter called puts, as long as the parameter responds to to_s

• We should be able to use a parameter called write, as long as this parameter is String

• We should be able to call without parameters to flush, to ensure the information is indeed being written to the log

Standard output is in line with the conditions:

use Rack::CommonLogger, $stderr

If no error stream is provided, then Rack::CommonLogger uses rack.errors from the environment variable to log errors.

(b) Rack::Lint

Rack::Lint - Ensure the app obeys the Rack specification. This specification aims to formalize the Rack protocol. You can (and should) use Rack::Lint to enforce it. When you develop middleware, be sure to add a Lint before and after to catch all mistakes.

Rack::Lint validates your application and the requests and responses according to the Rack spec.

Rack two types of inspection, a static check, call methods in its implementation; the other is dynamically check each method in its implementation.

If the check does not pass then Rack::Lint::LintError is thrown. It is easy to write code to define a new assert methods.

```
class LintError < RuntimeError; end
module Assertion
  def assert(message, &block)
    unless block.call
      raise LintError, message
    end
  end
end
include Assertion
```

Methods assert a message argument and a block of code, the code block is executed if the result is false (or nil), then throws LintError exception.

Check call behavior

```
def call(env=nil)
  dup._call(env)
end
```

```
def _call(env)
  ## It takes exactly one argument, the *environment*
  assert("No env given") { env }
  check_env env
  env['rack.input'] = InputWrapper.new(env['rack.input'])
  env['rack.errors'] = ErrorWrapper.new(env['rack.errors'])
  ## and returns an Array of exactly three values:
  status, headers, @body = @app.call(env)
  ## The *status*,
  check_status status
  ## the *headers*,
  check_headers headers
  ## and the *body*.
  check_content_type status, headers
  check_content_length status, headers, env
  [status, headers, self]
end
```

The code above checks the following:

• The environment (env) of the Request meets specifications

• Response status header field, content type and content length of the environment

Check the environment

```
def check_env(env)
  ## The environment must be an instance of Hash that includes
  ## CGI-like headers. The application is free to modify the
  ## environment.
  assert("env #{env.inspect} is not a Hash, but #{env.class}") {
    env.kind_of? Hash
```

```
  }
Environment must be a Hash.
  if session = env['rack.session']
    ##    store(key, value)         (aliased as []=);
    assert("session #{session.inspect} must respond to store and []=") {
      session.respond_to?(:store) && session.respond_to?(:[]=)
    }
    ##    fetch(key, default = nil) (aliased as []);
    assert("session #{session.inspect} must respond to fetch and []") {
      session.respond_to?(:fetch) && session.respond_to?(:[])
    }

  ##      delete(key);
    assert("session #{session.inspect} must respond to delete") {
      session.respond_to?(:delete)
    }
    ##    clear;
    assert("session #{session.inspect} must respond to clear") {
      session.respond_to?(:clear)
    }
end
```

If the environment includes rack.session keyword, it should save the request of the session, it must be a Hash like object, that is, if the object is called the session, then it must respond to the following methods:

• store methods and aliases [] = method, which can be used session.store (key, value) or a session [key] = value stored keyword, value pairs.

• fetch method and aliases [], which can be used session.fetch (key, default = nil) or session [key] for key corresponding value.

• delete method, that can be used session.delete (key) key to delete the corresponding key, value pairs.

• clear, that can clear all entries session.clear

Obviously, session is not necessarily Hash type, so as long as it responds to the required methods we are ok.

```
## <tt>rack.logger</tt>:: A common object interface for logging messages.
##        The object must implement:
if logger = env['rack.logger']
  ##      info(message, &block)
  assert("logger #{logger.inspect} must respond to info") {
    logger.respond_to?(:info)
  }

  ## debug(message, &block)
  assert("logger #{logger.inspect} must respond to debug") {
    logger.respond_to?(:debug)
  }

  ##      warn(message, &block)
  assert("logger #{logger.inspect} must respond to warn") {
    logger.respond_to?(:warn)
  }
  ## error(message, &block)
  assert("logger #{logger.inspect} must respond to error") {
    logger.respond_to?(:error)
  }
  ##      fatal(message, &block)
  assert("logger #{logger.inspect} must respond to fatal") {
    logger.respond_to?(:fatal)
  }
end
```

If the environment has rack.logger, the Rack can use it for logging. This object must respond to the following methods:

• info(message, &block)

• debug(message, &block)

• warn(message, &block)

• error(message, &block)

• fatal(message, &block)

This is a de facto standard log in many programming languages and frameworks .

```
%w[REQUEST_METHOD SERVER_NAME SERVER_PORT QUERY_STRING
        rack.version rack.input rack.errors
        rack.multithread rack.multiprocess rack.run_once].each do |header|
          assert("env missing required key #{header}") { env.include? header }
end
```

For Ruby application server, Rack demands at least the keyword env.

env can be divided into two categories:

• from the HTTP request, the first class of CGI. All caps, there is only one part (the middle of no use "."). It also contains two categories:

• "HTTP_" at the beginning of the keyword. These values are provided directly from the client HTTP request header fields, Web server before calling the Rack in these header fields must be added before "HTTP_", for example, if the request contains the Accept header field, it must be included in the environment "HTTP_ACCEPT". There are two heads exceptions: CONTENT_TYPE and CONTENT_LENGTH, they should not be added before "HTTP_".

```
## The environment must not contain the keys
## <tt>HTTP_CONTENT_TYPE</tt> or <tt>HTTP_CONTENT_LENGTH</tt>
## (use the versions without <tt>HTTP_</tt>).
%w[HTTP_CONTENT_TYPE HTTP_CONTENT_LENGTH].each { |header|
  assert("env contains #{header}, must use #{header[5,-1]}") {
    not env.include? header
  }
```

}

Other parts of the request from the user are, they do not "HTTP_" this prefix. REQUEST_METHOD, SERVER_NAME, SERVER_PORT, and QUERY_STRING these must be provided.

• not be obtained directly from the HTTP request message, and in general lower case. At least two parts, separated by "." Separated. It also includes two categories:

◦ Rack reserves, the prefix is "rack.". One rack.version, rack.input, rack.errors, rack.multithread, rack.multiprocess and rack.run_once these keywords must be provided.

◦ Web server's own use, the prefix must not be "rack.". For example, the server may use thin.xxx in such a situation.

Necessary for these keywords corresponding value, a series of specifications:

* <tt>rack.version</tt> must be an array of Integers.

```
  assert("rack.version must be an Array, was #{env["rack.version"].class}") {
    env["rack.version"].kind_of? Array
  }
  ## * <tt>rack.url_scheme</tt> must either be +http+ or +https+.
  assert("rack.url_scheme unknown: #{env["rack.url_scheme"].inspect}") {
    %w[http https].include? env["rack.url_scheme"]
  }
  ## * There must be a valid input stream in <tt>rack.input</tt>.
  check_input env["rack.input"]
  ## * There must be a valid error stream in <tt>rack.errors</tt>.
  check_error env["rack.errors"]
```

rack.version value must be an array, such as [1, 0], [1, 1], respectively Rack1.0 and Rack1.1; rack.url_scheme value must be http or https; rack.input and rack.errors the corresponding value must be a legal input and error stream and specific requirements will be discussed later.

* The <tt>REQUEST_METHOD</tt> must be a valid token.

```
assert("REQUEST_METHOD unknown: #{env["REQUEST_METHOD"]}") {
  env["REQUEST_METHOD"] =~ /\A[0-9A-Za-z!\#$%&'*+.^_`|~-]+\z/
}
```

* The <tt>SCRIPT_NAME</tt>, if non-empty, must start with <tt>/</tt>

```
assert("SCRIPT_NAME must start with /") {
  !env.include?("SCRIPT_NAME") || env["SCRIPT_NAME"] == "" || env["SCRIPT_NAME"] =~ /\A\//
}
## * The <tt>PATH_INFO</tt>, if non-empty, must start with <tt>/</tt>
assert("PATH_INFO must start with /") {
  !env.include?("PATH_INFO") || env["PATH_INFO"] == "" || env["PATH_INFO"] =~ /\A\//
}
## * 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/
}
## * One of <tt>SCRIPT_NAME</tt> or <tt>PATH_INFO</tt> must be
## set. <tt>PATH_INFO</tt> should be <tt>/</tt> if ##       <tt>SCRIPT_NAME</tt> is empty.
assert("One of SCRIPT_NAME or PATH_INFO must be set(make PATH_INFO '/' if SCRIPT_NAME is empty)") {
  env["SCRIPT_NAME"] || env["PATH_INFO"]
} ##     <tt>SCRIPT_NAME</tt> never should be <tt>/</tt>, but instead be empty.
assert("SCRIPT_NAME cannot be '/', make it '' and PATH_INFO '/'") {
  env["SCRIPT_NAME"] != "/"
}
```

Regular expression, "\ A" said the beginning of the string, "\ z", said the end of the string. Pointed out above, these statements separately:

• REQUEST_METHOD is a string, the value can not be blank, matching the regular expressions.
• SCRIPT_NAME and PATH_INFO the two must be at least one.

  ◦ SCRIPT_NAME if so, its value can be empty string, it can be a slash / start of string.

  ◦ the requirements of the PATH_INFO and SCRIPT_NAME exactly the same.

But SCRIPT_NAME represents the name of an application, so can not be a single slash, if necessary, an empty string can SCRIPT_NAME, PATH_INFO to make a single slash.

• CONTENT_LENGTH not required. However, if any, must be a string, which should be fully all figures.

Now go back to the corresponding input stream rack.input what conditions to be met:

```
## === The Input Stream
##
## The input stream is an IO-like object which contains the raw HTTP
## POST data.
def check_input(input)
  ## When applicable, its external encoding must be "ASCII-8BIT" and it
  ## must be opened in binary mode, for Ruby 1.9 compatibility.
  assert("rack.input #{input} does not have ASCII-8BIT as its external encoding") {
    input.external_encoding.name == "ASCII-8BIT"
  } if input.respond_to?(:external_encoding)
  assert("rack.input #{input} is not opened in binary mode") {
    input.binmode?
  } if input.respond_to?(:binmode?)
  ## The input stream must respond to +gets+, +each+, +read+ and +rewind+.
  [:gets, :each, :read, :rewind].each { |method|
    assert("rack.input #{input} does not respond to ##{method}") {
      input.respond_to? method
    }
  }
end
```

Input stream will contain the original HTTP POST data. If appropriate, it should be encoded using the ASCII-8BIT external (external_encoding, use binary mode. External code is a concept that appears Ruby 1.9, which indicates that the text stored in a file encoding. (With the corresponding internal coding is used Ruby said in the text in the code). binary mode to guarantee read the original data.

In addition to code requirements, the input stream must be able to respond: gets, each, read, rewind it four ways. rack.errors flow requirements corresponding to slightly more errors, to respond to the puts, write and flush method can

```
## === The Error Stream
```

```
def check_error(error)
  ## The error stream must respond to +puts+, +write+ and +flush+.
  [:puts, :write, :flush].each { |method|
    assert("rack.error #{error} does not respond to ##{method}") {
      error.respond_to? method
    }
  }
end
```

Checking the status code

```
## === The Status
def check_status(status)
  ## This is an HTTP status. When parsed as integer (+to_i+), it must be
  ## greater than or equal to 100.
  assert("Status must be >=100 seen as integer") { status.to_i >= 100 }
end
```

Response status code returned in the message must be converted to an integer using to_i, this integer must be greater than 100

Checking the response header

Response header must be able to respond to each method, and each generates a keyword and a corresponding value. Hash meets this condition, we talked earlier about Rack::Utils::HeaderHash used as a Rack middleware for response headers.

```
## === The Headers
def check_headers(header)
  ## The header must respond to +each+, and yield values of key and value.
  assert("headers object should respond to #each, but doesn't (got #{header.class} as headers)") {
    header.respond_to? :each
  }
  header.each { |key, value|
    ## The header keys must be Strings.
```

```
    assert("header key must be a string, was #{key.class}") { key.kind_of? String }
    ## The header must not contain a +Status+ key,
    assert("header must not contain Status") { key.downcase != "status" }
    ## contain keys with <tt>:</tt> or newlines in their name,
    assert("header names must not contain : or \\n") { key !~ /[:\n]/ }
    ## contain keys names that end in <tt>-</tt> or <tt>_</tt>,
    assert("header names must not end in - or _") { key !~ /[-_]\z/ }
    ## but only contain keys that consist of
    ## letters, digits, <tt>_</tt> or <tt>-</tt> and start with a letter.
    assert("invalid header name: #{key}") { key =~ /\A[a-zA-Z][a-zA-Z0-9_-]*\z/ }
    ## The values of the header must be Strings,
    assert("a header value must be a String, but the value of "+"'#{key}' is a #{value.class}") {
      value.kind_of? String
    }
    ## consisting of lines (for multiple header values, e.g. multiple
    ## <tt>Set-Cookie</tt> values) seperated by "\n".
    value.split("\n").each { |item|
      ## The lines must not contain characters below 037.
      assert("invalid header value #{key}: #{item.inspect}") { item !~ /[\000-\037]/}
    }
  }
end
```

For each keyword / value pairs, that is, each response header fields and their values, respectively, the following requirements:

- key terms of the keywords
  - key must be a string, not Symbol
  - key can not be a "Status" in this string
  - key can not include the ":" and "\ n" these two characters
  - key must begin with a letter followed by more letters, numbers, "-" or "_", but not to "-" or "_" character at the end of the two.

• the value of their (value) for

  ◦ value must be a string

  ◦ value value value can not contain less than 037 characters, control characters

Checking the content type

```
## === The Content-Type
def check_content_type(status, headers)
  headers.each { |key, value|
    ## There must be a <tt>Content-Type</tt>, except when the
    ## +Status+ is 1xx, 204 or 304, in which case there must be none given.
    if key.downcase == "content-type"
      assert("Content-Type header found in #{status} response, not allowed") {
        not Rack::Utils::STATUS_WITH_NO_ENTITY_BODY.include? status.to_i
      }
      return
    end
  }
  assert("No Content-Type header found") {
    Rack::Utils::STATUS_WITH_NO_ENTITY_BODY.include? status.to_i
  }
end
```

headers.each checks each head, if you find 'Content-Type', two situations may occur:

• If you assert that the status code is not 1xx, 204 or 304 of this condition does not hold - that they are really 1xx, 204 or 304 in a middle of a state, an exception is thrown

• If the status code is not 1xx, 204 or 304 of this condition is true, then the code behind the implementation of assert, returns,

The code will not execute if does not find Content-Type, the program will do that after each assert, this request must state is the 1xx, 204 or 304. Combination of the two words, status code 1xx, 204 or 304 response must not have Content-Type header, which all of his status must have Content-Type header.

Checking the content length

```
## === The Content-Length
def check_content_length(status, headers, env)
  headers.each { |key, value|
    if key.downcase == 'content-length'
       ## There must not be a <tt>Content-Length</tt> header when the +Status+ is 1xx, 204 or
304.
      assert("Content-Length header found in #{status} response, not allowed") {
        not Rack::Utils::STATUS_WITH_NO_ENTITY_BODY.include? status.to_i
      }
      ...
      ...
      ...
      return
    end
  }
end
```

Beginning of the code as part and Content-Type: status of that 1xx, 204 or 304 response, it must not set Content-Length.

If not the state, and indeed there is Content-Length, Content-Length must check whether the value of the actual content of the same length. The above code is omitted, the code below

```
bytes = 0
string_body = true
if @body.respond_to?(:to_ary)
  @body.each do |part|
    unless part.kind_of?(String)
      string_body = false
      break
    end
    bytes += Rack::Utils.bytesize(part)
  end
```

```
  if env["REQUEST_METHOD"] == "HEAD"
    assert("Response body was given for HEAD request, but should be empty") { bytes == 0 }
  else
    if string_body
          assert("Content-Length header was #{value}, but should be #{bytes}") { value == bytes.to_s }
    end
  end
end
```

Code to determine response body is only able to convert into an array or an array of situations. First calculate the length of this body, and the body completely to the string

```
@body.each do |part|
  unless part.kind_of?(String)
    string_body = false
    break
  end
  bytes += Rack::Utils.bytesize(part)
end
```

Next inspection when the request method is GET, the response body length must be 0. If not, GET and the entire response body is indeed a string, then the Content-Length response header value (value) must be equal to the actual length of response body.

Checking each

call method is a static check, each method is on the dynamic response of the body check.

```
## === The Body
def each
  @closed = false
  ## The Body must respond to +each+
  @body.each do |part|
    ## and must only yield String values.
    assert("Body yielded non-string value #{part.inspect}") { part.kind_of? String }
    yield part
```

```
  end
end
```

First check the response body must be able to respond to each method, and each must produce a string (part).

```
if @body.respond_to?(:to_path)
  assert("The file identified by body.to_path does not exist") { ::File.exist? @body.to_path }
end
```

And if the response body responds to to_path method, the return value should be a path name, and this path name of the file should exist.

Here is some code snippet showing how to use rspec with Rack::Lint:

```
  specify "notices status errors" do
    lambda {
      Rack::Lint.new(lambda {|env| ["cc", {}, ""]}).call(env({}))
    }.should.raise(Rack::Lint::LintError).message.should.match(/must be >= 100 seen
    as integer/)

    lambda {
      Rack::Lint.new(lambda {|env| [42, {}, ""]}).call(env({}))
    }.should.raise(Rack::Lint::LintError).
      message.should.match(/must be >= 100 seen as integer)
    end
```

(c) Rack::Reloader

Sometimes, when you make changes in your application, we wish that the framework can reload the modified code. For example, in the development process, we do not want to restart the Web server whenever we change the code. Because of this, Rail offers development and production environment. For example, we have a simple program, including the two files. One is the test-reloader.ru:

```
require 'simple'
```

```
run Simple.new
```

And the simple.rb:

```
class Simple
  def call(env)
    [200, {'Content-Type'=>'text/html'},["first"]]
  end
end
```

Now with rackup test-reloader.ru start the program. In the browser enter http://localhost:9292, the output will be: first. Then we modify the code to simple.rb the first to second. No matter how many times you refresh, the browser will always be display *first*, unless you exit and restart rackup.

Using Rack::Reloader is very simple, just add a line in the test-reloader.rb:

```
use Rack::Reloader
require 'simple'
run Simple.new
```

Restart rackup, but this time if the file simple.rb changes the output from first to second, the browser will immediately output the result of changes (may need to refresh the browser a few times, because a certain amount of time to reload - Here we can see how to configure this time. Browser may also cache the output so you may have to do shift+reload on the browser window).

The Rack::Reloader is very efficient, you can even use it in a staging environment to reload the source code.

```
def initialize(app, cooldown = 10, backend = Stat)
  @app = app
  @cooldown = cooldown
  @last = (Time.now - cooldown)
  @cache = {}
  @mtimes = {}
```

```
  extend backend
end
```

In the initialization Rack::Reloader space pieces, you can specify the time interval (cooldown), and a module backend. This module provides a rotation method to calculate all the loaded files and related information.

```
def call(env)
  if @cooldown and Time.now > @last + @cooldown
    if Thread.list.size > 1
      Thread.exclusive{ reload! }
    else
      reload!
    end
    @last = Time.now
  end
  @app.call(env)
end
```

Every time a request arrives Rack checks, only when the interval exceeds a set time, it is possible to do a reload. It also has more than one thread to determine whether there is currently (read.list.size > 1): If there is only one thread, then directly call reload!, Otherwise, a critical region in the implementation of reload!.

read.exclusive code execution in the critical region, the critical region for the entire Ruby process, in the critical region, all the existing threads will not be scheduled. Although it is not entirely correct, but you can generally think it makes Ruby Green Thread read all the other suspended operation.

```
def reload!(stderr = $stderr)
  rotation do |file, mtime|
    previous_mtime = @mtimes[file] ||= mtime
    safe_load(file, mtime, stderr) if mtime > previous_mtime
  end
end
```

rotation is provided by the Stat module, which provides all the files have been loaded and the corresponding time of last modification. If this file is the last time the last load before the last modified time, that is loaded and then modified, then the program calls safe_load perform real work load.

```
# A safe Kernel::load, issuing the hooks depending on the results
def safe_load(file, mtime, stderr = $stderr)
  load(file)
  stderr.puts "#{self.class}: reloaded `#{file}'"
  file
rescue LoadError, SyntaxError => ex
  stderr.puts ex
ensure
  @mtimes[file] = mtime
end
```

safe_load loading process to ensure that there's loads and grammatical errors will not throw an exception, while it ensures that the document load time of the last set to the current last modified.

Now let's look at Stat module to see how rotation is implemented.

```
def rotation
  files = [$0, *$LOADED_FEATURES].uniq
  paths = ['./', *$LOAD_PATH].uniq
  files.map{|file|
    next if file =~ /\.(so|bundle)$/ # cannot reload compiled files
    found, stat = figure_path(file, paths)
    next unless found && stat && mtime = stat.mtime
    @cache[file] = found
    yield(found, mtime)
  }.compact
end
```

rotation to obtain a current program name ($ 0) and all the files have been loaded ($ LOADED_FEATURES), Files saved to the variable. Then set the paths for all the loading path ($ LOAD_PATH). For each file, the program first to determine whether it is for the C library - they can not re-loaded (so is linux, and mac os x bundle is the following library file suffix.) Then ratation call figure_path method to find the file, if found the file, then use the file and its last modified time to call the rotation behind the block - we have reload! Program saw.figure_path methods are two cases:

• If the file is the file name given an absolute path, then directly call safe_stat (file)

• Otherwise, try all of the Ruby load path and file name File.join them up, again adjusting with safe_stat (file), until the real file to find the specific implementation date, see the Rack::Reloader's source code.

