A Quick Introduction to Rack

Rack aims to provide a minimal API for connecting web servers supporting Ruby (like WEBrick, Mongrel etc.) and Ruby web frameworks (like Rails, Sinatra etc.).

by RubyLearning
http://rubylearning.com/blog/

A Quick Introduction to Rack

What's Rack?

In the words of the author of Rack – **Christian Neukirchen**:
Rack aims to provide a minimal API for connecting web servers supporting Ruby (like WEBrick, Mongrel etc.) and Ruby web frameworks (like Rails, Sinatra etc.).

Web frameworks such as Sinatra are built on top of Rack or have a Rack interface for allowing web application servers to connect to them.

The premise of Rack is simple – it just allows you to easily deal with HTTP requests.

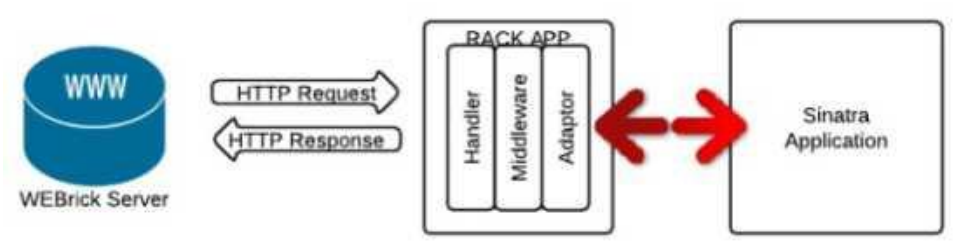
HTTP is a simple protocol: it basically describes the activity of a client sending a HTTP request to a server and the server returning a HTTP response. Both HTTP request and HTTP response in turn have very similar structures. A HTTP request is a triplet consisting of a method and resource pair, a set of headers and an optional body while a HTTP response is in triplet consisting of a response code, a set of headers and an optional body.

Rack maps closely to this. A Rack application is a Ruby object that has a `call` method, which has a single argument, the *environment*, (corresponding to a HTTP request) and returns an array of 3 elements, *status*, *headers* and *body* (corresponding to a HTTP response).

Rack includes *handlers* that connect Rack to all these web application servers (WEBrick, Mongrel etc.).

Rack includes *adapters* that connect Rack to various web frameworks (Sinatra, Rails etc.).

Between the server and the framework, Rack can be customized to your applications needs using *middleware*. The fundamental idea behind Rack middleware is – come between the calling client and the server, process the HTTP request before sending it to the server, and processing the HTTP response before returning it to the client.



A Rack App

Documentation

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

Installing Rack

Note that I have Ruby 1.9.2 installed on a Windows box and all the programs in this article have been tested using that.

Let's check if we already have rack with us. Open a command window and type:

```
irb --simple-prompt
>> require 'rack'
=> true
>>
```

Yes, rack's already there on your machine. If rack's not there you will get an error like:

```
LoadError: no such file to load -- rack
```

You can install rack by opening a new command window and typing:

```
gem install rack
```

A quick visit to Ruby's proc object

Remember the proc object from Ruby? *Blocks are not objects*, but they can be converted into objects of class `Proc`. This can be done by calling the `lambda` method of the class `Object`. A block created with `lambda` acts like a Ruby method. The class `Proc` has a method `call` that invokes the block.

In irb type:

```
>> my_rack_proc = lambda {puts 'Rack Intro'}
=> #<Proc:0x1fc9038@(irb):2(lambda)>
>> # method call invokes the block
?> my_rack_proc.call
Rack Intro
=> nil
>>
```

A simple Rack app – my_rack_proc

As mentioned earlier, our simple Rack application is a Ruby object (not a class) that responds to `call` and takes exactly one argument, the *environment*. The *environment* must be a true instance of `Hash`.

