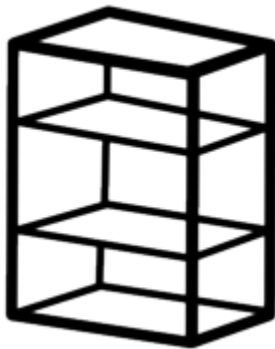


The Little Rack Book

**Rack a minimal interface between web servers
supporting Ruby and Ruby frameworks**



rack
powers web applications

by RubyLearning
<http://rubylearning.org>

IMPORTANT

You Do NOT Have Rights to Edit, Resell, Copy or Claim Ownership to this Book!

However...You DO Have the Right to Pass this Book Along to Others Who Might Benefit from it!

Feel free to share it with your blog readers and give it away as a freebie.

ALL RIGHTS RESERVED: You do not have any rights to sell or profit from this book. All content is to remain unedited and all links must stay in tact as they are. You can not claim any type of ownership without express written permission from the creator and only the creator, Satish Talim. All rights to this book belong to the author only.

DISCLAIMER: All information contained within this book is strictly the views represented by the author, at time of publication. Said author can and does reserve the right to add to, change, alter or update the thoughts and opinions stated herein. Every attempt has been made to accurately substantiate all information in the said book. However, the author, his partners, affiliates make no warranty to nor do they take responsibility for any errors or exclusions that may be contained in within.

ACKNOWLEDGEMENT: Konstantin Haase and Matt Aimonetti for helping me with some practical Rack examples.

Find more actionable tips and advice at:

<http://satishtalim.github.com/webruby/chapter5.html>

Learning Rack

Revisiting Ruby's proc object

Note that I have Ruby 1.9.3 installed on a Windows box and all the programs in this article have been tested using that.

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.

Open a command window and type:

```
$ irb --simple-prompt
>> my_proc = lambda {puts 'Proc Called'}
=> #<Proc:0x1fc9038@(irb):2(lambda)>
>> # method call invokes the block
?> my_proc.call
Proc Called
=> nil
>> exit
$
```

Close the command window.

my_proc1.rb

The above can be written in a Ruby program `my_proc1.rb`:

```
# my_proc1.rb
my_proc = lambda {puts 'Proc Called'}
my_proc.call
```

Open a command window and execute the `my_proc1.rb` program:

```
$ ruby my_proc1.rb #=> Proc Called
```

Close the command window.

Rack Specification

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).

That's the Rack specification in a nutshell. You can check out the full details [here](#). **Strictly speaking, you don't need the rack gem in order to write Rack ready applications.** Just stick to the specification and that's it.

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`. This hash contains all the relevant information about the request: this includes the HTTP verb used by the request, the path that is

requested, the headers that have been sent by the client, and so on. 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. Open a command window and type:

```
$ irb --simple-prompt
>> 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. In the above window 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. In the above window type:

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

Close the command window.

my_rack_proc.rb

What we saw above can be written in a Ruby program `my_rack_proc.rb`:

```
# my_rack_proc.rb
my_rack_proc = lambda { |env| [200, {"Content-Type" =>
"text/plain"}, ["Hello. The time is #{Time.now}"]] }
puts my_rack_proc.call({})
```

Open a command window and execute the `my_rack_proc.rb` program:

```
$ ruby my_rack_proc.rb
200
{"Content-Type"=>"text/plain"}
Hello. The time is 2011-12-07 09:42:31 +0530
```

Close the command window.

Rack Documentation

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

Rack Source Code

<https://github.com/rack/rack>

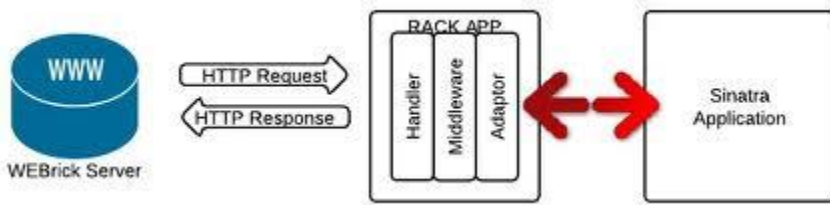
Installing Rack gem

Rack gem is a [collection of utilities and facilitating classes](#), to make life easier for anyone developing Rack applications. It includes basic implementations of request, response, cookies, sessions and a good number of useful middlewares.

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

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
```

In case you do not have rack, install it by typing:

```
$ gem install rack
```

We can run our previously written Rack application (`my Rack proc`) with any of the Rack handlers.