(d) Rack::Runtime

In the logs, performance evaluation and analysis process, we would like to know the processing time of a request, Runtime middleware calculate the time and put it on the X-Runtime header response. The code is self-explanatory.

```
class Runtime
  # Sets an "X-Runtime" response header, indicating the response
  # time of the request, in seconds
  #
  # You can put it right before the application to see the processing
  # time, or before all the other middlewares to include time for them,
  # too.
 def initialize(app, name = nil)
  @app = app
  @header_name = "X-Runtime"
  @header_name << "-#{name}" if name
 end


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

```
    request_time = Time.now - start_time

    if !headers.has_key?(@header_name)
      headers[@header_name] = "%0.6f" % request_time
    end
    [status, headers, body]
  end
end
```

One thing to note, call is not the only place to process the request, a lot of logic is often present in the body of *each* method implementation.

(e) Rack::Sendfile

Please note: This section is only used to understand mechanisms of SendFile, Rack::Sendfile implementation is not as described below. This section is incomplete, but also need to add lighttpd and apache. This section may be removed in subsequent versions. Web applications often need to handle large files, including images, PDF, Word, video and audio files for clients to download and display. If the file and the application logic does not have any relationship, you can use the proxy server (such as ngnix, lighttpd, apache, etc.) for file handling, bypassing Ruby application server completely.

But sometimes, you need to search based on user requests a specific file location URL, or you need to control access to the document. A typical example is a file can only be accessed by some users. This time the request must be after the proxy server to reach Ruby application servers, Web programs through the processing to the actual file transfer to the client.

The problem is, now responsible for Web application server reads the file, the contents of the file transmitted to the proxy server, proxy server then writes the contents of the file to the client. This obviously will cause CPU, memory, and wasting a lot of internal network resources.

To solve this problem, the vast majority of proxy servers support a mechanism of X-Sendfile, Ruby program only need to set the appropriate response headers, telling the proxy server where the file is located. When the proxy server in response header X-Sendfile found is set, then it will read directly under the path of the file contents of the file and send it to the client, since it can directly connect the client to read and write to files can greatly enhance transmission and performance, lower CPU and memory consumption.

Different proxy services have different X-Sendfile achieved, we take a look at some main requirements of the proxy server is to:

nginx Nginx to X-Sendfile called X-Accel-Redirect.

First, you must set

sendfile on;

Now suppose that the actual file system /files/images under a abc.jpg file. The first issue to consider is the /files/images directory can not be accessed directly by the user, otherwise, in accordance with the general set up nginx, nginx will deal directly with static files, we can not control the ruby application file transfer process. Therefore, we should /images directory to the internal directory, nginx's internal instructions to ensure this. The following is the corresponding configuration items:

```
location /images/ {
  internal;
  root /files;
}
```

Now, when the user requests http://www.somdomain.com/files/images/abc.jpg file, in accordance with our normal procedures of the configuration of Ruby, it will redirect the request to the Ruby application server. Ruby program receives the request, for treatment, should be included in the response header

X-Accel-Redirect: / images / abc.jpg

This connection nginx will set in the root (ie / files) and location (ie / images) get /files/images, and then read the actual file system /files/images/abc.jpg file and send it to client.

More common approach is to define a complete alias, so the user directory and the actual URL stored in the file directory structure can be completely different. For example, we want the user to request http://www.somdomain.com/images/abc.jpg can directly access the file. If the file is actually stored in the file system /files/images under, then you can set the following:

```
location /images/ {
  internal;
  alias /files/images/; # Note trailing slash
}
```

Note, alias the final slash is essential. Similarly, Ruby program only needs to set the response header:

X-Accel-Redirect: /images/abc.jpg

Nginx know /images corresponding to the path is the actual file system /files/images/, it will go to that directory to read the appropriate file.

4. Application Configuration & Routing Middleware

(a) Rack::Cascade

Rack::Cascade middleware to used to mount more than one application. It tries a request on several apps and returns the first response that is not 404 (or in a list of configurable status codes).

```
apps = [lambda {|env| [404, {}, ["File doesn't exists"]]},  lambda {|env| [200, {}, ["I'm ok"]]}]
use Rack::ContentLength
use Rack::ContentType
run Rack::Cascade.new(apps)
```

When you rackup this .ru file you will see the output "I'm ok" in the browser. What is the meaning of this? Consider we need a Rails application program embedded in a sinatra, consider a separate deal with file upload we want to achieve. Cache not want to load the entire Rails handling procedures, but the cache does not exist when the call is still able to Rails corresponding logic. Infinite possibilities, we will discuss some specific examples later.

(b) Rack::Lock

Some of the web framework Ruby or procedures within the process in the same multi-threaded parallel processing multiple requests, if the Web server also supports multi-threaded, then the rack.multithread env is set to true.

Some of the framework can not handle multiple threads, such as general Rails framework can only be single-threaded operation. Rack::Lock middleware will make the whole process a request for exclusive lock:

```
module Rack
  class Lock
    FLAG = 'rack.multithread'.freeze
    def initialize(app, lock = Mutex.new)
      @app, @lock = app, lock
    end
```

```
    def call(env)
      old, env[FLAG] = env[FLAG], false
      @lock.synchronize { @app.call(env) }
    ensure
      env[FLAG] = old
    end
  end
end
```

Before starting to process the request, Rack::Lock sets the rack.multithread value to false, then the synchronization block in the lock handle the entire request, the final recovery rack.multi-thread value. This ensures that the entire Ruby process can only handle one request at a time.

(c) Session Manager

Session we are talking about here refers to the server tracking users and which request belongs to which request in any given interaction. To accomplish this we must be able to determine the user's identity.

Since HTTP is stateless transaction, a request / response after the end of it and the next request / response has no relationship. In order to determine which request comes from a particular user.

HTTP did not consider issues related to how to identify a user, so people came up with a variety of techniques, including:

• Shove the HTTP header with the user identity information

• Use IP addresses to identify the customer

• Force users log on and use authentication to identify users

• Embed user's identity inside the URL

• Cookie, can be used to effectively protect the identity. Let's see how to use the Cookie to maintain the user's identity.

HTTP Cookies

When the user first visit Web application, server did not know any of the user information. In order to confirm the user's follow-up visit, the server sets a unique cookie to identify the user, and in response Set-Cookie header field value is set for this cookie. After the browser receives this response, save the Cookie in your cookie database. The next time, when users access the same

site, the browser selects the corresponding cookie value, uses it to set the Cookie request header field. Cookie can contain multiple name = value This keyword value pairs, for example:

id = "1234567"; name = "jack"; phone = "65452334"

Generally cookie has:

• domain: Cookie domain name.

• path: and the cookie path to the beginning of the relevant domain.

• secure: if it is safe, if it is set, then only when in https this cookie will be sent to the server.

• expiration: expiration time

• name: cookie name

• value: cookie value

Here is a simple program to set the cookie:

```
run lambda {|env| [200, {'Content-Type'=>'text/html',
'Set-Cookie'=>"id = 1234567\nname=jack\nphone=65452334"}, ['hello world!']]}
```

Run this program with rackup. Edit your /etc/hosts file to add a line makes example.com to point 127.0.0.1: 127.0.0.1 www.example.com open the browser, clear your cookie, enter http:/ /www.example.com:9292, and then look at the cookie.

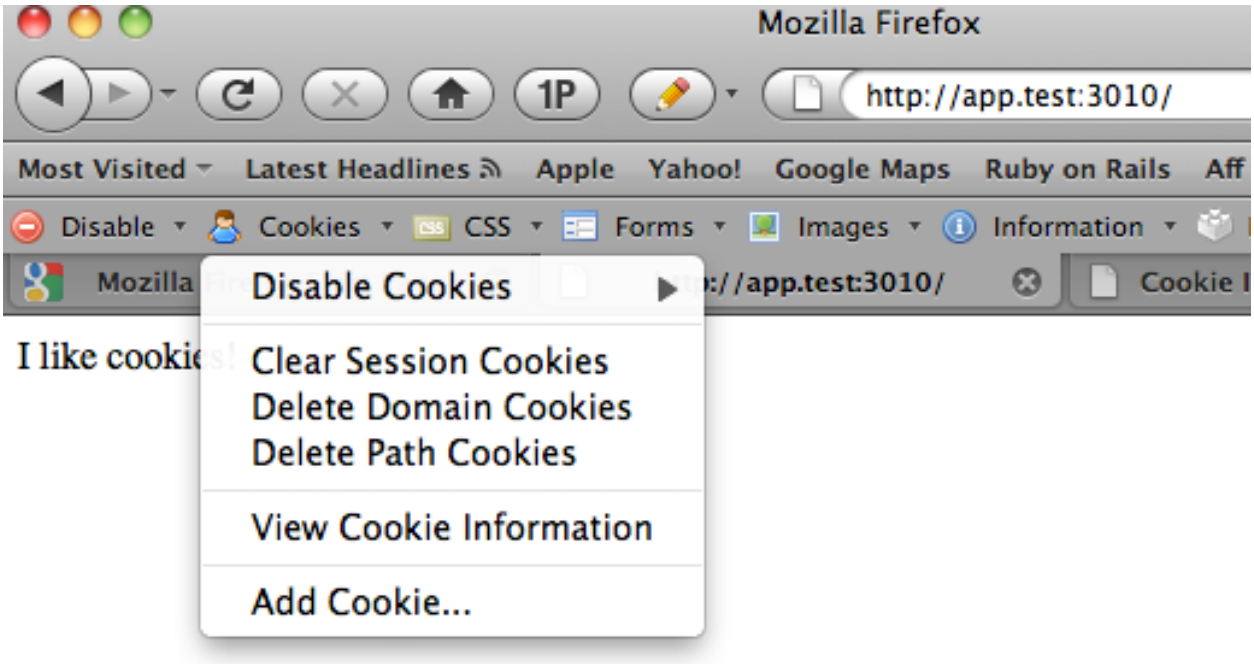
```
mbp2:~ bparanj$ cat /etc/hosts
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting.  Do not change this entry.
##
127.0.0.1       www.example.com example.com test.example.com app.test localhost
## 127.0.0.1    app.test localhost
255.255.255.255 broadcasthost
::1             localhost
fe80::1%lo0     localhost
```

Figure is based on the upper part of the domain name (domain) list of organizations, one cookie, and the lower half of the corresponding lists the name of the cookie (name), content (value), the host (host), path (path), sending conditions and expiration time (expiration).

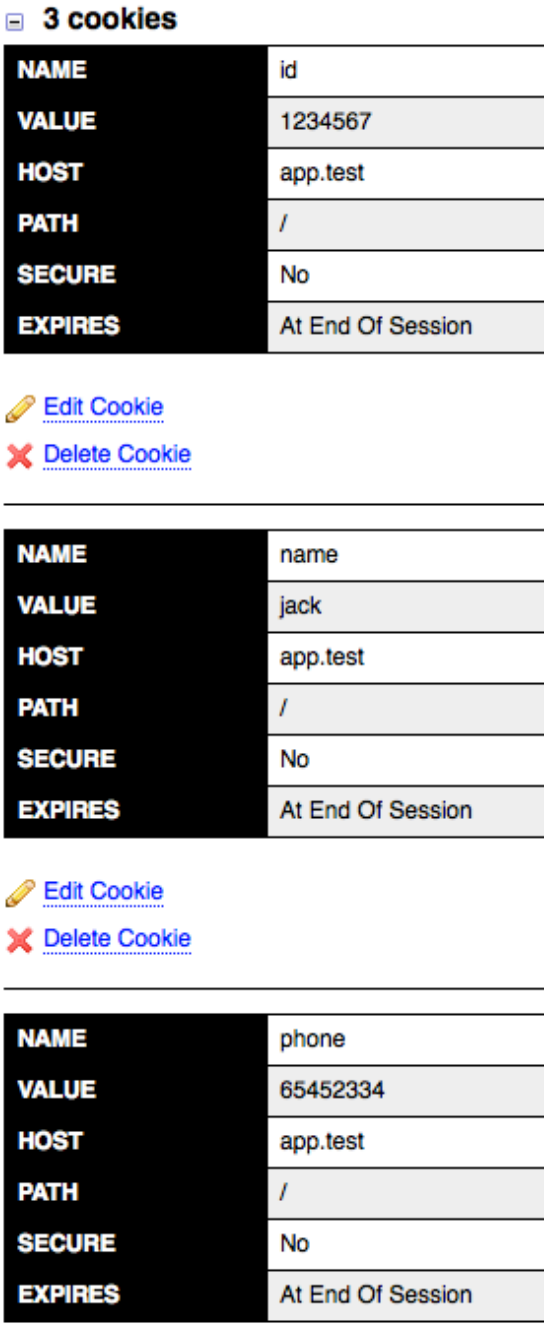
Our program is set using the three Set-Cookie cookie, between them with "\ n" separator.

We do not set the cookie expiration time the lower half, you can see is the end of the session expiration time. Cookie can be divided into two categories, the session cookie (session cookie) and persistent cookie (persistent cookie). This session is a browser open and close, that if you close your browser, reopen, then the previous session cookie will not exist. Persistent cookie to survive longer, even if closed, to re-open the browser will not be deleted. Clearly, we need to use persistent cookie to maintain the application session.

Click on "View Cookie Information" of the Web Developer toolbar in Firefox:



You can see three cookies has been set in the figure below.



Session cookie and persistent cookie is not essentially different, the only difference is that when they expire. If you set a cookie, a discard parameter, or that are not set not set Expire Max-Age parameter, it is a conversation session. Otherwise, is a persistent cookie.

Set Cookie property

There are two versions of the cookie specification, Version 0, respectively, and Version1 (RFC 2965). Commonly used version is Version 0, it came from Netscape, because Netscape cookie mechanism was first to introduce the client. We focus on the Version 0.

Server can be used in the following format to set cookie header in the response properties: Set-Cookie: name = value [; expires = date] [; path = path] [; domain = domain] [; secure]

Expires is not required. Setting determines whether this is a session cookie or a persistent cookie. And if there is value, then it determines the persistent cookie expiration time. Time zone must GMT: Weekday, DD-Mon-YY HH:: MM:: SS GMT DATE must be between the separator -. Here is an example of a set expires: Set-Cookie: foo = bar; expires = Wednesday, 09-Nov-99 23:12:40 GMT Ruby, you can meet this specification format by using the following snippet of code:

```
Time.now.gmtime.strftime("%a, %d-%b-%Y %H:%M:%S GMT")
```

Domain name is optional. Browser decides based on this value to determine whether to send a cookie to a domain name (domain) of the host or not. If the domain is example.com, then it can match www.example.com, blog.example.com so the host, but will not match www.abc.com.

If you do not set the domain, then it generates the default Set-Cookie response equal to that host. Note that a host of other domain names can not set the domain value. For example, if the response is generated in the host name wiki.example.com, it can not respond to the Set-Cookie domain is set to abc.com. Set up under the surface is a domain example:

Set-Cookie: foo = bar; domain = "example.com"

To better understand the domain to set the rules, we can do an experiment, edit the /etc/hosts file by adding the following line:

127.0.0.1 www.example.com example.com test.example.com

Thus, the three host names point to our local machine. Now write the following program:

```
run lambda {|env| [200, {'Content-Type'=>'text/html',
'Set-Cookie'=>"id = 1234567;domain=example.com"}, ['hello world!']]}
```

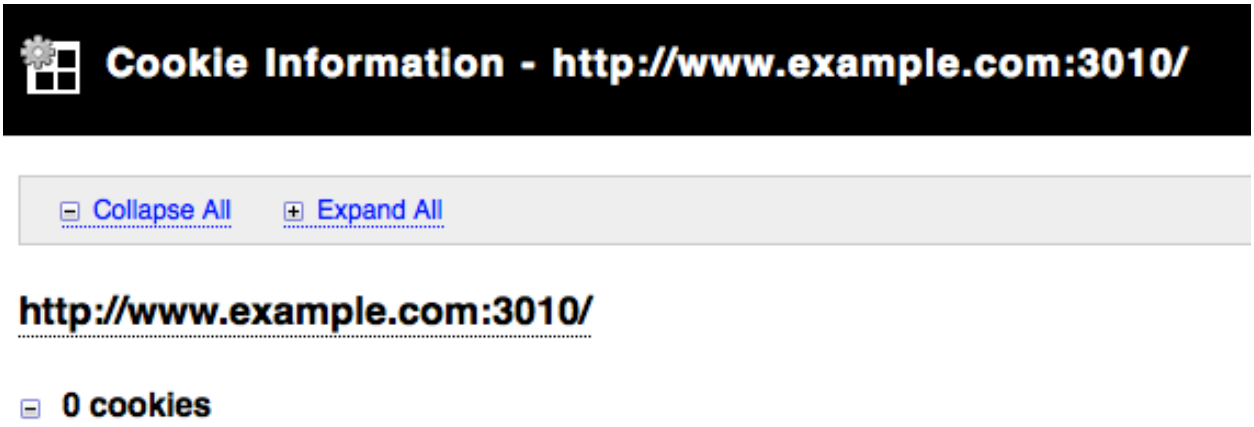
Whether we use www.example.com, example.com or test.example.com to access this program, you can see the cookie in the client's domain name is example.com. This means that any one of the same name (domain) within the host can set the cookie's domain name for their domain name. (Please test before each cookie is empty.)

Now modify the above program, set the domain to test.example.com:

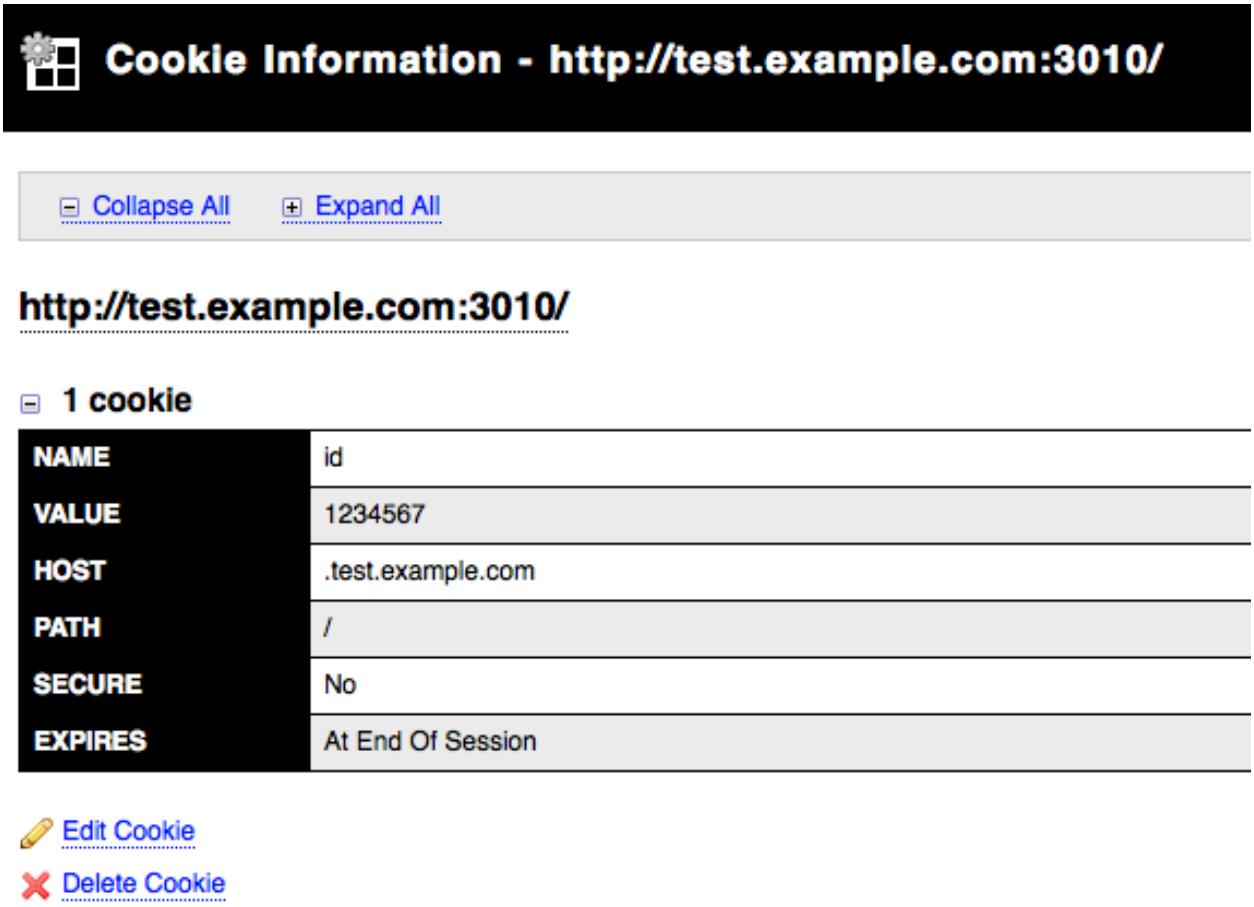
run lambda {|env| [200, {'Content-Type'=>'text/html',

'Set-Cookie'=>"id = 1234567;domain=test.example.com"}, ['hello world!']]}

Example.com and www.example.com are used to access the application, you will not see the client cookie. Only test.example.com to visit, before they can be set to test.example.com domain of the cookie.

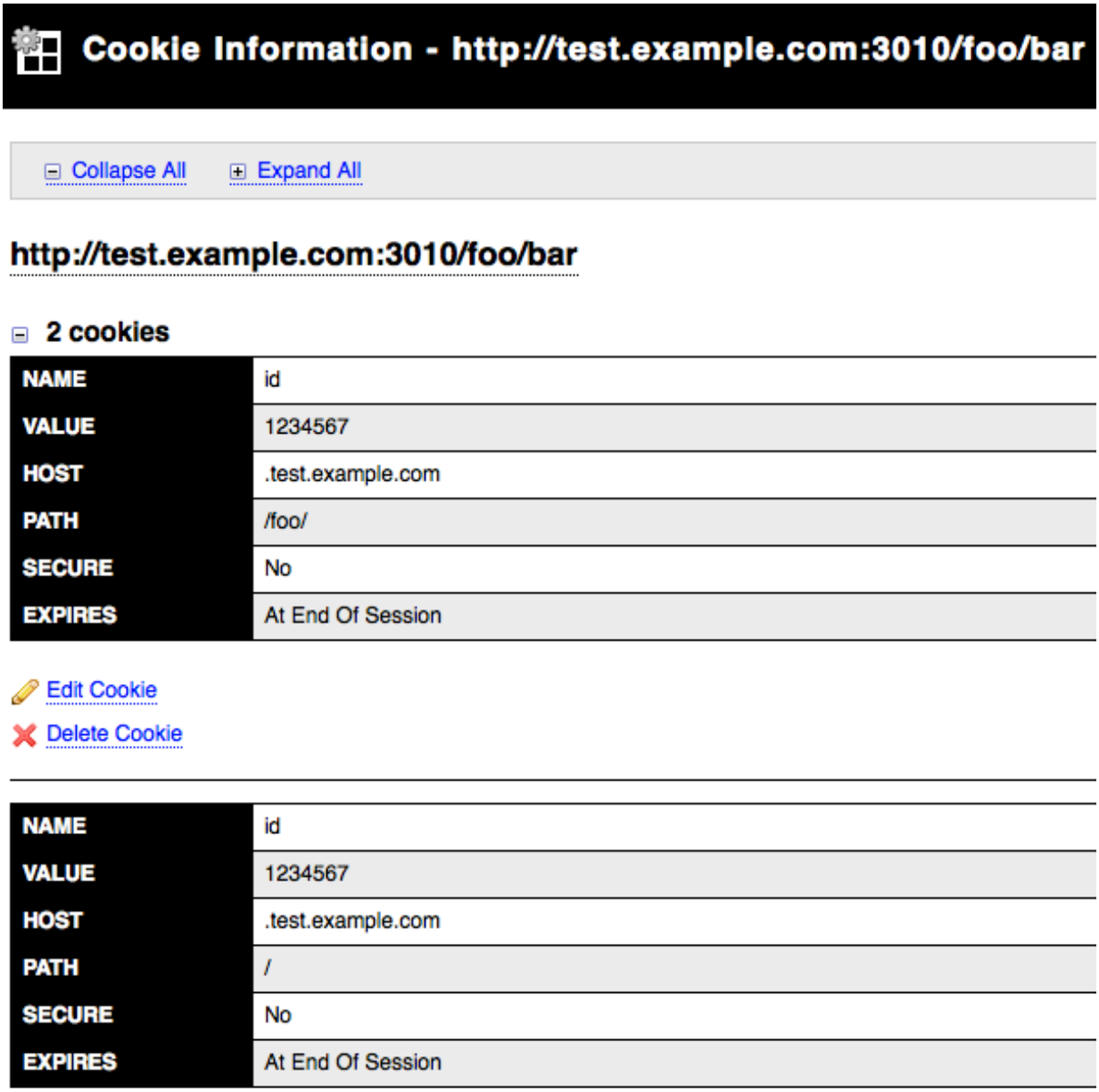


This time when domain2.ru is run you can see that when we hit the URL www.example.com, the cookie is not present whereas for test.example.com you can see the cookie as shown in the figure below:



Path is also optional. Path attributes can make your web site, part of the cookie and the association. If the path is set to "/", then it can match all documents within the domain. Other path values indicate a prefix, that is, if you set a path to /foo, then /foobar, /foo/sample.html so can match. If you do not set any Path value, then the default is Set-Cookie value generated URL.

We can test this, run the previous application, first enter the URL as http://www.example.com, you can see in the browser a cookie, its path is "/." Then enter http://www.example.com/foo, your client's Path or the cookie corresponding to "/." If you enter http://www.example.com/foo/bar, then the corresponding path was "/foo". This is shown in the following screenshot.



Secure is optional. If you set secue, for example: Set-Cookie: order_id = 519; secure so only you use https to access the site, the cookie will only be sent by the client to the server.

The client sends Cookie

Usually the client will save tens of thousands of cookie, it can not all the cookie sent to all sites. It is based on the host server settings, path, security options and compare the current visit of the URL, and then send the cookie to meet the conditions to the corresponding server.

The client's request will include a header field Cookie, like this:

Cookie: name1=value1 [;name2 =value2]

(d) Rack::Session::Cookie

Rack::Session::Cookie provides a simple cookie-based session management. The session is a Ruby Hash stored as base64 encoded marshalled data set to :key (default: rack.session). When the secret key is set, cookie data is checked for data integrity. When using this middleware, you can specify it in the env environment keyword, the default is rack.session. In addition, you can specify a security code that is :secret.

Let's look at a practical example:

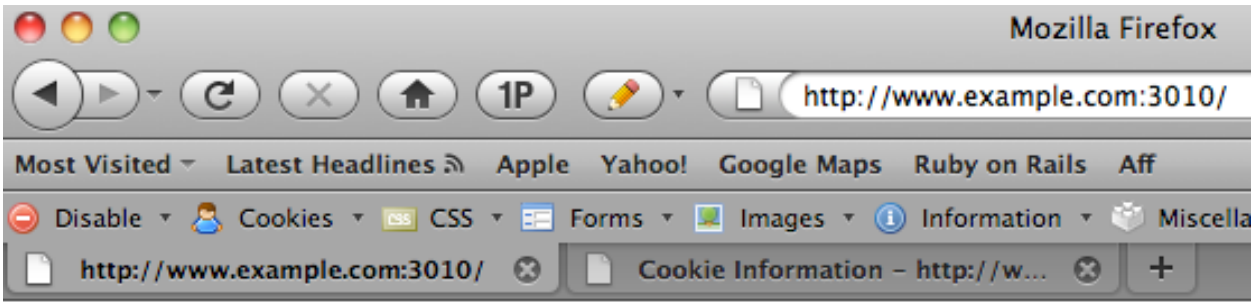
use Rack::Session::Cookie, :key => 'rack.session', :domain => 'example.com',

:path => '/', :expire_after => 2592000, :secret => 'any_secret_key'

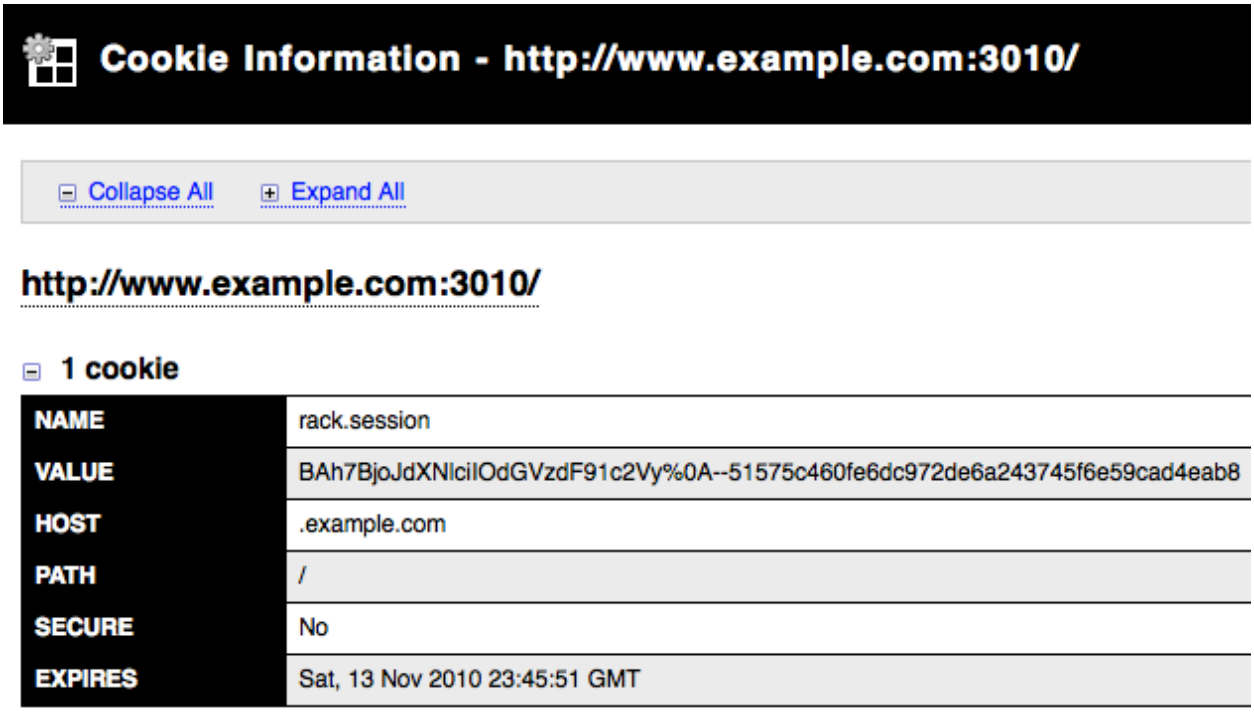
run lambda {|env| user = env['rack.session'][:user] env['rack.session'][:user] ||= 'test_user' [200, {"Content-Type" => "text/html"}, [user || "no current user"]]

}

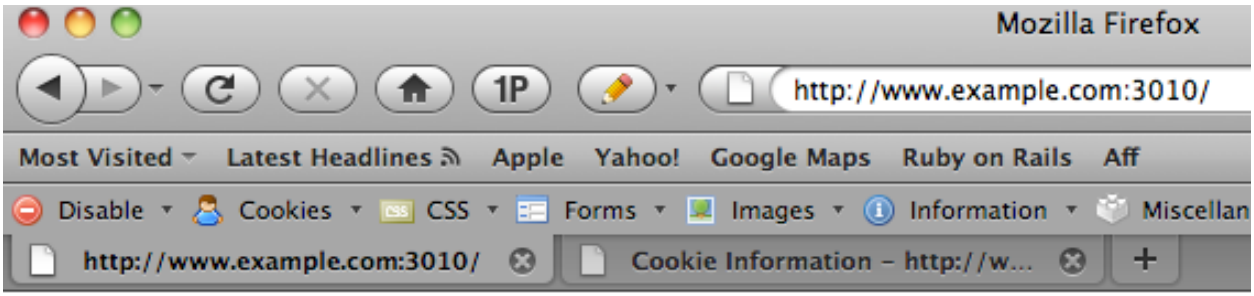
Your /etc/hosts file with a line and let us make it to example.com to point 127.0.0.1: 127.0.0.1 www.example.com open the browser, clear your cookie, enter http://www.example.com:3010, you will get no current user.



no current user



Program has been running, suggesting that env ['rack.session'] already exists, and is a Hash. Refresh your browser, you should get "test_user", which shows that we set the env ['rack.session']



test_user

Should be keyword :user value. This time, if you look at the local browser cookie, you can see the emergence of a ". Example.com" in the cookie.

If you clear your browser again, cookie, it will still be no current user, because the server can not get session information from the cookie. Let's look at the concrete implementation:

```
def initialize(app, options={})
  @app = app
  @key = options[:key] || "rack.session"
```

```
  @secret = options[:secret]
  @default_options = {:domain => nil, :path => "/", :expire_after => nil}.merge(options)
end
```

Use Rack::Session::Cookie time, allows you to set the following options:

:key key refers to the session in the cookie name (default is rack.session). That is, if the key is rack.session, then the last Set-Cookie header field is as follows: Set-Cookie: rack.session =.....

[; Expires = date] [; path = path] [; domain = domain] [; secure]

:secret allows you to set a security code :secret, used to encrypt your cookie data

:domain domain name

:path Path

:expire_after cookie is valid

The specific meaning of these options has been discussed in detail before. Let's take a look at the call where this middleware in in action.

```
def call(env)
  load_session(env)
  status, headers, body = @app.call(env)
  commit_session(env, status, headers, body)
end
```

In Rack::Session::Cookie the work can be divided into three stages:

1. Read from the request of the session cookie data, and set env [rack.session] the corresponding value.

2. Request Process - that is our Rack application may change the session content.

3. Request after the session finished with the data (ie, env [rack.session]) to write cookie, and set the response Set-Cookie header.

We are backwards, and the first session to see how the cookie data is converted to:

```
def commit_session(env, status, headers, body)
  session_data = Marshal.dump(env["rack.session"])
  session_data = [session_data].pack("m*")
```

```
  if @secret
    session_data = "#{session_data}--#{generate_hmac(session_data)}"
  end
  if session_data.size > (4096 - @key.size)
      env["rack.errors"].puts("Warning! Rack::Session::Cookie data size exceeds 4K. Content dropped.")
  else
    options = env["rack.session.options"]
    cookie = Hash.new
    cookie[:value] = session_data
    cookie[:expires] = Time.now + options[:expire_after] unless options[:expire_after].nil?
    Utils.set_cookie_header!(headers, @key, cookie.merge(options))
  end
  [status, headers, body]
end
```

When the procedure to this point when, env ['rack.session'] which already contains a session object, which is a Hash.

Line 2 to the session dump to a string session_data, line 2 pack ("m *") then all the Base64 encoded data.

5-8 line, if you set a security code, then generate_hmac method will increase based on your data in a password and session ssl encrypted hash and the encrypted data and previously obtained data with the "-" connected . The purpose of this session is the time to read the raw data and encrypted according to data validation, we will see this method in the load_session verification process.

9-11 is a special case, session_data data can not exceed 4k, otherwise there is no cookie data is written. By the way, session data that should be kept very small, only the primitive data types such as integers, strings, etc., otherwise the program will affect your performance. The behavior is nothing to subtract 9 @ key length, because the final data is written inside cookie "@ key = session_data" like that.

12-16 rows of data written to the session cookie. Here is a cookie Hash, its value is calculated at a few steps in front of the session_data, if you set the expiration time, then it is interpreted as the number of seconds from now. Last commit_session with Utils.set_cookie_header! This cookie is written to the response hash table header. Specifically, how to write, we go back into the discussion.

16 lines when writing, cookie also incorporates the values from the rack.session.options, the Hash is actually a middleware initialization of those parameters, including: domain,: path: and expire_after. (load_session below the line 18)

Now we can easily understand the work load the cookie, and it is basically a reverse operation commit_session

```
def load_session(env)
  request = Rack::Request.new(env)
  session_data = request.cookies[@key]

  if @secret && session_data
    session_data, digest = session_data.split("--")
    session_data = nil unless digest == generate_hmac(session_data)
  end
  begin session_data = session_data.unpack("m*").first
    session_data = Marshal.load(session_data)
    env["rack.session"] = session_data
  rescue
    env["rack.session"] = Hash.new
  end
  env["rack.session.options"] = @default_options.dup
end
```

1-2 lines to read session data from the request. Note that only the keyword for the @key out of the cookie. the process of how to parse cookie request relates to the relevant agreements cookie will be described in detail later.

If you set the security code secret, then decrypt 5-8, in front of commit_session the process, we use - the session of the raw data and encrypted data connection up. So here were the first to get the original data and the encrypted data to the session_data and digest, and compared. Only when:

digest == generate_hmac (session_data)

Condition holds, we can consider this session_data is legal, otherwise session_data will be set to nil-leading 15 lines to be executed, session will be an empty hash.

Line 11 using the previously commit_session pack decode Base64 encoded data, and then reload the original 12 lines of the Hash Table, the last 13 lines set to the env of rack.session keywords.

Memories commit_session line 16, cookie written from before the merger option rack.session.options

```
12      options = env["rack.session.options"]
16      Utils.set_cookie_header!(headers, @key, cookie.merge(options))
```

The data is from the initialization rack.session.options Rack::Session::Cookie middleware parameters.

Set cookie header

Written response to the real head of the cookie is Utils's set_cookie_header! Method. set_cookie_header! methods can be divided into two main parts. Hash value for the first dealing with the situation:

```
def set_cookie_header!(header, key, value)
  case value
  when Hash
    domain = "; domain=" + value[:domain] if value[:domain]
    path = "; path="      + value[:path]  if value[:path]
    # According to RFC 2109, we need dashes here. # N.B.: cgi.rb uses spaces...
    expires = "; expires=" + value[:expires].clone.gmtime.
    strftime("%a, %d-%b-%Y %H:%M:%S GMT") if value[:expires]
    secure = "; secure" if value[:secure] httponly = "; HttpOnly" if value[:httponly]
    value = value[:value]
  end
```

Rack::Session::Cookie middleware called set_cookie_header! method when, value is a Hash, which contains a variety of cookie and session options. According to earlier discussion that described the cookie-related specifications, code the following features:

• If the value [: domain] option exists, the cookie's domain property "; domain = value [: domain]"

• If the value [: path] option exists, the cookie's path attribute "; path = value [: path]"

• If the value [: expires] option exists, the cookie's expires attribute "; expires = value [: expires] is converted to GMT time format of the value"

• If the value [: secure] option exists, the cookie's secure property "; secure" (remember this is a boolen cookie attribute value)

• If the value [: httponly] option exists, the cookie's secure property "; HttpOnly" (HttpOnly cookie is a security-related options, not all browsers support)

• the hash value obtained from the key corresponding to the real value, that value [: value], and set it as the value of the variable value

Next task is to really set the Set-Cookie response header field value:

```
value = [value] unless Array === value
cookie = escape(key) + "=" +
value.map { |v| escape v }.join("&") + "#{domain}#{path}#{expires}#{secure}#{httponly}"

case header["Set-Cookie"]
when Array
  header["Set-Cookie"] << cookie
when String
  header["Set-Cookie"] = [header["Set-Cookie"], cookie]
when nil
  header["Set-Cookie"] = cookie
end
nil
```

If the value includes multiple values, use the "&" symbol to connect them together, then add all the cookie property at the back, we get a complete cookie value, the form:

rack.session =.....; domain =....; path =.....; expires =...; secure; HttpOnly final code already exists to determine whether the header Set-Cookie value: If so, header [Set-

Cookie] added into the array contains more than one cookie, otherwise, directly set to the current cookie. Good memory reader may notice that earlier, we have talked about have all the header values must be character String, including the Set-Cookie check:

```
## === The Headers
def check_headers(header)
  ..... header.each { |key, value|
    ........
    ## The values of the header must be Strings,
    assert("a header value must be a String, but the value of " +
     "'#{key}' is a #{value.class}") { value.kind_of? String } ## consisting of lines (for multiple header values, e.g. multiple ## <tt>Set-Cookie</tt> values) seperated by "\n". value.split("\n").each { |item|
    ## The lines must not contain characters below 037.
    assert("invalid header value #{key}: #{item.inspect}") {
      item !~ /[\000-\037]/
    }
  }
end
```

If the number of cookie, then the corresponding number of Set-Cookie cookie should be used "\n" separately, without is an array. Indeed, if the header is an ordinary Hash, then check the above error occurs. However, some intermediate pieces (We have already seen) will use a HeaderHash, it's each to achieve the following:

```
def each
  super do |k, v|
    yield(k, v.respond_to?(:to_ary) ? v.to_ary.join("\n") : v)
  end
end
```

If HeaderHash a certain value in the array, then this will first of all the members of this array with the "\n" connection into a string. This fits well with Rack::Lint requirements. So we write our own middleware, it should not make use of HeaderHash Hash processing response headers.

(e) ID Session

Rack::Session::Cookie provide a direct deposit in the Cookie Session of the method. Session when the data written in response to cookie stored in the back to the client, the client requested to resubmit the data when put back to the server.

This approach has some problems. First, if the session data contained too much, because every time a request / response are related to the object and the sequence of loading, the performance of the system will cause a relatively large impact. On the other hand, these data may be stored in the client and network security risks caused by the process. So, in general, we do not advocate in the session to save a lot of Ruby objects and data. The most common practice is to keep only one user ID.

Rack::Session::Abstract::ID class provides a simple framework, you can use it to id-based session management. Cookie of the session data includes only a simple id. You can override some parts of the framework to implement your own session management middleware. Default default parameter ID abstract middleware options include:

```
DEFAULT_OPTIONS = {
  :path =>        '/',
  :domain =>    nil,
  :expire_after => nil,
  :secure =>      false,
  :httponly =>  true,
  :defer =>       false,
  :renew =>      false,
  :sidbits =>     128
}
```

In addition to the three final surface, the other options we should have all the more familiar. Meaning of the remaining three options are:

• defer defer if set to true, then the response header will not set the cookie (not yet know what use)

• renew, if you set this option to true, then the specific implementation of session management should not be the original request sent by the client session_id, but each time generate a new session_id, and the session data corresponding to the original session_id and corresponds to the new

session_id. Note: renew higher priority than defer, that is, even if defer is set to true, as long as the set renew to true, then the cookie will be written to the response header.

• sidbits: session_id length of the resulting number of bit. ID class provides a practical generate_sid method for your specific implementation uses:

```
def generate_sid
  "%0#{@default_options[:sidbits] / 4}x" %
  rand(2**@default_options[:sidbits] - 1)
end
```

Generate a random session id

Method which uses rand to generate a random string of 16 hex. Of course, you can own the other

Write a method to generate foreign session_id. In addition, and Rack::Session::Cookie, like, you can specify the session initialize the name of the cookie said:

```
def initialize(app, options={})
  @app = app
  @key = options[:key] || "rack.session"
  @default_options = self.class::DEFAULT_OPTIONS.merge(options)
end
```

Class method ID main methods include:

• call
• load_session
• commit_session
• get_session
• set_session

Implementation of *call*:

```
def call(env)
  context(env)
end
def context(env, app=@app)
  load_session(env) status, headers, body = app.call(env)
  commit_session(env, status, headers, body)
end
```

In front of the main branches and we analyzed the Rack::Session::Cookie is not much difference between the session data is loaded, processing the request, the session data submitted by the three steps. load_session general and specific implementations Rack::Session::Cookie's load_session the same:

```
def load_session(env)
  request = Rack::Request.new(env)
  session_id = request.cookies[@key]
  begin
    session_id, session = get_session(env, session_id)
    env['rack.session'] = session
  rescue
    env['rack.session'] = Hash.new
  end
  env['rack.session.options'] = @default_options.merge(:id => session_id)
end
```

Code for: Rack::Session::Abstract::ID of load_session method

The main difference is:

1. From request.cookies [@ key] to obtain the client save session_id. Because we are only in the session cookie stored in a data id values.

2. With this client saved session_id call get_session obtained services side of the session_id and the session and get_session is a non-implemented methods, the specific session implementation

should override this method to determine how the client cookie in the session_id to services side of the session_id and session.

