FULL STACK WEB DEVELOPER PART XIV

Introduction to Web Development

about HTTP protocol

Panos M.

www.techcareerbooster.com

Full Stack Web Developer Part XIV: Introduction to Web Development

Panos Matsinopoulos

This book is for sale at
http://leanpub.com/full-stack-web-developer-part-xiv-introduction-to-web-development

This version was published on 2019-08-20

Contents

The Bundle and the TCB Subscription

Part of Bundle

This book is not sold alone. It is part of the bundle Full Stack Web Developer.

Goes with a TCB Subscription

When you purchase the bundle, then you have full access to the contents of the TCB Courses.

Each TCB Subscription Goes with the Bundle

Moreover, this goes vice-versa. If you purchase the subscription to the TCB Courses, then you are automatically eligible for the Full Stack Web Developer bundle.

Credits To Photo on Cover Page

Image by Dimitris Vetsikas from Pixabay.

About and Copyright Notice

Full Stack Web Developer - Part XIV - Introduction to Web Development 1st Edition, August 2019

by Panos M. for Tech Career Booster (https://www.techcareerbooster.com)

For more information: https://www.techcareerbooster.com

Introduction to Web Development

This book is getting you into Web development starting from the HTTP protocol which is the language that a Web browser talks to the Web server. It will lay the fundamental knowledge required in order to understand how Web tools work. Also, you will build Web pages and Web servers with tools that are used in production by many companies around the world. Hence, this is where real Web development starts.

1 - Introduction to HTTP

Summary

Introduction to HTTP

This chapter introduces you to HTTP, the Hypertext Transfer Protocol. It is the protocol that Web browsers are using to communicated with Web servers. It is the fundamental stone to your journey to Web development. In other words, you cannot claim to be a Web developer if you don't grasp, at least, the basics of HTTP. Also, this chapter is necessary for you to easier understand the next chapters and sections that follow on Web development.

Learning Goals

1. Learn about HTTP.
2. Learn about the HTTP well known port.

3. Learn about the HTTPS well known port.
4. Learn about the URL and its format.
5. Learn about the query and its params.
6. Learn about the HTTP verbs.
7. Learn about the HTTP Response Status Codes.
8. Learn about the HTTP Message Format.
9. Learn how you can inspect HTTP messages using Google Developer Tools.
10. Learn about the URL encoding of query params.
11. Learn how to practice HTTP using telnet.
12. Learn how to specify which Web site you want to access.
13. Learn how browsers fetch Web page from remote Web servers.
14. Learn about `curl`.

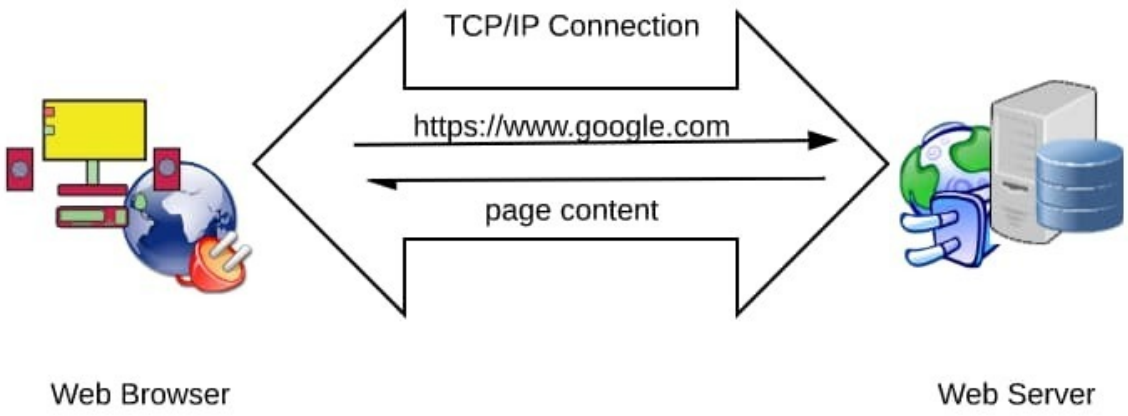
Introduction

HTTP, Hypertext Transfer Protocol is an application layer protocol that relies on TCP/IP and it is used for communication between clients and servers over the Web. It is the protocol that a browser uses to connect to a Web server and ask for information. Its secure version is the HTTPS and it is becoming ubiquitous little-by-little.

HTTP Request/Response

HTTP is a request/response paradigm of communication and relies on TCP/IP which guarantees the delivery of messages from sender to recipient and vice-versa.

There is a server that expects an HTTP Request from a client, e.g. a Web Browser. The client sends the HTTP Request and signals the end of the request transmission. It then puts itself into a reading mode (over the same connection, which is bidirectional) and reads the HTTP Response coming in from the server.



Browser Sends an HTTP Request to Get the Content of a Page

HTTP Port

The well-known port for TCP/IP connections that would serve the HTTP protocol is port number `80`. When you use your browser to get the contents of a page, e.g. to get the contents of `http://www.google.com`, your browser will use, by default, the port number `80`, to connect to the Web server that listens for connections behind the host `www.google.com`.

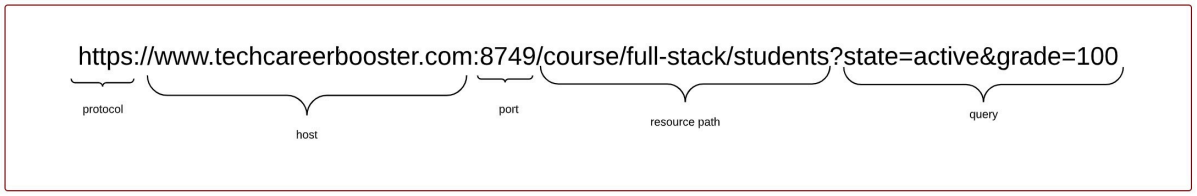
HTTPS Port

On the other hand, the well-known port for TCP/IP connections that HTTP Secure (HTTPS) protocol uses is `443`. So, whenever you access `https://www.google.com`, then your browser is setting up a connection to a remote server listening on port `443`.

URL - Uniform Resource Locator

An HTTP Request includes a Uniform Resource Locator, i.e. a URL which specifies what this request is interested in retrieving as an answer.

The components of the URL such as this: `https://www.techcareerbooster.com:8749/course/full-stack/students?state=active&grade=100` are the following:



URL Decomposition

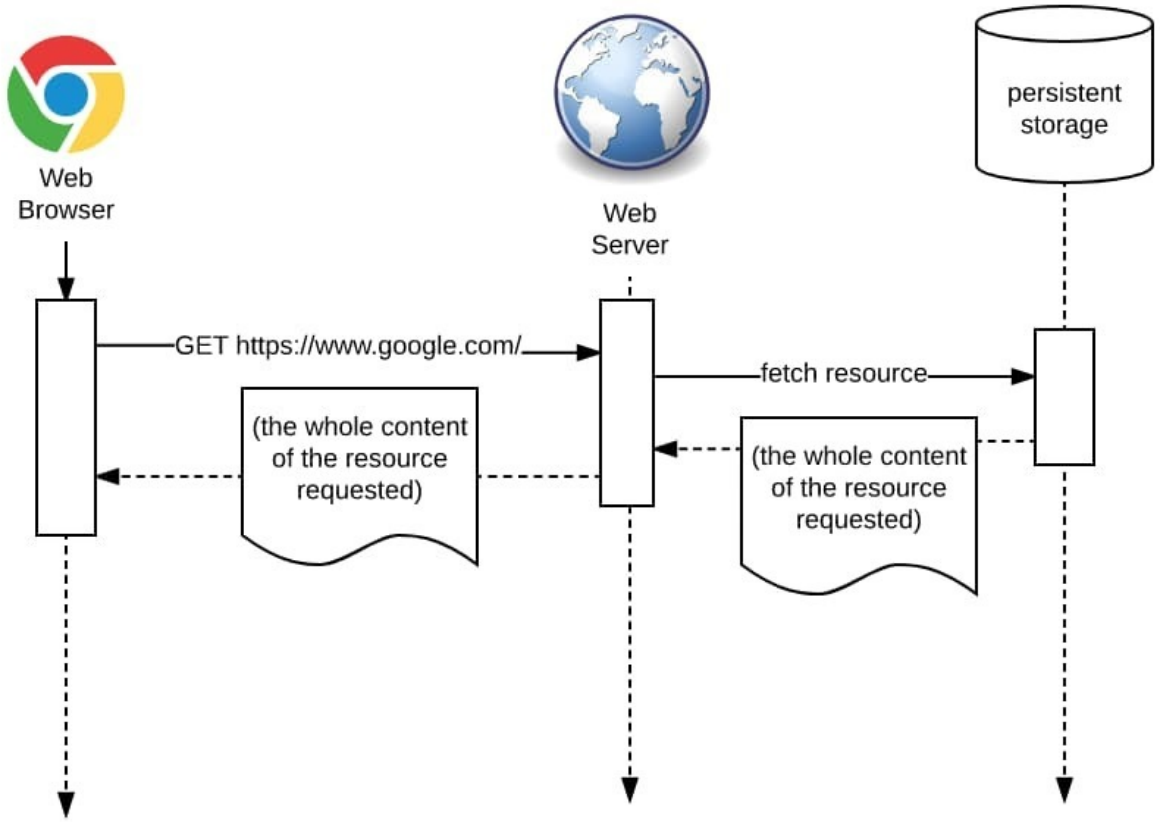
1. The protocol. In this example, this is `https`. The protocol can be either `http` or `https`. When the protocol is `https`, then any data that are sent from client to server, i.e. from your browser to the Web server, are being encrypted.
2. The host. In this example, this is `www.techcareerbooster.com`. The host name is being translated to an IP address with the help of Domain Name Servers.
3. The port. In this example, the port is `8749`. When the port is omitted and the protocol is `http`, then the port `80` is used. But when the protocol is `https`, then the default port is `443`.
4. Path to the resource. In this example, this is `/courses/full-stack/students`. It tells Web server which resource on that server the client is interested in.
5. The query and its params. In this example it is `state=active&grade=100`. The query is separated from the resource path using the symbol `?`. The query is composed of a series of query params. In this example with have `2` params, the `state` and the `grade`. Each param comes with a value which is separated from the param name using the symbol `=`. Each param/value pair is separated from the next using the symbol `&`. The params are not mandatory, but when present tell the Web server that the client is interested in getting the particular resource information under some kind of filtering based on the param values. It is the responsibility of the Web application processing those params to return back the requested information.

HTTP Verbs

As we said above, the client uses HTTP to send a request that bares a URL. But, besides the URL, that is the full address of the resource the client is interested in, HTTP requires the client to tell what they want to do with the particular resource. Client tells what they want to do with the particular resource using well-defined *HTTP verbs*.

`GET`

The most popular verb is the `GET` verb which tells server that the client wants to retrieve the information related to the resource specified in the URL. The URL needs to contain everything (host, path to resource, query parameters) the Web application needs in order to return back the information the client is asking to retrieve.



Web Browser issuing a GET Request

When you use your Web browser to access / fetch a Web page, it is the `GET` verb that is internally used by your browser.

`POST`

This verb is used when we want to tell the Web application to *create* a resource. Client sends a `POST` request with all the data encoded in the body of the request (we will talk about head and

body of the HTTP request later on). Some clients may also send the data as query params but this is not usually a good practice.

Web application will process the data sent with the HTTP request and will update its state. For example, it will update a record into a database. So `POST` requests are dangerous and whenever a client sends such a request, client needs to know that the server side of things will permanently be updated. (Have you ever seen your browser popping up a dialog asking whether you are sure that you want to resubmit? It tries to protect you from resending a `POST` request accidentally).

When you use your Web browser to fill in a form and submit your data, your browser will usually use a `POST` request to the remote server.

`PUT` or `PATCH`

This verb is used when the client wants to *update* the resource specified in the URL. Similar to `PUT` is the `PATCH` verb and can be used in the place of `PUT`.

`DELETE`

This verb is used when the client wants to *delete* a resource.

Note that `PUT` and `DELETE` might not be supported by some clients. In that case, the client might use `POST` instead and specifying the actual intention as part of the data sent in the body of the request.

`HEAD`

This is very useful because it can be used to check whether a resource exists and if it exists whether it has been changed from the previous try to get it. It does not return the whole resource itself. So, it minimizes the amount of information that travels from server end to client end.

Verbs and Usages

This is a table with the most common HTTP Verbs and their Usages

HTTP Verb	Usage
`GET`	Requesting a resource, e.g. a page
`POST`	Create a resource, e.g. a book on a bookstore application
`PUT` or `PATCH`	Update a resource, e.g. a book price
`DELETE`	Delete a resource, e.g. a book
`HEAD`	Checking whether a resource has been updated without actually fetching the resource

Response Status Codes

We have learned about the URL, that is part of the HTTP request, and about the HTTP verbs. There are other things that are part of the HTTP request, but before learning about these, let's see one very important part of the HTTP response. It is the HTTP Response Status code.

The HTTP Response Status code gives instructions to client on how to interpret the response content.

1XX

The HTTP Response Status code in the range `100 - 199` are informational messages. Usually, when sent by the server, this implies that the server is still processing the request and a final response will soon be returned.

