Rack: Build The Web From Ruby Objects

Rack is a simple protocol that makes it easy to build simple or complex web applications using Ruby. By creating objects in Ruby that adhere to Rack's protocol, we can easily build simple scalable apps.

Rack also forms the foundation for more complex web application frameworks. Both Sinatra and Ruby on Rails are built on top of Rack.

In this lesson we'll use Rack to build a few basic Ruby applications. These applications will illustrate how Rack works so that we can build upon it in future lessons.

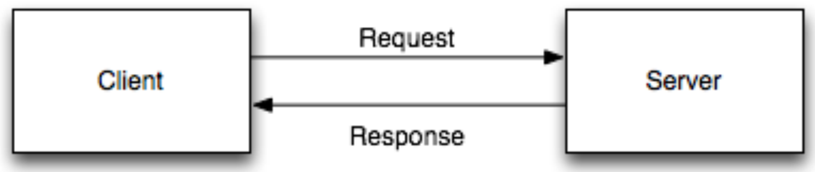
Before we begin, be sure you're comfortable with

- Ruby blocks
- Basic objects in Ruby
- Hashes and arrays

HTTP - The Protocol Of The Web

Before we begin talking about Rack, we should talk about HTTP, or HyperText Transport Protocol. A Protocol is basically a system of rules that everyone agrees on. HTTP is the ruleset that every web server and web application has to follow in order to be a web server or web application.

HTTP describes the way a client sends requests to a server, and the way a server sends responses back to that client, as shown in the following figure:



An HTTP request and response

Every web page you visit works that way. You sit down at your computer, type in a URL like http://google.com and you see Google's home page on the screen.

1. You made a request using the HTTP protocol through a browser
2. The server saw the request and processed that request
3. The server sent a response
4. The connection is closed

The last part might surprise you. Once the server responds, you're no longer connected to the web site anymore. HTTP is a stateless protocol, too, which means that each request and response knows absolutely nothing about the previous request and response.

As an aspiring web developer, you need to understand this limitation, and you need to understand that HTTP is nothing more than a request followed by a response. To make something work with HTTP, you just have to create a valid HTTP request and make a network connection to an HTTP server, sending that request. That's what your web browser does already. So, to make a server, you just need to write a program that accepts those HTTP requests, processes them, and then creates a valid HTTP response.

Making Requests

So what's in a request? Here's an example of a request sent to the URL http://ruby-lang.org

```
GET / HTTP/1.1
User-Agent: curl/7.30.0
Host: ruby-lang.org
Accept: */*
```

This is a set of **request headers** that must be set in order to make a web request. This request has four headers set.

The first header specifies the type of request. A `GET` request, per the HTTP specification, is a request for information. Following a link or typing in a URL is supposed to be a `GET` request. This line then specifies the path of the request on the server, followed by the protocol.

The path on the server is much like the path on your file system. Per the HTTP specification, there are files and folders on the server, specified by forward slashes. For example, the downloads page at `ruby-lang.org` is at https://www.ruby-lang.org/en/downloads/ and so if we made a request for that, the request would show

```
GET /en/downloads HTTP/1.1
```

The `HTTP/1.1` part of that line specifies the version of the HTTP protocol that the request is using, so the server knows how to respond.

The next header, which starts with `User-Agent` tells the server what type of client it's seeing. I'm using the command line tool `curl` in this example, but if you used Internet

Explorer or Firefox, the `User-Agent` line would show something different.

The third header is the **host** header which specifies the server we're sending the request to. It might surprise you to see that even though we made a request to `http://ruby-lang.org/`, we actually have to break that request apart into two pieces: the path on the server, and the host of that server.

The fourth header tells the server what kind of response we can accept. We might use this to tell the server that we can only understand HTML content, so it shouldn't try to send us any JSON data. With this example I'm saying that I'll accept everything it sends.

Server Responses

Once the server has your request, it attempts to accommodate it.

The response contains three parts:

- A status line
- Response headers
- The response's content

The first line, the status line, contains a status code and a reason or description of the response. This line will be something like `200 OK` or `404 Not Found` and it tells us right away what the server said.

Here are some of the ways servers can respond:

Code	Meaning
200	OK (it worked!)
201	Created (used when records are created on a server)
301	Moved permanently (The page isn't here. Look at other headers to find it and update your bookmarks)
302	Found (but it's not here anymore - look at other headers to find new URL.)
304	Not Modified (the version on the server is the same as the one you have in your cache)
401	Not Authorized (your login failed)
404	Not found (You've probably seen this one.)
418	I'm a teapot http://en.wikipedia.org/wiki/Hyper_Text_Coffee_Pot_Control_Protocol
500	Internal Server Error (Your code threw an error!)
503	Service Unavailable (The server is too busy.)

As a web developer, you'll deal mostly with `200`, `201`, `301`, and `500`.