To look at the Rack handlers available, in the already open `irb` window type:

```
>> require 'rack'
=> true
>> Rack::Handler.constants
=> [:CGI, :FastCGI, :Mongrel, :EventedMongrel,
:SwiftplyedMongrel, :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.

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

```
>> 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 1.9.3 (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 could have run this app at a different port, say 9876:

```
>> 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 1.9.3 (2011-07-09) [i386-
mingw32]
[2011-10-24 11:32:21] INFO WEBrick::HTTPServer#start:
pid=480 port=9876
```

In the browser window you would have typed the url: <http://localhost:9876/>

In the command window, press **Ctrl C** to stop the **WEBrick** server. Close the command window.

my_rack_proc2.rb

What we saw above can be written in a Ruby program `my_rack_proc2.rb`:

```
# my_rack_proc2.rb
require 'rack'
my_rack_proc = lambda { |env| [200, {"Content-Type" =>
"text/plain"}, ["Hello. The time is #{Time.now}"]] }
Rack::Handler::WEBrick.run my_rack_proc
```

Open a command window and execute the `my_rack_proc2.rb` program:

```
$ ruby my_rack_proc2.rb
[2011-12-07 09:54:48] INFO WEBrick 1.3.1
[2011-12-07 09:54:48] INFO ruby 1.9.3 (2011-10-30) [i386-
mingw32]
[2011-12-07 09:54:48] INFO WEBrick::HTTPServer#start:
pid=5992 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 2011-12-07 09:55:52 +0530
```

Press **Ctrl-C** in the command window to stop the **WEBrick** server.
Close the command window.

Another Rack app - my_method

A Rack app need not be a lambda; it could be a method. Open a command window and type:

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

In the open irb shell, 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 1.9.3 (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. Close the command window.

my_rack2.rb

What we saw above can be written in a Ruby program `my_rack2.rb`:

```
# my_rack2.rb
require 'rack'
def my_method env
  [200, {}, ["method called"]]
end
Rack::Handler::WEBrick.run method(:my_method)
```

Open a command window and execute the `my_rack2.rb` program:

```
$ ruby my_rack2.rb
[2011-12-07 10:11:48] INFO WEBrick 1.3.1
[2011-12-07 10:11:48] INFO ruby 1.9.3 (2011-10-30) [i386-
mingw32]
[2011-12-07 10:11:48] INFO WEBrick::HTTPServer#start:
pid=6664 port=80
```

Open a browser window and type the url: <http://localhost/>

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

```
method called
```

Press **Ctrl-C** in the command window to stop the **WEBrick** server.
Close the command window.

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 Rack Object 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`. Also note that you don't need to `require 'rack'` since you use `rackup` which already loads rack.

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 1.9.3 (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** in the command window to stop the **WEBrick** server.
Close the command window.

my_app.rb

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`, open a command window and type:

```
$ rackup config.ru
[2011-10-25 06:18:16] INFO WEBrick 1.3.1
[2011-10-25 06:18:16] INFO ruby 1.9.3 (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** in the command window to stop the **WEBrick** server. Close the command window.

Using Rack::Request and Rack::Response

We will use `cURL`, `Rack::Request` and `Rack::Response` in the code examples that follow.

First, let's look at the following `cURL` command:

```
curl -X request
```

(HTTP) Specifies a custom request (GET, POST) to use when communicating with the HTTP server. The specified request will be used instead of the standard GET.

```
curl -X POST -d data
```

(HTTP) Sends the specified data in a POST request to the HTTP server, in a way that can emulate as if a user has filled in a HTML form and pressed the submit button. Note that the data is sent exactly as specified with no extra processing (with all newlines cut off). The data is expected to be "url-encoded". This will cause `cURL` to pass the data to the server using the content-type `application/x-www-form-urlencoded`. If more than one `-d/--data`

option is used on the same command line, the data pieces specified will be merged together with a separating &-letter. Thus, using '-d name=daniel -d skill=lousy' would generate a post chunk that looks like 'name=daniel&skill=lousy'.

If you start the data with the letter @, the rest should be a file name to read the data from, or - if you want `cURL` to read the data from stdin. The contents of the file must already be url-encoded.

Do check out the source code for [Rack::Request](#) and [Rack::Response](#). Let us write a program to explore these two:

Our `config.ru` will be:

```
require './my_request'  
run MyRequest.new
```

Our application `my_request.rb` is as follows:

```
# my_request.rb
class MyRequest
  def call(env)
    req = Rack::Request.new(env)
    name = req.params['name']
    text = req.params['text']

    Rack::Response.new.finish do |res|
      res['Content-Type'] = 'text/plain'
      res.status = 200
      str = "Parameters sent: name - #{name} | text -
#{text}"
      res.write str
    end
  end
end
```

To run our rack app, in the same folder that contains `config.ru`, open a command window and type:

```
$ rackup config.ru
[2011-10-25 06:18:16] INFO WEBrick 1.3.1
[2011-10-25 06:18:16] INFO ruby 1.9.3 (2011-07-09) [i386-
mingw32]
[2011-10-25 06:18:16] INFO WEBrick::HTTPServer#start:
pid=2224 port=9292
```

Our client is `cURL`. Open a Bash shell and type:

```
$ curl -X POST -d 'name=Satish Talim&text=This course is awesome!' localhost:9292
```

In your Bash shell, you should see something like this:

```
Parameters sent: name - Satish Talim | text - This course is awesome!
```

Type `exit` in your Bash shell and press **Ctrl-C** in the command window to stop the **WEBrick** server. Close the command window.

A very basic practical Rack app

Credit: [Matt Aimonetti](#)

Matt has been kind enough to [write this app](#) for us. It's a very basic rack application showing how to use a router based on the uri and how to process requests based on the HTTP method used. The explanation is [here](#).

Another practical Rack app

Credit: [Konstantin Haase](#)

Konstantin has provided us with [another practical Rack app](#), where we use `cURL` to upload a text file and which then gets sorted.

Rack middleware

Credit: [Matt Aimonetti](#)

Remember: 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.

Write the following program `my_middleware.rb`:

```
# my_middleware.rb
module MyMiddleware
  class Hello
    def initialize(app)
      @app = app
    end

    def call(env)
      if env['PATH_INFO'] == '/hello'
        [200, {"Content-Type" => "text/plain"}, ["Hello
from the middleware!"]]
      else
        # forward the request
        @app.call(env)
      end
    end
  end
end
```

In the above program, we are using a [module](#). Also, we keep the application reference in the `@app` variable when a new instance of the middleware is created.

`my_middleware.rb` is a rack middleware living on its own, that you need to require it in your app and then call it by telling rack to use it.

Create our app `config.ru` as follows:

```
require './my_middleware'
use MyMiddleware::Hello # this comes in between
run Proc.new{|env| [200, {"Content-Type" => "text/plain"},
['Try accessing visiting /hello']] }
```

To run our rack app, in the same folder that contains `config.ru`, open a command window and type:

```
rackup config.ru
[2011-12-11 09:33:40] INFO WEBrick 1.3.1
[2011-12-11 09:33:40] INFO ruby 1.9.3 (2011-10-30) [i386-
mingw32]
[2011-12-11 09:33:40] INFO WEBrick::HTTPServer#start:
pid=3532 port=9292
```

Note: Under the hood, `rackup` converts your `config.ru` script to an instance of [Rack::Builder](#) (which we shall talk about

soon). `Rack::Builder#use` adds a middleware to the rack application stack created by `Rack::Builder`.

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

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

```
Try accessing visiting /hello
```

That's the normal response from your app.

Next, in your browser window type the url: <http://localhost:9292/hello>

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

```
Hello from the middleware!
```

That's your middleware talking!

Press **Ctrl-C** in the command window to stop the **WEBrick** server. Close the command window.

Using Lobster

Rack comes with little funny web application called "Lobster", that can be used as a demo.

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

```
# config.ru
require 'rack/lobster'
run Rack::Lobster.new
```

To run our rack app, in the same folder that contains `config.ru`, open a command window and type:

```
$ rackup config.ru
```

Open a browser window and type the url: <http://localhost:9292/>. You will see a lobster drawing!

Press **Ctrl-C** in the command window to stop the **WEBrick** server. Close the command window.

The following is credited to [Matt Aimonetti](#)

Previously we wrote a program `my_middleware.rb`. We can use this with lobster:

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