2XX

Any code in the range `200 - 299` denotes success of the processing on the server side. The most common code is `200` which is also called `OK`. For a `GET` request, the server replies with an HTTP Response with status code `200` and the resource data encoded in the body of the response. If the verb was `POST`, then a `200` response might contain a piece of information that would allow the client to fetch the created resource in the subsequent call. For example, it might contain the URL to the newly created resource.

Usually, we should be expecting a return status code of `201`, when a new resource is created. However, this is a rule that it may not be followed by all Web applications. There are Web applications that return a `200` instead of a `201` when they create a new resource.

The code `202` means that the request has been accepted, but not fulfilled immediately. The fulfillment would take place asynchronously. It may be fulfilled or not.

3XX

This response code indicates that a resource has been moved and that the client should be redirected to another resource (page).

In fact, `301` means that the resource has been moved permanently and the resource should now be looked at at a different URL. The different URL is included as part of the response.

`302`, on the other hand, usually means temporary redirection. In other words, although the current request should now be repeated to another URL, future ones should be done to the original URL.

`304` is very useful to implement some caching mechanism. It tells client that the resource has not been modified since the last request and, for that reason, the response body, besides the status code (`304`) does not contain the actual resource information, assuming that the client already has the cached copy.

4XX - Client Side Errors

These response status codes informing the client that there is something wrong on the client side. In other words, that something didn't go as expected due to incorrect or missing information provided by the client.

The most common response status code is `404`, which means `Not Found`. This is returned to the client that is requesting to get a resource that cannot be found.

Other common codes are:

- `400` which is a `Bad Request`. This is usually returned when the server believes that the client didn't form the request data correctly.
- `401` or `Unauthorized`. This is returned when the server cannot authenticate the client request, i.e. credentials provided are wrong. Note that although this response code is officially called `Unauthorized`, it should be returned when server does *authentication* and authentication fails. Not when server does *authorization*. For failed authorization, the response code should be `403` (`Forbidden`).
- `403` or `Forbidden`. This is returned when the client has been authenticated, so server knows who the client is, but this client is not allowed to access the resource requested.
- `405` or `Method Not Allowed`. This is returned when the client is sending an HTTP Verb that is not supported by the server for the particular resource.
- `406` or `Not Acceptable`. This is returned when the client is sending data in a format that the server does not support, or if the client is asking the server to return the data in a format that the server does not support.
- `422` or `Unprocessable Entity`. This error means that the client is asking server to carry out an action with data that are not valid, business-wise. For example, the client might be sending customer details to be saved as a new customer, without sending the customer last name, when customer last name is required by the Web application.

5XX - Server Side Errors

This family of response status codes is there to indicate to the client that a server-side error has taken place.

The most common error code is `500`, which is the `Internal Server Error`. This error code is returned when the server side cannot actually give any more specific information. Another common error is `503`, `Service Unavailable`. This usually means that the server cannot respond due to overload or because it is down due to maintenance. Usually, this is more temporary situation than the `500` case.

This is a table that summarizes the ranges of HTTP status codes:

HTTP Status Code Range	Purpose
`1XX`	Informational Response. Rare. Usually, not final.
`2XX`	Success
`3XX`	Redirection or Not Changed
`4XX`	Client Error
`5XX`	Server Error

> *Information:* Before I continue, let me just make clear, the difference between *authentication* and *authorization*. Authentication is carried out by the server in the input of a username and a password (usually). Using the credentials (username and password) server identifies who is behind the request. On the other hand, authorization makes sure that the authenticated and identified user has access to a particular resource or not. For example, an admin user might have access to admin pages of a Web application, whereas a standard user might not.

HTTP Message Format

We have learned about the main constituent parts of the HTTP protocol. Let's now see the actual HTTP message format.

The format of an HTTP message, either request or response, is the same and has as follows:

Request Line Or Status Line

The first line is either called the *Request Line*, if the HTTP is a request, or *Status Line*, if the HTTP is a response. In any case, the first line is separated from the second one using a new line character

Message Headers

After the first line (request or status), the message may be composed of 0 or more message headers. Each message header is sent in its own line. The message headers give information to the recipient about the resource and may also help recipient decode the message body that follows.

An Empty Line Separator

After the (optional) message headers, or after the first line (if the message headers are not present), there has to be an empty line. This empty line is mandatory and either marks the end of the HTTP message, when a message body does not follow, or separates the head from the message body if message body exists.

The Message Body

The message body is optional and, if present, follows the empty line separator described above.

HTTP Request Message Example

This is an example of an HTTP Request Message:

```
POST https://www.techcareerbooster.com/contact_us_emails HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6,el;q=0.4
Connection: keep-alive
Content-Length: 309
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Cookie:_techcareerbooster_session=RVVhbl3779b43660; _ga=GA1.2.1475987; _gid=GA1.2\
.052; _gat=1; __zlcmid=gFgOzLF
Host: www.techcareerbooster.com
Origin: https://www.techcareerbooster.com
```

```
12 Referer: https://www.techcareerbooster.com/
13 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (K\
14 HTML, like Gecko) Chrome/58.0.3029.96 Safari/537.36
15 X-Requested-With: XMLHttpRequest
16
17 contact_us_email%5Bname%5D=Panos&contact_us_email%5Bsubject%5D=Test+Subject&conta\
18 ct_us_email%5Bemail%5D=panosm%2B201705072117%40techcareerbooster.com&contact_us_e\
19 mail%5Bmessage%5D=Test+Body&authenticity_token=PSaRpPT2YVjumsCTMTVapim7%2B4GJR%2B\
20 N%2B36usQwmA8iBT%2FYFNywdS5UUiA%3D%3D
```

(the above code snippet online)

(The line numbers above are not part of the message)

Let's see some details of the above example:

1. The line `POST https://www.techcareerbooster.com/contact_us_emails HTTP/1.1` is the *request line*. The request line always starts with the HTTP verb. In this particular case is `POST`. Then we have the URL that we want to refer to. After that, we have the HTTP version specification. Currently, having the value `HTTP/1.1` denoting that we are using HTTP version 1.1. which is the most popular version today.
2. Lines 2 till 13 are message headers. The message headers give extra information about the request and the client itself. Server will process the information inside the headers and act accordingly. The headers have a name and a value. The name ends with a `:` character that separates the name from the value. The value starts after a single space following the `:` character. In the above example we can see headers like:
    1. `Accept-Encoding` with value `gzip, deflate, br`
    2. `Content-Length` with value `309`
    3. `Content-Type` with value `application/x-www-form-urlencoded; charset=UTF-8`
    4. and many more
3. Line 14 is the empty line that delineates the end of the message headers.
4. Line 15 is the start of the message body. Note that in the particular example above, the message body has `309` characters, equal to the number of characters specified in the `Content-Length` header. I guess that you get the point here. The `Content-Length` is very important because it will tell the server how long the message body is and, hence, server will check the consistency of the message body, by comparing its actual length to the expected length given in `Content-Length`.

It is also very important to understand that each header may have its own encoding of their values. For example `Accept-Encoding` has a value which is a comma-separated list of other values (e.g.: `gzip, deflate, br`). On another example, the `Content-Type` has values that are semi-colon-separated list of other values (e.g.: `application/x-www-form-urlencoded; charset=UTF-8`). Hence, the more you do HTTP and Web programming, the more you will get familiar with these headers and their values. You only have to google for HTTP headers and you will find an abundance of information about these pieces of information in the request and response messages.

HTTP Response Message Example

The following is an HTTP response message example:

```
1  HTTP/1.1 200 OK
2  Cache-Control: no-cache
3  Connection: keep-alive
4  Content-Type: text/html
5  Date: Sun, 07 May 2017 18:48:22 GMT
6  Server: Cowboy
7  Set-Cookie:_techcareerbooster_session=U2lDeEdlQVJnT1ZBPT0tLW1lbTgyN1RRcFNQSVFtOWN\
8  OcGplR3c9PQ%3D%3D--a727e75bc6e61b2f98681fb58; path=/; secure; HttpOnly
9  Strict-Transport-Security: max-age=31536000
10 Transfer-Encoding: chunked
11 Via: 1.1 vegur
12 X-Content-Type-Options: nosniff
13 X-Frame-Options: SAMEORIGIN
14 X-Request-Id: c68c6ab0-79e5-422d-ad33-a2bb89162c55
15 X-Runtime: 0.026712
16 X-Xss-Protection: 1; mode=block
```

(the above code snippet online)

Let's see the details of it:

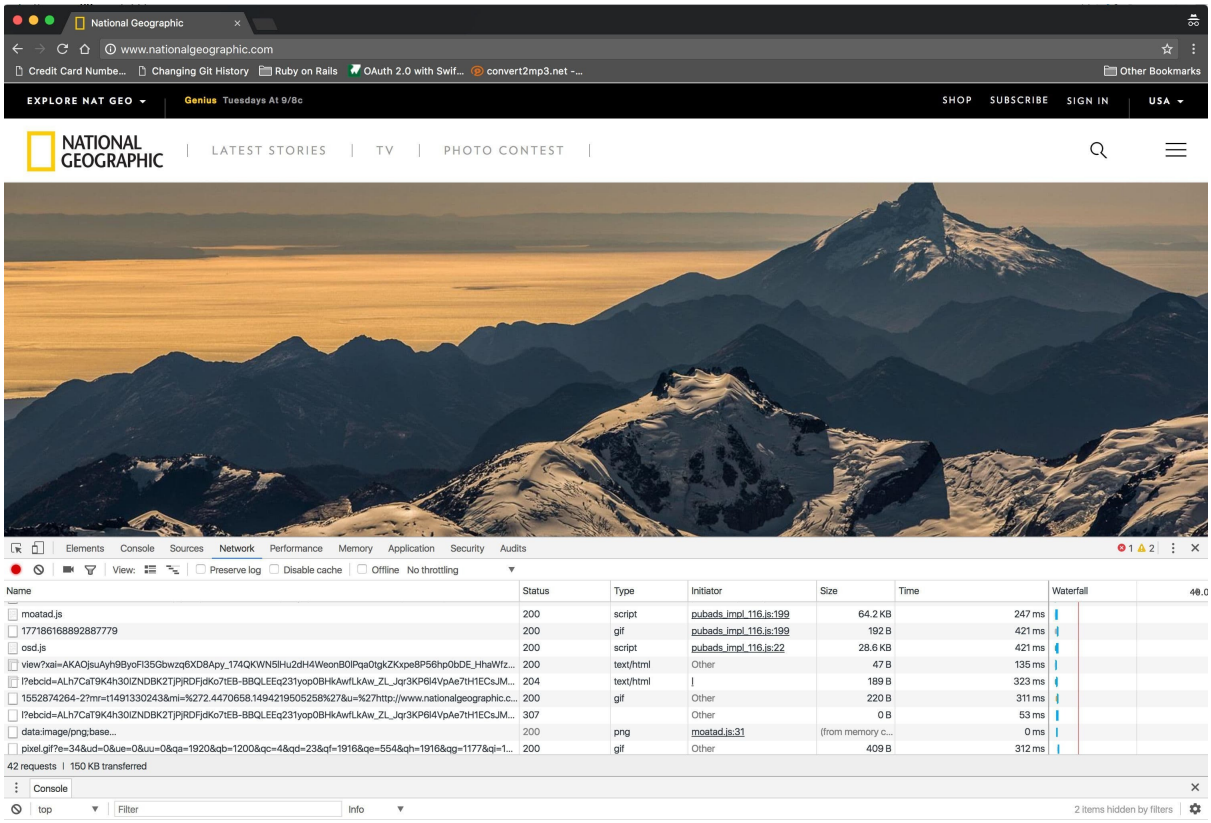
1. The first line is the *status line* of the HTTP response message. It is composed of three parts. The HTTP version, here being `HTTP/1.1`. Then is the Status Code, here being `200`. Finally, we see the word `OK` which is the *Reason Phrase*.
2. Lines 2 till 15 are the response headers. These are encoded, one header per line. And the name of the header is separated from the value using the `:` character.
3. This example response does not have message body.

A `GET` Request Response Example

We continue with another example, which is a `GET` request that will be issued by our browser to a Web server. In order to inspect the message details, we will use our Google Developer Tools, and in particular, the *Network* tab.

(1) Open your browser and visit the page https://www.nationalgeographic.com.

