# Preface

If you happen to stumble into the web page of this book, I assume you know something about Rack, at least you should heard the name before. Rack is a specification which define a unified interface between the web server and your application, so for an application, if it conforms the Rack spec, it can run in different web servers without modification. A lot of web frameworks have support for Rack, including Rails and Sinatra. In addition to be a spec, Rack is also a rubygem. Although the interface is very simple, you can add a lot of different features by creating middleware, and the Rack gem already includes a lot of them. In this book I'll introduce the Rack specification in detail and invite you to walk through the source code of Rack gem with me. And also I'll show you how Rails is organizing its middleware and how to customize the behavior of your Rails application by adding Rack middleware. After finishing this book you will have a better understanding of how the web server and application interact with each other, and also how the ActionController works in Rails.

This book is divided into following chapters,

*Chapter 1* gives a general introduction to the Rack interface, and you will implement a simple Rack application using a lambda, an object and a method, then in *Chapter 2* we will take a detailed look at the Rack specification. In *Chapter 3* we will introduce the concept of middleware and also learn a simple DSL in Rack gem and how to invoke it from the *rackup* command. In *Chapter 4* and *Chapter 5* we will explore the source code of some important middleware in Rack gem, you will not only learn how to use them but also I'll walk you through the source code to show you how they are implemented. *Chapter 6* talks about the Rack interface in Rails and how the middleware in a Rails application is configured, and then in *Chapter 7* I'll show some examples to implement some features by creating Rack middleware for Rails application. And in addition, in *the Appendix* I'll show you how to explore the source code of the Rack rubygem.

## Target reader

I assume you are a Rails developer who have a basic knowledge of Ruby and Rails, you should know what's a block and what's a lambda, and also you know the structure of a Rails application. But when we explore the Rack and Rails source code, if we meet some uncommon syntax of Ruby, such as parallel assignment, underscore variable, I'll explain it in the text. If you are new to ruby, I recommend you read *Programming Ruby 1.9 & 2.0: The Pragmatic Programmers' Guide* by Dave Thomas, Andy Hunt and Chad Fowler, which is published by Pragmatic Bookshelf. For learning Rails, *Agile Web Development with Rails 4* by Sam Ruby, Dave Thomas and David Heinemeier Hansson is a good start point. However, previous knowledge of Rack is not required. That being said, if you just want to look cool by buying a niche book, you are very welcome :)

# Rubygem version

All examples in this book are tested in ruby 2.0. And also since Rack is a spec, the interface should be consistent across different versions. However this book shows a lot of source code from some rubygems, in such case the implementation may be different if you look into other version.The following list shows the version of the rubygems which are discussed in this book.

- Rack: 1.5
- Ruby on Rails: 4.0

# Conventions

In the text if I want to emphasize something, for example a method, it will be *emphatic*. If I want to run some script from a console, the text will be `monospaced`, the command line is prefixed with prompt $ and what the user typed will be **`bold`**, like following,

```
$ irb
2.0.0-p353 :001 > 3.times { puts "Hello Rack" }
Hello Rack
Hello Rack
Hello Rack
=> 3
2.0.0-p353 :002 >
```

Ok, enough talking, let’s start exploring the world of Rack now!

# Chapter 1 Introduction to Rack

## Why Rack

Rack is a specification which defines a minimal, modular and pluggable interface for developping web applications in Ruby. According to the author of Rack Christian Neukirchen[1], "…I noticed that 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 webserver he wants to use. Hopefully, the framework users are satisfied with that choice. …Rack aims to provide a minimal API for connecting web servers and web frameworks.". Think about without Rack, if you create a web framework, you need to create a handler for each web server, however, if your web framework conforms to the Rack interface, it supports all web servers which has a Rack Handler. And as you can guess, Rack already includes a set of handlers for different web servers:

- Mongrel
- EventedMongrel
- SwiftipliedMongrel
- WEBrick
- FCGI
- CGI
- SCGI
- LiteSpeed
- Thin

And the following web servers include Rack handlers in their distributions:

- Ebb
- Fuzed
- Glassfish v3
- Phusion Passenger (which is mod_rack for Apache and for nginx)
- Puma
- Rainbows!
- Reel
- Unicorn
- unixrack
- uWSGI
- Zbatery