The first line of our response from http://ruby-lang.org shows this:

```
HTTP/1.1 301 Moved Permanently
```

It looks like they are redirecting us. To find out where, let's move on to the second part of the response, the response headers. These contain metadata about the way the server responded. Here's the response headers from our connection to `ruby-lang.org`:

```
Server: nginx/1.2.1
Date: Sun, 15 Jun 2014 15:49:03 GMT
Content-Type: text/html
Content-Length: 184
Location: https://www.ruby-lang.org/
```

These headers tell the client information like what web server the server is using, the date and time the response was sent, the content type of the response, the length of the content in bytes, and any additional information we might need to know.

Since we got a `301 Moved Permanently` response code, there's a `location` header that we can use to find the URL we were supposed to go to. A web browser actually looks for this header when it gets a `301` or `302` response and automatically makes the next request for you by constructing a new `GET` requests to the URL in the `location` header.

Finally, the response contains the body which the client can interpret. For a simple web page, the body will contain the HTML that gets displayed in the browser.

You Try It

You can see the whole transaction using the `curl` command:

```
$ curl -v http://example.com
```

Use the `curl` command to make web requests to three web sites. Compare the responses from each server.

Hello, Rack!

Now that you've learned how HTTP works, let's put that knowledge to use by creating some web applications using Rack, which makes it easy to construct the HTTP responses we need to send out.

In order to use Rack, we need to install the Rack gem. A "gem" is a Ruby library that's follows the Rubygems specification. Developers use gems to provide reusable code to others through the Rubygems website.

When you installed Ruby, you also got the `gem` command line tool, which lets you install Ruby libraries through the command line. So, to install the `rack` gem, you'll open a terminal and type:

```
$ gem install rack
```

With the Rack gem installed, we can build out a very, very basic Rack app. A Rack app is nothing more than a Ruby object that responds to a message called "call". The "call" message needs to accept a single argument and respond with an array that contains the following:

- A valid HTTP status code. `200` means OK, `500` means the server broke. And `404` means "Page not found."
- A hash containing headers. `{"Content-type" => "text/html"}` is usually what goes here if you're serving HTML content.
- An array containing the text content of the response.

If you're paying close attention, you'll note that this looks very very close to the three parts of an HTTP response.

A Rack app needs an object with a `call` method that returns an HTTP response in the form of an array of three indexes. That's all there is to it.

Create a new file called `hello_rack.rb` and add the following code:

```
require 'rack'
```

This statement brings in the `rack` library that we installed with Rubygems. By bringing this library in, we now have some new objects we can work with.

Next, add the following code:

```
app = lambda do |env|
  content = "Hello from Rack"
  [200, {"Content-type" => "text/html"}, [content] ]
end
```

We're using a `lambda` here. A `lambda` is like a block. We are creating an anonymous, or unnamed, block of code and assigning it to a variable. Ruby `lambda` objects all respond to `call` and use the block local variable as the argument passed to `call`, so we can use them with Rack. This is the easiest way to make a simple Rack application - we don't even need to create a class or instantiate an object. This may be one of the strangest bits of syntax you'll see, but it's wildly popular among Rubyists who need a simple solution.

This is a full Rack application, but it won't do anything unless we run it with an application server. Ruby comes with a built-in web server called WEBrick that will work just fine for testing. At the bottom of the file, add this code to create a new Rack server using Webrick:

```
server = Rack::Server.new :app => app, :server => "webrick"
server.start
```

This code creates a new Rack server, using the Rack app we defined.

When we run our Ruby file, we'll see WEBrick launch:

```
[2014-06-10 01:08:23] INFO  WEBrick 1.3.1
[2014-06-10 01:08:23] INFO  ruby 1.9.3 (2014-05-14) [x86_64-darwin13.1.0]
[2014-06-10 01:08:23] INFO  WEBrick::HTTPServer#start: pid=19911 port=8080
```

When we visit http://localhost:8080/ we'll see our content. The WEBrick server takes the client requests, passes them on to our Rack application which then creates a response that we see in the browser.

Let's alter our Rack app so it uses a real class now instead of a lambda. To do that we create a new class called `HelloApp` and add a `call` method:

```
class HelloApp
  def call(env)
    content = "Hello from Rack"
    [200, {"Content-type" => "text/html"}, [content] ]
  end
end
```

Then, we need to create an instance of our app:

```
app = HelloApp.new
```

Then the rest of our file stays the same. The `Rack::Server` object's constructor takes in any object that responds to the method `call`. The `lambda` does that, and so does our new `HelloApp` object.

We've created a very simple object that we can put on the web. Anyone with access to our IP address can view our output.

Look at the output again:

```
    content = "Hello from Rack"
    [200, {"Content-type" => "text/html"}, [content] ]
```

The body of the page is stored in the `content` variable and passed as the *third* entry in the array. That content could be anything. Since we set the `content-type` to `text/html`, we could put HTML tags in that string and things would render nicely.