The app should return an Array of exactly three values:
the *status* code (it must be greater than or equal to 100),
the *headers* (must be a hash), and the *body* (the body commonly is an Array of Strings, the application instance itself, or a File-like object. The *body* must respond to method `each` and must only yield `String` values.) Let us create our new `proc` object. Type:

```
>> my_rack_proc = lambda { |env| [200, {}, ["Hello. The
time is #{Time.now}"]] }
=> #<Proc:0x1f4c358@(irb):5(lambda)>
>>
```

Now we can call the proc object my_rack_proc with
the `call` method. Type:

```
>> my_rack_proc.call({})
=> [200, {}, ["Hello. The time is 2011-10-24 09:18:56
+0530"]]
>>
```

my_rack_proc is our single line Rack application.
In the above example, we have used an empty hash for headers.
Instead, let's have something in the header as follows:

```
>> my_rack_proc = lambda { |env| [200, {"Content-Type" =>
"text/plain"}, ["Hello. The time is #{Time.now}"]] }
=> #<Proc:0x1f4c358@(irb):5(lambda)>
>>
```

Now we can call the proc object my_rack_proc with
the `call` method. Type:

```
>> my_rack_proc.call({})
=> [200, {"Content-Type" => "text/plain"}, ["Hello. The
time is 2011-10-24 09:18:56 +0530"]]
>>
```

We can run our Rack application (**my_rack_proc**) with any of the Rack handlers.

To look at the Rack handlers available, in irb type:

```
>> Rack::Handler.constants
=> [:CGI, :FastCGI, :Mongrel, :EventedMongrel,
:SwiftipliedMongrel, :WEBrick, :LSWS, :SCGI, :Thin]
>>
```

To get a handler for say **WEBrick** (the default **WEBrick**, web application server, that comes along with Ruby), type:

```
>> Rack::Handler::WEBrick
=> Rack::Handler::WEBrick
>>
```

All of these handlers have a common method called `run` to run all the Rack based applications.

```
>> Rack::Handler::WEBrick.run my_rack_proc
[2011-10-24 10:00:45] INFO WEBrick 1.3.1
[2011-10-24 10:00:45] INFO ruby 1.9.2 (2011-07-09) [i386-
mingw32]
[2011-10-24 10:00:45] INFO WEBrick::HTTPServer#start:
pid=1788 port=80
```

Open a browser window and type the url: http://localhost/
In your browser window, you should see a string, something like this:

```
Hello. The time is 2011-10-24 10:02:20 +0530
```

Note: If you already have something running at port 80, you can run this app at a different port, say 9876. Type:

```
>> Rack::Handler::WEBrick.run my_rack_proc, :Port => 9876
[2011-10-24 11:32:21] INFO  WEBrick 1.3.1
[2011-10-24 11:32:21] INFO  ruby 1.9.2 (2011-07-09) [i386-
mingw32]
[2011-10-24 11:32:21] INFO  WEBrick::HTTPServer#start:
pid=480 port=9876
```

Open a browser window and type the url: http://localhost:9876/
In your browser window, you should see a string, something like this:

```
Hello. The time is 2011-10-24 10:02:20 +0530
```

Another Rack app – my_method

A Rack app need not be a lambda; it could be a method. Type:

```
>> def my_method env
>> [200, {}, ["method called"]]
>> end
=> nil
```

We declare a method my_method that takes an argument env. The method returns three values.

Next type:

```
>> Rack::Handler::WEBrick.run method(:my_method)
[2011-10-24 14:32:05] INFO  WEBrick 1.3.1
[2011-10-24 14:32:05] INFO  ruby 1.9.2 (2011-07-09) [i386-mingw32]
[2011-10-24 14:32:05] INFO  WEBrick::HTTPServer#start: pid=1644 port=80
```

Open a browser window and type the url: http://localhost/
In your browser window, you should see something like this:

```
method called
```

`Method` objects are created by `Object#method`. They are associated with a particular object (not just with a class). They may be used to invoke the method within the object. The `Method.call` method invokes the method with the specified arguments, returning the method's return value.

Press **Ctrl-C** in irb to stop the **WEBrick** server.

Using rackup

The rack gem comes with a bunch of useful stuff to make life easier for a rack application developer. rackup is one of them.

rackup is a useful tool for running Rack applications. 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.

To use rackup, you'll need to supply it with a rackup config file. By convention, you should use .ru extension for a rackup config file. Supply it a run RackObject and you're ready to go:

```
$ rackup config.ru
```

By default, rackup will start a server on port 9292.