---

[1] http://chneukirchen.org/blog/archive/2007/02/introducing-rack.html

Rack is not just a specification, it's also an implementation. There is a Rack rubygem which already includes a lot of middleware which provide support for functionalities that each framework needs, such as HTTP request parsing, session management etc. So if you create a new framework, you don't need to reinvent the wheel and can put the focus on more interesting part. If a web framework conforms the Rack interface, then it can run in all handlers without modification. Now the following frameworks already include a Rack adapter:

- Camping
- Coset
- Espresso
- Halcyon
- Mack
- Maveric
- Merb
- Racktools::SimpleApplication
- Ramaze
- Ruby on Rails
- Rum
- Sinatra
- Sin
- Vintage
- Waves
- Wee
- … and many others.

Between the server handler and the framework, Rack can be customized to your applications needs using middleware. For example, the Rack rubygem already includes a set of middleware:

- Rack::URLMap, to route to multiple applications inside the same process.
- Rack::CommonLogger, for creating Apache-style logfiles.
- Rack::ShowException, for catching unhandled exceptions and presenting them in a nice and helpful way with clickable backtrace.
- Rack::ContentLength, for set a 'Content-Length' header to the response if one doesn't exist.
- …many others!

So suppose you create a Rack middleware, it can be reused in different web frameworks. For example, if you create an authentication middleware (one already exists[2]!), it can be used in both Rails and Sinatra applications.

[2] https://github.com/hassox/warden

# Why learning Rack

As a RoR or Sinatra web developper, you may be wondering why you need to learn Rack, since normally it's hidden behind the web framework and you may not work with Rack in day-to-day development. However, by learning Rack, you can understand more deeply how your web framework works and interacts with the web server. And also sometimes you have to create middleware for some tasks that can't be done in the web framework. For example, for a RoR controller which accepts and returns JSON, you want to return a meaningful response if the client sends an invalid JSON string, you can't do it in RoR, as before the request reaches the RoR controller the parsing already occured, in such case you have to create a middleware for that. And in addition to that, by learning Rack you can also learn a lot of things about HTTP, and it's also fun. I have spent several months doing the research on Rack and Rails source code, and I must say I have learnt a lot from it.

# Install Rack

Now let's do some experiments on Rack. I assume you already installed Ruby, and since Rack is implemented as a ruby gem, you just type the following command to install Rack,

```
$ gem install rack
```

After installation you can execute following command to check if rack is installed successfully, your version may be different if you installed another version.

```
$ rackup -v
Rack 1.2 (Release: 1.5)
```

# A simple example

The Rack is a very simple specification which fills only on 2 or 3 A4 pages. According to the spec, 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*,
- the *body*.

Since a Rack application is a Ruby object (not a class) that responds to call, so this object can be any of the following options, * A lambda or Proc object * Any object that defines a call method * A method object

## Implement Rack application using lambda

A lambda or Proc is an object which encapsulate a piece of code, and later you can invoke *call* to execute it. Let's create a simple Rack application by using a lambda object in irb. Firstly let's fire up irb,

```
$ irb
2.0.0-p353 :001 >
```

You can type ruby code after the prompt '>' and irb will interprete it directly. So let's create a lambda by typing following code,

```
2.0.0-p353 :001 > l = lambda { |env| [200, { 'Content-Type' => 'text/plain' }, ["Hello from a lambda"]] }
=> #<Proc:0x007fe1a19a5058@(irb):1 (lambda)>
```

In the above code, we create a lambda which accepts a env parameter, and returns an array which has 3 values. The first is an integer 200, which represents the HTTP status code OK, the second element is a hash instance, and it contains one entry, the key is 'Content-Type' string, and value is 'text/plain', which tells the browser that it should render the body as plain text instead of HTML. And the third value is an array, which contains a string object. This is the body of the response, and according to the Rack spec, the body should respond to `each` method, and each element should yield a string. So here we use an array object wich contains String objects. After we create the lambda object we assign it to a local variable l.