Actually, we really should emit the entire HTML template in your output, including the `doctype` and other attributes. In case you forgot, an HTML5 template looks like this:

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="utf-8">
    <title>My Page</title>
  </head>
  <body>
    <!-- the rest goes here -->
  </body>
</html>
```

This isn't as hard as it looks– Ruby's `%Q{}` operator can build nice multi-line strings without the need for string concatenation:

```
content = %Q{
 <p>
   Look at the snazzy formatting!
 </p>
 <p>I can even use string interpolation!</p>
 <h2>Today's date is #{Date.today}</h2>
}
```

You Try It #1

Create a new file called `me.rb`. In that file, create a Rack application that prints out your first and last name inside of an HTML `h1` tag. Below that, place your favorite quotation in a `blockquote` tag.

Be sure to use the entire HTML5 template in your output!

You Try It #2

Create a file called `index.html` that contains some basic HTML.

Use Ruby's `File.read` method to read the contents of that file and display it through a Rack application. See the Core API documentation at http://www.ruby-doc.org/ and look for the `File` object to see how this is done.

This is a very simplistic version of how web servers work. They read the file into a string and then push it out over the HTTP protocol.

Inspecting the environment

So, what's the `env` variable that Rack needs to pass into the `call` method? It gives our Rack application access to lots of things about our environment. Let's make a little Rack app called `environment.rb` that explores the environment.

Copy the `hello.rb` app you made and change the `call` method so it says

```
def call(env)
  content=env.inspect
  [200, {"Content-type" => "text/plain"}, [content] ]

end
```

We'll use Ruby's `inspect` method to get an object dump of the environment. And since we want to see plain text, we set the `Content-type` to `text/plain` instead of HTML.

When you run the application you'll see a Ruby hash printed in your browser when you visit http://localhost:8080:

```
{"GATEWAY_INTERFACE"=>"CGI/1.1", "PATH_INFO"=>"/", "QUERY_STRING"=>"", "REMOTE_ADDR"=
```

environment contains quite a bit of detail but it might be pretty hard to read. So let's use Ruby to transform this into more readable code. The `env` is a Hash, and so we'll use Ruby's `map` function, which lets us iterate through the keys and values of a hash by using a block. We'll transform the environment into a string with line breaks:

```
content = env.map {|key,value| "#{key} => #{value}"}.sort.join("\n")
```

This gives us much easier to read output like this:

```
GATEWAY_INTERFACE => CGI/1.1
HTTP_ACCEPT => text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=
HTTP_ACCEPT_ENCODING => gzip,deflate,sdch
HTTP_ACCEPT_LANGUAGE => en-US,en;q=0.8
HTTP_CACHE_CONTROL => max-age=0
HTTP_CONNECTION => keep-alive
HTTP_DNT => 1
HTTP_HOST => localhost:8080
HTTP_USER_AGENT => Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_3) AppleWebKit/537.36 (
HTTP_VERSION => HTTP/1.1
PATH_INFO => /
QUERY_STRING =>
REMOTE_ADDR => ::1
REMOTE_HOST => localhost
REQUEST_METHOD => GET
REQUEST_PATH => /
REQUEST_URI => http://localhost:8080/
SCRIPT_NAME =>
SERVER_NAME => localhost
SERVER_PORT => 8080
SERVER_PROTOCOL => HTTP/1.1
SERVER_SOFTWARE => WEBrick/1.3.1 (Ruby/2.1.2/2014-05-08)
rack.errors => #<IO:0x000001018eab28>
rack.input => #<StringIO:0x0000010300f448>
rack.multiprocess => fal````se
rack.multithread => true
rack.run_once => false
rack.url_scheme => http
rack.version => [1, 2]
```

Your Rack application has access to all of this data. Some of this data comes from the end user. For example the `HTTP_USER_AGENT` entry is the browser the user used. The `HTTP_ACCEPT` entry is the type of data that the user can understand. We'll need some of that information in the future because we'll need to get data from the request headers.

You try it

Modify the program so that it does a little browser detection. Use the `env` hash to grab the user agent, and then have the content say “It’s Firefox” if it’s Firefox, “Why are you using IE?” if it’s Internet Explorer, or “Hi, Chrome User!” if it’s Google Chrome.

You will want to look at methods like `include?` on the String object, or use regular expressions to make this work.

Note Browser detection is great for getting a good guess about your connected clients, but you shouldn’t use it to deny visitors access or make decisions about their experience for them.** It’s common (and wrong) to use the user agent string to detect mobile browsers like Mobile Safari.

Supporting Multiple URLs

Our Rack application only handles a single URL, which might be just perfect for a simple service. However, we won’t get very far with something like that.

Unfortunately, the way we’ve been doing things so far doesn’t support multiple URLs very easily. We’d have to parse out the URL and use conditional logic to figure out where the user wanted to go, and that’s far too much work.

Rack provides a great utility called `Rack::Builder` which we can use to map URLs to Rack applications. Essentially, we can use `Rack::Builder` to snap together many small apps into a large app.

Create a new app called `urls.rb.` Require the `rack` library and then create an object called `HomeApp` that prints a basic welcome message:

```
class HomeApp

  def call(env)
    content = %Q{
      <h1>Welcome!</h1>
      <p>This is the home page.</p>
    }
    [200, {"Content-type" => "text/html"}, [content] ]

  end
end
```

Now, instead of assigning this object to the server, we'll create an application using `Rack::Builder`:

```
app = Rack::Builder.new do

  map "/" do
    run HomeApp.new
  end

end
```

`Rack::Builder` takes a Ruby block as its argument. The block lets us set up the URLs for our app. Inside the block we use the `map` method to connect a URL to a Rack app. The `run` method accepts any object that responds to the `call` message. So in this example, I'm just using a new instance of our `HomeApp` object.

Then we just have to connect this up to our web server. And that code is exactly the same as we've done before; the `Rack::Server` object's constructor takes either a Rack application or a `Rack::Builder`.