```
def get_session (env, sid)
  raise '# get_session not implemented.'
end
```

3. Session_id together with the middleware server's default parameters are set to env ['rack.session.options'] value for client access to services behind the commit_session session_id and other uses.

commit_session general and specific implementations Rack::Session::Cookie's commit_session the same:

```
def commit_session(env, status, headers, body)
  session = env['rack.session']
  options = env['rack.session.options']
  session_id = options[:id]
  if not session_id = set_session(env, session_id, session, options)
        env["rack.errors"].puts("Warning! #{self.class.name} failed to save session. Content dropped.")
  elsif options[:defer] and not options[:renew]
    env["rack.errors"].puts( "Defering cookie for #{session_id}") if $VERBOSE
  else
    cookie = Hash.new
    cookie[:value] = session_id
    cookie[:expires] = Time.now + options[:expire_after] unless options[:expire_after].nil?
    Utils.set_cookie_header!(headers, @key, cookie.merge(options))
  end
  [status, headers, body]
end
```

The main difference is:

1. env ['rack.session.options'] [: id] get to the server session_id, its value is in the load_session set.

2. session_id side with this service call set_session obtained client session_id, session_id then be written to the client cookie. One exception, that set the defer option and renew option is not set - this time will not be written session_id cookie. And get_session as, set_session is a method not implemented, the need for specific middleware to override it:

```
def set_session(env, sid, session, options)
  raise '#set_session not implemented.'
end
```

Implementing specific middleware

Therefore, if we are to achieve a specific ID-based middleware, we can inherit the Rack::Session::Abstact::ID type, and at least achieve:

• get_session (env, sid): cookie for the client where sid to get the session id, your implementation can be based on the id stored in the corresponding server-side session data, and returns [session_id, session]. Respectively as follows:

  ◦ server returned session_id is the session id, based on your needs, services, and client side of the session_id session_id can be the same or different, as long as they establish a one to one relationship.

  ◦ server to the client session id corresponding to the specific session data is stored where, which look at different implementations, and may be in the database, files, cache and so on.

Another point to note is that sometimes the client's session id may be nil (apparently for the first time the user requests to this is the case), then your implementation should generate a new id.

• set_session (env, sid, session, options): where sid is the service end of the session id, options are included in the parameters using this middleware. This method should be used to update session parameters stored in the server side session data, and returns the corresponding client session id. Next sections is the ID session of the two concrete implementation.

(f) Memcache Session

Rack::Session::Memcache is based on Rack::Session::Abstract::ID of the specific implementation. Therefore, it is through the session_id cookie on the client and server to pass between, and its session data is stored in a memcached cache server (http://memcached.org/).

Parameters

In addition to the default class ID parameters, the middleware, there are two additional default parameters, and memcached related:

DEFAULT_OPTIONS = Abstract::ID::DEFAULT_OPTIONS.merge :namespace => 'rack:session', :memcache_server => 'localhost:11211'

These two parameters and the memcached server-related:

namespace: memcached is the equivalent of a large hash table. For example: foo => bar1

Keyword in the memcached server settings corresponding to the value of foo bar.[14]

Problem is that different parts of an application or different applications may need to set the value of the corresponding keyword foo. In order to prevent conflict between them, you can specify a name space. For example, you set the name space for rack.session, then when you use the keyword foo to set the time, in fact, saved memcached is this:

rack.session: foo => bar

memcache_server: memcached server specified host and port. You can specify multiple hosts and

Port combination, such as: ['localhost:11211',"127.0.0.1:11211"]

Middleware testing whether the initialization process and the memcached server connection:

```
def initialize(app, options={})
  super

  @mutex = Mutex.new
  mserv = @default_options[:memcache_server]
  mopts = @default_options.
  reject{|k,v| MemCache::DEFAULT_OPTIONS.include? k }
  @pool = MemCache.new mserv, mopts
  unless @pool.active? and @pool.servers.any?{|c| c.alive? }
    raise 'No memcache servers'
  end
end
```

14. This is just a hint, in fact, has its own specific protocol memcached set

One of the @mutex variable for later use synchronous logic. The middleware uses MemCache Client.[15]

MemCache.new mserv, mopts

establish and memcached server link. Two parameters are:

mserv server address and port list, is what we provide when memcache_server middleware initialization parameter.

mopts is a memcached server to operate some of the options. This means that you can provide these parameters for the middleware, for example:

```
use Rack::Session::Memcache,
  :memcache_server => 'localhost:11211',
  :namespace => 'rack.session',
  :multithread=>true,
  :failover=>true
```

And so on. Specific parameters, please refer to the code or documentation MemCache (http://github.com/mperham/ memcache-client/blob/master/lib/memcache.rb).

To ask where the @default_options parameters come from, see Rack::Session::Abstract::ID of the initialize implementation.

get_session

The first method is to cover get_session, has been mentioned: get_session (env, sid): cookie for the client where sid to get the session id, you can achieve Id be preserved in accordance with the corresponding session in the server data, and returns [session_id, session]. Let's take a look at get_session the general framework, the specific data acquisition process of the first session was omitted:

```
def get_session(env, session_id)
  @mutex.lock if env['rack.multithread']
  ...
  ...
```

15. http://github.com/mperham/memcache-client

```
  ...
rescue MemCache::MemCacheError, Errno::ECONNREFUSED
  # MemCache server cannot be contacted
  warn "#{self} is unable to find memcached server."
  warn $!.inspect
  return [ nil, {} ]
ensure
  @mutex.unlock if @mutex.locked?
end
```

If env ['rack.multithread'] value is true, then that code may run in multiple threads, so the beginning and end, respectively, and @mutex.unlock @mutex.lock used to protect this critical area. memcached caching rescue operation clause error on the situation, or can not, and memcached to connect, then log in addition to the work of some accidents, and finally return [nil, {}] - the session_id is nil, the session is an empty Hash-ID code in the abstract class in the follow-up session will lead to cookie data is not sent back to the client.

Replaced with a concrete realization of ellipses as follows:

```
unless session_id and session = @pool.get(session_id)
  session_id, session = generate_sid, {}
  unless /^STORED/ =~ @pool.add(session_id, session)
    raise "Session collision on '#{session_id.inspect}'"
  end
end
session.instance_variable_set '@old', @pool.get(session_id, true)
return [session_id, session]
```

Line 3 that in both cases:

• session_id is nil, usually this is the first time the user visits. Or:

• @ pool.get (session_id) do not correspond to the value of the session_id - is likely to memcached server

In, session_id the corresponding item has been removed from the cache. We need to regenerate session_id, while the session data can only be an empty hash. Otherwise, session will be included from the cache server, this session_id the corresponding session data.

4 session_id, session = generate_sid, {}

re-generated using generate_sid session_id generate session after the make the session_id and add to the cache. Memcache the add method:

add (key, value)

Memcached will only cache server does not exist in this key when adding a new key / value corresponding to entry and return to the "STORED". Otherwise return "NOT_STORED" said it could not save, this time it will throw an exception on line 6, line 15 will return [nil, {}].

In line 9, the current session to set an instance variable @ old, it is @ pool.get (session_id, true)

MemCache the get method has two parameters, the first parameter is the key, the second boolean parameter indicates whether to obtain the original data (raw).

Memcached cache server does not know what a Ruby object. So before you save any object, Memcache client will first use Marshal.dump export it to a string and saved to the cache server - this is the raw data. The MemCache client data obtained from the memcached server is stored in the raw data. If raw is set to false (the default), then the client will use Marshal.load Memcache it to reload the object. If raw is set to true, then return directly from memcached server to get the raw data.

Finally, line 10 to return the session_id and the session. The reason why this code should be placed within a critical area because:

• to prevent different users use the same session_id, generate_sid method is defined as follows:

```
def generate_sid
  loop do
    sid = super
    break sid unless @pool.get(sid, true)
  end
end
```

call the abstract super class ID provided (covered previously) random generation method. Although less likely, but still have the opportunity to different users generate the same session_id. So here's generate_sid generated under the super-class generate_sid

• Multiple threads may also exist in determining whether the session and set a new session to be scheduled between the data - which may lead to the same user repeatedly generate different session_id, different session data - which is obviously not allowed.

However, even mutually exclusive conflicts can not be avoided session_id, because Ruby is a different process may also set the session for a one user data, may result in two different processes for the same Ruby.

(g) Routing

How does Rack figure out which app to run? Routing libraries comes to our rescue:

Rack::Builder

Rack::Mount

Rack::URLMap

Using Rack::Builder

```
map '/' do
  run lambda {|env| [200, {}, ["The root is on fire!"]]}
end

map '/files' do
  # Mount an app that serves your files here
end

map "/blog" do
  run Blog::Public
end

map "/db" do
  run Blog::DBAdmin
end
```

Use Rack::Builder to add some more sophisticated behaviour, it's a DSL to construct Rack applications. It provides free logging, http authentication, directory serving etc. Here is another example:

```
app = Rack::Builder.new {
  use Rack::CommonLogger
  use Rack::ShowExceptions
  map "/" do
    use Rack::Lint
    run MyCustomMicroApp.new
  end
}
```

Using Rack::Mount

A stackable dynamic tree based Rack router. Developed by Josh Peek.

Rack::Mount supports Rack's +X-Cascade+ convention to continue trying routes if the response returns pass. This allows multiple routes to be nested or stacked on top of each other. Since the application endpoint can trigger the router to continue matching, middleware can be used to add arbitrary conditions to any route. This allows you to route based on other request attributes, session information, or even data dynamically pulled from a database.

```
require 'rack/mount'

app = Rack::Mount::RouteSet.new do |set|
  inner_app = lambda{|env| [200, {}, ["Using Rack mount to run Rack Application"]]}
  set.add_route(inner_app, :path => "/", :method => "get")
end

use Rack::ContentType
use Rack::ContentLength
# The RouteSet itself is a simple rack app you mount
run app
```

add_route takes a Rack application and conditions to match with. Conditions may be strings or regexps. See Rack::Mount

Usage

Rack::Mount provides a plugin API to build custom DSLs on top of.

The API is extremely minimal and only 3 methods are exposed as the public API.

Rack::Mount::RouteSet#add_route: builder method for adding routes to the set

Rack::Mount::RouteSet#call: Rack compatible recognition and dispatching method

Rack::Mount::RouteSet#generate: generates a route condition from identifiers or significant keys

Using Rack::URLMap

Rack::URLMap.new "/one" => app1,

"/two" => app2,

"/one/foo" => app3

Also can do virtual hosts:

Rack::URLMap.new "http://one.example.org" => app1, "http://two.example.org" => app2,

"https://example.org/secure" => secureap

Rack::URLMap takes a hash mapping urls or paths to apps, and dispatches accordingly. Support for HTTP/1.1 host names exists if the URLs start with http:// or https://.

URLMap modifies the SCRIPT_NAME and PATH_INFO such that the part relevant for dispatch is in the SCRIPT_NAME, and the rest in the PATH_INFO. This should be taken care of when you need to reconstruct the URL in order to create links.

URLMap dispatches in such a way that the longest paths are tried first, since they are most specific.

Rack::URLMap is used to route to multiple applications inside the same process.

5. Testing

Testing with Rack::MockRequest

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

describe "The sample application 3 slides ago" do
  it "Should reply with a welcome on GET" do
    req = Rack::MockRequest.new(myapp)
    res = req.get("/?name=Euruko")
    res.should.be.ok
    res.should.match /Hello, Euruko/
  end
end
```

VI. Appendix

A - Rack Specification

Rack applications

A Rack application is an Ruby object (not a class) that responds to `call`. It takes exactly one argument, the **environment** and returns an Array of exactly three values: The **status**, the **headers**, and the **body**.

The Environment

The environment must be an instance of Hash that includes CGI-like headers. The application is free to modify the environment. The environment is required to include these variables (adopted from PEP333), except when they'd be empty, but see below.

`REQUEST_METHOD`:	The HTTP request method, such as "GET" or "POST". This cannot ever be an empty string, and so is always required.
`SCRIPT_NAME`:	The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This may be an empty string, if the application corresponds to the "root" of the server.
`PATH_INFO`:	The remainder of the request URL's "path", designating the virtual "location" of the request's target within the application. This may be an empty string, if the request URL targets the application root and does not have a trailing slash. This value may be percent-encoded when I originating from a URL.
`QUERY_STRING`:	The portion of the request URL that follows the `?`, if any. May be empty, but is always required!

`SERVER_NAME`, `SERVER_PORT`:	When combined with `SCRIPT_NAME` and `PATH_INFO`, these variables can be used to complete the URL. Note, however, that `HTTP_HOST`, if present, should be used in preference to `SERVER_NAME` for reconstructing the request URL. `SERVER_NAME` and `SERVER_PORT` can never be empty strings, and so are always required.
`HTTP_` Variables:	Variables corresponding to the client-supplied HTTP request headers (i.e., variables whose names begin with `HTTP_`). The presence or absence of these variables should correspond with the presence or absence of the appropriate HTTP header in the request.

In addition to this, the Rack environment must include these Rack-specific variables:

Rack

`rack.version`:	The Array [1,1], representing this version of .
`rack.url_scheme`:	`http` or `https`, depending on the request URL.
`rack.input`:	See below, the input stream.
`rack.errors`:	See below, the error stream.
`rack.multithread`:	true if the application object may be simultaneously invoked by another thread in the same process, false otherwise.
`rack.multiprocess`:	true if an equivalent application object may be simultaneously invoked by another process, false otherwise.
`rack.run_once`:	true if the server expects (but does not guarantee!) that the application will only be invoked this one time during the life of its containing process. Normally, this will only be true for a server based on CGI (or something similar).

Additional environment specifications have approved to standardized middleware APIs. None of these are required to be implemented by the server.

`rack.session`:	A hash like interface for storing request session data. The store must implement: store(key, value) (aliased as []=); fetch(key, default = nil) (aliased as []); delete(key); clear;

`rack.logger`:	A common object interface for logging messages. The object must implement:
	`info(message, &block)`
	`debug(message, &block)`
	`warn(message, &block)`
	`error(message, &block)`
	`fatal(message, &block)`

The server or the application can store their own data in the environment, too. The keys must contain at least one dot, and should be prefixed uniquely. The prefix `rack.` is reserved for use with the Rack core distribution and other accepted specifications and must not be used otherwise. The environment must not contain the keys `HTTP_CONTENT_TYPE` or `HTTP_CONTENT_LENGTH` (use the versions without `HTTP_`). The CGI keys (named without a period) must have String values. There are the following restrictions:

- `rack.version` must be an array of Integers.
- `rack.url_scheme` must either be `http` or `https`.
- There must be a valid input stream in `rack.input`.
- There must be a valid error stream in `rack.errors`.
- The `REQUEST_METHOD` must be a valid token.
- The `SCRIPT_NAME`, if non-empty, must start with `/`
- The `PATH_INFO`, if non-empty, must start with `/`
- The `CONTENT_LENGTH`, if given, must consist of digits only.
- One of `SCRIPT_NAME` or `PATH_INFO` must be set. `PATH_INFO` should be `/` if `SCRIPT_NAME` is empty. `SCRIPT_NAME` never should be `/`, but instead be empty.

The Input Stream

The input stream is an IO-like object which contains the raw HTTP POST data. When applicable, its external encoding must be "ASCII-8BIT" and it must be opened in binary mode, for Ruby 1.9 compatibility. The input stream must respond to `gets`, `each`, `read` and `rewind`.

- `gets` must be called without arguments and return a string, or `nil` on EOF.
- `read` behaves like IO#read. Its signature is `read([length, [buffer]])`. If given, `length` must be an non-negative Integer (>= 0) or `nil`, and `buffer` must be a String and may not be nil. If `length` is given and not nil, then this method reads at most `length` bytes from the input stream. If `length` is not given or nil, then this method reads all data until EOF. When EOF is reached, this method returns nil if `length` is

given and not nil, or "" if `length` is not given or is nil. If `buffer` is given, then the read data will be placed into `buffer` instead of a newly created String object.

- `each` must be called without arguments and only yield Strings.
- `rewind` must be called without arguments. It rewinds the input stream back to the beginning. It must not raise Errno::ESPIPE: that is, it may not be a pipe or a socket. Therefore, handler developers must buffer the input data into some rewindable object if the underlying input stream is not rewindable.
- `close` must never be called on the input stream.

The Error Stream

The error stream must respond to `puts`, `write` and `flush`.

- `puts` must be called with a single argument that responds to `to_s`.
- `write` must be called with a single argument that is a String.
- `flush` must be called without arguments and must be called in order to make the error appear for sure.
- `close` must never be called on the error stream.

The Response

The Status

This is an HTTP status. When parsed as integer (`to_i`), it must be greater than or equal to 100.

The Headers

The header must respond to `each`, and yield values of key and value. The header keys must be Strings. The header must not contain a `Status` key, contain keys with `:` or newlines in their name, contain keys names that end in `-` or `_`, but only contain keys that consist of letters, digits, `_` or `-` and start with a letter. The values of the header must be Strings, consisting of lines (for multiple header values, e.g. multiple `Set-Cookie` values) seperated by "\n". The lines must not contain characters below 037.

The Content-Type

There must be a `Content-Type`, except when the `Status` is 1xx, 204 or 304, in which case there must be none given.

The Content-Length

There must not be a `Content-Length` header when the `Status` is 1xx, 204 or 304.

The Body

The Body must respond to `each` and must only yield String values. The Body itself should not be an instance of String, as this will break in Ruby 1.9. If the Body responds to `close`, it will be called after iteration. If the Body responds to `to_path`, it must return a String identifying the location of a file whose contents are identical to that produced by calling `each`; this may be used by the server as an alternative, possibly more efficient way to transport the response. The Body commonly is an Array of Strings, the application instance itself, or a File-like object.

B - Example Middleware

Reverse Proxy

Like Squid or Varnish, but small and simple. It will set correct http headers. That is really important. The main basis of this is that your application pages can be cached by Rack::Cache and stop requests from even having to hit your Ruby process.

JSON-P

This is for when you want to get callbacks for your json / ajax requests from your server. The data will be returned wrapped within a callback method. This can make writing a javascript based interface much faster and easier to implement. This is part of rack-contrib.

C - part of Hypertext Transfer Protocol -- HTTP/1.1

RFC 2616 Fielding, et al.

14 Header Field Definitions

This section defines the syntax and semantics of all standard HTTP/1.1 header fields. For entity-header fields, both sender and recipient refer to either the client or the server, depending on who sends and who receives the entity.