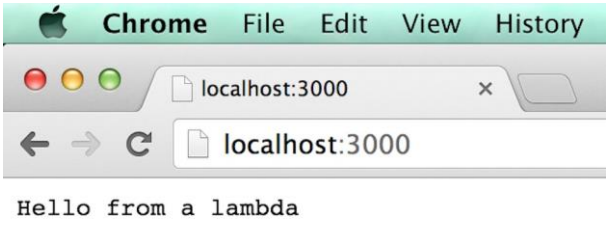
Now lets run this lambda by using the WEBrick handler (in later chapters we will see how to run Rack application by using the rackup command), but before that we should require the Rack rubygem in our system,

```
2.0.0-p353 :002 > require 'rack'
=> true
```

The response *true* tells us that the Rack gem is loaded successfully, now let's run this lambda using the WEBrick handler by calling *Rack::Handler::WEBrick#run* class method,

```
2.0.0-p353 :005 > Rack::Handler::WEBrick.run l, Port: 3000
[2014-02-01 21:59:28] INFO WEBrick 1.3.1
[2014-02-01 21:59:28] INFO ruby 2.0.0 (2013-11-22) [x86_64-darwin13.0.2]
[2014-02-01 21:59:28] WARN TCPServer Error: Address already in use - bind(2)
[2014-02-01 21:59:28] INFO WEBrick::HTTPServer#start: pid=3368 port=3000
```

The Rack rubygem already includes a set of handlers and WEBrick is one of them, and all Rack inherent handlers are in Rack::Handler module. Normally a handler will have a *run* class method, and it accepts two parameters, the first is the rack object, here we pass the local variable l, which points to the lambda we just defined, and the second parameter normally is a hash which you can pass options, here we pass the Port: 3000 to change the WEBrick server port to 3000, otherwise by default it's 8080. After the WEBrick is started we can access our web app by typing *http://localhost:3000* in our favorite browser.




**A Rack application using lambda**

We can see that in the browser it displays the string we specified in the body array: "Hello from a lambda".

## Implement Rack application using an object

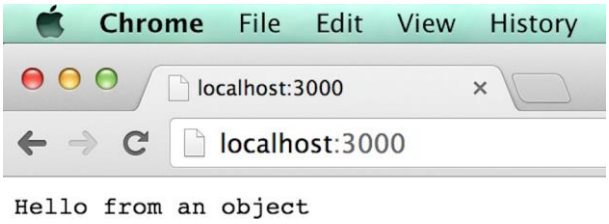
Now let's try implement a Rack application using an object. Firstly lets stop the WEBrick server by typing 'Ctrl+C', it will print some stack trace to say that it was interrupted, but we can ignore it. Now we create a class App which has the instance method `call`,

```
2.0.0-p353 :005 > class App
2.0.0-p353 :006?> def call(env)
2.0.0-p353 :007?> [200, { 'Content-Type' => 'text/plain' }, ["Hello from an object"]]
2.0.0-p353 :008?> end
2.0.0-p353 :009?> end
=> nil
```

we create a class App inside which we define an instance method *call*. According to the Rack specification, it accepts an env parameter, and returns an array which has 3 values. The content of the array is almost the same as the lambda, but in the last element, we specifiy the content as "Hello from an object", now let's start the WEBrick by typing the following command,

```
2.0.0-p353 :010 > Rack::Handler::WEBrick.run App.new, Port: 3000
[2014-02-01 22:58:17] INFO WEBrick 1.3.1
[2014-02-01 22:58:17] INFO ruby 2.0.0 (2013-11-22) [x86_64-darwin13.0.2]
[2014-02-01 22:58:17] WARN TCPServer Error: Address already in use - bind(2)
[2014-02-01 22:58:17] INFO WEBrick::HTTPServer#start: pid=3453 port=3000
```

Since in Rack specification, a Rack application should be an object, not a class which responds to the call method, we created an instance of app and passed it as the first parameter of WEBrick#run by calling App.new. Now we can refresh our browser to access http://localhost:3000 again, and it should display the page as the following figure, we can see that now the content is changed to "Hello from an object".