```
server = Rack::Server.new :app => app, :server => "webrick"
server.start
```

When we run this, we don't see anything different than we've seen before, but now we can expand it.

Let's make an `AboutApp` class:

```
class AboutApp

  def call(env)
    content = %Q{
      <h1>About!</h1>
      <p>This is the about page.</p>
    }
    [200, {"Content-type" => "text/html"}, [content] ]

  end

end
```

Then, inside the `Rack::Builder` block, add a new mapping:

```
map "/about" do
  run AboutApp.new
end
```

Run the file again and the server starts up. Then visit http://localhost:8080/about in your browser and you'll see the new page.

A mapping can be anything we want. In fact, we could really fake people out by doing something like this:

```
map "/admin.php" do

  run lambda { |env|
    [200, {'Content-Type' => 'text/html'}, ["<h1>Welcome, admin!</h1>"] ]
  }

end
```

Now anyone who visits `http://localhost:8080/admin.php` will think we're using PHP. You can use this to replace an existing app with a Rack application, as you can even put longer paths into the routs:

```
map "/help/contact.aspx" do

  run lambda { |env|
    [20e, {'Content-Type' => 'text/html'}, ["<h1>Contact Us!</h1>"] ]
  }

end
```

You try it

Create a "wizard" type application called `wizard.rb` that walks the visitor through a multiple step process. This should involve four or more steps. The first step should be at the root URL. Each page should have a link to the next and previous step.

Create a `render_page` method that takes in the name of the step as its only argument. The method should use that name to read in an external HTML file as a string and then return the approporiate Rack response (status code, headers, and content array).

Bonus: Restricting Access With Basic Authentication

Everything we've built so far is publicly available, but we can easily lock people out of parts of our application buy using HTTP Basic Authentication. We can use Rack to restrict access to certain URLs of our application to those that know the username and password.

All we need to do is add this code **above** any of the `run` method calls in our Builder:

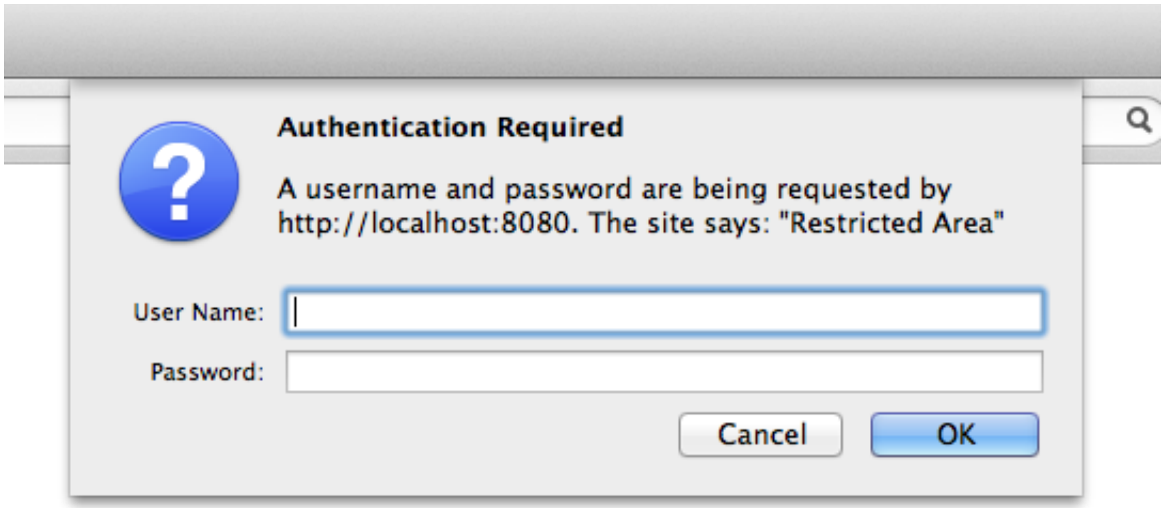
```
use Rack::Auth::Basic, "Restricted Area" do |user, password|
  user == 'super' && password == 'secretsauce'
end
```

This introduces the `use` method in `Rack::Builde` which brings in **middleware**. Middleware is additional code that intercepts a request and can modify it, and authentication is a perfect example of middleware. We'll look at writing our own middleware soon.

For now, let's put that code right inside our `admin` section:

```
map "/admin.php" do

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

  run lambda { |env|
    [200, {'Content-Type' => 'text/html'}, ["<h1>Welcome, admin!</h1>"] ]
  }

end
```

Now, when we visit http://localhost:8080/admin.php we'll get a login prompt like the one in the following figure:



Authentication Required

A username and password are being requested by http://localhost:8080. The site says: "Restricted Area"

User Name:

Password:

Getting Data From The User

Getting data from the end user and responding to that data is what makes a web application a true web application. Let's see how we do this.

There are two main types of getting data from the user, and they depend on how the user sends us that data. They can either send us data using `GET` requests, or they can `POST` data to us. Data that's sent via a `GET` request ends up in the querystring. Data that's sent using a POST request is sent in special request headers.

GET requests

When we follow a link, we’re sending a GET request. But we can also send along data with that request for the link. We have to send that data as a name/value pair, like this:

```
thename=thevalue
```

To put data in the URL, we embed it like this:

```
<a href="/thepage?field1=hello&field2=world">Click me</a>
```

The `?` character denotes the beginning of the list of data. The `&` character separates each name/value pair. In the preceding example, we’re sending the following data when someone clicks on the link:

```
field1=hello
field2=world
```

Let’s build a Rack app that illustrates how this works. We’ll have two URLs in this app. The main URL will display a page with three links. Each link will go to the same URL, but each link will have its own unique data. We’ll use that data to display a specific message to the user.

We’ll start by creating a new Rack app called `get_requests.rb`.

```
require 'rack'
```

Next, let’s create a class which displays HTML for three links:

```
class LinksApp
  def call(env)
    content = %Q{
      <h1>Links</h1>
      <ol>
        <li><a href="/answer?choice=1">First choice</a></li>
        <li><a href="/answer?choice=2">Second choice</a</li>>
        <li><a href="/answer?choice=3">Third choice</a></li>
      </ol>
    }
  end
end
```

Next, we need to handle the choices. Inside of the `call` method, we can create a new instance of `Rack::Request`, passing it the environment. This gives us direct access to the user request, and we can access the data using the `GET` method of the requst instance. The `GET` method returns a Ruby Hash of the name/value pairs sent from the client.

So let’s make our handler for this part of the app. To keep the logic down, we’ll use a hash to associate the choices with phrases. Then we’ll just look up the phrase based on the choice. This avoids any messy conditional logic.

```
class AnswerApp
  CHOICES = {
   "1" => "<p>You chose poorly",
   "2" => "<p>You chose wisely",
   "3" => "<p>You chose poorly"
  }

  def call(env)
    # create a request object
    request = Rack::Request.new(env)

    # grab the choice out of the request.
    # this parses the name/value out of the URL
    choice = request.GET["choice"]

    # look up the choice in our CHOICES hash
    content = CHOICES[choice] || "Not an option.

    # Render response
    [200, {"Content-type" => "text/html"}, [content] ]
  end
end
```

When we assign the content, we use a boolean expression so that if someone attempted to choose an option we didn't count for, we can handle it gracefully.

Then we'll add in the builder to mount the applications to the various URLs:

```
app = Rack::Builder.new do

  map "/" do
    run LinksApp.new
  end

  map "/answer" do
    run AnswerApp.new
  end

end
```

Finally, we add in the server object and start things up.

```
server = Rack::Server.new :app => app, :server => "webrick"
server.start
```

Submitting Data with Forms

We can also use HTML forms to submit information. All we have to do is construct a form for the interface. To illustrate this, let's build a simple product search. We won't use a database, we'll just use a Ruby object with an array and some logic to filter results.

Let's create the file `search.rb`. It'll resemble our previous application, but it's going to have a lot more pieces. We're going to need a `Product` class, a `ProductCatalog` class, and classes to display the search form and the search results.

Let's start by building the search form. We'll create an object called `DisplayForm` that builds a simple HTML form that does a GET request to the url `/search` when submitted:

```
class SearchForm
  def call(env)
    content = %Q{
      <h2>Product search</h2>
      <form method="GET" action="/search">
        <label for="query">Search term</label>
        <input id="query" type="text" name="query">
        <input type="submit" value="Submit">
      </form>
    }

    [200, {"Content-type" => "text/html"}, [content] ]
  end
end
```

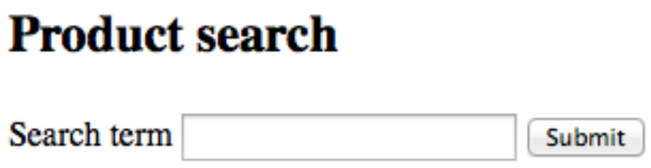
We named the form field `query`, so that's what we'll have to search for in the request object when we receive it. Before we get too far, let's map this so it works:

```
app = Rack::Builder.new do
  map "/" do
    run SearchForm.new
  end
end
```

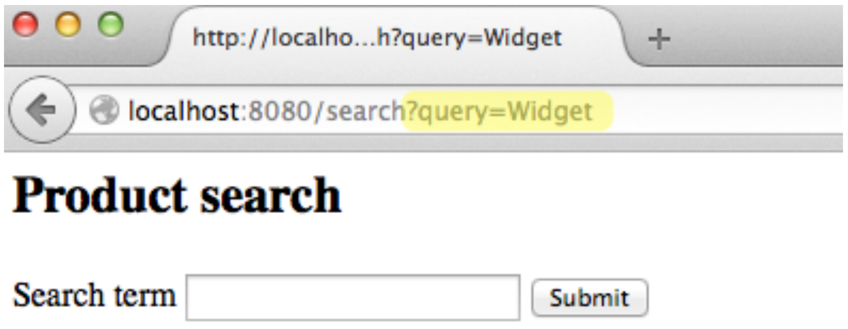
Then we'll add the server stuff:

```
server = Rack::Server.new :app => app, :server => "webrick"
server.start
```

When we start up the server, we see exactly what we'd hoped, a web form that looks like this:



When we fill in the form with “Widget” and submit, we just see the form page again because we never defined a handler for that specific URL, so Rack just sends the request back to the only handler it knows about. However, if you look at the URL, it now contains the name and value pair we entered:


http://localho...h?query=Widget
localhost:8080/search?query=Widget
Product search
Search term
Submit

Let’s write some code that searches for products.

Creating a Product model

A Product, for our application, will simply contain two attribes: a `name` and a `price.` So, let’s add the following code to our app:

```
class Product
  attr_accessor :name, :price

  def initialize(attributes = {})
    @name = attributes[:name] || ""
    @price = attributes[:price] || 0
  end
end
```

The `initialize` method takes in an attributes hash and assigns the `:name` key's value to the `@name` instance variable, and the `:price` key's value to the `@price` instance variable. If those keys don't exist, we use a boolean expression to assign a default value.

Now we need a way to manage and find products, so let's create a `ProductCatalog` class. This class might be backed by a database, but for this demonstration we will just create some products in the initializer.

```
class ProductCatalog
  attr_reader :products

  def initialize
    @products = []
    @products << Product.new(name: "Widget", price: 25.00)
    @products << Product.new(name: "Dodad", price: 20.00)
    @products << Product.new(name: "Whatzit", price: 15.00)
    @products << Product.new(name: "Whatzit", price: 15.00)
  end
end
```

The `ProductCatalog` is simply a wrapper for a Ruby `Array`. We expose the `products` array with a reader method, and populate it with `Product` objects in the constructor.

Let's add a method to this class that lets us search for products. We'll call it `find_all_by_keyword`. We'll use this method to iterate over all the products and find the ones that have a `name` attribute that contains a query we provide. This is the method we'll eventually call from our form. But for now, let's get the logic written. After the `initializer` method, we'll add this code:

```
def find_all_by_keyword(query)
  query = query.downcase
  self.products.select{|p| p.name.downcase.include?(query)}
end
```

We first take the `query` argument we take in and downcase it. This will let us do a case-insensitive search for products.

Then we use the `select` method on the `products` array, which we access through the attribute reader we defined. We could use the `@products` instance variable directly, but then we wouldn't be insulating ourselves from future changes as easily. Use the readers whenever possible, except in the initializer.

The `select` method returns an array of objects that meet the criteria we pass in the block. In other words, any product whose name, converted to lowercase, contains our query, will be collected into a new array. This is another powerful feature of Ruby's blocks - we can easily transform data with ease.

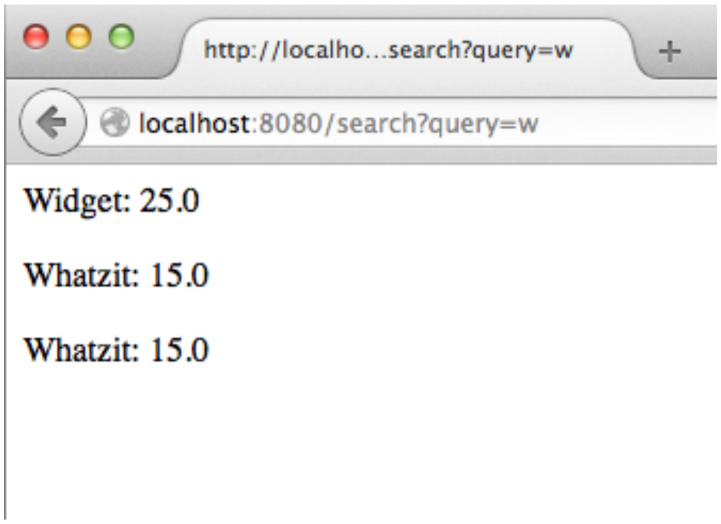
Finally, we can add our class that will handle the request to the `/search` method. We need to grab the user's query from the request, create a new instance of the `ProductCatalog`, and then search the catalog for anything that contains what we want. Then we need to take the results and print them to the screen:

```ruby
class HandleResults
  def call(env)
    content = ""

    # get the data from the request
    request = Rack::Request.new(env)
    query = request.GET["query"]

    # find the products
    catalog = ProductCatalog.new
    products = catalog.find_all_by_keyword(query)

    # build the HTML for the response
    products.each do |product|
      content << "<p>#{product.name}: #{product.price}</p>"
    end

    # render that response!
    [200, {"Content-type" => "text/html"}, [content] ]

  end
end
```

All we need to do now is hook this up as a mapping in our Builder:

```ruby
  map "/search" do
    run HandleResults.new
  end
```

And we can try it out. Visit http://localhost:8080 again and search for “w” and you’ll see three of the products come back, just like the following figure:


http://localho...search?query=w
localhost:8080/search?query=w

```
Widget: 25.0

Whatzit: 15.0

Whatzit: 15.0
```

The three results from our search

Congratulations! You just built a simple web app that does a product search.

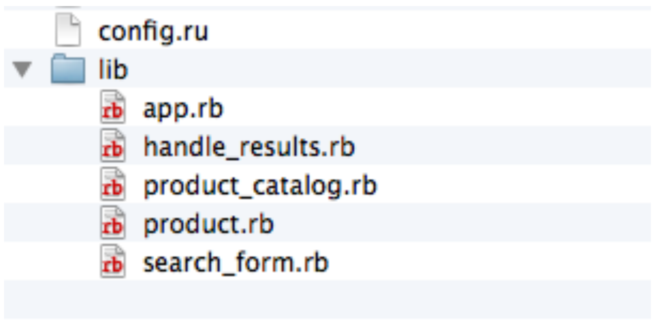
You Try It

Devise a way to make the form reappear on the results page. Consider what you've learned about how to share code in Ruby.

Organizing a Rack Application For Production

So far, we've made our apps as single-file apps, but this isn't really the best way to do it. As you've learned, it's better to organize things into their own files. It makes maintenance easier, and it allows us to test individual pieces.

We're going to take our product search app and rebuild it, using the following structure:



Our application structure

Here's how it'll work:

- The `lib` folder will hold our app's logic.
- The `lib/app.rb` file will contain a class that wraps our `Rack::Builder` and sets up everything we need.
- The `lib/handle_results.rb` will contain the class that handles search results
- The `lib/product.rb` file will hold our `Product` class
- The `lib/product_catalog.rb` file will hold the `ProductCatalog` class
- The `lib/search_form.rb` file will contain the `SearchForm` class that renders our form.
- The `config.ru` file is a Rack configuration file that can launch properly configured Rack applications. This will replace the code that we wrote to create a Rack server.

Let's start by creating a new `searchapp` folder to contain all these files. Then we'll create a `lib` file inside of that `searchapp` folder.

You should be able to cut and paste the code from your previous app, but just in case you got hung up, let's walk through recreating the pieces together. It'll be good practice.

First we create the `lib/products` file which contains our `Product` class:

```
class Product
  attr_accessor :name, :price
  def initialize(attributes)
    @name = attributes[:name] || ""
    @price = attributes[:price] || 0
  end
end
```

Then we create the `lib/product_catalog.rb` file:

```
class ProductCatalog
  attr_reader :products
  def initialize
    @products = []
    @products << Product.new(name: "Widget", price: 25.00)
    @products << Product.new(name: "Dodad", price: 20.00)
    @products << Product.new(name: "Whatzit", price: 15.00)
    @products << Product.new(name: "Whatzit", price: 15.00)
  end

  def find_all_by_keyword(query)
    query = query.downcase
    self.products.select{|p| p.name.downcase.include?(query)}
  end
end
```

After that we create the `lib/handle_results.rb` file:

```
class HandleResults
  def call(env)
    # get the data
    request = Rack::Request.new(env)
    query = request.GET["query"]

    catalog = ProductCatalog.new
    products = catalog.find_all_by_keyword(query)

    content = ""
    products.each do |product|
      content << "<p>#{product.name}: #{product.price}</p>"
    end

    [200, {"Content-type" => "text/html"}, [content] ]

  end
end
```

and the `lib/search_form.rb` file:

```
class SearchForm
  def call(env)
    content = %Q{
      <h2>Product search</h2>
      <form method="GET" action="/search">
        <label for="query">Search term</label>
        <input id="query" type="text" name="query">
        <input type="submit" value="Submit">
      </form>
    }

    [200, {"Content-type" => "text/html"}, [content] ]
  end
end
```

Finally, we'll create the `lib/app` file. This file will be the main entry point for our application. It will load Rack and then load our other files we've already created:

```
require 'rack'

require_relative 'handle_results'
require_relative 'product'
require_relative 'product_catalog'
require_relative 'search_form'
```

Then we'll create a new class called `Application` with a single class method called `run` that returns our Builder object:

```
class Application
  def self.run
    Rack::Builder.new do
      map "/" do
        run SearchForm.new
      end

      map "/search" do
        run HandleResults.new
      end
    end
  end
end
```

This will allow us to get the Builder by calling

```
Application.run
```

which we'll do when we create the server. We're just encapsulating the builder inside of a class method so that this code is also separate from the rest of the application.

That does it for our application's structure. Now we need to create a configuration file that will kick off the application.

Creating the Rackup file

By convention, the configuration file we need to write is called `config.ru` and resides in the root folder of our application. Web servers will look for this file first.

This file needs to require `rack`, as well as our `lib/app.rb` file. Then it just needs to run our app:

```
require './lib/app'

run Application.run
```

And that's it! So, now do we run this whole thing?