```
# config.ru
require 'rack/lobster'
use MyMiddleware::Hello
run Rack::Lobster.new
```

To run our rack app, in the same folder that contains `config.ru`, open a command window and type:

```
$ rackup config.ru
```

After you have finished executing this program, press **Ctrl-C** in the command window to stop the **WEBrick** server. Close the command window.

Rack::Builder

Under the hood, `rackup` converts your `config.ru` script to an instance of [Rack::Builder](#).

`Rack::Builder` is the thing that glues various Rack middlewares and applications together and converts them into a single entity/rack application. A good analogy is comparing `Rack::Builder` object with a stack, where at the very bottom is your actual rack application and all middlewares on top of it, and the whole stack itself is a rack application too.

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

```
rack_time = Proc.new { |env| [200, {"Content-Type" =>
"text/plain"}, ["Hello. The time is #{Time.now}"]] }
Rack::Handler::WEBrick.run rack_time, :Port => 9292
```

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 1.9.3 (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. The time is 2011-12-06 11:19:49 +0530
```

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

In `config.ru`, let us convert `rack_time` to use `Rack::Builder`. We will use the block form of `Rack::Builder`:

```
rack_time = Proc.new { |env| [200, {"Content-Type" =>
"text/plain"}, ["Hello. The time is #{Time.now}"]] }
builder = Rack::Builder.new do
  run rack_time
end
Rack::Handler::WEBrick.run rack_time, :Port => 9292
```

`Rack::Builder#run` specifies the actual rack application you're wrapping with `Rack::Builder`.

Here `Rack::Builder#initialize` accepts a block argument, which is evaluated within the context of the newly created instance using `instance_eval`.

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 1.9.3 (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. The time is 2011-12-06 11:19:49 +0530
```

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

`Rack::Builder#use` adds a middleware to the rack application stack created by `Rack::Builder`. Rack has many useful middlewares and one of them is `Rack::CommonLogger`, which logs a single line to the supplied log file in the Apache common log format.

Let's add `Rack::CommonLogger` to `rack_time`:

```

require 'logger'

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

builder = Rack::Builder.new do
  use Rack::CommonLogger
  Logger.new('rack.log')
  run rack_time
end

Rack::Handler::WEBrick.run rack_time, :Port => 9292

```

We create an explicit logger `rack.log`. This log file is created in the same folder as `config.ru` and contains:

```

# Logfile created on 2011-12-06 13:54:42 +0530 by
logger.rb/31641

```

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 1.9.3 (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. The time is 2011-12-06 11:19:49 +0530
```

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

Let's see how to use `Rack::Builder#map` to map urls to given actions.

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

```

require 'logger'

rack_app = Rack::Builder.new do
  use Rack::CommonLogger
  Logger.new('rack.log')

  map "/" do
    run Proc.new {|env| [200, {"Content-Type" =>
"text/html"}, ["This is public page"]] }
  end

  map "/secret" do
    use Rack::Auth::Basic, "Restricted Area" do |user,
password|
      user == 'super' && password == 'secretsauce'
    end

    run Proc.new {|env| [200, {"Content-Type" =>
"text/html"}, ["This is a secret page"]] }
  end

  map "/files" do
    run Proc.new {|env| [200, {"Content-Type" =>
"text/html"}, ["Here are the secret files"]] }
  end
end

Rack::Handler::WEBrick.run rack_app, :Port => 9292

```

When you nest map blocks, you'll need to specify URI relative to the enclosing mapping block, as you can clearly see in the example above. We use HTTP Basic Authentication for securing things.

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 1.9.3 (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:

```
This is public page
```

If you type the url: <http://localhost:9292/secret>

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



HTTP Basic Authentication

For User Name type super and for Password type secretsauce. Press the OK button. In your browser window, you should see something like this:

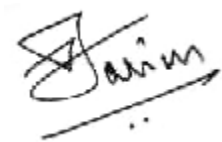
```
This is a secret page
```

Now, if you type the url: <http://localhost:9292/secret/files>, in your browser window, you should see something like this:

```
Here are the secret files
```

Do run all the examples in this Rack tutorial, to get a solid understanding of Rack.

Best,

A handwritten signature in black ink, appearing to read "Satish Talim". The signature is stylized with a large, looped initial 'S' and a horizontal line underneath. There are two small dots below the line.

<http://rubylearning.org/>