**A Rack application using object**

## Implement the Rack application using a method

In Object class there is a *method* method, and you can call this method and pass a symbol or string which is the method name, and then the method will return a Method object. And later you can invoke the 'call' on the method object and pass the parameters that the original method accepts, which is like calling the original method directly. So this time we try to implement a Rack application by using a method object. Firstly let's stop the WEBrick server by typing 'Ctrl+C' again, and then create a new class App2 as following,

```
2.0.0-p353 :012 > class App2
2.0.0-p353 :013?> def render env
2.0.0-p353 :014?> [200, {'Content-Type' => 'text/plain'}, ["Hello from a method"]]
2.0.0-p353 :015?> end
2.0.0-p353 :016?> end
=> nil
```

Here we defined another class App2, and instead of defining a call method, we create a method named *render*, it also accepts an env parameter, and returns an array which has 3 elements. In the last array element, we specify the string value as "Hello from a method".

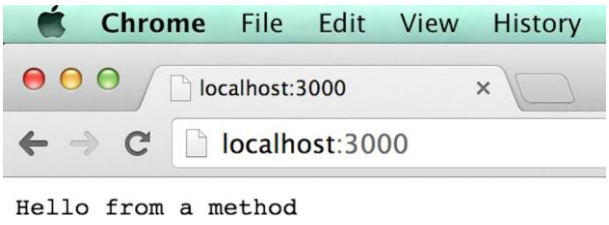
Now lets try to create an instance of App2 and call method on that object and pass :render to get the method object which points to *render* method,

```
2.0.0-p353 :018 > m = App2.new.method(:render)
=> #<Method: App2#render>
```

We can see that it returns an instance of Method object, and now we start the WEBrick server again pass this method object,

```
2.0.0-p353 :010 > Rack::Handler::WEBrick.run m, Port: 3000
[2014-02-01 23:20:02] INFO WEBrick 1.3.1
[2014-02-01 23:20:02] INFO ruby 2.0.0 (2013-11-22) [x86_64-darwin13.0.2]
[2014-02-01 23:20:02] WARN TCPServer Error: Address already in use - bind(2)
[2014-02-01 23:20:02] INFO WEBrick::HTTPServer#start: pid=3598 port=3000
```

Now when we refresh our browser again, we can see that now it displays "Hello from a method", as we expected.




**A Rack application using method**

# Summary

You see, a Rack application is that simple, right? In this chapter we gave a gentle introduction to the Rack interface, which is just the *call* method and its signature, and then we implement a Rack application by using a lambda, an object and a method. In next chapter we will have a detail look at the Rack specification. Don't worry, as I said the Rack specification is only 2 or 3 A4 pages, so it's not that difficult.

# Chapter 2 Rack Specification

In last chapter we had an overlook at the Rack interface, so in this chapter we will have a detailed look at the Rack specification. The specification can be viewed at the Rack website[3]. If a web framework or application conforms to the spec, then it can run without modification under all web servers which have a Rack handler. Let's repeat the requirements of Rack application here, 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 which represents the HTTP response:

- the *status*,
- the *headers*,
- the *body*.

Now let's have a detailed look at the environment and the response.

## The Environment

The environment must be an instance of Hash that includes CGI-like headers. And also the application is free to modifiy the environment.

### CGI and FastCGI

CGI (Common Gateway Interface) is a protocl for web servers to interfacing with external applications. CGI applications run in separate processes, which are created at the start of each request and torn down at the end. When the web server creates the process, it will pass a set of *environment variables* to the CGI program, for example, the server specific variable SERVER_-NAME which denotes the hostname of the server, and request specific parameter REQUEST_-METHOD which is the method of the HTTP request. The CGI program then get the variables from its environment and execute its logic. CGI programs can be implemented by interpreted languages such as Perl or compiled languages such as C/C++.