14.1 Accept

The Accept request-header field can be used to specify certain media types which are acceptable for the response. Accept headers can be used to indicate that the request is specifically limited to a small set of desired types, as in the case of a request for an in-line image.

```
Accept         = "Accept" ":"
                 #( media-range [ accept-params ] )
media-range    = ( "*/*"
                 | ( type "/" "*" )
                 | ( type "/" subtype )
                 ) *( ";" parameter )
accept-params  = ";" "q" "=" qvalue *( accept-extension )
```

accept-extension = ";" token ["=" (token | quoted-string)]

The asterisk "*" character is used to group media types into ranges, with "*/*" indicating all media types and "type/*" indicating all subtypes of that type. The media-range MAY include media type parameters that are applicable to that range.

Each media-range MAY be followed by one or more accept-params, beginning with the "q" parameter for indicating a relative quality factor. The first "q" parameter (if any) separates the media-range parameter(s) from the accept-params. Quality factors allow the user or user agent to indicate the relative degree of preference for that media-range, using the qvalue scale from 0 to 1 (section 3.9). The default value is q=1.

Note: Use of the "q" parameter name to separate media type parameters from Accept extension parameters is due to historical practice. Although this prevents any media type parameter named "q" from being used with a media range, such an event is believed to be unlikely given the lack of any "q" parameters in the IANA media type registry and the rare usage of any media type parameters in Accept. Future media types are discouraged from registering any parameter named "q".

The example

Accept: audio/*; q=0.2, audio/basic

SHOULD be interpreted as "I prefer audio/basic, but send me any audio type if it is the best available after an 80% mark-down in quality."

If no Accept header field is present, then it is assumed that the client accepts all media types. If an Accept header field is present, and if the server cannot send a response which is acceptable according to the combined Accept field value, then the server SHOULD send a 406 (not acceptable) response.

A more elaborate example is

Accept: text/plain; q=0.5, text/html,
    text/x-dvi; q=0.8, text/x-c

Verbally, this would be interpreted as "text/html and text/x-c are the preferred media types, but if they do not exist, then send the text/x-dvi entity, and if that does not exist, send the text/plain entity."

Media ranges can be overridden by more specific media ranges or specific media types. If more than one media range applies to a given type, the most specific reference has precedence. For example,

Accept: text/*, text/html, text/html;level=1, */*

have the following precedence:

1) text/html;level=1

2) text/html

3) text/*

4) */*

The media type quality factor associated with a given type is determined by finding the media range with the highest precedence which matches that type. For example,

Accept: text/*;q=0.3, text/html;q=0.7, text/html;level=1,

text/html;level=2;q=0.4, */*;q=0.5

would cause the following values to be associated:

text/html;level=1 = 1

text/html = 0.7

text/plain = 0.3

image/jpeg = 0.5

text/html;level=2 = 0.4

text/html;level=3 = 0.7

Note: A user agent might be provided with a default set of quality values for certain media ranges. However, unless the user agent is a closed system which cannot interact with other rendering agents, this default set ought to be configurable by the user.

14.2 Accept-Charset

The Accept-Charset request-header field can be used to indicate what character sets are acceptable for the response. This field allows clients capable of understanding more comprehensive or special- purpose character sets to signal that capability to a server which is capable of representing documents in those character sets.

Accept-Charset = "Accept-Charset" ":"

1#((charset | "*")[";" "q" "=" qvalue])

Character set values are described in section 3.4. Each charset MAY be given an associated quality value which represents the user's preference for that charset. The default value is q=1. An example is

Accept-Charset: iso-8859-5, unicode-1-1;q=0.8

The special value "*", if present in the Accept-Charset field, matches every character set (including ISO-8859-1) which is not mentioned elsewhere in the Accept-Charset field. If no "*" is

present in an Accept-Charset field, then all character sets not explicitly mentioned get a quality value of 0, except for ISO-8859-1, which gets a quality value of 1 if not explicitly mentioned.

If no Accept-Charset header is present, the default is that any character set is acceptable. If an Accept-Charset header is present, and if the server cannot send a response which is acceptable according to the Accept-Charset header, then the server SHOULD send an error response with the 406 (not acceptable) status code, though the sending of an unacceptable response is also allowed.

14.3 Accept-Encoding

The Accept-Encoding request-header field is similar to Accept, but restricts the content-codings (section 3.5) that are acceptable in the response.

```
Accept-Encoding  = "Accept-Encoding" ":"
                   1#( codings [ ";" "q" "=" qvalue ] )
codings          = ( content-coding | "*" )
```

Examples of its use are:

```
Accept-Encoding: compress, gzip
Accept-Encoding:
Accept-Encoding: *
Accept-Encoding: compress;q=0.5, gzip;q=1.0
Accept-Encoding: gzip;q=1.0, identity; q=0.5, *;q=0
```

A server tests whether a content-coding is acceptable, according to an Accept-Encoding field, using these rules:

1. If the content-coding is one of the content-codings listed in the Accept-Encoding field, then it is acceptable, unless it is accompanied by a qvalue of 0. (As defined in section 3.9, a qvalue of 0 means "not acceptable.")
2. The special "*" symbol in an Accept-Encoding field matches any available content-coding not explicitly listed in the header field.
3. If multiple content-codings are acceptable, then the acceptable content-coding with the highest non-zero qvalue is preferred.
4. The "identity" content-coding is always acceptable, unless specifically refused because the Accept-Encoding field includes "identity;q=0", or because the field includes "*;q=0" and does

not explicitly include the "identity" content-coding. If the
Accept-Encoding field-value is empty, then only the "identity"
encoding is acceptable.

If an Accept-Encoding field is present in a request, and if the server cannot send a response which is acceptable according to the Accept-Encoding header, then the server SHOULD send an error response with the 406 (Not Acceptable) status code.

If no Accept-Encoding field is present in a request, the server MAY assume that the client will accept any content coding. In this case, if "identity" is one of the available content-codings, then the server SHOULD use the "identity" content-coding, unless it has additional information that a different content-coding is meaningful to the client.

Note: If the request does not include an Accept-Encoding field,
and if the "identity" content-coding is unavailable, then
content-codings commonly understood by HTTP/1.0 clients (i.e.,
"gzip" and "compress") are preferred; some older clients
improperly display messages sent with other content-codings. The
server might also make this decision based on information about
the particular user-agent or client.

Note: Most HTTP/1.0 applications do not recognize or obey qvalues
associated with content-codings. This means that qvalues will not
work and are not permitted with x-gzip or x-compress.

14.4 Accept-Language

The Accept-Language request-header field is similar to Accept, but restricts the set of natural languages that are preferred as a response to the request. Language tags are defined in section 3.10.

```
Accept-Language = "Accept-Language" ":"
                  1#( language-range [ ";" "q" "=" qvalue ] )
language-range  = ( ( 1*8ALPHA *( "-" 1*8ALPHA ) ) | "*" )
```

Each language-range MAY be given an associated quality value which represents an estimate of the user's preference for the languages specified by that range. The quality value defaults to "q=1". For example,

Accept-Language: da, en-gb;q=0.8, en;q=0.7

would mean: "I prefer Danish, but will accept British English and other types of English." A language-range matches a language-tag if it exactly equals the tag, or if it exactly equals a prefix of the tag such that the first tag character following the prefix is "-". The special range "*", if

present in the Accept-Language field, matches every tag not matched by any other range present in the Accept-Language field.

Note: This use of a prefix matching rule does not imply that language tags are assigned to languages in such a way that it is always true that if a user understands a language with a certain tag, then this user will also understand all languages with tags for which this tag is a prefix. The prefix rule simply allows the use of prefix tags if this is the case.

The language quality factor assigned to a language-tag by the Accept-Language field is the quality value of the longest language- range in the field that matches the language-tag. If no language-range in the field matches the tag, the language quality factor assigned is 0. If no Accept-Language header is present in the request, the server

SHOULD assume that all languages are equally acceptable. If an Accept-Language header is present, then all languages which are assigned a quality factor greater than 0 are acceptable.

It might be contrary to the privacy expectations of the user to send an Accept-Language header with the complete linguistic preferences of the user in every request. For a discussion of this issue, see section 15.1.4.

As intelligibility is highly dependent on the individual user, it is recommended that client applications make the choice of linguistic preference available to the user. If the choice is not made available, then the Accept-Language header field MUST NOT be given in the request.

Note: When making the choice of linguistic preference available to the user, we remind implementors of the fact that users are not familiar with the details of language matching as described above, and should provide appropriate guidance. As an example, users might assume that on selecting "en-gb", they will be served any kind of English document if British English is not available. A user agent might suggest in such a case to add "en" to get the best matching behavior.

14.5 Accept-Ranges

The Accept-Ranges response-header field allows the server to indicate its acceptance of range requests for a resource:

```
Accept-Ranges     = "Accept-Ranges" ":" acceptable-ranges
acceptable-ranges = 1#range-unit | "none"
```

Origin servers that accept byte-range requests MAY send

    Accept-Ranges: bytes

but are not required to do so. Clients MAY generate byte-range requests without having received this header for the resource involved. Range units are defined in section 3.12.

Servers that do not accept any kind of range request for a resource MAY send

    Accept-Ranges: none

to advise the client not to attempt a range request.

14.6 Age

The Age response-header field conveys the sender's estimate of the amount of time since the response (or its revalidation) was generated at the origin server. A cached response is "fresh" if its age does not exceed its freshness lifetime. Age values are calculated as specified in section 13.2.3.

    Age = "Age" ":" age-value
    age-value = delta-seconds

Age values are non-negative decimal integers, representing time in seconds.

If a cache receives a value larger than the largest positive integer it can represent, or if any of its age calculations overflows, it MUST transmit an Age header with a value of 2147483648 (2^31). An HTTP/1.1 server that includes a cache MUST include an Age header field in every response generated from its own cache. Caches SHOULD use an arithmetic type of at least 31 bits of range.

14.7 Allow

The Allow entity-header field lists the set of methods supported by the resource identified by the Request-URI. The purpose of this field is strictly to inform the recipient of valid methods

associated with the resource. An Allow header field MUST be present in a 405 (Method Not Allowed) response.

Allow = "Allow" ":" #Method

Example of use:

Allow: GET, HEAD, PUT

This field cannot prevent a client from trying other methods. However, the indications given by the Allow header field value SHOULD be followed. The actual set of allowed methods is defined by the origin server at the time of each request.

The Allow header field MAY be provided with a PUT request to recommend the methods to be supported by the new or modified resource. The server is not required to support these methods and SHOULD include an Allow header in the response giving the actual supported methods.

A proxy MUST NOT modify the Allow header field even if it does not understand all the methods specified, since the user agent might have other means of communicating with the origin server.

14.8 Authorization

A user agent that wishes to authenticate itself with a server--usually, but not necessarily, after receiving a 401 response--does so by including an Authorization request-header field with the request. The Authorization field value consists of credentials containing the authentication information of the user agent for the realm of the resource being requested.

Authorization = "Authorization" ":" credentials

HTTP access authentication is described in "HTTP Authentication: Basic and Digest Access Authentication" [43]. If a request is authenticated and a realm specified, the same credentials SHOULD be valid for all other requests within this realm (assuming that the authentication scheme itself does not require otherwise, such

as credentials that vary according to a challenge value or using synchronized clocks).

When a shared cache (see section 13.7) receives a request containing an Authorization field, it MUST NOT return the corresponding response as a reply to any other request, unless one of the following specific exceptions holds:

1. If the response includes the "s-maxage" cache-control directive, the cache MAY use that response in replying to a subsequent request. But (if the specified maximum age has passed) a proxy cache MUST first revalidate it with the origin server, using the request-headers from the new request to allow the origin server to authenticate the new request. (This is the defined behavior for s-maxage.) If the response includes "s-maxage=0", the proxy MUST always revalidate it before re-using it.
2. If the response includes the "must-revalidate" cache-control directive, the cache MAY use that response in replying to a subsequent request. But if the response is stale, all caches MUST first revalidate it with the origin server, using the request-headers from the new request to allow the origin server to authenticate the new request.
3. If the response includes the "public" cache-control directive, it MAY be returned in reply to any subsequent request.

14.9 Cache-Control

The Cache-Control general-header field is used to specify directives that MUST be obeyed by all caching mechanisms along the request/response chain. The directives specify behavior intended to prevent caches from adversely interfering with the request or response. These directives typically override the default caching algorithms. Cache directives are unidirectional in that the presence of a directive in a request does not imply that the same directive is to be given in the response.

Note that HTTP/1.0 caches might not implement Cache-Control and might only implement Pragma: no-cache (see section 14.32).

Cache directives MUST be passed through by a proxy or gateway application, regardless of their significance to that application, since the directives might be applicable to all recipients along the request/response chain. It is not possible to specify a cache- directive for a specific cache.

```
Cache-Control   = "Cache-Control" ":" 1#cache-directive

cache-directive = cache-request-directive
    | cache-response-directive

cache-request-directive =
      "no-cache"                      ; Section 14.9.1
    | "no-store"                      ; Section 14.9.2
    | "max-age" "=" delta-seconds     ; Section 14.9.3, 14.9.4
    | "max-stale" [ "=" delta-seconds ]   ; Section 14.9.3
    | "min-fresh" "=" delta-seconds   ; Section 14.9.3
    | "no-transform"                  ; Section 14.9.5
    | "only-if-cached"                ; Section 14.9.4
    | cache-extension                 ; Section 14.9.6

 cache-response-directive =
      "public"                        ; Section 14.9.1
    | "private" [ "=" <"> 1#field-name <"> ] ; Section 14.9.1
    | "no-cache" [ "=" <"> 1#field-name <"> ]; Section 14.9.1
    | "no-store"                      ; Section 14.9.2
    | "no-transform"                  ; Section 14.9.5
    | "must-revalidate"               ; Section 14.9.4
    | "proxy-revalidate"              ; Section 14.9.4
    | "max-age" "=" delta-seconds     ; Section 14.9.3
    | "s-maxage" "=" delta-seconds    ; Section 14.9.3
    | cache-extension                 ; Section 14.9.6

cache-extension = token [ "=" ( token | quoted-string ) ]
```

When a directive appears without any 1#field-name parameter, the directive applies to the entire request or response. When such a directive appears with a 1#field-name parameter, it applies only to the named field or fields, and not to the rest of the request or response. This mechanism

supports extensibility; implementations of future versions of the HTTP protocol might apply these directives to header fields not defined in HTTP/1.1.

The cache-control directives can be broken down into these general categories:

- Restrictions on what are cacheable; these may only be imposed by the origin server.
- Restrictions on what may be stored by a cache; these may be imposed by either the origin server or the user agent.
- Modifications of the basic expiration mechanism; these may be imposed by either the origin server or the user agent.
- Controls over cache revalidation and reload; these may only be imposed by a user agent.
- Control over transformation of entities.
- Extensions to the caching system.

14.9.1 What is Cacheable

By default, a response is cacheable if the requirements of the request method, request header fields, and the response status indicate that it is cacheable. Section 13.4 summarizes these defaults for cacheability. The following Cache-Control response directives allow an origin server to override the default cacheability of a response:

public

Indicates that the response MAY be cached by any cache, even if it would normally be non-cacheable or cacheable only within a non- shared cache. (See also Authorization, section 14.8, for additional details.)

private

Indicates that all or part of the response message is intended for a single user and MUST NOT be cached by a shared cache. This allows an origin server to state that the specified parts of the response are intended for only one user and are not a valid response for requests by other users. A private (non-shared) cache MAY cache the response.

Note: This usage of the word private only controls where the response may be cached, and cannot ensure the privacy of the message content.

no-cache

If the no-cache directive does not specify a field-name, then a cache MUST NOT use the response to satisfy a subsequent request without successful revalidation with the origin server. This allows an origin server to prevent caching even by caches that have been configured to return stale responses to client requests.

If the no-cache directive does specify one or more field-names, then a cache MAY use the response to satisfy a subsequent request, subject to any other restrictions on caching. However, the specified field-name(s) MUST NOT be sent in the response to a subsequent request without successful revalidation with the origin server. This allows an origin server to prevent the re-use of certain header fields in a response, while still allowing caching of the rest of the response.

Note: Most HTTP/1.0 caches will not recognize or obey this directive.

14.9.2 What May be Stored by Caches

no-store

The purpose of the no-store directive is to prevent the inadvertent release or retention of sensitive information (for example, on backup tapes). The no-store directive applies to the entire message, and MAY be sent either in a response or in a request. If sent in a request, a cache MUST NOT store any part of either this request or any response to it. If sent in a response, a cache MUST NOT store any part of either this response or the request that elicited it. This directive applies to both non- shared and shared caches. "MUST NOT store" in this context means that the cache MUST NOT intentionally store the information in non-volatile storage, and MUST make a best-effort attempt to remove the information from volatile storage as promptly as possible after forwarding it.

Even when this directive is associated with a response, users might explicitly store such a response outside of the caching system (e.g., with a "Save As" dialog). History buffers MAY store such responses as part of their normal operation.

The purpose of this directive is to meet the stated requirements of certain users and service authors who are concerned about accidental releases of information via unanticipated accesses to cache data structures. While the use of this directive might improve privacy in some cases, we caution that it is NOT in any way a reliable or sufficient mechanism for ensuring privacy. In particular, malicious or compromised caches might not recognize or obey this directive, and communications networks might be vulnerable to eavesdropping.

14.9.3 Modifications of the Basic Expiration Mechanism

The expiration time of an entity MAY be specified by the origin server using the Expires header (see section 14.21). Alternatively, it MAY be specified using the max-age directive in a response. When the max-age cache-control directive is present in a cached response, the response is stale if its current age is greater than the age value given (in seconds) at the time of a new request for that resource. The max-age directive on a response implies that the response is cacheable (i.e., "public") unless some other, more restrictive cache directive is also present.

If a response includes both an Expires header and a max-age directive, the max-age directive overrides the Expires header, even if the Expires header is more restrictive. This rule allows an origin server to provide, for a given response, a longer expiration time to an HTTP/1.1 (or later) cache than to an HTTP/1.0 cache. This might be useful if certain HTTP/1.0 caches improperly calculate ages or expiration times, perhaps due to desynchronized clocks.

Many HTTP/1.0 cache implementations will treat an Expires value that is less than or equal to the response Date value as being equivalent to the Cache-Control response directive "no-cache". If an HTTP/1.1 cache receives such a response, and the response does not include a Cache-Control header field, it SHOULD consider the response to be non-cacheable in order to retain compatibility with HTTP/1.0 servers.

Note: An origin server might wish to use a relatively new HTTP cache control feature, such as the "private" directive, on a network including older caches that do not understand that feature. The origin server will need to combine the new feature with an Expires field whose value is less than or equal to the Date value. This will prevent older caches from improperly caching the response.

s-maxage

If a response includes an s-maxage directive, then for a shared cache (but not for a private cache), the maximum age specified by this directive overrides the maximum age specified by either the max-age directive or the Expires header. The s-maxage directive also implies the semantics of the proxy-revalidate directive (see section 14.9.4), i.e., that the shared cache must not use the entry after it becomes stale to respond to a subsequent request without first revalidating it with the origin server. The s- maxage directive is always ignored by a private cache.

Note that most older caches, not compliant with this specification, do not implement any cache-control directives. An origin server wishing to use a cache-control directive that restricts, but does not prevent, caching by an HTTP/1.1-compliant cache MAY exploit the requirement that the max-age directive overrides the Expires header, and the fact that pre-HTTP/1.1-compliant caches do not observe the max-age directive.

Other directives allow a user agent to modify the basic expiration mechanism. These directives MAY be specified on a request:

max-age

Indicates that the client is willing to accept a response whose age is no greater than the specified time in seconds. Unless max- stale directive is also included, the client is not willing to accept a stale response.

min-fresh

Indicates that the client is willing to accept a response whose freshness lifetime is no less than its current age plus the specified time in seconds. That is, the client wants a response that will still be fresh for at least the specified number of seconds.

max-stale

Indicates that the client is willing to accept a response that has exceeded its expiration time. If max-stale is assigned a value, then the client is willing to accept a response that has exceeded its expiration time by no more than the specified number of seconds. If no value is assigned to max-stale, then the client is willing to accept a stale response of any age.

If a cache returns a stale response, either because of a max-stale directive on a request, or because the cache is configured to override the expiration time of a response, the cache MUST attach a Warning header to the stale response, using Warning 110 (Response is stale).

A cache MAY be configured to return stale responses without validation, but only if this does not conflict with any "MUST"-level requirements concerning cache validation (e.g., a "must-revalidate" cache-control directive).

If both the new request and the cached entry include "max-age" directives, then the lesser of the two values is used for determining the freshness of the cached entry for that request.

14.9.4 Cache Revalidation and Reload Controls

Sometimes a user agent might want or need to insist that a cache revalidate its cache entry with the origin server (and not just with the next cache along the path to the origin server), or to reload its cache entry from the origin server. End-to-end revalidation might be necessary if either the cache or the origin server has overestimated the expiration time of the cached response. End-to-end reload may be necessary if the cache entry has become corrupted for some reason.

End-to-end revalidation may be requested either when the client does not have its own local cached copy, in which case we call it "unspecified end-to-end revalidation", or when the client does have a local cached copy, in which case we call it "specific end-to-end revalidation."

The client can specify these three kinds of action using Cache- Control request directives:

End-to-end reload

The request includes a "no-cache" cache-control directive or, for compatibility with HTTP/1.0 clients, "Pragma: no-cache". Field names MUST NOT be included with the no-cache directive in a request. The server MUST NOT use a cached copy when responding to such a request.