To view rackup help, open a command window and type:

```
$ rackup –help
```

Let us create a config.ru file that contains the following:

```
run lambda { |env| [200, {"Content-Type" => "text/plain"},
["Hello. The time is #{Time.now}"]] }
```

This file contains `run`, which can be called on anything that responds to a `.call`.

To run our rack app, in the same folder that contains config.ru, type:

```
$ rackup config.ru
[2011-10-24 15:18:03] INFO WEBrick 1.3.1
[2011-10-24 15:18:03] INFO ruby 1.9.2 (2011-07-09) [i386-
mingw32]
[2011-10-24 15:18:03] INFO WEBrick::HTTPServer#start:
pid=3304 port=9292
```

Open a browser window and type the url: http://localhost:9292/
In your browser window, you should see something like this:

```
Hello. The time is 2011-10-24 15:18:10 +0530
```

Press **Ctrl-C** to stop the **WEBrick** server.

Now let's move our application from the config.ru file to my_app.rb file as follows:

```
# my_app.rb
class MyApp
  def call env
    [200, {"Content-Type" => "text/html"}, ["Hello Rack
Participants"]]
  end
end
```

Also, our config.ru will change to:

```
require './my_app'
run MyApp.new
```

To run our rack app, in the same folder that contains config.ru, type:

```
rackup config.ru
[2011-10-25 06:18:16] INFO  WEBrick 1.3.1
[2011-10-25 06:18:16] INFO  ruby 1.9.2 (2011-07-09) [i386-
mingw32]
[2011-10-25 06:18:16] INFO  WEBrick::HTTPServer#start:
pid=2224 port=9292
```

Open a browser window and type the url: http://localhost:9292/
In your browser window, you should see something like this:

```
Hello Rack Participants
```

Press **Ctrl-C** to stop the **WEBrick** server.

Deploying Pure Rack Apps to Heroku

We shall now download and install Git. The precompiled packages are available here: http://git.or.cz/

Select the relevant package for your operating system.

Git still has some issues on the Windows platform but for normal usage the msysgit package shouldn’t let you down. Download and install it from the url:
http://code.google.com/p/msysgit/downloads/list

Select the current version available.

Install by running the EXE installer. When asked, it is recommended for Windows users to use the “Use Git Bash only” option.

Please ensure that you are connected to the internet and then create an account on Heroku (obviously do this only once) if you don’t have one – http://heroku.com/signup.

Open a new command window to install the Heroku gem file. Type:

```
gem install heroku
```

Next, create a new folder, say `rackheroku`. Assuming that you have Git installed, open a Bash shell in that folder. You now need to identify yourself to Git (you need to do this only once). With the bash shell still open type in the following:

```
git config --global user.name "Your name here"
git config --global user.email "Your email id"
```

Git does not allow accented characters in user name. This will set the info stored when you commit to a Git repository. Git has now been set up.

The first step in using Git is to create your SSH Key. This will be used to secure communications between your machine and other machines, and to identify your source code changes. (If you already have an SSH key for other reasons, you can use it here, there is nothing Git-specific about this.)

To create our ssh key, open a new command window and type:

```
$ ssh-keygen -C "username@email.com" -t rsa
```

(with your own email address, of course).

Accept the default key file location. When prompted for a passphrase, make one up and enter it. If you feel confident that your own machine is secure, you can use a blank passphrase, for more convenience and less security. Note where it told you it stored the file. On my Windows box, it was stored in "C:\Documents and Settings\A\.ssh\". Memorize your passphrase carefully. If you forget it, you will NOT be able to recover it.

Open the public file id_rsa.pub with a text editor. The text in there is your "public SSH key".

Upload your public key (do it only once):

```
$ heroku keys:add
```

You'll be prompted for your username and password the first time you run a heroku command; they'll be saved on ~/.heroku/credentials so you won't be prompted on future runs. It will also upload your public key to allow you to push and pull code.