FastCGI is a variation of CGI protocol, which aims to reduce the overhead of creating processes. Instead of creating a new process for each request, FastCGI uses persistent processes to handle a series of requests. More details can be got from Wikipedia . So for Rack environment it's a Hash object which contains headers like the CGI environment variables.

http://en.wikipedia.org/wiki/Common_Gateway_Interface

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

Now let's try to create a Rack application which displays all variables in the environment hash. This time we don't use irb, instead we create a file named print_env.rb which has following content,

**A Rack application which prints the env hash**

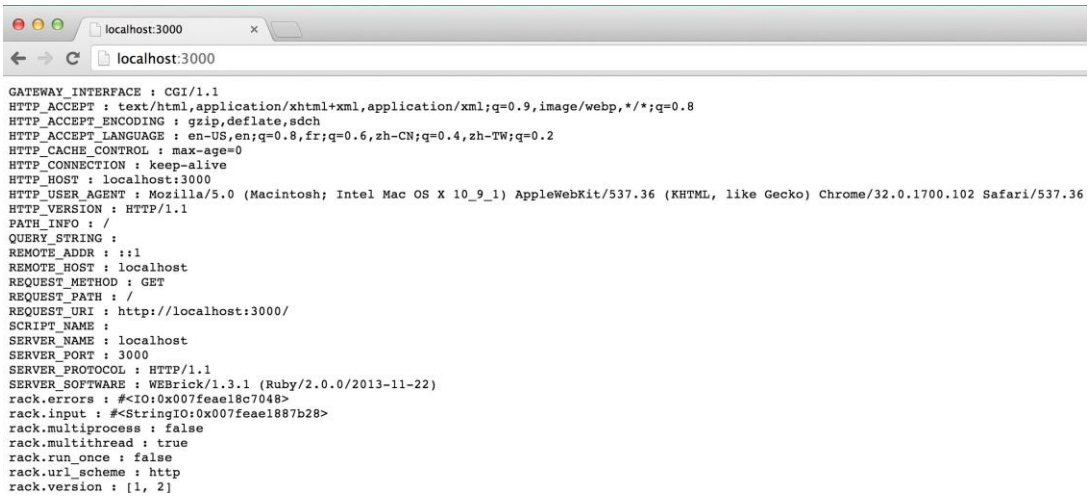
```
1  require 'rack'
2
3  class PrintEnv
4
5    def call(env)
6      body = []
7      env.sort_by {|key, value| key }.each do |key, value|
8        body << "#{key} : #{value} \n"
9      end
10     [200, {'Content-Type' => 'text/plain'}, body]
11   end
12
13 end
14
15 Rack::Handler::WEBrick.run PrintEnv.new, Port: 3000
```

In the script on line 1 firstly we require the Rack rubygem since the WEBrick handler is inside this gem. And then we create a class PrintEnv. In this class we define an instance method call, since this method conforms the Rack spec, it accepts a single parameter *env*. In this method on line 6, we initialize a local variable *body* which is an empty array. And then to make it easier to check the variables in the environment, we firstly call *env.sort_by {|key, value| key }* to sort by the vairable names. Then we iterate the hash using its *each* method. In the block of each method, we initialize a string which prints the content of the key and value and then appends the string to the body array. And lastly we return an array which contains 3 elements, the 200 HTTP status code, a hash which contains a header of 'Content-Type', and the body. And on line 15 we run the Rack application by calling Rack::Handler::WEBrick.run method and set the port to 3000.

Now let's run the script. In the console change to the folder where this script resides, and then run it using following command,

```
$ ruby print_env.rb
[2014-02-18 21:58:45] INFO WEBrick 1.3.1
[2014-02-18 21:58:45] INFO ruby 2.0.0 (2013-11-22) [x86_64-darwin13.0.2]
[2014-02-18 21:58:45] WARN TCPServer Error: Address already in use - bind(2)
[2014-02-18 21:58:45] INFO WEBrick::HTTPServer#start: pid=31475 port=3000
```

Now in our favorite browser we type http://localhost:3000, and the page is like the following figure,

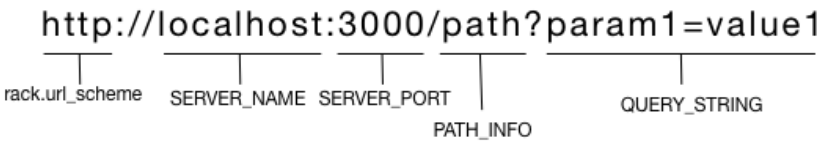



**A Rack application which prints all variables in env**

We can see that in the environment there is already a set of variables, which is actually populated by our WEBrick handler:

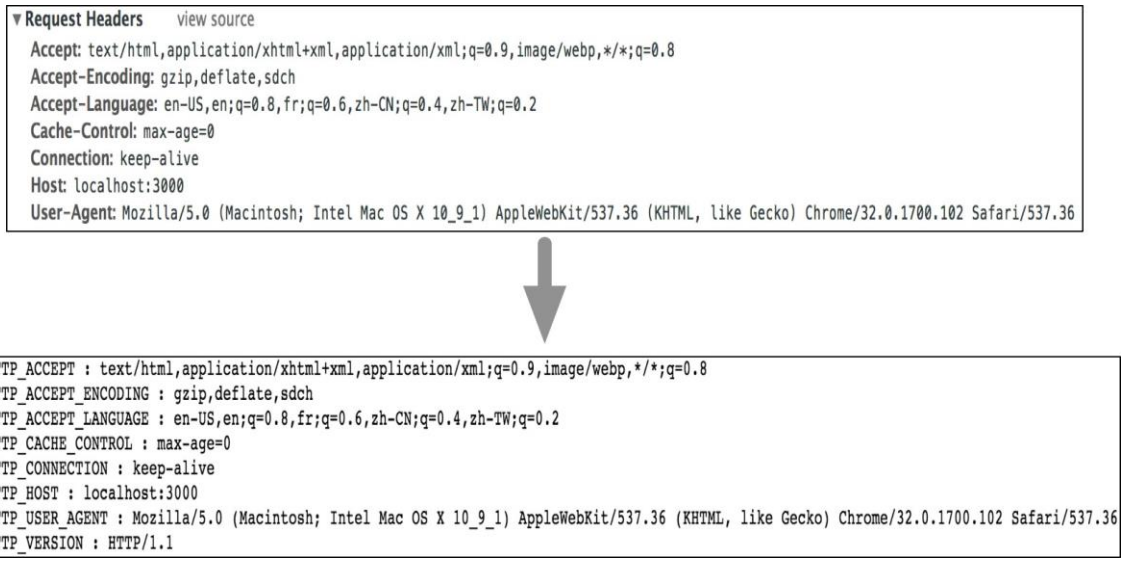
- a set of CGI like variables such as REQUEST_PATH, SERVER_NAME, SERVER_PORT
- all HTTP headers sent by the client
- a set of 'rack.*' variables

We will give an explanation of these variables later. Now let's focus on some important variables for developers, the PATH_INFO and QUERY_STRING. If the browser accesses http://localhost:3000, the value of PATH_INFO is '/' and the QUERY_STRING is an empty string. Now we type http://localhost:3000/path?param1=value1, we can see that the PATH_INFO now changes to 'path', and the QUERY_STRING is now 'param1=value1' which is after the question mark '?'. When the server handler receives the HTTP request, it will parse the URL and set different parts to different variables in the environment like the following figure,




**HTTP path and parameters are decomposed in different vairables**

And then let's look at the headers, when the browser sends a HTTP request to the server, it could include a set of HTTP headers. If you use Chrome, you can view the HTTP requests and responses by using the Chrome Developer Tools (You can use Firebug plugin if you use Firefox). And as the following figure, we can see that the Rack handler maps the HTTP headers to a set of variables prefixed with *HTTP_*. And also all header names are transformed to upper case and the dash in the names is transformed to underscore.




**HTTP headers are transformed to variables starting with HEADER_**

Now let's see what the specification is saying. In the Rack specification, the environment is required to include the following variables,

| Variable name | Variable meaning |
|---|---|
| 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 originating from a URL. |
| QUERY_STRING | The portion of the request URL that follows the '?', if any. May be empty, but is always required! |

| Variable name | Variable meaning |
|---|---|
| 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. The presence or absence of these variables should correspond with the presence or absence of the appropriate HTTP header in the request. See RFC3875 section 4.1.18[4] for specific behavior |

In addition to this, the Rack environment must include the following Rack-specific variables, all variables have a *rack.* prefix.

| Variable name | Variable meaning |
|---|---|
| rack.version | The Array representing this version of Rack. See Rack::VERSION, that corresponds to the version of this SPEC. |
| rack.url_scheme | *http* or *https*, depending on the request URL. |
| rack.input | See the input stream section below |
| rack.errors | See the rack.errors section below |
| 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). |

## the input stream

The vairable *rack.input* is an input stream. The input stream is an IO-like object which contains the raw HTTP POST data. The input stream must respond to 4 methods: *gets*, *read*, *each* and *rewind*.