Specific end-to-end revalidation

The request includes a "max-age=0" cache-control directive, which forces each cache along the path to the origin server to revalidate its own entry, if any, with the next cache or server. The initial request includes a cache-validating conditional with the client's current validator.

Unspecified end-to-end revalidation

The request includes "max-age=0" cache-control directive, which forces each cache along the path to the origin server to revalidate its own entry, if any, with the next cache or server. The initial request does not include a cache-validating

conditional; the first cache along the path (if any) that holds a cache entry for this resource includes a cache-validating conditional with its current validator.

max-age

When an intermediate cache is forced, by means of a max-age=0 directive, to revalidate its own cache entry, and the client has supplied its own validator in the request, the supplied validator might differ from the validator currently stored with the cache entry. In this case, the cache MAY use either validator in making its own request without affecting semantic transparency.

However, the choice of validator might affect performance. The best approach is for the intermediate cache to use its own validator when making its request. If the server replies with 304 (Not Modified), then the cache can return its now validated copy to the client with a 200 (OK) response. If the server replies with a new entity and cache validator, however, the intermediate cache can compare the returned validator with the one provided in the client's request, using the strong comparison function. If the client's validator is equal to the origin server's, then the intermediate cache simply returns 304 (Not Modified). Otherwise, it returns the new entity with a 200 (OK) response.

If a request includes the no-cache directive, it SHOULD NOT include min-fresh, max-stale, or max-age.

only-if-cached

In some cases, such as times of extremely poor network connectivity, a client may want a cache to return only those responses that it currently has stored, and not to reload or revalidate with the origin server. To do this, the client may include the only-if-cached directive in a request. If it receives this directive, a cache SHOULD either respond using a cached entry that is consistent with the other constraints of the request, or respond with a 504 (Gateway Timeout) status. However, if a group of caches is being operated as a unified system with good internal connectivity, such a request MAY be forwarded within that group of caches.

must-revalidate

Because a cache MAY be configured to ignore a server's specified expiration time, and because a client request MAY include a max- stale directive (which has a similar effect), the protocol also includes a mechanism for the origin server to require revalidation of a cache entry on any subsequent use. When the must-revalidate directive is present in a response received by a cache, that cache MUST NOT use the entry after it becomes stale to respond to a

subsequent request without first revalidating it with the origin server. (I.e., the cache MUST do an end-to-end revalidation every time, if, based solely on the origin server's Expires or max-age value, the cached response is stale.)

The must-revalidate directive is necessary to support reliable operation for certain protocol features. In all circumstances an HTTP/1.1 cache MUST obey the must-revalidate directive; in particular, if the cache cannot reach the origin server for any reason, it MUST generate a 504 (Gateway Timeout) response.

Servers SHOULD send the must-revalidate directive if and only if failure to revalidate a request on the entity could result in incorrect operation, such as a silently unexecuted financial transaction. Recipients MUST NOT take any automated action that violates this directive, and MUST NOT automatically provide an unvalidated copy of the entity if revalidation fails.

Although this is not recommended, user agents operating under severe connectivity constraints MAY violate this directive but, if so, MUST explicitly warn the user that an unvalidated response has been provided. The warning MUST be provided on each unvalidated access, and SHOULD require explicit user confirmation.

proxy-revalidate

The proxy-revalidate directive has the same meaning as the must- revalidate directive, except that it does not apply to non-shared user agent caches. It can be used on a response to an authenticated request to permit the user's cache to store and later return the response without needing to revalidate it (since it has already been authenticated once by that user), while still requiring proxies that service many users to revalidate each time (in order to make sure that each user has been authenticated). Note that such authenticated responses also need the public cache control directive in order to allow them to be cached at all.

14.9.5 No-Transform Directive

no-transform

Implementors of intermediate caches (proxies) have found it useful to convert the media type of certain entity bodies. A non- transparent proxy might, for example, convert between image formats in order to save cache space or to reduce the amount of traffic on a slow link.

Serious operational problems occur, however, when these transformations are applied to entity bodies intended for certain kinds of applications. For example, applications for medical

imaging, scientific data analysis and those using end-to-end authentication, all depend on receiving an entity body that is bit for bit identical to the original entity-body.

Therefore, if a message includes the no-transform directive, an intermediate cache or proxy MUST NOT change those headers that are listed in section 13.5.2 as being subject to the no-transform directive. This implies that the cache or proxy MUST NOT change any aspect of the entity-body that is specified by these headers, including the value of the entity-body itself.

14.9.6 Cache Control Extensions

The Cache-Control header field can be extended through the use of one or more cache-extension tokens, each with an optional assigned value. Informational extensions (those which do not require a change in cache behavior) MAY be added without changing the semantics of other directives. Behavioral extensions are designed to work by acting as modifiers to the existing base of cache directives. Both the new directive and the standard directive are supplied, such that applications which do not understand the new directive will default to the behavior specified by the standard directive, and those that understand the new directive will recognize it as modifying the requirements associated with the standard directive. In this way, extensions to the cache-control directives can be made without requiring changes to the base protocol.

This extension mechanism depends on an HTTP cache obeying all of the cache-control directives defined for its native HTTP-version, obeying certain extensions, and ignoring all directives that it does not understand.

For example, consider a hypothetical new response directive called community which acts as a modifier to the private directive. We define this new directive to mean that, in addition to any non-shared cache, any cache which is shared only by members of the community named within its value may cache the response. An origin server wishing to allow the UCI community to use an otherwise private response in their shared cache(s) could do so by including

Cache-Control: private, community="UCI"

A cache seeing this header field will act correctly even if the cache does not understand the community cache-extension, since it will also see and understand the private directive and thus default to the safe behavior.

Unrecognized cache-directives MUST be ignored; it is assumed that any cache-directive likely to be unrecognized by an HTTP/1.1 cache will be combined with standard directives (or the response's default cacheability) such that the cache behavior will remain minimally correct even if the cache does not understand the extension(s).

14.10 Connection

The Connection general-header field allows the sender to specify options that are desired for that particular connection and MUST NOT be communicated by proxies over further connections.

The Connection header has the following grammar:

Connection = "Connection" ":" 1#(connection-token)

connection-token = token

HTTP/1.1 proxies MUST parse the Connection header field before a message is forwarded and, for each connection-token in this field, remove any header field(s) from the message with the same name as the connection-token. Connection options are signaled by the presence of a connection-token in the Connection header field, not by any corresponding additional header field(s), since the additional header field may not be sent if there are no parameters associated with that connection option.

Message headers listed in the Connection header MUST NOT include end-to-end headers, such as Cache-Control.

HTTP/1.1 defines the "close" connection option for the sender to signal that the connection will be closed after completion of the response. For example,

Connection: close

in either the request or the response header fields indicates that the connection SHOULD NOT be considered `persistent' (section 8.1) after the current request/response is complete.

HTTP/1.1 applications that do not support persistent connections MUST include the "close" connection option in every message.

A system receiving an HTTP/1.0 (or lower-version) message that includes a Connection header MUST, for each connection-token in this field, remove and ignore any header field(s) from the message with the same name as the connection-token. This protects against mistaken forwarding of such header fields by pre-HTTP/1.1 proxies. See section 19.6.2.

14.11 Content-Encoding

The Content-Encoding entity-header field is used as a modifier to the media-type. When present, its value indicates what additional content codings have been applied to the entity-body, and thus what decoding mechanisms must be applied in order to obtain the media-type referenced by the

Content-Type header field. Content-Encoding is primarily used to allow a document to be compressed without losing the identity of its underlying media type.

Content-Encoding = "Content-Encoding" ":" 1#content-coding

Content codings are defined in section 3.5. An example of its use is

Content-Encoding: gzip

The content-coding is a characteristic of the entity identified by the Request-URI. Typically, the entity-body is stored with this encoding and is only decoded before rendering or analogous usage. However, a non-transparent proxy MAY modify the content-coding if the new coding is known to be acceptable to the recipient, unless the "no-transform" cache-control directive is present in the message.

If the content-coding of an entity is not "identity", then the response MUST include a Content-Encoding entity-header (section 14.11) that lists the non-identity content-coding(s) used.

If the content-coding of an entity in a request message is not acceptable to the origin server, the server SHOULD respond with a status code of 415 (Unsupported Media Type).

If multiple encodings have been applied to an entity, the content codings MUST be listed in the order in which they were applied. Additional information about the encoding parameters MAY be provided by other entity-header fields not defined by this specification.

14.12 Content-Language

The Content-Language entity-header field describes the natural language(s) of the intended audience for the enclosed entity. Note that this might not be equivalent to all the languages used within the entity-body.

Content-Language = "Content-Language" ":" 1#language-tag

Language tags are defined in section 3.10. The primary purpose of Content-Language is to allow a user to identify and differentiate entities according to the user's own preferred language. Thus, if the body content is intended only for a Danish-literate audience, the appropriate field is

Content-Language: da

If no Content-Language is specified, the default is that the content is intended for all language audiences. This might mean that the sender does not consider it to be specific to any natural language, or that the sender does not know for which language it is intended.

Multiple languages MAY be listed for content that is intended for multiple audiences. For example, a rendition of the "Treaty of Waitangi," presented simultaneously in the original Maori and English versions, would call for

Content-Language: mi, en

However, just because multiple languages are present within an entity does not mean that it is intended for multiple linguistic audiences. An example would be a beginner's language primer,

such as "A First Lesson in Latin," which is clearly intended to be used by an English-literate audience. In this case, the Content-Language would properly only include "en".

Content-Language MAY be applied to any media type -- it is not limited to textual documents.

14.13 Content-Length

The Content-Length entity-header field indicates the size of the entity-body, in decimal number of OCTETs, sent to the recipient or, in the case of the HEAD method, the size of the entity-body that would have been sent had the request been a GET.

Content-Length = "Content-Length" ":" 1*DIGIT

An example is

Content-Length: 3495

Applications SHOULD use this field to indicate the transfer-length of the message-body, unless this is prohibited by the rules in section 4.4.

Any Content-Length greater than or equal to zero is a valid value. Section 4.4 describes how to determine the length of a message-body if a Content-Length is not given.

Note that the meaning of this field is significantly different from the corresponding definition in MIME, where it is an optional field used within the "message/external-body" content-type. In HTTP, it SHOULD be sent whenever the message's length can be determined prior to being transferred, unless this is prohibited by the rules in section 4.4.

14.14 Content-Location

The Content-Location entity-header field MAY be used to supply the resource location for the entity enclosed in the message when that entity is accessible from a location separate from the requested resource's URI. A server SHOULD provide a Content-Location for the variant corresponding to the response entity; especially in the case where a resource has multiple entities associated with it, and those entities actually have separate locations by which they might be individually accessed, the server SHOULD provide a Content-Location for the particular variant which is returned.

Content-Location = "Content-Location" ":"

(absoluteURI | relativeURI)

The value of Content-Location also defines the base URI for the entity.

The Content-Location value is not a replacement for the original requested URI; it is only a statement of the location of the resource corresponding to this particular entity at the time of the request. Future requests MAY specify the Content-Location URI as the request- URI if the desire is to identify the source of that particular entity.

A cache cannot assume that an entity with a Content-Location different from the URI used to retrieve it can be used to respond to later requests on that Content-Location URI. However, the

Content- Location can be used to differentiate between multiple entities retrieved from a single requested resource, as described in section 13.6.

If the Content-Location is a relative URI, the relative URI is interpreted relative to the Request-URI.

The meaning of the Content-Location header in PUT or POST requests is undefined; servers are free to ignore it in those cases.

14.15 Content-MD5

The Content-MD5 entity-header field, as defined in RFC 1864 [23], is an MD5 digest of the entity-body for the purpose of providing an end-to-end message integrity check (MIC) of the entity-body. (Note: a MIC is good for detecting accidental modification of the entity-body in transit, but is not proof against malicious attacks.)

Content-MD5 = "Content-MD5" ":" md5-digest

md5-digest = <base64 of 128 bit MD5 digest as per RFC 1864>

The Content-MD5 header field MAY be generated by an origin server or client to function as an integrity check of the entity-body. Only origin servers or clients MAY generate the Content-MD5 header field; proxies and gateways MUST NOT generate it, as this would defeat its value as an end-to-end integrity check. Any recipient of the entity- body, including gateways and proxies, MAY check that the digest value in this header field matches that of the entity-body as received.

The MD5 digest is computed based on the content of the entity-body, including any content-coding that has been applied, but not including any transfer-encoding applied to the message-body. If the message is received with a transfer-encoding, that encoding MUST be removed prior to checking the Content-MD5 value against the received entity.

This has the result that the digest is computed on the octets of the entity-body exactly as, and in the order that, they would be sent if no transfer-encoding were being applied.

HTTP extends RFC 1864 to permit the digest to be computed for MIME composite media-types (e.g., multipart/* and message/rfc822), but this does not change how the digest is computed as defined in the preceding paragraph.

There are several consequences of this. The entity-body for composite types MAY contain many body-parts, each with its own MIME and HTTP headers (including Content-MD5, Content-Transfer-Encoding, and Content-Encoding headers). If a body-part has a Content-Transfer- Encoding or Content-Encoding header, it is assumed that the content of the body-part has had the encoding applied, and the body-part is included in the Content-MD5 digest as is -- i.e., after the application. The Transfer-Encoding header field is not allowed within body-parts.

Conversion of all line breaks to CRLF MUST NOT be done before computing or checking the digest: the line break convention used in the text actually transmitted MUST be left unaltered when computing the digest.

Note: while the definition of Content-MD5 is exactly the same for HTTP as in RFC 1864 for MIME entity-bodies, there are several ways in which the application of Content-MD5 to HTTP entity-bodies differs from its application to MIME entity-bodies. One is that HTTP, unlike MIME, does not use Content-Transfer-Encoding, and does use Transfer-Encoding and Content-Encoding. Another is that HTTP more frequently uses binary content types than MIME, so it is worth noting that, in such cases, the byte order used to compute the digest is the transmission byte order defined for the type. Lastly, HTTP allows transmission of text types with any of several line break conventions and not just the canonical form using CRLF.

14.16 Content-Range

The Content-Range entity-header is sent with a partial entity-body to specify where in the full entity-body the partial body should be applied. Range units are defined in section 3.12.

```
Content-Range = "Content-Range" ":" content-range-spec
content-range-spec      = byte-content-range-spec
byte-content-range-spec = bytes-unit SP
                          byte-range-resp-spec "/"
                          ( instance-length | "*" )

byte-range-resp-spec = (first-byte-pos "-" last-byte-pos)
                       | "*"
instance-length          = 1*DIGIT
```

The header SHOULD indicate the total length of the full entity-body, unless this length is unknown or difficult to determine. The asterisk "*" character means that the instance-length is unknown at the time when the response was generated.

Unlike byte-ranges-specifier values (see section 14.35.1), a byte- range-resp-spec MUST only specify one range, and MUST contain absolute byte positions for both the first and last byte of the range.

A byte-content-range-spec with a byte-range-resp-spec whose last- byte-pos value is less than its first-byte-pos value, or whose instance-length value is less than or equal to its last-byte-pos val-

ue, is invalid. The recipient of an invalid byte-content-range- spec MUST ignore it and any content transferred along with it.

A server sending a response with status code 416 (Requested range not satisfiable) SHOULD include a Content-Range field with a byte-range- resp-spec of "*". The instance-length specifies the current length of

the selected resource. A response with status code 206 (Partial Content) MUST NOT include a Content-Range field with a byte-range- resp-spec of "*".

Examples of byte-content-range-spec values, assuming that the entity contains a total of 1234 bytes:

. The first 500 bytes:

bytes 0-499/1234

. The second 500 bytes:

bytes 500-999/1234

. All except for the first 500 bytes:

bytes 500-1233/1234

. The last 500 bytes:

bytes 734-1233/1234

When an HTTP message includes the content of a single range (for example, a response to a request for a single range, or to a request for a set of ranges that overlap without any holes), this content is transmitted with a Content-Range header, and a Content-Length header showing the number of bytes actually transferred. For example,

HTTP/1.1 206 Partial content

Date: Wed, 15 Nov 1995 06:25:24 GMT

Last-Modified: Wed, 15 Nov 1995 04:58:08 GMT

Content-Range: bytes 21010-47021/47022

Content-Length: 26012

Content-Type: image/gif

When an HTTP message includes the content of multiple ranges (for example, a response to a request for multiple non-overlapping ranges), these are transmitted as a multipart message. The multipart media type used for this purpose is "multipart/byteranges" as defined in appendix 19.2. See appendix 19.6.3 for a compatibility issue.

A response to a request for a single range MUST NOT be sent using the multipart/byteranges media type. A response to a request for multiple ranges, whose result is a single range, MAY be sent as a multipart/byteranges media type with one part. A client that cannot decode a multipart/byteranges message MUST NOT ask for multiple byte-ranges in a single request.

When a client requests multiple byte-ranges in one request, the server SHOULD return them in the order that they appeared in the request.

If the server ignores a byte-range-spec because it is syntactically invalid, the server SHOULD treat the request as if the invalid Range header field did not exist. (Normally, this means return a 200 response containing the full entity).

If the server receives a request (other than one including an If- Range request-header field) with an unsatisfiable Range request- header field (that is, all of whose byte-range-spec values have a first-byte-pos value greater than the current length of the selected resource), it SHOULD return a response code of 416 (Requested range not satisfiable) (section 10.4.17).

Note: clients cannot depend on servers to send a 416 (Requested range not satisfiable) response instead of a 200 (OK) response for an unsatisfiable Range request-header, since not all servers implement this request-header.

14.17 Content-Type

The Content-Type entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.

Content-Type = "Content-Type" ":" media-type

Media types are defined in section 3.7. An example of the field is

Content-Type: text/html; charset=ISO-8859-4

Further discussion of methods for identifying the media type of an entity is provided in section 7.2.1.

14.18 Date

The Date general-header field represents the date and time at which the message was originated, having the same semantics as orig-date in RFC 822. The field value is an HTTP-date, as described in section 3.3.1; it MUST be sent in RFC 1123 [8]-date format.

Date = "Date" ":" HTTP-date

An example is

Date: Tue, 15 Nov 1994 08:12:31 GMT

Origin servers MUST include a Date header field in all responses, except in these cases:

1. If the response status code is 100 (Continue) or 101 (Switching Protocols), the response MAY include a Date header field, at the server's option.

2. If the response status code conveys a server error, e.g. 500 (Internal Server Error) or 503 (Service Unavailable), and it is inconvenient or impossible to generate a valid Date.
3. If the server does not have a clock that can provide a reasonable approximation of the current time, its responses MUST NOT include a Date header field. In this case, the rules in section 14.18.1 MUST be followed.

A received message that does not have a Date header field MUST be assigned one by the recipient if the message will be cached by that recipient or gatewayed via a protocol which requires a Date. An HTTP implementation without a clock MUST NOT cache responses without revalidating them on every use. An HTTP cache, especially a shared cache, SHOULD use a mechanism, such as NTP [28], to synchronize its clock with a reliable external standard.

Clients SHOULD only send a Date header field in messages that include an entity-body, as in the case of the PUT and POST requests, and even then it is optional. A client without a clock MUST NOT send a Date header field in a request.

The HTTP-date sent in a Date header SHOULD NOT represent a date and time subsequent to the generation of the message. It SHOULD represent the best available approximation of the date and time of message generation, unless the implementation has no means of generating a reasonably accurate date and time. In theory, the date ought to represent the moment just before the entity is generated. In practice, the date can be generated at any time during the message origination without affecting its semantic value.

14.18.1 Clockless Origin Server Operation

Some origin server implementations might not have a clock available. An origin server without a clock MUST NOT assign Expires or Last- Modified values to a response, unless these values were associated with the resource by a system or user with a reliable clock. It MAY assign an Expires value that is known, at or before server configuration time, to be in the past (this allows "pre-expiration" of responses without storing separate Expires values for each resource).

14.19 ETag

The ETag response-header field provides the current value of the entity tag for the requested variant. The headers used with entity tags are described in sections 14.24, 14.26 and 14.44. The entity tag MAY be used for comparison with other entities from the same resource (see section 13.3.3).

ETag = "ETag" ":" entity-tag

Examples:

ETag: "xyzzy"

ETag: W/"xyzzy"

ETag: ""

14.20 Expect

The Expect request-header field is used to indicate that particular server behaviors are required by the client.

```
Expect       =  "Expect" ":" 1#expectation

expectation  =  "100-continue" | expectation-extension

expectation-extension =  token [ "=" ( token | quoted-string )
                         *expect-params ]

expect-params =  ";" token [ "=" ( token | quoted-string ) ]
```

A server that does not understand or is unable to comply with any of the expectation values in the Expect field of a request MUST respond with appropriate error status. The server MUST respond with a 417 (Expectation Failed) status if any of the expectations cannot be met or, if there are other problems with the request, some other 4xx status.

This header field is defined with extensible syntax to allow for future extensions. If a server receives a request containing an Expect field that includes an expectation-extension that it does not support, it MUST respond with a 417 (Expectation Failed) status.

Comparison of expectation values is case-insensitive for unquoted tokens (including the 100-continue token), and is case-sensitive for quoted-string expectation-extensions.