In order for Heroku to know what to do with your Rack app, create a config.ru (ru stands for Rack up) in the rackheroku folder. The contents are:

```
require './my_app'
run MyApp.new
```

Also, copy the my_app.rb file to the rackheroku folder. It’s contents are:

```
# my_app.rb
class MyApp
  def call env
    [200, {"Content-Type" => "text/html"}, ["Hello Rack
Participants"]]
  end
end
```

In the already open command window, we will install bundler. Type:

```
gem install bundler
```

Close the command window. Next, we will install the required gems (if any) via bundler. In the already open Git Bash shell for folder rackheroku type:

```
$ bundle init
Writing new Gemfile to c:/rackheroku/Gemfile
```

Edit the created Gemfile with your preferred text editor to let it look like this:

```
source "http://rubygems.org"
gem 'rack'
```

Now we need to tell Bundler to check if we're missing the gems our application depends on, if so, tell it to install them. In your open Bash shell type:

```
$ bundle check
The Gemfile's dependencies are satisfied
```

Finally in the open Bash shell, type:

```
$ bundle install
```

This will ensure all gems specified on Gemfile, together with their dependencies, are available for your application. Running “bundle install” will also generate a “Gemfile.lock” file. The Gemfile.lock ensures that your deployed versions of gems on Heroku match the version installed locally on your development machine.

Next we set up our local app to use Git. Type:

```
$ git init
$ git add .
$ git commit -m "Rack app first commit"
```

Let's create our Rack app on Heroku. Type:

```
$ heroku create
Creating quiet-winter-3741... done, stack is bamboo-mri-
1.9.2
http://quiet-winter-3741.heroku.com/ |
git@heroku.com:quiet-winter-3741.git
Git remote heroku added
```

The app has been created and two URLs are provided. One is for the web face of your new app i.e. http://quiet-winter-3741.heroku.com/ If you visit that URL now, you'll see a standard welcome page, until you push your application up. The other one is for the Git repository that you will push your code to. Normally you would need to add this as a git remote; the "heroku create" command has done this for you automatically. Do note that the output from the create command *will be different for each one of you.*

Now let us deploy our code to Heroku. Type:

```
git push heroku master
```

At this stage we can rename our app to rackheroku. Type:

```
$ heroku rename rackheroku
http://rackheroku.heroku.com/ |
git@heroku.com:rackheroku.git
Git remote heroku updated
```

Our app is now deployed to Heroku. Open a new browser window and type http://rackheroku.heroku.com/
In the browser window, you should see:

```
Hello Rack Participants from across the globe
```

Using Rack middleware

We mentioned earlier that between the server and the framework, Rack can be customized to your applications needs using *middleware*. The fundamental idea behind Rack middleware is – come between the calling client and the server, process the HTTP request before sending it to the server, and processing the HTTP response before returning it to the client.

Rackup also has a `use` method that accepts a middleware. Let us use one of Rack's built-in middleware.

You must have observed that every time we make a change to our config.ru or my_app.rb files, we need to restart the WEBrick server.

To avoid this, let's use one of Rack's built-in middleware – `Rack::Reloader`. Edit config.ru to have:

```
require './my_app'
use Rack::Reloader
run MyApp.new
```

In the same folder, we have the my_app.rb file. It's contents are:

```
# my_app.rb
class MyApp
  def call env
    [200, {"Content-Type" => "text/html"}, ["Hello Rack
Participants from across the globe"]]
  end
end
```

To run our rack app, in the same folder that contains config.ru, type:

```
$ rackup config.ru
```

Open a browser window and type the url: http://localhost:9292/. In your browser window, you should see something like this:

```
Hello Rack Participants from across the globe
```

Modify my_app.rb and instead of “Hello Rack Participants from across the globe” type “Hello Rack Participants from across the world!

Refresh the browser window and you should see:

```
Hello Rack Participants from across the world!
```

There was no need to re-start the WEBrick server.

You can now press **Ctrl-C** to stop the **WEBrick** server.

Writing our own Middleware

For all practical purposes, a middleware is a rack application that wraps up an inner application.

Our middleware is going to do something very simple. We will append some text to the body being sent by our inner application. Let us write our own middleware. We will create a middleware class called `MyRackMiddleware`. Here’s the skeleton program:

```
class MyRackMiddleware
  def initialize(appl)
    @appl = appl
  end
  def call(env)
  end
end
```

In the code above, to get the original body, we initialize the class `MyRackMiddleware` by passing in the inner application (appl).