[4] https://tools.ietf.org/html/rfc3875#section-4.1.18

The input stream doesn't have to be an IO object, but the above four methods should respect the semantic of the ones in IO class. You can check the documentation of these methods on ruby-doc.org[5], so I won't repeat it here.

In Rack rubygem, there is a class Rack::Request, which wraps the env and provides a convinient interface for accessing variables in the env. We will have a detailed look at this class later. But now let's focus on its *POST* method, this method parses the POST data and return all parameters within.

**Rack::Request#POST shows how to use *rack.input* variable**

```
1  module Rack
2    class Request
3      def initialize(env)
4        @env = env
5      end
6
7      def POST
8        if @env["rack.input"].nil?
9          raise "Missing rack.input"
10       elsif @env["rack.request.form_input"].eql? @env["rack.input"]
11         @env["rack.request.form_hash"]
12       elsif form_data? || parseable_data?
13         @env["rack.request.form_input"] = @env["rack.input"]
14         unless @env["rack.request.form_hash"] = parse_multipart(env)
15           form_vars = @env["rack.input"].read
16
17           @env["rack.request.form_vars"] = form_vars
18           @env["rack.request.form_hash"] = parse_query(form_vars)
19
20           @env["rack.input"].rewind
21         end
22         @env["rack.request.form_hash"]
23       else
24         {}
25       end
26     end
27   end
28 end
```

The *initialize* method accepts the env param and set it as an instance variable *@env*, so later we could access the environment inside the class. Then let's digest the *POST* method little by little. On line 8 it firstly check that if the *rack.input* variable is set, if not, it throws an exception since this variable is mandatory in specification. Then on line 10 it checks if the *rack.request.form_input* variable is equal to *rack.input* variable, if yes, it means that the POST data is already

[5] http://ruby-doc.org/core-2.0/IO.html

parsed and stored in *rack.request.form_hash* variable, so we just return that variable directly. If the *rack.request.form_input* is not equal to *rack.input* variable, it means we don't parse the form data yet, so on line 12 if there is form data and it's parseable, it will first try to parse the HTTP request as multipart by calling *parse_multipart*, if the request is not multipart, this method will return nil. Then the real work on *rack.input* begins. On line 15, it calls *@env["rack.input"].read* and return the read content to the local variable *form_vars*. According to the ruby doc[6], if the *read* method has no arguments, it will read all content from the input stream until it reaches EOF, so the *form_vars* variable contains all form data as a string. And then on line 17 it sets the *form_vars* to the variable *rack.request.form_vars* in the environment. And then it calls the method *parse_query* to parse the form data into a hash structure and set the hash to the variable *rack.request.form_hash* in the environment. Then on line 20 it calls *@env["rack.input"].rewind* to reset the current read position to the beginning of the input stream, so later the input stream could be read again. And lastly on line 22 it returns the already parsed *rack.request.form_hash* variable.

## The Error stream

The *rack.errors* variable points to an output stream which is to output error messages during the processing of the request. It must respond to *puts*, *write* and *flush* methods. In the Rack specification it defines the following rules,

- *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.

In Rack rubygem there is a middleware called Rack::ShowExceptions which catches all exceptions raised from the app it wraps, then it outputs the backtrace to the error stream. We will introduce what's middleware in next chapter but now let's just see how to use the *rack.errors* variable.

**Rack::ShowExceptions shows how to use the *rack.errors* to output backtrace**

```
1  module Rack
2    class ShowExceptions
3
4      def call(env)
5        @app.call(env)
6      rescue StandardError, LoadError, SyntaxError => e
7        exception_string = dump_exception(e)
8
9        env["rack.errors"].puts(exception_string)
10       env["rack.errors"].flush
11
```

[6] http://ruby-doc.org/core-2.0/IO.html#method-i-read