The Expect mechanism is hop-by-hop: that is, an HTTP/1.1 proxy MUST return a 417 (Expectation Failed) status if it receives a request with an expectation that it cannot meet. However, the Expect request-header itself is end-to-end; it MUST be forwarded if the request is forwarded.

Many older HTTP/1.0 and HTTP/1.1 applications do not understand the Expect header.

See section 8.2.3 for the use of the 100 (continue) status.

14.21 Expires

The Expires entity-header field gives the date/time after which the response is considered stale. A stale cache entry may not normally be returned by a cache (either a proxy cache or a user agent cache) unless it is first validated with the origin server (or with an intermediate cache that has a fresh copy of the entity). See section 13.2 for further discussion of the expiration model.

The presence of an Expires field does not imply that the original resource will change or cease to exist at, before, or after that time.

The format is an absolute date and time as defined by HTTP-date in section 3.3.1; it MUST be in RFC 1123 date format:

```
Expires = "Expires" ":" HTTP-date
```

An example of its use is

Expires: Thu, 01 Dec 1994 16:00:00 GMT

Note: if a response includes a Cache-Control field with the max-

age directive (see section 14.9.3), that directive overrides the

Expires field.

HTTP/1.1 clients and caches MUST treat other invalid date formats, especially including the value "0", as in the past (i.e., "already expired").

To mark a response as "already expired," an origin server sends an Expires date that is equal to the Date header value. (See the rules for expiration calculations in section 13.2.4.)

To mark a response as "never expires," an origin server sends an Expires date approximately one year from the time the response is sent. HTTP/1.1 servers SHOULD NOT send Expires dates more than one year in the future.

The presence of an Expires header field with a date value of some time in the future on a response that otherwise would by default be non-cacheable indicates that the response is cacheable, unless indicated otherwise by a Cache-Control header field (section 14.9).

14.22 From

The From request-header field, if given, SHOULD contain an Internet e-mail address for the human user who controls the requesting user agent. The address SHOULD be machine-usable, as defined by "mailbox" in RFC 822 [9] as updated by RFC 1123 [8]:

From = "From" ":" mailbox

An example is:

From: webmaster@w3.org

This header field MAY be used for logging purposes and as a means for identifying the source of invalid or unwanted requests. It SHOULD NOT be used as an insecure form of access protection. The interpretation of this field is that the request is being performed on behalf of the person given, who accepts responsibility for the method performed. In particular, robot agents SHOULD include this header so that the person responsible for running the robot can be contacted if problems occur on the receiving end.

The Internet e-mail address in this field MAY be separate from the Internet host which issued the request. For example, when a request is passed through a proxy the original issuer's address SHOULD be used.

The client SHOULD NOT send the From header field without the user's approval, as it might conflict with the user's privacy interests or their site's security policy. It is strongly recommended that the user be able to disable, enable, and modify the value of this field at any time prior to a request.

14.23 Host

The Host request-header field specifies the Internet host and port number of the resource being requested, as obtained from the original URI given by the user or referring resource (generally an HTTP URL,

as described in section 3.2.2). The Host field value MUST represent the naming authority of the origin server or gateway given by the original URL. This allows the origin server or gateway to differentiate between internally-ambiguous URLs, such as the root "/" URL of a server for multiple host names on a single IP address.

Host = "Host" ":" host [":" port] ; Section 3.2.2

A "host" without any trailing port information implies the default port for the service requested (e.g., "80" for an HTTP URL). For example, a request on the origin server for <http://www.w3.org/pub/WWW/> would properly include:

GET /pub/WWW/ HTTP/1.1

Host: www.w3.org

A client MUST include a Host header field in all HTTP/1.1 request messages . If the requested URI does not include an Internet host name for the service being requested, then the Host header field MUST be given with an empty value. An HTTP/1.1 proxy MUST ensure that any request message it forwards does contain an appropriate Host header field that identifies the service being requested by the proxy. All Internet-based HTTP/1.1 servers MUST respond with a 400 (Bad Request) status code to any HTTP/1.1 request message which lacks a Host header field.

See sections 5.2 and 19.6.1.1 for other requirements relating to Host.

14.24 If-Match

The If-Match request-header field is used with a method to make it conditional. A client that has one or more entities previously obtained from the resource can verify that one of those entities is current by including a list of their associated entity tags in the If-Match header field. Entity tags are defined in section 3.11. The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead. It is also used, on updating requests, to prevent inadvertent modification of the wrong version of a resource. As a special case, the value "*" matches any current entity of the resource.

If-Match = "If-Match" ":" ("*" | 1#entity-tag)

If any of the entity tags match the entity tag of the entity that would have been returned in the response to a similar GET request (without the If-Match header) on that resource, or if "*" is given

and any current entity exists for that resource, then the server MAY perform the requested method as if the If-Match header field did not exist.

A server MUST use the strong comparison function (see section 13.3.3) to compare the entity tags in If-Match.

If none of the entity tags match, or if "*" is given and no current entity exists, the server MUST NOT perform the requested method, and MUST return a 412 (Precondition Failed) response.

This behavior is most useful when the client wants to prevent an updating method, such as PUT, from modifying a resource that has changed since the client last retrieved it.

If the request would, without the If-Match header field, result in anything other than a 2xx or 412 status, then the If-Match header MUST be ignored.

The meaning of "If-Match: *" is that the method SHOULD be performed if the representation selected by the origin server (or by a cache, possibly using the Vary mechanism, see section 14.44) exists, and MUST NOT be performed if the representation does not exist.

A request intended to update a resource (e.g., a PUT) MAY include an If-Match header field to signal that the request method MUST NOT be applied if the entity corresponding to the If-Match value (a single entity tag) is no longer a representation of that resource. This allows the user to indicate that they do not wish the request to be successful if the resource has been changed without their knowledge. Examples:

If-Match: "xyzzy"

If-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"

If-Match: *

The result of a request having both an If-Match header field and either an If-None-Match or an If-Modified-Since header fields is undefined by this specification.

14.25 If-Modified-Since

The If-Modified-Since request-header field is used with a method to make it conditional: if the requested variant has not been modified since the time specified in this field, an entity will not be returned from the server; instead, a 304 (not modified) response will be returned without any message-body.

If-Modified-Since = "If-Modified-Since" ":" HTTP-date

An example of the field is:

If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT

A GET method with an If-Modified-Since header and no Range header requests that the identified entity be transferred only if it has been modified since the date given by the If-Modified-Since header. The algorithm for determining this includes the following cases:

a) If the request would normally result in anything other than a
200 (OK) status, or if the passed If-Modified-Since date is
invalid, the response is exactly the same as for a normal GET.
A date which is later than the server's current time is
invalid.

b) If the variant has been modified since the If-Modified-Since
date, the response is exactly the same as for a normal GET.

c) If the variant has not been modified since a valid If-Modified-Since date, the server SHOULD return a 304 (Not Modified) response.

The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead.

Note: The Range request-header field modifies the meaning of If-Modified-Since; see section 14.35 for full details.

Note: If-Modified-Since times are interpreted by the server, whose clock might not be synchronized with the client.

Note: When handling an If-Modified-Since header field, some servers will use an exact date comparison function, rather than a less-than function, for deciding whether to send a 304 (Not Modified) response. To get best results when sending an If-Modified-Since header field for cache validation, clients are advised to use the exact date string received in a previous Last-Modified header field whenever possible.

Note: If a client uses an arbitrary date in the If-Modified-Since header instead of a date taken from the Last-Modified header for the same request, the client should be aware of the fact that this date is interpreted in the server's understanding of time. The client should consider unsynchronized clocks and rounding problems due to the different encodings of time between the client and server. This includes the possibility of race conditions if the document has changed between the time it was first requested and the If-Modified-Since date of a subsequent request, and the possibility of clock-skew-related problems if the If-Modified-Since date is derived from the client's clock without correction to the server's clock. Corrections for different time bases between client and server are at best approximate due to network latency.

The result of a request having both an If-Modified-Since header field and either an If-Match or an If-Unmodified-Since header fields is undefined by this specification.

14.26 If-None-Match

The If-None-Match request-header field is used with a method to make it conditional. A client that has one or more entities previously obtained from the resource can verify that none of those entities is current by including a list of their associated entity tags in the If-None-Match header field. The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead. It is also used to prevent a method (e.g. PUT) from inadvertently modifying an existing resource when the client believes that the resource does not exist.

As a special case, the value "*" matches any current entity of the resource.

If-None-Match = "If-None-Match" ":" ("*" | 1#entity-tag)

If any of the entity tags match the entity tag of the entity that would have been returned in the response to a similar GET request (without the If-None-Match header) on that resource, or if "*" is given and any current entity exists for that resource, then the server MUST NOT perform the requested method, unless required to do so because the resource's modification date fails to match that supplied in an If-Modified-Since header field in the request. Instead, if the request method was GET or HEAD, the server SHOULD respond with a 304 (Not Modified) response, including the cache- related header fields (particularly ETag) of one of the entities that matched. For all other request methods, the server MUST respond with a status of 412 (Precondition Failed).

See section 13.3.3 for rules on how to determine if two entities tags match. The weak comparison function can only be used with GET or HEAD requests.

If none of the entity tags match, then the server MAY perform the requested method as if the If-None-Match header field did not exist, but MUST also ignore any If-Modified-Since header field(s) in the request. That is, if no entity tags match, then the server MUST NOT return a 304 (Not Modified) response.

If the request would, without the If-None-Match header field, result in anything other than a 2xx or 304 status, then the If-None-Match header MUST be ignored. (See section 13.3.4 for a discussion of server behavior when both If-Modified-Since and If-None-Match appear in the same request.)

The meaning of "If-None-Match: *" is that the method MUST NOT be performed if the representation selected by the origin server (or by a cache, possibly using the Vary mechanism, see section 14.44) exists, and SHOULD be performed if the representation does not exist. This feature is intended to be useful in preventing races between PUT operations.

Examples:

If-None-Match: "xyzzy"

If-None-Match: W/"xyzzy"

If-None-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"

If-None-Match: W/"xyzzy", W/"r2d2xxxx", W/"c3piozzzz"

If-None-Match: *

The result of a request having both an If-None-Match header field and either an If-Match or an If-Unmodified-Since header fields is undefined by this specification.

14.27 If-Range

If a client has a partial copy of an entity in its cache, and wishes to have an up-to-date copy of the entire entity in its cache, it could use the Range request-header with a conditional GET (using either or both of If-Unmodified-Since and If-Match.) However, if the condition fails because the entity has been modified, the client would then have to make a second request to obtain the entire current entity-body.

The If-Range header allows a client to "short-circuit" the second request. Informally, its meaning is `if the entity is unchanged, send me the part(s) that I am missing; otherwise, send me the entire new entity'.

If-Range = "If-Range" ":" (entity-tag | HTTP-date)

If the client has no entity tag for an entity, but does have a Last- Modified date, it MAY use that date in an If-Range header. (The server can distinguish between a valid HTTP-date and any form of entity-tag by examining no more than two characters.) The If-Range header SHOULD only be used together with a Range header, and MUST be ignored if the request does not include a Range header, or if the server does not support the sub-range operation.

If the entity tag given in the If-Range header matches the current entity tag for the entity, then the server SHOULD provide the specified sub-range of the entity using a 206 (Partial content) response. If the entity tag does not match, then the server SHOULD return the entire entity using a 200 (OK) response.

14.28 If-Unmodified-Since

The If-Unmodified-Since request-header field is used with a method to make it conditional. If the requested resource has not been modified since the time specified in this field, the server SHOULD perform the requested operation as if the If-Unmodified-Since header were not present.

If the requested variant has been modified since the specified time, the server MUST NOT perform the requested operation, and MUST return a 412 (Precondition Failed).

If-Unmodified-Since = "If-Unmodified-Since" ":" HTTP-date

An example of the field is:

If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT

If the request normally (i.e., without the If-Unmodified-Since header) would result in anything other than a 2xx or 412 status, the If-Unmodified-Since header SHOULD be ignored.

If the specified date is invalid, the header is ignored.

The result of a request having both an If-Unmodified-Since header field and either an If-None-Match or an If-Modified-Since header fields is undefined by this specification.

14.29 Last-Modified

The Last-Modified entity-header field indicates the date and time at which the origin server believes the variant was last modified.

Last-Modified = "Last-Modified" ":" HTTP-date

An example of its use is

Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT

The exact meaning of this header field depends on the implementation of the origin server and the nature of the original resource. For files, it may be just the file system last-modified time. For entities with dynamically included parts, it may be the most recent of the set of last-modify times for its component parts. For database gateways, it may be the last-update time stamp of the record. For virtual objects, it may be the last time the internal state changed.

An origin server MUST NOT send a Last-Modified date which is later than the server's time of message origination. In such cases, where the resource's last modification would indicate some time in the future, the server MUST replace that date with the message origination date.

An origin server SHOULD obtain the Last-Modified value of the entity as close as possible to the time that it generates the Date value of its response. This allows a recipient to make an accurate assessment of the entity's modification time, especially if the entity changes near the time that the response is generated.

HTTP/1.1 servers SHOULD send Last-Modified whenever feasible.

14.30 Location

The Location response-header field is used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource. For 201 (Created) responses, the Location is that of the new resource which was created by the request. For 3xx responses, the location SHOULD indicate the server's preferred URI for automatic redirection to the resource. The field value consists of a single absolute URI.

Location = "Location" ":" absoluteURI

An example is:

Location: http://www.w3.org/pub/WWW/People.html

Note: The Content-Location header field (section 14.14) differs

from Location in that the Content-Location identifies the original

location of the entity enclosed in the request. It is therefore

possible for a response to contain header fields for both Location

and Content-Location. Also see section 13.10 for cache

requirements of some methods.

14.31 Max-Forwards

The Max-Forwards request-header field provides a mechanism with the TRACE (section 9.8) and OPTIONS (section 9.2) methods to limit the number of proxies or gateways that can forward the request to the next inbound server. This can be useful when the client is attempting to trace a request chain which appears to be failing or looping in mid-chain.

```
Max-Forwards   = "Max-Forwards" ":" 1*DIGIT
```

The Max-Forwards value is a decimal integer indicating the remaining number of times this request message may be forwarded.

Each proxy or gateway recipient of a TRACE or OPTIONS request containing a Max-Forwards header field MUST check and update its value prior to forwarding the request. If the received value is zero (0), the recipient MUST NOT forward the request; instead, it MUST respond as the final recipient. If the received Max-Forwards value is greater than zero, then the forwarded message MUST contain an updated Max-Forwards field with a value decremented by one (1).

The Max-Forwards header field MAY be ignored for all other methods defined by this specification and for any extension methods for which it is not explicitly referred to as part of that method definition.

14.32 Pragma

The Pragma general-header field is used to include implementation- specific directives that might apply to any recipient along the request/response chain. All pragma directives specify optional behavior from the viewpoint of the protocol; however, some systems MAY require that behavior be consistent with the directives.

```
Pragma           = "Pragma" ":" 1#pragma-directive

pragma-directive  = "no-cache" | extension-pragma

extension-pragma  = token [ "=" ( token | quoted-string ) ]
```

When the no-cache directive is present in a request message, an application SHOULD forward the request toward the origin server even if it has a cached copy of what is being requested. This pragma directive has the same semantics as the no-cache cache-directive (see section 14.9) and is defined here for backward compatibility with HTTP/1.0. Clients SHOULD include both header fields when a no-cache request is sent to a server not known to be HTTP/1.1 compliant.

Pragma directives MUST be passed through by a proxy or gateway application, regardless of their significance to that application, since the directives might be applicable to all recipients along the request/response chain. It is not possible to specify a pragma for a specific recipient; however, any pragma directive not relevant to a recipient SHOULD be ignored by that recipient.

HTTP/1.1 caches SHOULD treat "Pragma: no-cache" as if the client had sent "Cache-Control: no-cache". No new Pragma directives will be defined in HTTP.

Note: because the meaning of "Pragma: no-cache as a response

header field is not actually specified, it does not provide a

reliable replacement for "Cache-Control: no-cache" in a response

14.33 Proxy-Authenticate

The Proxy-Authenticate response-header field MUST be included as part of a 407 (Proxy Authentication Required) response. The field value consists of a challenge that indicates the authentication scheme and parameters applicable to the proxy for this Request-URI.

Proxy-Authenticate = "Proxy-Authenticate" ":" 1#challenge

The HTTP access authentication process is described in "HTTP Authentication: Basic and Digest Access Authentication" [43]. Unlike WWW-Authenticate, the Proxy-Authenticate header field applies only to the current connection and SHOULD NOT be passed on to downstream clients. However, an intermediate proxy might need to obtain its own credentials by requesting them from the downstream client, which in some circumstances will appear as if the proxy is forwarding the Proxy-Authenticate header field.

14.34 Proxy-Authorization

The Proxy-Authorization request-header field allows the client to identify itself (or its user) to a proxy which requires authentication. The Proxy-Authorization field value consists of credentials containing the authentication information of the user agent for the proxy and/or realm of the resource being requested.

Proxy-Authorization = "Proxy-Authorization" ":" credentials

The HTTP access authentication process is described in "HTTP Authentication: Basic and Digest Access Authentication" [43] . Unlike Authorization, the Proxy-Authorization header field applies only to the next outbound proxy that demanded authentication using the Proxy- Authenticate field. When multiple proxies are used in a chain, the

Proxy-Authorization header field is consumed by the first outbound proxy that was expecting to receive credentials. A proxy MAY relay the credentials from the client request to the next proxy if that is the mechanism by which the proxies cooperatively authenticate a given request.

14.35 Range

14.35.1 Byte Ranges

Since all HTTP entities are represented in HTTP messages as sequences of bytes, the concept of a byte range is meaningful for any HTTP entity. (However, not all clients and servers need to support byte- range operations.)

Byte range specifications in HTTP apply to the sequence of bytes in the entity-body (not necessarily the same as the message-body).

A byte range operation MAY specify a single range of bytes, or a set of ranges within a single entity.

ranges-specifier = byte-ranges-specifier

byte-ranges-specifier = bytes-unit "=" byte-range-set

byte-range-set = 1#(byte-range-spec | suffix-byte-range-spec)

byte-range-spec = first-byte-pos "-" [last-byte-pos]

first-byte-pos = 1*DIGIT

last-byte-pos = 1*DIGIT

The first-byte-pos value in a byte-range-spec gives the byte-offset of the first byte in a range. The last-byte-pos value gives the byte-offset of the last byte in the range; that is, the byte positions specified are inclusive. Byte offsets start at zero.

If the last-byte-pos value is present, it MUST be greater than or equal to the first-byte-pos in that byte-range-spec, or the byte- range-spec is syntactically invalid. The recipient of a byte-range-set that includes one or more syntactically invalid byte-range-spec values MUST ignore the header field that includes that byte-range- set.

If the last-byte-pos value is absent, or if the value is greater than or equal to the current length of the entity-body, last-byte-pos is taken to be equal to one less than the current length of the entity-body in bytes.

By its choice of last-byte-pos, a client can limit the number of bytes retrieved without knowing the size of the entity.

suffix-byte-range-spec = "-" suffix-length

suffix-length = 1*DIGIT

A suffix-byte-range-spec is used to specify the suffix of the entity-body, of a length given by the suffix-length value. (That is, this form specifies the last N bytes of an entity-body.) If the entity is shorter than the specified suffix-length, the entire entity-body is used.

If a syntactically valid byte-range-set includes at least one byte- range-spec whose first-byte-pos is less than the current length of the entity-body, or at least one suffix-byte-range-spec with a non- zero suffix-length, then the byte-range-set is satisfiable. Otherwise, the byte-range-set is unsatisfiable. If the byte-range-set is unsatisfiable, the server SHOULD return a response with a status of 416 (Requested range not satisfiable). Otherwise, the server SHOULD return a response with a status of 206 (Partial Content) containing the satisfiable ranges of the entity-body.

Examples of byte-ranges-specifier values (assuming an entity-body of length 10000):

- The first 500 bytes (byte offsets 0-499, inclusive): bytes=0-
  499
- The second 500 bytes (byte offsets 500-999, inclusive):
  bytes=500-999
- The final 500 bytes (byte offsets 9500-9999, inclusive):
  bytes=-500

- Or bytes=9500-
- The first and last bytes only (bytes 0 and 9999): bytes=0-0,-1
- Several legal but not canonical specifications of the second 500 bytes (byte offsets 500-999, inclusive):
  bytes=500-600,601-999
  bytes=500-700,601-999

14.35.2 Range Retrieval Requests

HTTP retrieval requests using conditional or unconditional GET methods MAY request one or more sub-ranges of the entity, instead of the entire entity, using the Range request header, which applies to the entity returned as the result of the request:

Range = "Range" ":" ranges-specifier

A server MAY ignore the Range header. However, HTTP/1.1 origin servers and intermediate caches ought to support byte ranges when possible, since Range supports efficient recovery from partially failed transfers, and supports efficient partial retrieval of large entities.