Next we need to call this appl:

```
def call(env)
  status, headers, body = @appl.call(env) # we now call the
inner application
end
```

In the above code, we now have access to the original body, which we can now modify. Rack does not guarantee you that the body would be a string. It could be an object too. However, we are guaranteed that if we call `.each` on the body, everything it returns will be a string. Let's use this in our `call` method:

```
def call(env)
  status, headers, body = @appl.call(env)
  append_s = "... greetings from RubyLearning!!"
  [status, headers, body << append_s]
end
```

Here’s our completed middleware class:

```
class MyRackMiddleware
  def initialize(appl)
    @appl = appl
  end
  def call(env)
    status, headers, body = @appl.call(env)
    append_s = "... greetings from RubyLearning!!"
    [status, headers, body << append_s]
  end
end
```

Our my_app.rb file remains the same. It’s contents are:

```
# my_app.rb
class MyApp
  def call(env)
    [200, {"Content-Type" => "text/html"}, ["Hello Rack
Participants from across the globe"]]
  end
end
```

Edit config.ru to have:

```
require './my_app'
require './myrackmiddleware'
use Rack::Reloader
use MyRackMiddleware
run MyApp.new
```

To run our rack app, in the same folder that contains config.ru, type:

```
$ rackup config.ru
```

Open a browser window and type the url: http://localhost:9292/.

In your browser window, you should see something like this:

```
Hello Rack Participants from across the globe... greetings
from RubyLearning!!
```

Rack and Sinatra

Let us see how to use Rack with a Sinatra app. Create a folder `racksinatra`. Next, let us install Sinatra:

```
gem install Sinatra
```

Here's a trivial Sinatra program `my_sinatra.rb` in the folder `racksinatra`:

```
require 'sinatra'
get '/' do
  'Welcome to all'
End
```

Sinatra can use Rack middleware. Our trivial middleware class will intercept the request from the user, calculate and display the response time on the console and pass on the unaltered request to the Sinatra app. In the folder `racksinatra` write the middleware class namely `rackmiddleware.rb`.

```
 class RackMiddleware
  def initialize(appl)
    @appl = appl
  end
  def call(env)
    start = Time.now
    status, headers, body = @appl.call(env) # call our
Sinatra app
    stop = Time.now
    puts "Response Time: #{stop-start}" # display on
console
    [status, headers, body]
  end
end
```

In order for Heroku to know what to do with our Sinatra app, create a config.ru in the folder `racksinatra`:

```
require "./my_sinatra"
run Sinatra::Application
```

Modify the Sinatra app to use our middleware:

```
# my_sinatra.rb
require 'sinatra'
require './rackmiddleware'
use RackMiddleware
get '/' do
  'Welcome to all'
End
```

Now open a Git Bash shell for folder `racksinatra` and type:

```
$ bundle init
```

Edit the created Gemfile with your preferred text editor to let it look like this:

```
source "http://rubygems.org"
gem 'sinatra'
```

In your open Bash shell type:

```
$ bundle check
```

Finally in the open Bash shell, type:

```
$ bundle install
```

Set up your local app to use Git. Type:

```
$ git init
$ git add .
$ git commit -m "SinatraRack app"
```

Create the app on Heroku. In the open bash shell, type:

```
$ heroku create
```

On my machine it showed:

```
Creating warm-winter-6785... done, stack is bamboo-mri-
1.9.2
http://warm-winter-6785.heroku.com/ | git@heroku.com:warm-
winter-6785.git
Git remote heroku added
```

Let us rename the app to racksinatra:

```
$ heroku rename racksinatra
```

Finally, let us push our app to Heroku:

```
$ git push heroku master
```

The app is now running on Heroku. You can take a look at it, in your browser type: http://racksinatra.heroku.com/.

To check whether the *** Response Time ***: has been displayed on the Heroku console, type:

```
$ heroku logs -s app -n 5
```

where `-s app` shows the output from your app and `-n 5` retrieves 5 log lines.

For me it showed:

```
?[36m2011-10-26T05:40:06+00:00 app[web.1]:?[0m *** Response
Time ***: 0.000297385
```

That's it. I hope you liked this introductory article on Rack. Happy Racking!!

If you have any questions, feel free to leave a comment on my article here.