(2) Open your developer tools and make sure that you have the *Network* tab open. You should see something like this:



National Geographic With Dev Tools Open On Network Tab

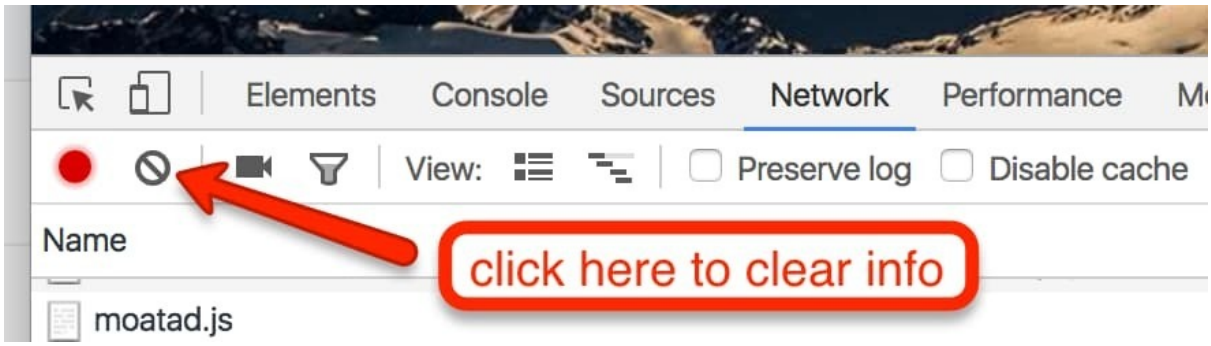
(3) Hover over (but **don't click**) the link *Latest Stories* or

over any link that you feel like. Keep a note of the URL that appears at the bottom. I.e. the URL that the browser would follow if you were to click that link.

On my case the URL is `www.nationalgeographic.com/latest-stories/?source=sitenavstories`. This is going to be the URL for the `GET` request that will take place as soon as we click on the link.

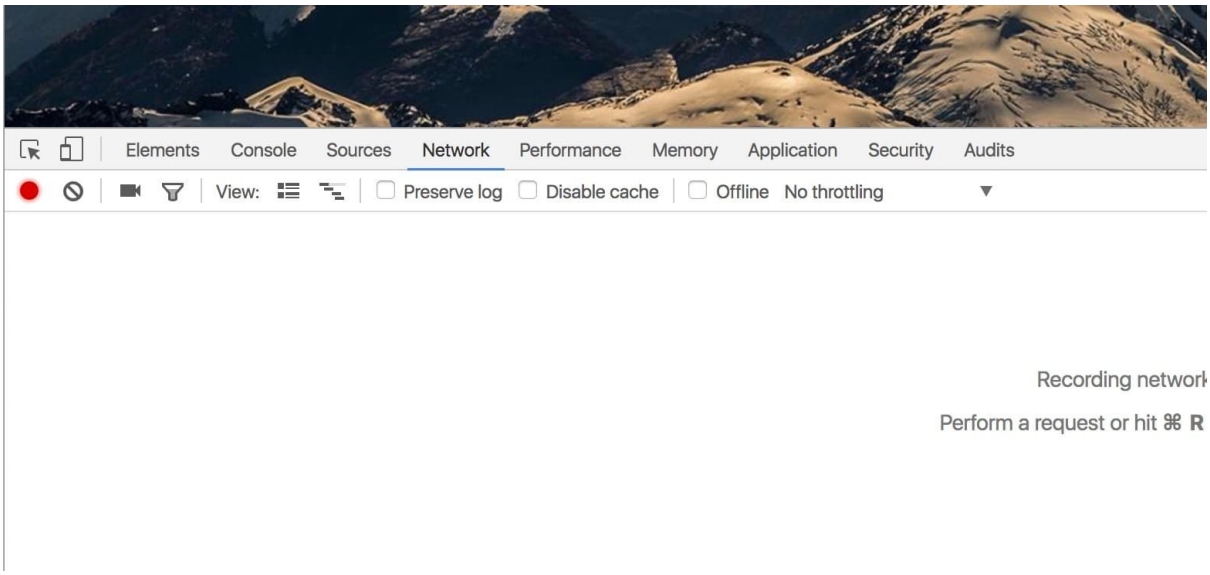
(4) Clear the Network Tab information:

Click on the corresponding button in order to clear the network tab information.



Clear the Network Tab Information

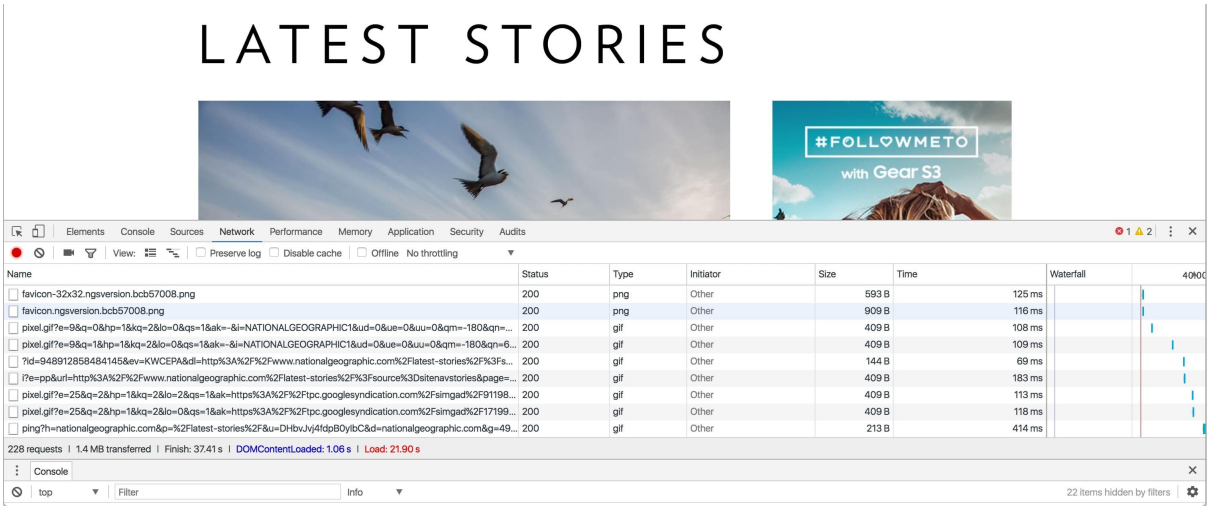
You should see something like this:



Cleared Network Tab

(5) Now Click on The Link ...

... and watch the network tab being filled in with lots of HTTP requests. Something like this:



Latest Stories Network Tab with HTTP Requests

When you load a page, usually numerous HTTP requests are being sent to the corresponding Web server. And this was an example of such a case. You can click on any of them and see its details, like the headers, the request body and the response status code.

(6) Click on the First Entry...

... which will correspond to the first HTTP request, i.e. to the `GET` request to visit that link that you spotted on step (3) above.

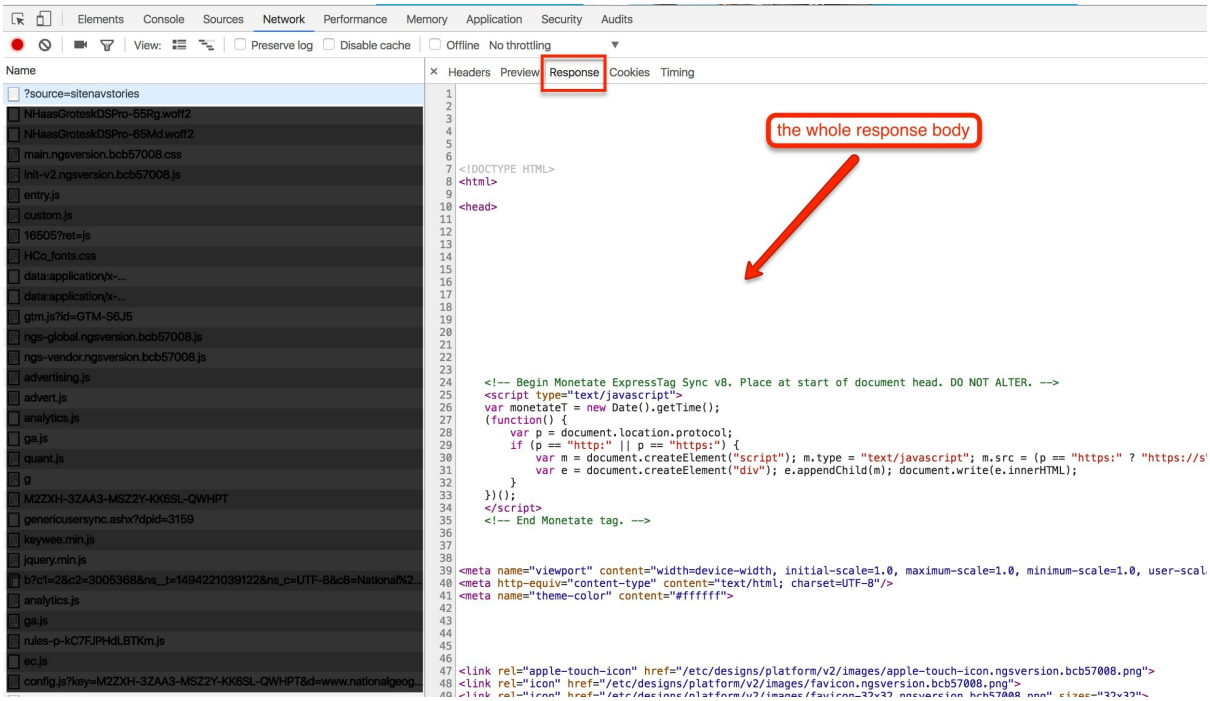
![Analyze HTTP Request Data in Dev Tools - Headers Tab]images/sections/14-introduction-to-web-development/chapters/01-http/analyze-http-request-data-in-dev-tools.jpg)

The *Headers* tab bears a lot of information about your HTTP request. You can clearly see the URL, the HTTP Verb and the Status Code. The Response Headers, the Request Headers and the

Query parameters.

(7) Click on the *Response* Tab...

... and you will see the response body, which, in this particular case is an HTML document.



Response Body viewed From Response Tab

URL Encoding Of Query Param Names and Values

We saw earlier that the URL specifying the resource we want to act upon, might contain a *query*. We also saw that the query is being encoded following specific rules.

1. It starts with the `?` symbol.
2. Each param/value pair is given using a `=` to separate the name from the value.
3. One pair from the other is separated using the `&`.

But what happens if the value of a param includes the `&`? Or the `=` symbol? Or what happens if the name of a param has a space?

```
first name=Panos&last name=Matsinopoulos&address=18st Cross Street&Angel Street&c\
ity=London
```

(the above code snippet online)

In the above example, there are two parameters that have names that include a blank space as part of the name. Also, the `address` parameter has value that includes both spaces and the symbol `&`.

Trying to use a URL that would include the above query wouldn't work.

This is where the idea of URL encoding comes into place. You need to encode the names and the values of your query so that they can be part of the URL.

Here is an online site that does URL encoding and decoding. Use that to encode the names and values of the above example:

1. `first name` becomes `first%20name`.
2. `last name` becomes `last%20name`.
3. `address` value `18st Cross Street&Angel Street` becomes `18st%20Cross%20Street%26Angel%20Street`

Now, the query, with proper encoding to be used in the URL has the value:

```
first%20name=Panos&last%20name=Matsinopoulos&address=18st%20Cross%20Street%26Angel%20Street&city=
```

The URL encoding is also called *Percent encoding* and you can google for that. In Ruby, you can use the `CGI.escape()` method call to encode a value following the URL encoding rules.

Practice HTTP using `telnet`

Now that we know the basics of the HTTP protocol, let's do some practice using `telnet`. `telnet` allows you to connect to a remote server (by defining its IP and port) and exchange text messages. It uses the Telnet protocol, but we don't have to know the details of it in order to practice some HTTP requests. We will use it to exchange HTTP protocol messages.

Telnet Example 1

On your bash terminal issue the following command:

```
$ telnet rubyonrails.org 80
Trying 192.30.252.153...
Connected to rubyonrails.org.
Escape character is '^]'.
```

(the above code snippet online)

`telnet` starts and connects to host `rubyonrails.org` on port `80`. This means that it connects to the Web server sitting behind this IP and port. Then, `telnet` waits for you to give the strings/data to send over to the remote server.

Type the following:

```
GET / HTTP/1.1
```

(the above code snippet online)

… together with an empty line. This is an example of an HTTP request that does not have headers. Only the Request Line.