If the server supports the Range header and the specified range or ranges are appropriate for the entity:

- The presence of a Range header in an unconditional GET modifies what is returned if the GET is otherwise successful. In other words, the response carries a status code of 206 (Partial Content) instead of 200 (OK).
- The presence of a Range header in a conditional GET (a request using one or both of If-Modified-Since and If-None-Match, or one or both of If-Unmodified-Since and If-Match) modifies what is returned if the GET is otherwise successful and the condition is true. It does not affect the 304 (Not Modified) response returned if the conditional is false.

In some cases, it might be more appropriate to use the If-Range header (see section 14.27) in addition to the Range header.

If a proxy that supports ranges receives a Range request, forwards the request to an inbound server, and receives an entire entity in reply, it SHOULD only return the requested range to its client. It SHOULD store the entire received response in its cache if that is consistent with its cache allocation policies.

14.36 Referer

The Referer[sic] request-header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained (the "referrer", although the header field is misspelled.) The Referer request-header allows a server to generate lists of back-links to resources for interest, logging, optimized caching, etc. It also allows obsolete or mistyped links to be traced for maintenance. The Referer field MUST NOT be sent if the Request-URI was obtained from a source that does not have its own URI, such as input from the user keyboard.

Referer = "Referer" ":" (absoluteURI | relativeURI)

Example:

Referer: http://www.w3.org/hypertext/DataSources/Overview.html

If the field value is a relative URI, it SHOULD be interpreted relative to the Request-URI. The URI MUST NOT include a fragment. See section 15.1.3 for security considerations.

14.37 Retry-After

The Retry-After response-header field can be used with a 503 (Service Unavailable) response to indicate how long the service is expected to be unavailable to the requesting client. This field MAY also be used with any 3xx (Redirection) response to indicate the minimum time the user-agent is asked wait before issuing the redirected request. The value of this field can be either an HTTP-date or an integer number of seconds (in decimal) after the time of the response.

Retry-After = "Retry-After" ":" (HTTP-date | delta-seconds)

Two examples of its use are

Retry-After: Fri, 31 Dec 1999 23:59:59 GMT

Retry-After: 120

In the latter example, the delay is 2 minutes.

14.38 Server

The Server response-header field contains information about the software used by the origin server to handle the request. The field can contain multiple product tokens (section 3.8) and comments identifying the server and any significant subproducts. The product tokens are listed in order of their significance for identifying the application.

Server = "Server" ":" 1*(product | comment)

Example:

Server: CERN/3.0 libwww/2.17

If the response is being forwarded through a proxy, the proxy application MUST NOT modify the Server response-header. Instead, it SHOULD include a Via field (as described in section 14.45).

Note: Revealing the specific software version of the server might allow the server machine to become more vulnerable to attacks against software that is known to contain security holes. Server implementors are encouraged to make this field a configurable option.

14.39 TE

The TE request-header field indicates what extension transfer-codings it is willing to accept in the response and whether or not it is willing to accept trailer fields in a chunked transfer-coding. Its value may consist of the keyword "trailers" and/or a comma-separated list of extension transfer-coding names with optional accept parameters (as described in section 3.6).

TE = "TE" ":" #(t-codings)

t-codings = "trailers" | (transfer-extension [accept-params])

The presence of the keyword "trailers" indicates that the client is willing to accept trailer fields in a chunked transfer-coding, as defined in section 3.6.1. This keyword is reserved for use with transfer-coding values even though it does not itself represent a transfer-coding.

Examples of its use are:

TE: deflate

TE:

TE: trailers, deflate;q=0.5

The TE header field only applies to the immediate connection. Therefore, the keyword MUST be supplied within a Connection header field (section 14.10) whenever TE is present in an HTTP/1.1 message.

A server tests whether a transfer-coding is acceptable, according to a TE field, using these rules:

1. The "chunked" transfer-coding is always acceptable. If the keyword "trailers" is listed, the client indicates that it is willing to accept trailer fields in the chunked response on behalf of itself and any downstream clients. The implication is that, if given, the client is stating that either all downstream clients are willing to accept trailer fields in the forwarded response, or that it will attempt to buffer the response on behalf of downstream recipients.
   Note: HTTP/1.1 does not define any means to limit the size of a chunked response such that a client can be assured of buffering

the entire response.

2. If the transfer-coding being tested is one of the transfer-codings listed in the TE field, then it is acceptable unless it is accompanied by a qvalue of 0. (As defined in section 3.9, a qvalue of 0 means "not acceptable.")
3. If multiple transfer-codings are acceptable, then the acceptable transfer-coding with the highest non-zero qvalue is preferred. The "chunked" transfer-coding always has a qvalue of 1.

If the TE field-value is empty or if no TE field is present, the only transfer-coding is "chunked". A message with no transfer-coding is always acceptable.

14.40 Trailer

The Trailer general field value indicates that the given set of header fields is present in the trailer of a message encoded with chunked transfer-coding.

Trailer = "Trailer" ":" 1#field-name

An HTTP/1.1 message SHOULD include a Trailer header field in a message using chunked transfer-coding with a non-empty trailer. Doing so allows the recipient to know which header fields to expect in the trailer.

If no Trailer header field is present, the trailer SHOULD NOT include any header fields. See section 3.6.1 for restrictions on the use of trailer fields in a "chunked" transfer-coding.

Message header fields listed in the Trailer header field MUST NOT include the following header fields:

. Transfer-Encoding

. Content-Length

. Trailer

14.41 Transfer-Encoding

The Transfer-Encoding general-header field indicates what (if any) type of transformation has been applied to the message body in order to safely transfer it between the sender and the recipient. This differs from the content-coding in that the transfer-coding is a property of the message, not of the entity.

Transfer-Encoding = "Transfer-Encoding" ":" 1#transfer-coding

Transfer-codings are defined in section 3.6. An example is:

Transfer-Encoding: chunked

If multiple encodings have been applied to an entity, the transfer- codings MUST be listed in the order in which they were applied. Additional information about the encoding parameters MAY be provided by other entity-header fields not defined by this specification.

Many older HTTP/1.0 applications do not understand the Transfer- Encoding header.

14.42 Upgrade

The Upgrade general-header allows the client to specify what additional communication protocols it supports and would like to use if the server finds it appropriate to switch protocols. The server MUST use the Upgrade header field within a 101 (Switching Protocols) response to indicate which protocol(s) are being switched.

Upgrade = "Upgrade" ":" 1#product

For example,

Upgrade: HTTP/2.0, SHTTP/1.3, IRC/6.9, RTA/x11

The Upgrade header field is intended to provide a simple mechanism for transition from HTTP/ 1.1 to some other, incompatible protocol. It does so by allowing the client to advertise its desire to use another protocol, such as a later version of HTTP with a higher major version number, even though the current request has been made using HTTP/1.1. This eases the difficult transition between incompatible protocols by allowing the client to initiate a request in the more commonly supported protocol while indicating to the server that it would like to use a "better" protocol if available (where "better" is determined by the server, possibly according to the nature of the method and/or resource being requested).

The Upgrade header field only applies to switching application-layer protocols upon the existing transport-layer connection. Upgrade cannot be used to insist on a protocol change; its acceptance and use by the server is optional. The capabilities and nature of the application-layer communication after the protocol change is entirely dependent upon the new protocol chosen, although the first action after changing the protocol MUST be a response to the initial HTTP request containing the Upgrade header field.

The Upgrade header field only applies to the immediate connection. Therefore, the upgrade keyword MUST be supplied within a Connection header field (section 14.10) whenever Upgrade is present in an HTTP/1.1 message.

The Upgrade header field cannot be used to indicate a switch to a protocol on a different connection. For that purpose, it is more appropriate to use a 301, 302, 303, or 305 redirection response.

This specification only defines the protocol name "HTTP" for use by the family of Hypertext Transfer Protocols, as defined by the HTTP version rules of section 3.1 and future updates to this specification. Any token can be used as a protocol name; however, it will only be useful if both the client and server associate the name with the same protocol.

14.43 User-Agent

The User-Agent request-header field contains information about the user agent originating the request. This is for statistical purposes, the tracing of protocol violations, and automated recognition of user agents for the sake of tailoring responses to avoid particular user agent limitations. User agents SHOULD include this field with requests. The field can contain multiple product tokens (section 3.8) and comments identifying the agent and any subproducts which form a significant part of the user agent. By convention, the product tokens are listed in order of their significance for identifying the application.

User-Agent = "User-Agent" ":" 1*(product | comment)

Example:

User-Agent: CERN-LineMode/2.15 libwww/2.17b3

14.44 Vary

The Vary field value indicates the set of request-header fields that fully determines, while the response is fresh, whether a cache is permitted to use the response to reply to a subsequent request without revalidation. For uncacheable or stale responses, the Vary field value advises the user agent about the criteria that were used to select the representation. A Vary field value of "*" implies that a cache cannot determine from the request headers of a subsequent request whether this response is the appropriate representation. See section 13.6 for use of the Vary header field by caches.

Vary = "Vary" ":" ("*" | 1#field-name)

An HTTP/1.1 server SHOULD include a Vary header field with any cacheable response that is subject to server-driven negotiation. Doing so allows a cache to properly interpret future requests on that resource and informs the user agent about the presence of negotiation

on that resource. A server MAY include a Vary header field with a non-cacheable response that is subject to server-driven negotiation, since this might provide the user agent with useful information about the dimensions over which the response varies at the time of the response.

A Vary field value consisting of a list of field-names signals that the representation selected for the response is based on a selection algorithm which considers ONLY the listed request-header field values in selecting the most appropriate representation. A cache MAY assume that the same selection will be made for future requests with the same values for the listed field names, for the duration of time for which the response is fresh.

The field-names given are not limited to the set of standard request-header fields defined by this specification. Field names are case-insensitive.

A Vary field value of "*" signals that unspecified parameters not limited to the request-headers (e.g., the network address of the client), play a role in the selection of the response representation. The "*" value MUST NOT be generated by a proxy server; it may only be generated by an origin server.

14.45 Via

The Via general-header field MUST be used by gateways and proxies to indicate the intermediate protocols and recipients between the user agent and the server on requests, and between the origin server and the client on responses. It is analogous to the "Received" field of RFC 822 [9] and is intended to be used for tracking message forwards, avoiding request loops, and identifying the protocol capabilities of all senders along the request/response chain.

```
Via =  "Via" ":" 1#( received-protocol received-by [ comment ] )

received-protocol = [ protocol-name "/" ] protocol-version

protocol-name     = token

protocol-version  = token

received-by       = ( host [ ":" port ] ) | pseudonym

pseudonym         = token
```

The received-protocol indicates the protocol version of the message received by the server or client along each segment of the request/response chain. The received-protocol version is appended to the Via field value when the message is forwarded so that information about the protocol capabilities of upstream applications remains visible to all recipients.

The protocol-name is optional if and only if it would be "HTTP". The received-by field is normally the host and optional port number of a recipient server or client that subsequently forwarded the message. However, if the real host is considered to be sensitive information, it MAY be replaced by a pseudonym. If the port is not given, it MAY be assumed to be the default port of the received-protocol.

Multiple Via field values represents each proxy or gateway that has forwarded the message. Each recipient MUST append its information such that the end result is ordered according to the sequence of forwarding applications.

Comments MAY be used in the Via header field to identify the software of the recipient proxy or gateway, analogous to the User-Agent and Server header fields. However, all comments in the Via field are optional and MAY be removed by any recipient prior to forwarding the message.

For example, a request message could be sent from an HTTP/1.0 user agent to an internal proxy code-named "fred", which uses HTTP/1.1 to forward the request to a public proxy at nowhere.com, which completes the request by forwarding it to the origin server at www.ics.uci.edu. The request received by www.ics.uci.edu would then have the following Via header field:

```
Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)
```

Proxies and gateways used as a portal through a network firewall SHOULD NOT, by default, forward the names and ports of hosts within the firewall region. This information SHOULD only be propagated if explicitly enabled. If not enabled, the received-by host of any host behind the firewall SHOULD be replaced by an appropriate pseudonym for that host.

For organizations that have strong privacy requirements for hiding internal structures, a proxy MAY combine an ordered subsequence of Via header field entries with identical received-protocol values into a single such entry. For example,

Via: 1.0 ricky, 1.1 ethel, 1.1 fred, 1.0 lucy

could be collapsed to

Via: 1.0 ricky, 1.1 mertz, 1.0 lucy

Applications SHOULD NOT combine multiple entries unless they are all under the same organizational control and the hosts have already been replaced by pseudonyms. Applications MUST NOT combine entries which have different received-protocol values.

14.46 Warning

The Warning general-header field is used to carry additional information about the status or transformation of a message which might not be reflected in the message. This information is typically used to warn about a possible lack of semantic transparency from caching operations or transformations applied to the entity body of the message.

Warning headers are sent with responses using:

```
Warning    = "Warning" ":" 1#warning-value
warning-value = warn-code SP warn-agent SP warn-text
                                [SP warn-date]
warn-code  = 3DIGIT
warn-agent = ( host [ ":" port ] ) | pseudonym
                ; the name or pseudonym of the server adding
                ; the Warning header, for use in debugging
warn-text  = quoted-string
warn-date  = <"> HTTP-date <">
```

A response MAY carry more than one Warning header.

The warn-text SHOULD be in a natural language and character set that is most likely to be intelligible to the human user receiving the response. This decision MAY be based on any available knowledge, such as the location of the cache or user, the Accept-Language field in a request, the Content-Language field in a response, etc. The default language is English and the default character set is ISO-8859-1.

If a character set other than ISO-8859-1 is used, it MUST be encoded in the warn-text using the method described in RFC 2047 [14].

Warning headers can in general be applied to any message, however some specific warn-codes are specific to caches and can only be applied to response messages. New Warning headers SHOULD be added after any existing Warning headers. A cache MUST NOT delete any Warning

header that it received with a message. However, if a cache successfully validates a cache entry, it SHOULD remove any Warning headers previously attached to that entry except as specified for specific Warning codes. It MUST then add any Warning headers received in the validating response. In other words, Warning headers are those that would be attached to the most recent relevant response.

When multiple Warning headers are attached to a response, the user agent ought to inform the user of as many of them as possible, in the order that they appear in the response. If it is not possible to inform the user of all of the warnings, the user agent SHOULD follow these heuristics:

- Warnings that appear early in the response take priority over
  those appearing later in the response.
- Warnings in the user's preferred character set take priority
  over warnings in other character sets but with identical warn-
  codes and warn-agents.

Systems that generate multiple Warning headers SHOULD order them with this user agent behavior in mind.

Requirements for the behavior of caches with respect to Warnings are stated in section 13.1.2.

This is a list of the currently-defined warn-codes, each with a recommended warn-text in English, and a description of its meaning.

110 Response is stale MUST be included whenever the returned response is stale.

111 Revalidation failed MUST be included if a cache returns a stale response because an attempt to revalidate the response failed, due to an inability to reach the server.

112 Disconnected operation SHOULD be included if the cache is intentionally disconnected from the rest of the network for a period of time.

113 Heuristic expiration MUST be included if the cache heuristically chose a freshness lifetime greater than 24 hours and the response's age is greater than 24 hours.

199 Miscellaneous warning The warning text MAY include arbitrary information to be presented to a human user, or logged. A system receiving this warning MUST NOT take any automated action, besides presenting the warning to the user.

214 Transformation applied MUST be added by an intermediate cache or proxy if it applies any transformation changing the content-coding (as specified in the Content-Encoding header) or media-type (as specified in the Content-Type header) of the response, or the entity-body of the response, unless this Warning code already appears in the response.

299 Miscellaneous persistent warning The warning text MAY include arbitrary information to be presented to a human user, or logged. A system receiving this warning MUST NOT take any automated action.

If an implementation sends a message with one or more Warning headers whose version is HTTP/1.0 or lower, then the sender MUST include in each warning-value a warn-date that matches the date in the response.

If an implementation receives a message with a warning-value that includes a warn-date, and that warn-date is different from the Date value in the response, then that warning-value MUST be deleted from the message before storing, forwarding, or using it. (This prevents bad consequences of naive caching of Warning header fields.) If all of the warning-values are deleted for this reason, the Warning header MUST be deleted as well.

14.47 WWW-Authenticate

The WWW-Authenticate response-header field MUST be included in 401 (Unauthorized) response messages. The field value consists of at least one challenge that indicates the authentication scheme(s) and parameters applicable to the Request-URI.

WWW-Authenticate = "WWW-Authenticate" ":" 1#challenge

The HTTP access authentication process is described in "HTTP Authentication: Basic and Digest Access Authentication" [43]. User agents are advised to take special care in parsing the WWW-Authenticate field value as it might contain more than one challenge, or if more than one WWW-Authenticate header field is provided, the contents of a challenge itself can contain a comma-separated list of authentication parameters.

Content Negotiation

Most HTTP responses include an entity which contains information for interpretation by a human user. Naturally, it is desirable to supply the user with the "best available" entity corresponding to the request. Unfortunately for servers and caches, not all users have the same preferences for what is "best," and not all user agents are equally capable of rendering all entity types. For that reason, HTTP has provisions for several mechanisms for "content negotiation" -- the process of selecting the best representation for a given response when there are multiple representations available.

Note: This is not called "format negotiation" because the

alternate representations may be of the same media type, but use

different capabilities of that type, be in different languages,

etc.

Any response containing an entity-body MAY be subject to negotiation, including error responses.

There are two kinds of content negotiation which are possible in HTTP: server-driven and agent-driven negotiation. These two kinds of negotiation are orthogonal and thus may be used separately or in combination. One method of combination, referred to as transparent negotiation, occurs when a cache uses the agent-driven negotiation information provided by the origin server in order to provide server-driven negotiation for subsequent requests.

12.1 Server-driven Negotiation

If the selection of the best representation for a response is made by an algorithm located at the server, it is called server-driven negotiation. Selection is based on the available representations of the response (the dimensions over which it can vary; e.g. language, content-coding, etc.) and the contents of particular header fields in the request message or on other information pertaining to the request (such as the network address of the client).

Server-driven negotiation is advantageous when the algorithm for selecting from among the available representations is difficult to describe to the user agent, or when the server desires to send its "best guess" to the client along with the first response (hoping to avoid the round-trip delay of a subsequent request if the "best guess" is good enough for the user). In order to improve the server's guess, the user agent MAY include request header fields (Accept, Accept-Language, Accept-Encoding, etc.) which describe its preferences for such a response.

Server-driven negotiation has disadvantages:

1. It is impossible for the server to accurately determine what might be "best" for any given user, since that would require complete knowledge of both the capabilities of the user agent and the intended use for the response (e.g., does the user want to view it on screen or print it on paper?).
2. Having the user agent describe its capabilities in every request can be both very inefficient (given that only a small percentage of responses have multiple representations) and a potential violation of the user's privacy.
3. It complicates the implementation of an origin server and the algorithms for generating responses to a request.
4. It may limit a public cache's ability to use the same response for multiple user's requests.

HTTP/1.1 includes the following request-header fields for enabling server-driven negotiation through description of user agent capabilities and user preferences: Accept (section 14.1), Accept- Charset (section 14.2), Accept-Encoding (section 14.3), Accept- Language (section 14.4), and User-Agent (section 14.43). However, an origin server is not limited to these dimensions and MAY vary the response based on any aspect of the request, including information outside the request-header fields or within extension header fields not defined by this specification.

The Vary header field can be used to express the parameters the server uses to select a representation that is subject to server- driven negotiation. See section 13.6 for use of the Vary header field by caches and section 14.44 for use of the Vary header field by servers.

12.2 Agent-driven Negotiation

With agent-driven negotiation, selection of the best representation for a response is performed by the user agent after receiving an initial response from the origin server. Selection is based on a list of the available representations of the response included within the header fields or entity-body of the initial response, with each representation identified by its own URI. Selection from among the representations may be performed automatically (if the user agent is capable of doing so) or manually by the user selecting from a generated (possibly hypertext) menu.

Agent-driven negotiation is advantageous when the response would vary over commonly-used dimensions (such as type, language, or encoding), when the origin server is unable to determine a user agent's capabilities from examining the request, and generally when public caches are used to distribute server load and reduce network usage.

Agent-driven negotiation suffers from the disadvantage of needing a second request to obtain the best alternate representation. This second request is only efficient when caching is used. In addition, this specification does not define any mechanism for supporting automatic selection, though it also does not prevent any such mechanism from being developed as an extension and used within HTTP/1.1.

HTTP/1.1 defines the 300 (Multiple Choices) and 406 (Not Acceptable) status codes for enabling agent-driven negotiation when the server is unwilling or unable to provide a varying response using server-driven negotiation.

12.3 Transparent Negotiation

Transparent negotiation is a combination of both server-driven and agent-driven negotiation. When a cache is supplied with a form of the list of available representations of the response (as in agent-driven negotiation) and the dimensions of variance are completely understood by the cache, then the cache becomes capable of performing server- driven negotiation on behalf of the origin server for subsequent requests on that resource.

Transparent negotiation has the advantage of distributing the negotiation work that would otherwise be required of the origin server and also removing the second request delay of agent-driven negotiation when the cache is able to correctly guess the right response.

This specification does not define any mechanism for transparent negotiation, though it also does not prevent any such mechanism from being developed as an extension that could be used within HTTP/1.1.

About Me

I have been working on Rails since May 2006 and I am available for consulting. Please contact me at rack@zepho.com with any contract opportunities, feedback, constructive criticism etc.