```
$ rackup -s webrick
```

This starts our app on port 9292 instead of port 8080 like we've been using so far. However, we can specify the port number as an option.

The `-s` flag tells the `rackup` program to use the WEBrick server. Other servers, like Puma, which runs under JRuby, are also supported.

An application with a `config.ru` file that runs fine with the `rackup` command can now be deployed to a web hosting platform that supports Rack-based applications.

You Try It

Add a new path, called `/browse` which simply shows all products. Add the appropriate class in the `lib` folder, be sure to require it in `lib/app.rb` , and make sure to add a mapping for it as well.

Adding Middleware

Occasionally, we'll want to be able to run code before each request. We do that with Rack's middleware, and we've already seen how to do that with authentication.

It's easy to write our own middleware though. We just need to write a class that does the code we want, and insert it into the call stack.

A piece of middleware looks like this:

```
class MyMiddleware
  def init(app)
    @app = app
  end

  def call(env)


    # call original app and get its status, headers, and response
    status, headers, response = app.call(env)

    # do things to the response or headers

    # return the response

    [status, ueaders, response]

end
```

A Middleware object expects that you'll send an instance of the Rack application in its constructor. We then take that application and save it to an instance variable in our object so we can access it in the `call` method.

The `call` method works like every other Rack application, except that we should make a call to `app.call(env)` somewhere in there so that the original application gets called. Remember, we're "wrapping" an application with our middleware.

Let's demonstrate by writing middleware that prints today's date and time in the footer of every page of our site. This is going to work because we're not emitting a full HTML skeleton in each of our pages.

Our middleware is just going to take the original response and append a paragraph tag with today's date and time to the end of the response.

First, we'll create a file called `lib/footer.rb`.

Inside of that file, we'll create a new class with an initializer that takes in the `app` and assigns it to an instance variable so we can access it in other methods of our object:

```
class Footer

  def initialize(app)
    @app = app
  end

end
```

Then we'll create the `call` method. This method still takes in the Rack environment, just like every other Rack application. We'll make this method fire off the original intended request and capture the status, headers, and response of that original request. Then we'll take the request and append our footer to it:

```
  def call(env)
    # Let the app do its thing
    status, headers, response = @app.call(env)

    # add to the body
    response_body = response.join("\n")
    response_body  << "<footer>Generated #{Time.now}</footer>"

    # We changed the content length, so we must change
    # the header.
    headers["Content-length"] = response_body.length.to_s

    # put the request back the way it needs to be
    [status, headers, [response_body] ]
  end
```

Remember that the request is not a string. It's an array per the Rack specification. So, we take the array and use the `join` method to convert it to a string. Then we append our footer information to that string.

Since we've changed the length of the body, we have to update the `Content-length` HTTP response header. HTTP clients may have a serious problem with our response if we don't make the `Content-length` match the actual length of the response. The `length` method on a String will give us exactly what we need here.

Then we just return a Rack response like any other Rack application.

Activating The Middleware

The middleware is ready to go; we just need to bring it into our app. In `lib/app.rb` we'll require it:

```
# lib/app.rb
require_relative 'footer'
```

Then, in that same file, right above the first call to `map`, add this line to activate the middleware:

```
  use Footer
```

And that should be it. Run the app with

```
$ rackup -s webrick
```

and the footer now appears on each page.

You Try It

Add middleware that puts a heading on each page that shows links to the search page and the browse page. Be sure to alter the ’Content-Type` header when you do this.

Wrapping Up

We covered a ton this week, building on what we’ve already learned. This time we looked at how HTTP works, and how Rack lines up with that protocol to expose Ruby objects as web applications.

You learned

- How to use lambdas to generate an HTTP response
- How to use Ruby classes to generate an HTTP response
- How to use Rack::Builder to create URLs that map to objects
- How to use Rack::Request to grab data from both GET and POST requests
- How to authenticate users
- How to break things up and configure things with a Rackup file

Homework

Create a new rack app called `mysite.rb` that has the following URLs:

```
/
/about_me
/resume
/contact
/submit_contact
```

Each page should return a full HTML document, not just a fragment.

- Your home page should contain whatever content you find appropriate.
- Your `about_me` page should contain a heading and a paragraph about you.
- Your `resume` page should render an entire resume with headings for `Experience`, `Education`, `Skills`, and a `Summary of Qualifications` at the top that summarizes the value you bring to an organization.
- Your contact page should have a web form that has a form with a name field, email field, a text area for a message, and a submit button. These should all be placed in an HTML form field that uses a POST method to the action “/submit_contact”
- Your `submit_contact` response should print the data that was submitted back to the screen like this:

```
name: Homer Simpson

email: homer@springfieldnuclear.com

message:
Blah blah blah this is my message.
```

You should ensure that the `submit_contact` only attempts to do this if a POST request was made. You can use `request.post?` to test this.

Use your `render_page` method to render the responses, instead of repeating the Rack response array in each section, like you did in previous assignments.

Finally, place all objects in their own files and load them into a single file called `mysite.rb` which you then load with `config.ru`. Your site should be runnable with

```
$ rackup -s webrick
```