You will get back this:

```
HTTP/1.1 400 Bad Request
Server: GitHub.com
Date: Mon, 08 May 2017 11:39:06 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 166
X-GitHub-Request-Id: FA97:4F47:591010:7A14B3:591058D5

<html>
<head><title>400 Bad Request</title></head>
<body bgcolor="white">
<center><h1>400 Bad Request</h1></center>
<hr><center>nginx</center>
</body>
</html>
Connection closed by foreign host.
```

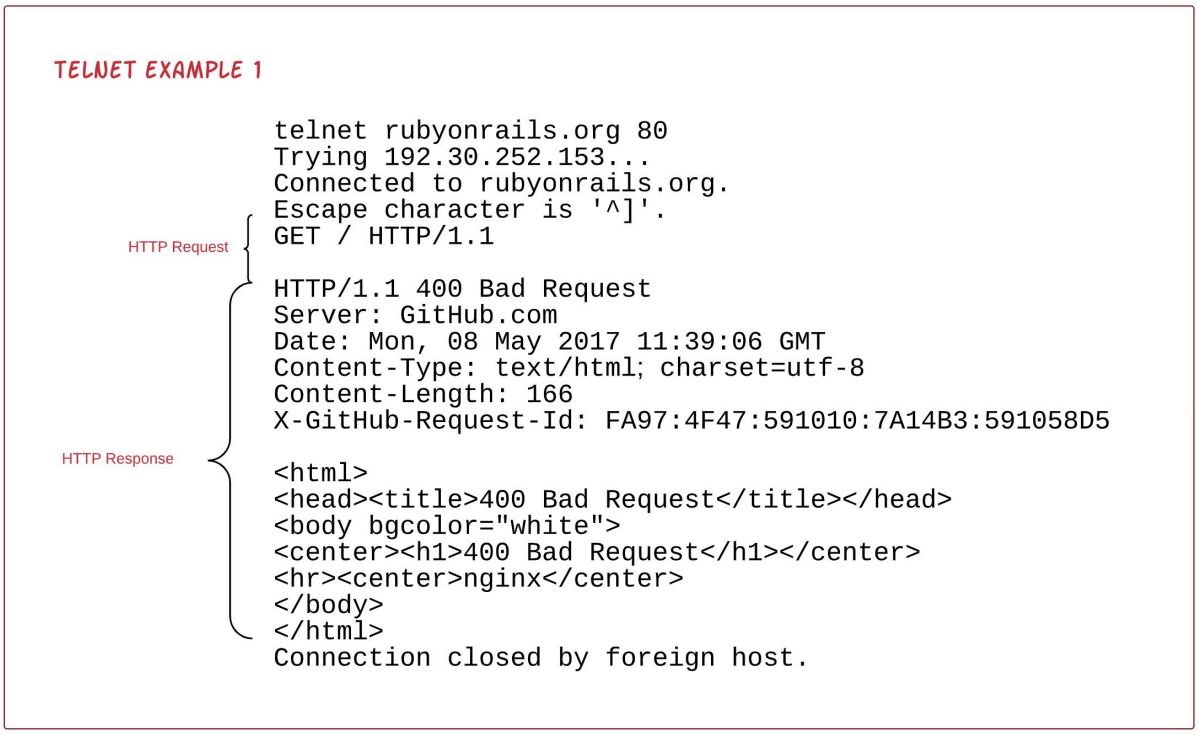
(the above code snippet online)

Great! You just sent an HTTP request to the `rubyonrails.org:80` server, to retrieve the `/` root resource sitting behind this Web server. And you just got back a response with status code `400 Bad Request`. The response has a `Server` header, which reveals where the reponse came from and also, it tells you that it has some reponse body data with length equal to `166` bytes. It tells you also, that the type of the response body is `text/html`, which means that it is a proper HTML page:

```
<html>
<head><title>400 Bad Request</title></head>
<body bgcolor="white">
<center><h1>400 Bad Request</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

(the above code snippet online)

This is the full interaction with the `rubyonrails.org` Web server as you experienced it above:



Telnet with HTTP Request and Response - Example 1

Telnet Example 2

The problem with the previous request that we did was that we didn't specify which Web site we wanted to fetch. We just gave `/`, in the resource URL. So, we basically said to the remote server that we need the default resource for the default Web site. And this didn't actually work. Most of the Web servers serve many Web sites and we need to tell them which Web site we are interested in. Hence, with HTTP, the pair IP/Port is not usually enough. We need to specify the Web site too.

> Please, don't get confused. The fact that we gave `rubyonrails.org` on the command line on telnet, didn't actually tell anything to the HTTP request that we issued using telnet. It was only used by `telnet` at the beginning to try to locate the remote server (by doing a host-to-IP translation using DNS).

Try the following telnet session:

```
1 $ telnet rubyonrails.org 80
2 Trying 192.30.252.153...
3 Connected to rubyonrails.org.
4 Escape character is '^]'.
5 GET / HTTP/1.1
6 Host: rubyonrails.org
```

(the above code snippet online)

You can see that except from the request line `GET http://www.rubyonrails.org HTTP/1.1`, we also send the header `Host` with value `rubyonrails.org`. This header is used to tell the remote server listening on port 80, which Web site we are interested in.

Now, after you issue the empty line (following the `Host: rubyonrails.org` line), you will get back this:

```
HTTP/1.1 200 OK
Server: GitHub.com
Date: Mon, 08 May 2017 12:00:49 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 5564
Last-Modified: Fri, 28 Apr 2017 11:51:53 GMT
Access-Control-Allow-Origin: *
Expires: Mon, 08 May 2017 12:10:49 GMT
Cache-Control: max-age=600
Accept-Ranges: bytes
X-GitHub-Request-Id: D141:4F4D:111CBDD:175FB4C:59105DEA

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <link rel="stylesheet" href="/style.css" type="text/css" media="screen" chars\
et="utf-8">
    <meta name="google-site-verification" content="Jknoaf3CvRzba3wXMxqgurDErIGZ9e\
1L1luuDMhQYEI" />
    <!-- Begin Jekyll SEO tag v2.2.0 -->
<title>Ruby on Rails | A web-application framework that includes everything neede\
d to create database-backed web applications according to the Model-View-Controll\
er (MVC) pattern.</title>
<meta property="og:title" content="Ruby on Rails" />
<meta property="og:locale" content="en_US" />
<meta name="description" content="A web-application framework that includes every\
thing needed to create database-backed web applications according to the Model-Vi\
ew-Controller (MVC) pattern." />
<meta property="og:description" content="A web-application framework that include\
s everything needed to create database-backed web applications according to the M\
odel-View-Controller (MVC) pattern." />
<link rel="canonical" href="http://rubyonrails.org/" />
<meta property="og:url" content="http://rubyonrails.org/" />
<meta property="og:site_name" content="Ruby on Rails" />
<meta name="twitter:card" content="summary" />
<meta name="twitter:site" content="@rails" />
<script type="application/ld+json">
{"@context": "http://schema.org",
```

```
"@type": "WebSite",
"name": "Ruby on Rails",
"headline": "Ruby on Rails",
"description": "A web-application framework that includes everything needed to cr\
eate database-backed web applications according to the Model-View-Controller (MVC\
) pattern.",
"url": "http://rubyonrails.org/"}</script>
<!-- End Jekyll SEO tag -->

  </head>

  <body>
    <div class="container">
          <div class="nav">
      <ul>
        <li class="first"><a href="http://weblog.rubyonrails.org/">Blog</a></li>
        <li><a href="http://guides.rubyonrails.org/">Guides</a></li>
        <li><a href="http://api.rubyonrails.org/">API</a></li>
        <li><a href="http://stackoverflow.com/questions/tagged/ruby-on-rails">Ask\
 for help</a></li>
        <li><a href="https://github.com/rails/rails">Contribute on GitHub</a></li>
      </ul>
    </div>

    <section>
      <p class="mobile-center">
        <img src="/images/rails-logo.jpg" width="220" height="78" alt="Ruby on Ra\
ils">
      </p>
    </section>

    <section>
      <figure class="right">
        <img src="/images/imagine.png" alt="So many possibilities.">
      </figure>

      <h1>Imagine what you could build if you learned Ruby on Rails&amp;#8230;</h1>
      <p>Learning to build a modern web application is daunting. Ruby on Rails ma\
kes it much easier and more fun. It includes <a href="everything-you-need">everyt\
hing you need</a> to build fantastic applications, and <a href="http://guides.rub\
yonrails.org/getting_started.html">you can learn it</a> with the support of <a hr\
ef="community">our large, friendly community</a>.</p>
    </section>

    <section class="version">
      <p><a href="http://weblog.rubyonrails.org/2017/4/27/Rails-5-1-final/">Lates\
```

```
t version &mdash; Rails 5.1.0 <span class="hide-mobile">released April 27, 2017</\
span></a></p>
      <p class="show-mobile"><small>Released April 27, 2017</small></p>
    </section>

    <section class="video-container">
      <iframe src="https://www.youtube.com/embed/OaDhY_y8WTo" frameborder="0" all\
owfullscreen class="video"></iframe>
    </section>

    <section class="interior">
      <figure class="left">
        <img src="/images/rubyrails.png" alt="Rails does it all.">
      </figure>

      <p><strong>You've probably already used many of the applications that were \
built with Ruby on Rails:</strong> <a href="https://basecamp.com">Basecamp</a>, <\
a href="https://github.com">GitHub</a>, <a href="https://shopify.com">Shopify</a>\
, <a href="https://airbnb.com">Airbnb</a>, <a href="https://twitch.tv">Twitch</a>\
, <a href="https://soundcloud.com">SoundCloud</a>, <a href="https://hulu.com">Hul\
u</a>, <a href="https://zendesk.com">Zendesk</a>, <a href="https://square.com">Sq\
uare</a>, <a href="https://highrisehq.com">Highrise</a>. Those are just some of t\
he big names, but there are literally hundreds of thousands of applications built\
 with the framework since its release in 2004.</p>

      <p><strong>Ruby on Rails is open source software</strong>, so not only is i\
t free to use, you can also help make it better. <a href="http://contributors.rub\
yonrails.org">More than 4,500 people</a> already have contributed code to Rails. \
It's easier than you think to become one of them.</p>

      <p><strong>Optimizing for programmer happiness with Convention over Configu\
ration</strong> is how we roll. Ruby on Rails has been popularizing both concepts\
 along with a variety of other controversial points since the beginning. To learn\
 more about why Rails is so different from many other web-application frameworks \
and paradigms, examine <a href="doctrine">The Rails Doctrine</a>.</p>
    </section>

    <section class="more">
      <figure>
        <img src="/images/more.png" alt="More resources">
      </figure>

      <p>Keep up to date with <a href="https://twitter.com/rails">Rails on Twitte\
r</a> and <a href="https://rails-weekly.ongoodbits.com">This Week in Rails</a></p\
>

```

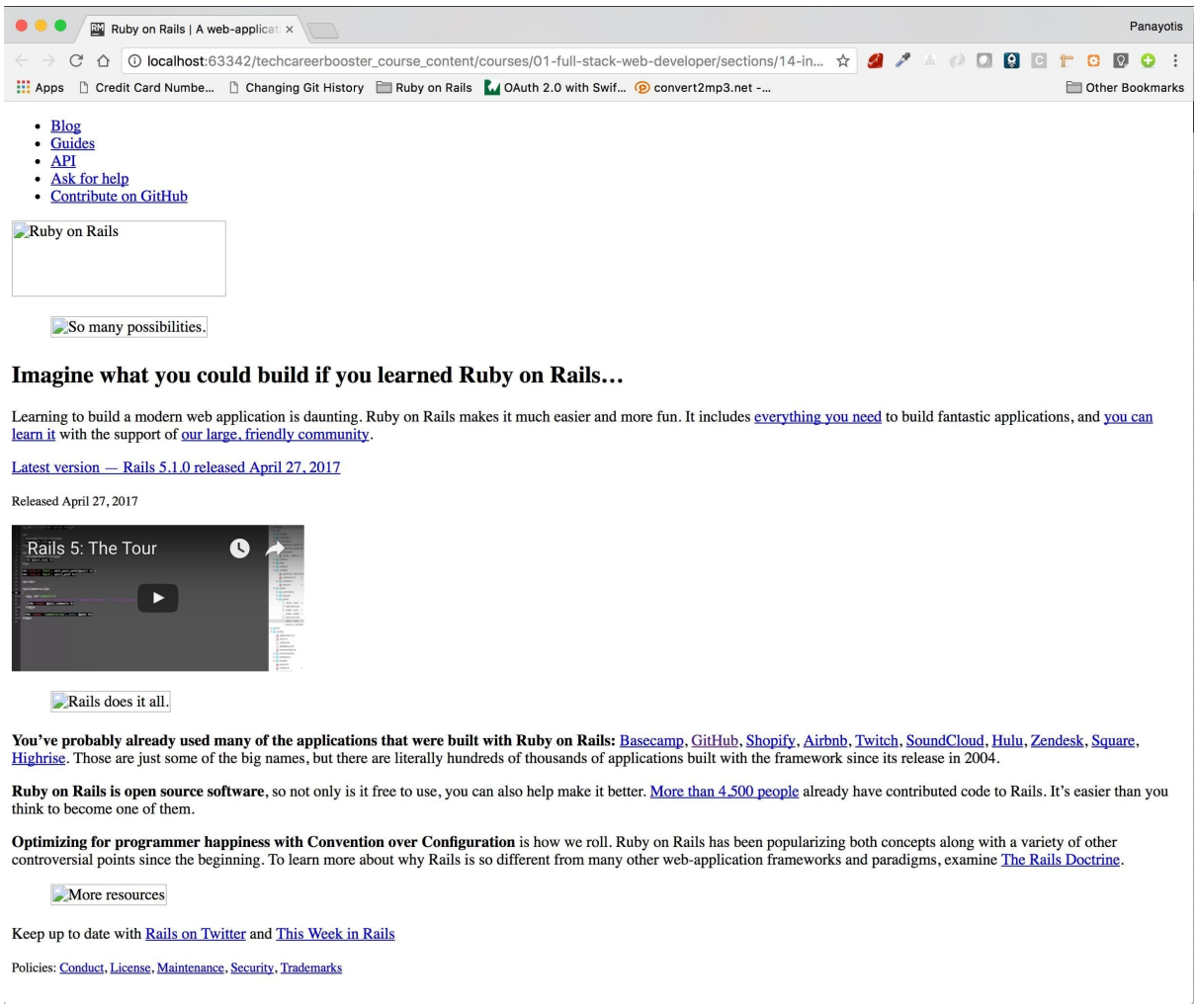
```
133       <p><small>Policies: <a href="conduct">Conduct</a>, <a href="http://opensour\
134 ce.org/licenses/MIT">License</a>, <a href="maintenance">Maintenance</a>, <a href=\
135 "security">Security</a>, <a href="trademarks">Trademarks</a></small></p>
136     </section>
137 
138     </div> <!-- /.container -->
139   </body>
140 </html>
```

(the above code snippet online)

which is the whole page that corresponds to `http://rubyonrails.org`.

The first line (`HTTP/1.1 200 OK`) tells us that the `GET` request was successfully served. Also the `Content-Type` header in the response tells us that the response body is an HTML page. We also have its length being `5564` bytes.

Try to save the response body into an `.html` document. e.g. `ror.html` and then try to open this document with your browser. You will see something like this:



RoR Plain HTML

Telnet Example 3

We used `telnet` to get the HTML content of the page `rubyonrails.org`. And we saved the content and loaded the page into our browser. Why the page is not displayed as it has to? It seems that it is missing the style sheets. But this is how browsers work more or less. They first fetch the HTML document and then they fetch the related other assets, based on the references in the HTML document itself.

For example, the HTML that we have got as response back from the example include a line like this:

```
<link rel="stylesheet" href="/style.css" type="text/css" media="screen" charset="\
utf-8">
```

(the above code snippet online)

This means that the browser needs to issue another HTTP request to get the file `style.css`.

Let's do this with our `telnet` technique:

```
$ telnet rubyonrails.org 80
Connected to rubyonrails.org.
Escape character is '^]'.
GET /style.css HTTP/1.1
Host: rubyonrails.org

HTTP/1.1 200 OK
Server: GitHub.com
Date: Mon, 08 May 2017 12:12:35 GMT
Content-Type: text/css
Content-Length: 4882
Last-Modified: Fri, 28 Apr 2017 11:51:49 GMT
Access-Control-Allow-Origin: *
Expires: Mon, 08 May 2017 12:22:35 GMT
Cache-Control: max-age=600
Accept-Ranges: bytes
X-GitHub-Request-Id: DC79:4F4D:1128EE4:17705D4:591060A9

/*--------------------------------------------------
GENERAL
--------------------------------------------------*/

body {
  font-family: Georgia, serif;
  line-height: 2rem;
  font-size: 1.3rem;
  background-color: white;
  margin: 0;
```

```
  padding: 0;
  color: #000;
}

small { font-size: 1rem; }

img { border: 0; }

a { color: #cc0000; text-decoration: none; }

p a:hover { text-decoration: underline; }

h1 {
  font-weight: normal;
  line-height: 2.8rem;
  font-size: 2.5rem;
  letter-spacing: -1px;
  color: black;
  margin-bottom: 1rem;
}

h2 {
  font-weight: normal;
  font-size: 1.8rem;
  color: black;
  margin-top: 2rem;
  margin-bottom: -0.5rem;
}

.container {
  width: 960px;
  margin: 0 auto 40px;
  overflow: hidden;
}

.show-mobile { display: none; }

/*-------------------------------------------------
NAV
-------------------------------------------------*/

.logo { text-align: center; margin: 1rem 0; }
.logo a { border: 0; }

.nav { margin-top: 30px; width: 500px; float: right; font-size: 14px; }
.nav ul { margin: 0; padding: 0; }
```

```
75  .nav ul li { display: inline; list-style-type: none; margin-left: 20px; }
76  .nav ul li.first { margin-left: 5px; }
77  .nav ul li a { padding-bottom: 1px; font-weight: bold; border-bottom: 2px solid #\
78  cc0000; }
79  .nav ul li a:hover { color: #000; border-bottom: 2px solid #000; }
80
81  .back { margin: 4rem 0; text-align: center; font-size: 1rem; }
82
83  /*-------------------------------------------------
84  LAYOUT
85  -------------------------------------------------*/
86
87  section {
88    margin: 0 auto 2rem;
89    width: 700px;
90  }
91
92  section.more { overflow: hidden; text-align: center;}
93  section.more figure img { width: 250px; height: 120px;}
94
95  figure { margin: 0; }
96  figure img { width: 375px; height: 375px; }
97  figure.right { float: right; margin-right: -5rem; }
98  figure.left { float:left; margin-left: -5rem; }
99
100 .version {
101   text-align: center;
102   overflow: hidden;
103   margin: 3rem auto;
104 }
105
106 .version a {
107   font-size: 1.1rem;
108   color: white;
109   border-bottom: 4px solid #ff9999;
110   background-color: #cc0000;
111   padding: 1.3rem 2rem;
112   border-radius: 20px;
113 }
114
115 .version a:hover {
116   opacity: 0.8;
117   text-decoration: none;
118 }
119
120 .video-container {
```

```
121     position: relative;
122     width: 100%;
123     height: 0;
124     padding-bottom: 56.25%;
125 }
126 .video {
127     position: absolute;
128     top: 0;
129     left: 0;
130     width: 100%;
131     height: 100%;
132 }
133 
134 /*-------------------------------------------------
135 CONTRIBUTORS
136 -------------------------------------------------*/
137 
138 .rails-core {
139   width: 100%;
140   overflow: hidden;
141   text-align: center;
142 }
143 
144 .person {
145   display: inline-block;
146   padding: 0.5rem;
147   text-align: center;
148 }
149 
150 .person a, .person em {
151   font-family: Helvetica, sans-serif;
152   font-size: 1rem;
153 }
154 
155 .person em { color: #777; }
156 
157 .meta {
158   font-family: Helvetica, sans-serif;
159   font-size: 1rem;
160   margin-top: 2.5rem;
161   line-height: 1.25rem;
162 }
163 
164 figure.contributor {
165   background-size: 700px 420px;
166   background-image: url('images/rails-core.jpg');
```

```
  width: 140px;
  height: 0;
  padding-top: 140px;
  box-sizing: border-box;
  font-family: Helvetica, sans-serif;
  font-size: 1rem;
  line-height: 1.25rem;
}

.david { background-position: 0 0; }
.jeremy { background-position: 560px 0; }
.santiago { background-position: 420px 0; }
.aaron { background-position: 280px 0; }
.xavier { background-position: 140px 0; }
.rafael { background-position: 0 280px; }
.andrew { background-position: 560px 280px; }
.guillermo { background-position: 420px 280px; }
.carlos { background-position: 280px 280px; }
.yves { background-position: 140px 280px; }
.godfrey { background-position: 0 560px; }
.matthew { background-position: 560px 560px; }
.kasper { background-position: 420px 140px; }
.eileen { background-position: 280px 140px; }

figure p { margin: 0; }

/*-----------------------------------------------------
RESPONSIVE CSS
-----------------------------------------------------*/

/* iPhone */
@media only screen and (max-device-width: 480px) {
  body {
    font-size: 1.2rem;
  }
  .container { width: auto; margin: 0; }
  .mobile-center { text-align: center; }
  section { width: auto; }
  .version { margin: 0 auto; }
  .version a { padding: 1rem; }
  .nav { width: 100%; margin: 20px 0 40px; overflow: hidden; text-align: center; }
  .nav ul li, .nav ul li.first { margin: 15px; line-height: 2rem; }
  .nav ul li.back { margin-right: 10px; }
  section { padding: 10px;}
  h1 { font-size: 32px; margin-top: 10px; }
  h2 { font-size: 21px; font-weight: bold; margin-bottom: -18px; }
```

```
213   figure img { width: 380px; height: 380px; }
214   figure.right, figure.left { margin: 0; float: none; }
215   .show-mobile { display: block; }
216   .hide-mobile { display: none; }
217 }
218
219 /* iPad */
220 @media only screen and (min-device-width: 481px) and (max-device-width: 1024px) a\
221 nd (orientation:portrait) {
222 }
223
224 @media only screen and (min-device-width: 481px) and (max-device-width: 1024px) a\
225 nd (orientation:landscape) {
226 }
```

(the above code snippet online)

Using `telnet`, we issued the request

```
1 GET /style.css HTTP/1.1
2 Host: rubyonrails.org
```

(the above code snippet online)

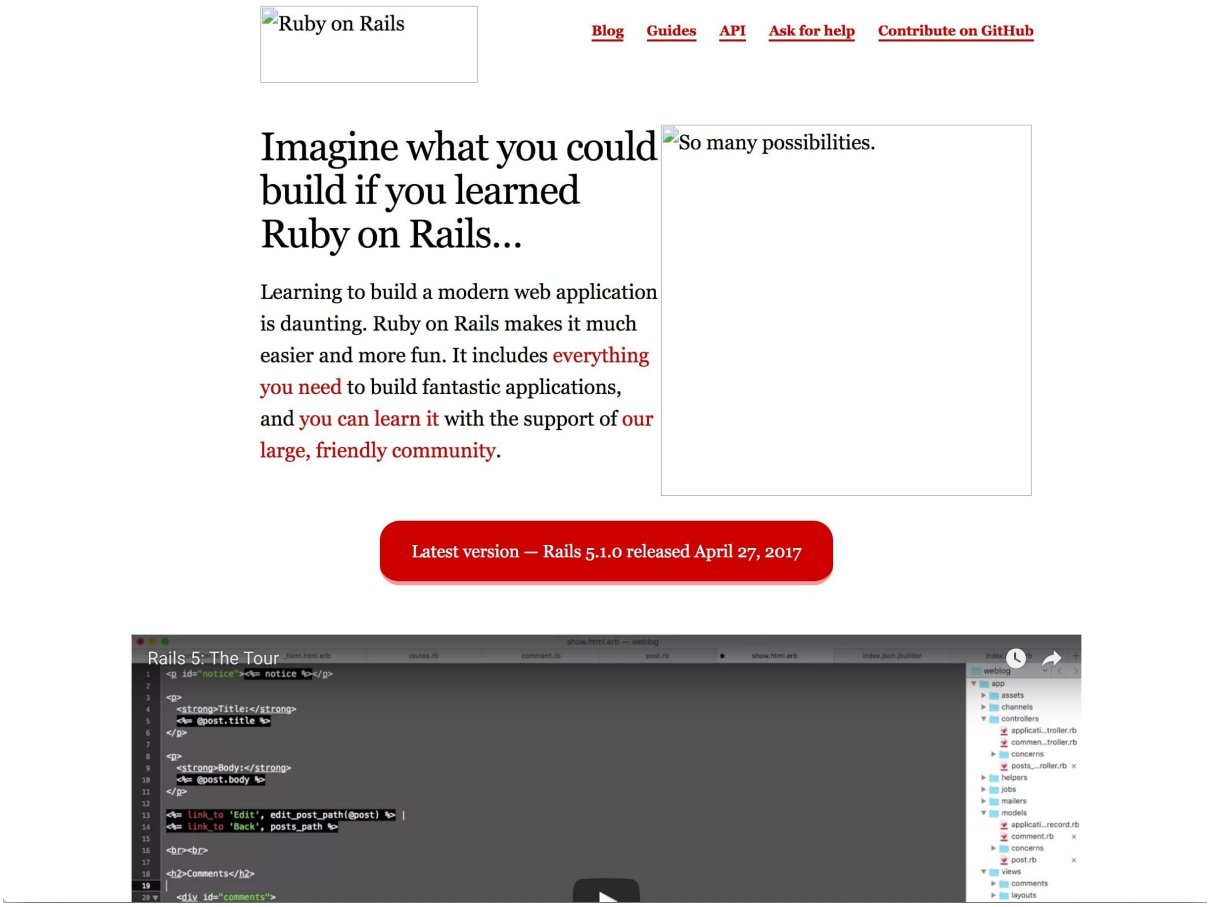
and we've got back the content of the `style.css` file. Let's save the response body into the file `style.css` (in the same folder that the `ror.html` file exists).

Then change the line

```
1 <link rel="stylesheet" href="/style.css" type="text/css" media="screen" charset="\
2 utf-8">
```

(the above code snippet online)

to have `href="style.css"` instead of `href="/style.css"`. This will make sure that the `ror.html` file binds to the local `style.css` file. Then reload the `ror.html` file in your browser. You will see something like this:



RoR Page with Style Included

As you can see, the page `ror.html` is much better displayed. The images are missing, but the style is now present to style the HTML document.

Then we can follow the same technique to download locally the images. And this is how the browser works and downloads all the necessary information in order to draw the page properly.

We will not download the content of the images using `telnet`, you can do that yourself as an exercise.

Issuing HTTP Requests with `curl`

Another tool that you can use to issue HTTP requests and get back responses is curl. Actually, `curl` is used to exchange data in various protocols, not only HTTP.

`curl` Example 1

Let's get the `rubyonrails.org` main page using `curl`.

```
$ curl -v rubyonrails.org
* Rebuilt URL to: http://rubyonrails.org/
*   Trying 192.30.252.153...
* Connected to rubyonrails.org (192.30.252.153) port 80 (#0)
> GET / HTTP/1.1
> Host: rubyonrails.org
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: GitHub.com
< Date: Mon, 08 May 2017 17:48:10 GMT
< Content-Type: text/html; charset=utf-8
< Content-Length: 5564
< Last-Modified: Fri, 28 Apr 2017 11:51:53 GMT
< Access-Control-Allow-Origin: *
< Expires: Mon, 08 May 2017 17:58:10 GMT
< Cache-Control: max-age=600
< Accept-Ranges: bytes
< X-GitHub-Request-Id: EEF7:4F53:1FDCE0A:2B8D773:5910AF5A
<
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <link rel="stylesheet" href="/style.css" type="text/css" media="screen" chars\
et="utf-8">
    <meta name="google-site-verification" content="Jknoaf3CvRzba3wXMxqgurDErIGZ9e\
1L1luuDMhQYEI" />
    <!-- Begin Jekyll SEO tag v2.2.0 -->
<title>Ruby on Rails | A web-application framework that includes everything neede\
d to create database-backed web applications according to the Model-View-Controll\
er (MVC) pattern.</title>
<meta property="og:title" content="Ruby on Rails" />
<meta property="og:locale" content="en_US" />
<meta name="description" content="A web-application framework that includes every\
thing needed to create database-backed web applications according to the Model-Vi\
ew-Controller (MVC) pattern." />
<meta property="og:description" content="A web-application framework that include\
s everything needed to create database-backed web applications according to the M\
odel-View-Controller (MVC) pattern." />
<link rel="canonical" href="http://rubyonrails.org/" />
<meta property="og:url" content="http://rubyonrails.org/" />
<meta property="og:site_name" content="Ruby on Rails" />
<meta name="twitter:card" content="summary" />
```

```
<meta name="twitter:site" content="@rails" />
<script type="application/ld+json">
{"@context": "http://schema.org",
"@type": "WebSite",
"name": "Ruby on Rails",
"headline": "Ruby on Rails",
"description": "A web-application framework that includes everything needed to cr\
eate database-backed web applications according to the Model-View-Controller (MVC\
) pattern.",
"url": "http://rubyonrails.org/"}</script>
<!-- End Jekyll SEO tag -->

  </head>

  <body>
    <div class="container">
          <div class="nav">
      <ul>
        <li class="first"><a href="http://weblog.rubyonrails.org/">Blog</a></li>
        <li><a href="http://guides.rubyonrails.org/">Guides</a></li>
        <li><a href="http://api.rubyonrails.org/">API</a></li>
        <li><a href="http://stackoverflow.com/questions/tagged/ruby-on-rails">Ask\
 for help</a></li>
        <li><a href="https://github.com/rails/rails">Contribute on GitHub</a></li>
      </ul>
    </div>

    <section>
      <p class="mobile-center">
        <img src="/images/rails-logo.jpg" width="220" height="78" alt="Ruby on Ra\
ils">
      </p>
    </section>

    <section>
      <figure class="right">
        <img src="/images/imagine.png" alt="So many possibilities.">
      </figure>

      <h1>Imagine what you could build if you learned Ruby on Rails&#8230;</h1>
      <p>Learning to build a modern web application is daunting. Ruby on Rails ma\
kes it much easier and more fun. It includes <a href="everything-you-need">everyt\
hing you need</a> to build fantastic applications, and <a href="http://guides.rub\
yonrails.org/getting_started.html">you can learn it</a> with the support of <a hr\
ef="community">our large, friendly community</a>.</p>
    </section>
```

```
93
94      <section class="version">
95        <p><a href="http://weblog.rubyonrails.org/2017/4/27/Rails-5-1-final/">Lates\
96  t version &mdash; Rails 5.1.0 <span class="hide-mobile">released April 27, 2017</\
97  span></a></p>
98        <p class="show-mobile"><small>Released April 27, 2017</small></p>
99      </section>
100
101     <section class="video-container">
102       <iframe src="https://www.youtube.com/embed/OaDhY_y8WTo" frameborder="0" all\
103 owfullscreen class="video"></iframe>
104     </section>
105
106     <section class="interior">
107       <figure class="left">
108         <img src="/images/rubyrails.png" alt="Rails does it all.">
109       </figure>
110
111       <p><strong>You've probably already used many of the applications that were \
112 built with Ruby on Rails:</strong> <a href="https://basecamp.com">Basecamp</a>, <\
113 a href="https://github.com">GitHub</a>, <a href="https://shopify.com">Shopify</a>\
114 , <a href="https://airbnb.com">Airbnb</a>, <a href="https://twitch.tv">Twitch</a>\
115 , <a href="https://soundcloud.com">SoundCloud</a>, <a href="https://hulu.com">Hul\
116 u</a>, <a href="https://zendesk.com">Zendesk</a>, <a href="https://square.com">Sq\
117 uare</a>, <a href="https://highrisehq.com">Highrise</a>. Those are just some of t\
118 he big names, but there are literally hundreds of thousands of applications built\
119  with the framework since its release in 2004.</p>
120
121       <p><strong>Ruby on Rails is open source software</strong>, so not only is i\
122 t free to use, you can also help make it better. <a href="http://contributors.rub\
123 yonrails.org">More than 4,500 people</a> already have contributed code to Rails. \
124 It's easier than you think to become one of them.</p>
125
126       <p><strong>Optimizing for programmer happiness with Convention over Configu\
127 ration</strong> is how we roll. Ruby on Rails has been popularizing both concepts\
128  along with a variety of other controversial points since the beginning. To learn\
129  more about why Rails is so different from many other web-application frameworks \
130 and paradigms, examine <a href="doctrine">The Rails Doctrine</a>.</p>
131     </section>
132
133     <section class="more">
134       <figure>
135         <img src="/images/more.png" alt="More resources">
136       </figure>
137
138       <p>Keep up to date with <a href="https://twitter.com/rails">Rails on Twitte\
```

```
r</a> and <a href="https://rails-weekly.ongoodbits.com">This Week in Rails</a></p\
>

      <p><small>Policies: <a href="conduct">Conduct</a>, <a href="http://opensour\
ce.org/licenses/MIT">License</a>, <a href="maintenance">Maintenance</a>, <a href=\
"security">Security</a>, <a href="trademarks">Trademarks</a></small></p>
    </section>

    </div> <!-- /.container -->
  </body>
</html>
```

(the above code snippet online)

The `-v` switch for the `curl` command is for `verbose` and allows you to see the HTTP request headers and the HTTP response headers.

Closing Note

That was an introduction to HTTP. HTTP has a lot more details for you to learn, but these are enough for you as a start. You will learn more about HTTP while you will be leveling up your knowledge on Web development in the next chapters and sections.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

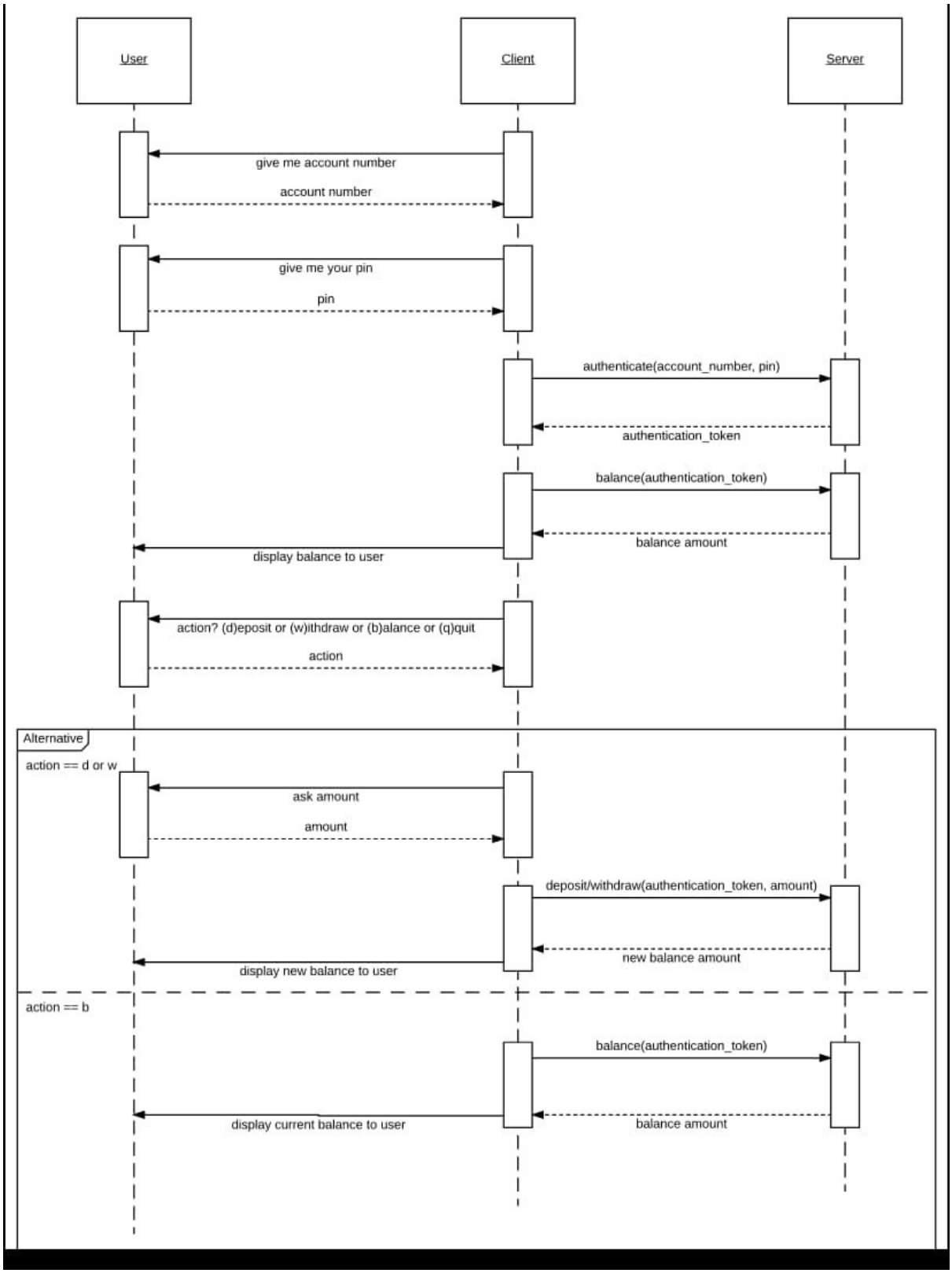
Task Details

You need to use HTTP protocol over TCP/IP sockets to develop a server that would function as a bank ATM machine. It will have to respond to the following questions:

1. Deposit
2. Withdraw
3. Balance

There is going to be a client that would allow the user to connect to the server and ask for their balance and deposit or withdraw money.

Here is a typical conversation between the client and the server.



Typical Conversation Between Client and Server

1. Initially, user is asked to give their account number and their pin.
2. Then client authenticates user using the `authenticate` service of the server.

3. Then client asks for the current balance of the account, and informs user about it.
4. Then client asks the user to tell it which action they want to carry out.
    1. Deposit
    2. Withdraw
    3. Balance
    4. Quit
5. If deposit or withdraw, client is asking the server to carry out the corresponding action.
6. If balance, client is asking the server to return back the current balance.
7. If quit, then client terminates.

Here is a short video displaying this interaction. In this video we show that we start the server and then we start the client which interacts with the user.

Task - TCP Sockets - ATM Client Server - Typical Interaction

With regards to the user interaction with the client, some things that are not obvious from the video, and that you need to take care of are:

1. If the user does not give correct account number/pin combination, client should terminate by informing user that the credentials were wrong.
2. User selection of an action (deposit, withdraw, balance or quit) should be done with the press of a key without the need to click on Enter. (Hint: The `$stdin.getch` method does that. But you will need to require `io/console` - Google for it).
3. If the server is not running the client should gracefully terminate. (Try to rescue the `Errno::ECONNREFUSED` exception)
4. The server keeps track of the accounts, their pins and their balances using a local *database*. Its database can be either a `yml` or a `JSON` file. Choose whatever you like. Again, the database with the accounts pre-exists.
5. When the server accepts a command to update the balance of an account (deposit or withdraw), then it saves the new balance in its database. So, if you stop the server and you load it again, it will have the new balances read from its database.
6. When a client authenticates, server sends back to the client an authentication token which uniquely identifies the account authenticated for the particular client interaction. Then subsequent calls to update the balance or get the balance use the authentication token returned. Server is able to identify the account from the authentication token and does not need the account number and pin again.

Some hints that will help you implement this project:

1. You will need to send HTTP formatted messages. So, instead of writing a string of any format, this string needs to be formatted according to the HTTP message formatting rules.
2. Try to make your code, client and server, as DRY (Do Not Repeat Yourself) as possible. And as clean as possible. This is very important.
3. Server should respond with HTTP status code `200` if the data sent by the client are correct. Otherwise, it should respond with `400`.

4. For the authentication case, when the credentials are incorrect it should respond with `401` status code. Same goes for other requests that have wrong authentication token.

Upload your code to your Github account so that your mentor can evaluate your work

Note: This is the same exercise like the one for chapter *Introduction to TCP IP Sockets*, but with HTTP protocol used instead.

2 - Sinatra

Summary

Sinatra Summary

Sinatra is a very famous Ruby Web framework that relies on Rack. In this chapter we are going to explain to you how you can write your first Web application! Not only that, we are going to tell you how to deploy that into a production Web applications hosting server. You will really enjoy this chapter being the first that allows you to process the data that you collect using HTML forms. And it is a great foundation chapter for the next big section on Ruby on Rails.

Learning Goals

1. Learn what is a Web framework.
2. Learn about Rack.
3. Learn what is a Rack application.
4. Learn about the `rackup` tool.
5. Learn about Sinatra gem.
6. Learn about routes.
7. Learn how you can submit data using a form and process this data at the server side.
8. Learn how to enable reloading of changed files while doing development.
9. Learn how you can set the environment your application is running in.

10. Learn how to generate content dynamically with view template and template languages like ERB.
11. Learn about layouts.
12. Learn where to put your public static files.
13. Learn about writing and reading cookies.
14. Learn about helpers.
15. Learn about the session.
16. Learn how to deploy your Rack application to Heroku.

Introduction

Sinatra is a Ruby Web framework. A Ruby Web framework is composed of a set of Ruby libraries. It has solved many problems for you. So, you need less things to worry about the technology to build a Web application.

Sinatra is based on Rack, which is a Ruby Web Interface. When a Web framework like Sinatra implements the Rack interface, then this framework can be deployed into Web servers that support Ruby via Rack.

In other words.

1. We have the Web Server. E.g. Apache.
2. We have a module in Web Server that makes it capable of hosting a Rack compatible application.
3. We have Sinatra being Rack compatible and with hooks for you to build your Web application.

All this makes it what we actually call a deployed Web application.

Rack

Let's talk about Rack. Rack will allow to quickly build a Web application, i.e. an application that accepts HTTP requests and responds with HTTP responses.

Rack Hello World Project

Create the project `hello_rack` in your RubyMine environment. Add the files to support `rvm` (try using Ruby 2.x and gemset `hello_rack`). Also, create the `Gemfile` with the following content:

```
1  # File: Gemfile
2  #
3  source 'https://rubygems.org'
4
5  gem 'rack'
```

(the above code snippet online)

Make sure that you have `bundler` installed (`gem install bundle --no-ri --no-rdoc`) and then run `bundle` to install the `Gemfile` specified gems.

The Gemfile specifies the gem `rack` to be installed. This brings in all the functionality necessary to start with a Web application based on `rack`.

```
hello_rack > $ bundle
Fetching gem metadata from https://rubygems.org/.............
Fetching version metadata from https://rubygems.org/.
Resolving dependencies...
Installing rack 2.0.2
Using bundler 1.14.6
Bundle complete! 1 Gemfile dependency, 2 gems now installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
hello_rack > $
```

(the above code snippet online)

Now create the file `main.rb`. This is going to be our starting script for our Web application:

```
# File: main.rb
#
require 'rack'

html_content =<<-HTML
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello Rack Page</title>

  </head>
  <body>
    <h1>Hello Rack!</h1>
  </body>
</html>
HTML

app = Proc.new do |environment|
  ['200', {'Content-Type' => 'text/html'}, [html_content]]
end

Rack::Handler::WEBrick.run app
```

(the above code snippet online)

Before we explain what is going on here, let's just see this in action. Start your Web application like this:

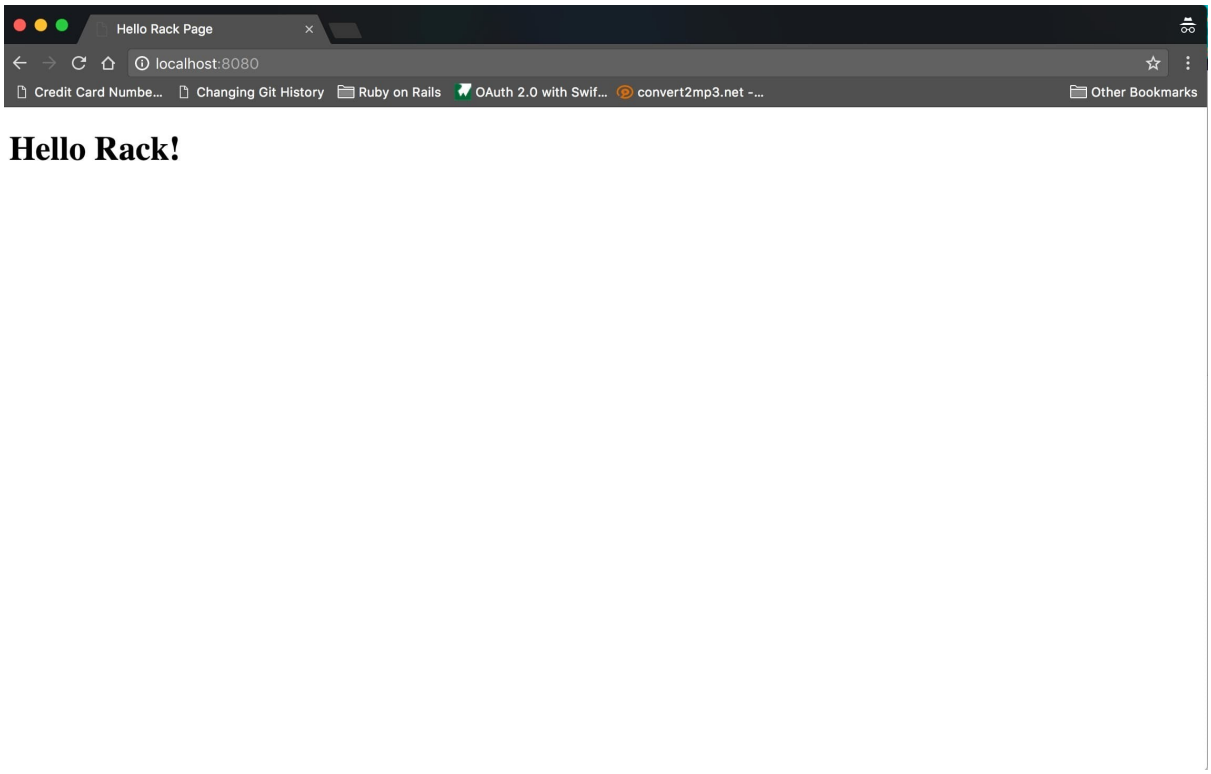
```
hello_rack > $ bundle exec ruby main.rb
[2017-05-10 18:55:02] INFO  WEBrick 1.3.1
[2017-05-10 18:55:02] INFO  ruby 2.2.4 (2015-12-16) [x86_64-darwin14]
[2017-05-10 18:55:02] INFO  WEBrick::HTTPServer#start: pid=22774 port=8080
```

(the above code snippet online)

You will see something like the above. This is the explanation of what you see:

1. The above command starts your Web application.
2. It internally uses a Web server tool which is called `WEBrick`. `WEBrick` is a primitive Web server that you can host your Rack application. It is not used in production deployments. But it is very useful for development purposes. At least at this level. Later on we will learn how we can use other Web servers.
3. The most important line of output is the last one, which says that an HTTP Server has started listening on port `8080`. The process of the HTTP Server is the process with process id `22774` (this number will be different in your case).
4. As you can see, the terminal does not return back to you. This terminal is now occupied by the Web server who is waiting for HTTP requests to arrive.

Now, let's open our Web browser and visit the page `http://localhost:8080`. This is what we will see:



Hello Rack!

Fantastic! This is our first Web application without having to deal with sockets and HTTP message formatting!

What Is a Rack Application

A Rack application is a Ruby object that

1. responds to the `#call` instance method
2. Its `#call` method takes exactly one argument. Usually conventionally called `environment` or `env`. It is a `Hash` instance that is populated with environment specific keys and HTTP request/response related keys. See later on about it.
3. Its `#call` method should return an array of 3 elements.
    1. The first element should be a string corresponding to a valid HTTP Status Response Code.
    2. The second element is a Hash of response headers. This is optional.
    3. The third element, which is optional too, is the response body.

In our `main.rb` Ruby application, we have a Rack application stored in the variable `app`.

The line 19 is the point at which the `app` is instantiated. It is instantiated with the help of the `Proc.new` call which creates instances that respond to `#call`. The `#call` implementation is going to be what the block given to `Proc.new` defines.

> *Information:* This is another example of the `Proc.new` usage. Start `irb` and issue the following commands:

```
2.2.4 :001 > a = Proc.new do |foo, bar|
2.2.4 :002 >     "#{foo.upcase}/#{bar.upcase}"
2.2.4 :003?>   end
 => #<Proc:0x007fc7cd219980@(irb):1>
2.2.4 :004 > a.call('John', 'Woo')
 => "JOHN/WOO"
2.2.4 :005 >
```

> As you can see from above, we can invoke the `#call` on the instance returned by `Proc.new`, giving run-time arguments as per the typical arguments defined in the block given to `Proc.new`.

Hence, lines 19 till 21 define a Rack application compatible instance.

Then on line 23, we ask a `WEBrick` HTTPServer to load this application and serve corresponding requests.

By default, the HTTP server starts listening for connections on port `8080`.

> *Hint*: This Ruby command will start the HTTP server on port `8090`: `ruby Rack::Handler::WEBrick.run app, Port: 8089`

The `environment` Hash

This is an example of the `environment` value that is given to the `#call` invocation:

```
{"GATEWAY_INTERFACE"=>"CGI/1.1",
 "PATH_INFO"=>"/",
 "QUERY_STRING"=>"",
 "REMOTE_ADDR"=>"::1",
 "REMOTE_HOST"=>"localhost",
 "REQUEST_METHOD"=>"GET",
 "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/2.2.4/2015-12-16)",
 "HTTP_HOST"=>"localhost:8080",
 "HTTP_CONNECTION"=>"keep-alive",
 "HTTP_CACHE_CONTROL"=>"max-age=0",
 "HTTP_UPGRADE_INSECURE_REQUESTS"=>"1",
 "HTTP_USER_AGENT"=>"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/\
537.36 (KHTML, like Gecko) Chrome/58.0.3029.96 Safari/537.36",
 "HTTP_ACCEPT"=>"text/html,application/xhtml+xml,application/xml;q=0.9,image/webp\
,*/*;q=0.8",
 "HTTP_ACCEPT_ENCODING"=>"gzip, deflate, sdch, br",
 "HTTP_ACCEPT_LANGUAGE"=>"en-GB,en-US;q=0.8,en;q=0.6,el;q=0.4",
 "rack.version"=>[1, 3],
 "rack.input"=>#<StringIO:0x007fedcca71f68>,
 "rack.errors"=>#<IO:<STDERR>>,
 "rack.multithread"=>true,
 "rack.multiprocess"=>false,
 "rack.run_once"=>false,
 "rack.url_scheme"=>"http",
 "rack.hijack?"=>true,
 "rack.hijack"=>#<Proc:0x007fedcca71e28@/...rack/handler/webrick.rb:74 (lambda)>,
 "rack.hijack_io"=>nil,
 "HTTP_VERSION"=>"HTTP/1.1",
 "REQUEST_PATH"=>"/"}
```

(the above code snippet online)

As you can read from the above list, Rack makes sure we have everything that we need to handle the HTTP request accordingly.

These are some of the most important keys here:

1. `REQUEST_METHOD`. This is the HTTP verb. In our example it is `GET`.
2. `QUERY_STRING`. This has the query params portion of the URL. For example, if the call was to `http://localhost:8080?foo=bar`, then this would have been `foo=bar`.
3. `HTTP_HOST`. This is important because it tells us which Web site the user is requesting.

4. `REQUEST_PATH`. This is the part of the URL that corresponds to the path. For example, if the call was to `http://localhost:8089/path/to/resource?foo=bar`, then this would have been `/path/to/resource`.

Using `rackup` tool

Instead of using a Ruby script like `main.rb`, that uses a `Rack::Handler` to start a Rack application, you can instead follow another technique:

Create the file `config.ru` with the following content (Note that this is a Ruby file with extension `.ru` instead of `.rb`):

```
# File: config.ru
#
html_content =<<-HTML
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello Rack Page</title>

  </head>
  <body>
    <h1>Hello Rack!</h1>
  </body>
</html>
HTML

app = Proc.new do |environment|
  ['200', {'Content-Type' => 'text/html'}, [html_content]]
end

run app
```

(the above code snippet online)

This is exactly the same file like the `main.rb` but it does not require `rack` and calls `run app` instead of `Rack::Handler::WEBrick.run app`.

Now in order to start your Web application, you need to invoke the `rackup` tool as follows:

```
hello_rack > $ bundle exec rackup
[2017-05-10 19:48:00] INFO  WEBrick 1.3.1
[2017-05-10 19:48:00] INFO  ruby 2.2.4 (2015-12-16) [x86_64-darwin14]
[2017-05-10 19:48:00] INFO  WEBrick::HTTPServer#start: pid=23647 port=9292
```

(the above code snippet online)

and you will see the HTTP Server starting at the default port `9292` instead.

If you want to start your server using another port you can invoke it like this:

```
hello_rack > $ bundle exec rackup --port 8080
[2017-05-10 19:49:22] INFO  WEBrick 1.3.1
[2017-05-10 19:49:22] INFO  ruby 2.2.4 (2015-12-16) [x86_64-darwin14]
[2017-05-10 19:49:22] INFO  WEBrick::HTTPServer#start: pid=23676 port=8080
```

(the above code snippet online)

... giving the `--port 8080` as an example to start on port `8080`.

Sinatra and Rack

As we said earlier, Sinatra is a Web framework that builds on top of Rack. Let's now switch to Sinatra.

Hello Sinatra

Let's create a new RubyMine project with name `hello_sinatra`. Make sure you integrate with `rvm` (ruby version 2.X, gemset `hello_sinatra`) and that you install `bundler` (`gem install bundler --no-ri --no-rdoc`).

Then create the file `Gemfile` with the following contents:

```
# File: Gemfile
#
source 'https://rubygems.org'

gem 'sinatra'
```

(the above code snippet online)

Having specified in the `Gemfile` that you want to use the `sinatra` gem, run `bundle`:

```
hello_sinatra > $ bundle
Fetching gem metadata from https://rubygems.org/..........
Fetching version metadata from https://rubygems.org/.
Resolving dependencies...
Installing mustermann 1.0.0
Installing rack 2.0.2
Installing tilt 2.0.7
Using bundler 1.14.6
Installing rack-protection 2.0.0
Installing sinatra 2.0.0
Bundle complete! 1 Gemfile dependency, 6 gems now installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
hello_sinatra > $
```

(the above code snippet online)

Now that Sinatra gem is installed, let's write our `main.rb` program:

```
# File: main.rb
#
require 'sinatra'

get '/' do
  'Hello Sinatra!'
end
```

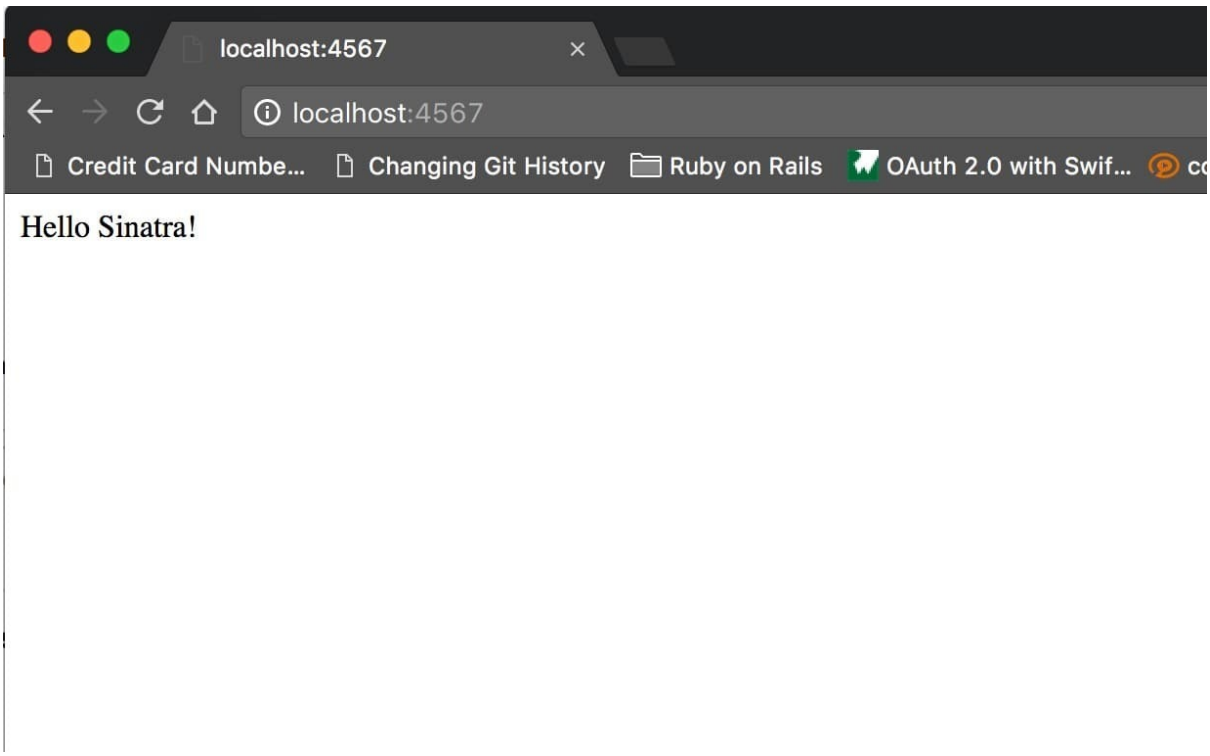
(the above code snippet online)

With the `main.rb` file ready, lets start our Web application:

```
hello_sinatra > $ bundle exec ruby main.rb
[2017-05-10 21:17:27] INFO  WEBrick 1.3.1
[2017-05-10 21:17:27] INFO  ruby 2.2.4 (2015-12-16) [x86_64-darwin14]
== Sinatra (v2.0.0) has taken the stage on 4567 for development with backup from \
WEBrick
[2017-05-10 21:17:27] INFO  WEBrick::HTTPServer#start: pid=24478 port=4567
```

(the above code snippet online)

Perfect! You can now see, that an HTTP Server is running on port `4567`. Let's open our Web browser and visit the page `http://localhost:4567`:



Hello Sinatra!

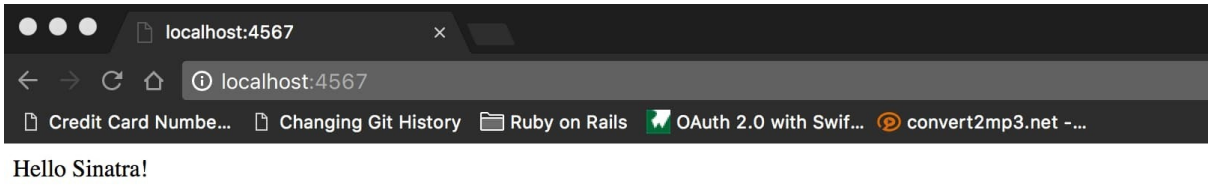
Pretty cool! Isn't it?

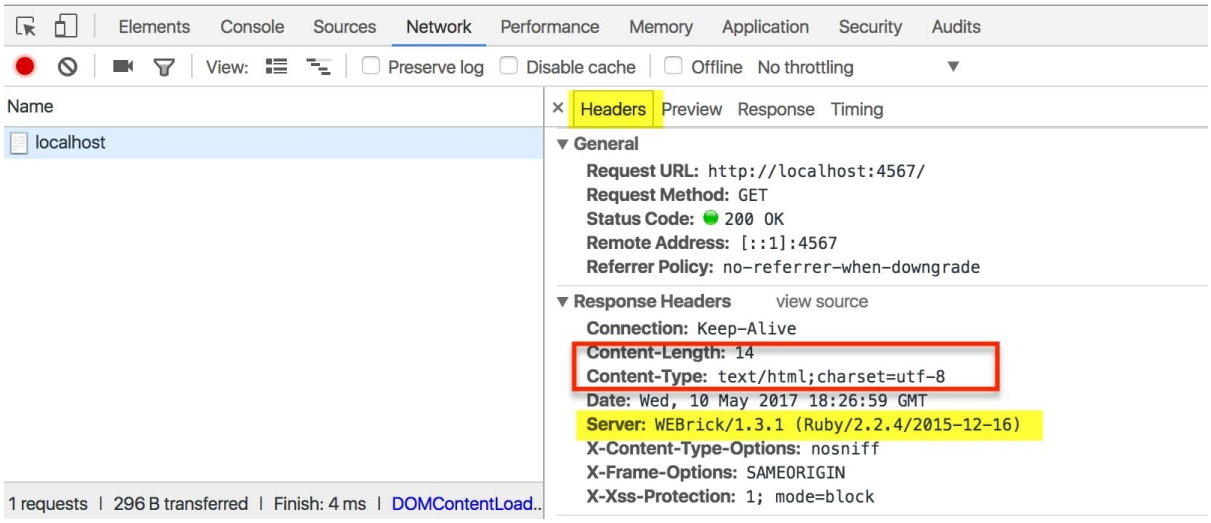
What's Inside Our `main.rb`

The Web application that we have developed above is quite simple and straightforward.

1. On line `5` we tell it hat it has to respond to HTTP verb `GET` for the root `/` path of our application.
2. In the block given to `get '/'` call we tell what we have to return. Currently, returning a simple text string.

If you inspect the HTTP request using the Google Developer Tools Network tab, you will see something like this:





Inspect Network - Headers

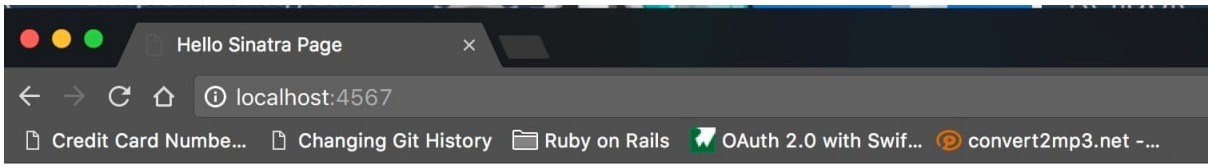
You can see that the `Content-Length` is 14, equal to the `Hello Sinatra!` string. The response code was `200`. Also, the `Content-Type` specified for the response is `text/html` although it is not a proper HTML document.

Let's try some real HTML response like we did with the Rack application. Update the `main.rb` to be like this:

```ruby
# File: main.rb
#
require 'sinatra'

html_content =<<-HTML
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello Sinatra Page</title>

  </head>
  <body>
    <h1>Hello Sinatra!</h1>
  </body>
</html>
HTML

get '/' do
  html_content
end
```

(the above code snippet online)

If you save this, restart your HTTP Server (press <kbd>Ctrl + C</kbd> to stop it and then issue `bundle exec ruby main.rb` again), and then reload the page on your browser, you will see this:



Hello Sinatra With HTML Content

Routes

The combination of an HTTP verb and a URL path constitutes a route for your Sinatra application. When a request is coming in, Sinatra matches the verb and the path of the request against one route defined in your code. It scans the routes from top to bottom and the first that matches takes responsibility to handle the request.

Let's write a new Sinatra application, called `crm`. Create the RubyMine project and add the `Gemfile` to bring `sinatra` in.

Then write the `main.rb` as follows:

```
1  # File: main.rb
2  #
3  require 'sinatra'
4
5  def welcome
6    File.read('views/welcome.html')
7  end
8
9  def contact_us
10   File.read('views/contact_us.html')
11 end
```

```
12
13 get '/' do
14   welcome
15 end
16
17 get '/contact_us' do
18   contact_us
19 end
```

(the above code snippet online)

This application defines two routes:

1. `/` for `GET` verb.
2. `/contact_us` for 'GET' verb.

Both route handlers read the HTML content to return from a corresponding HTML file. These files are inside the sub-folder with name `views`.

So, let's create the sub-folder `views`:

```
1 crm > $ mkdir views
2 crm > $
```

(the above code snippet online)

And then create the files `contact_us.html` and `welcome.html` inside this folder as follows:

```
1  <!DOCTYPE html>
2  <!-- File: views/contact_us.html -->
3  <!--                            -->
4  <html>
5    <head>
6      <meta charset="utf-8">
7      <title>Contact Us Form</title>
8    </head>
9    <body>
10     <h1>Fill In Your Details</h1>
11     <form>
12       <label for="first_name">First Name:</label>
13       <input type="text" name="first_name" id="first_name"/>
14       <button type="submit">Save</button>
15     </form>
16   </body>
17 </html>
```

(the above code snippet online)

and

```
<!DOCTYPE html>
<!-- File: views/welcome.html -->
<!--                          -->
<html>
  <head>
    <meta charset="utf-8">
    <title>Welcome</title>

  </head>
  <body>
    <h1>ACM CRM</h1>
    <a href="/contact_us">Contact us</a>
  </body>
</html>
```

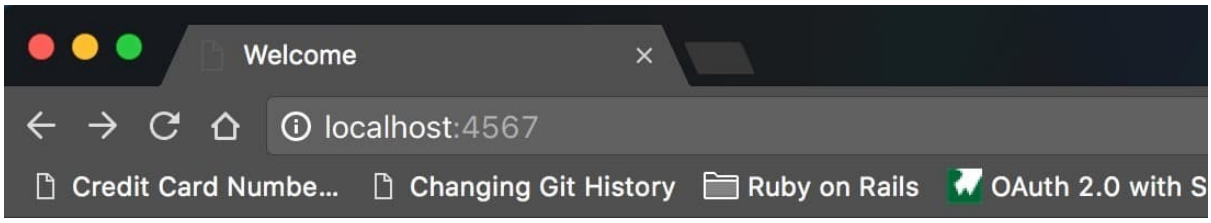
(the above code snippet online)

Now everything is ready to start your Web application:

```
crm > $ bundle exec ruby main.rb
[2017-05-10 22:33:34] INFO  WEBrick 1.3.1
[2017-05-10 22:33:34] INFO  ruby 2.2.4 (2015-12-16) [x86_64-darwin14]
== Sinatra (v2.0.0) has taken the stage on 4567 for development with backup from \
WEBrick
[2017-05-10 22:33:34] INFO  WEBrick::HTTPServer#start: pid=25762 port=4567
```

(the above code snippet online)

Now visit the page `http://localhost:4567`. You will see this:

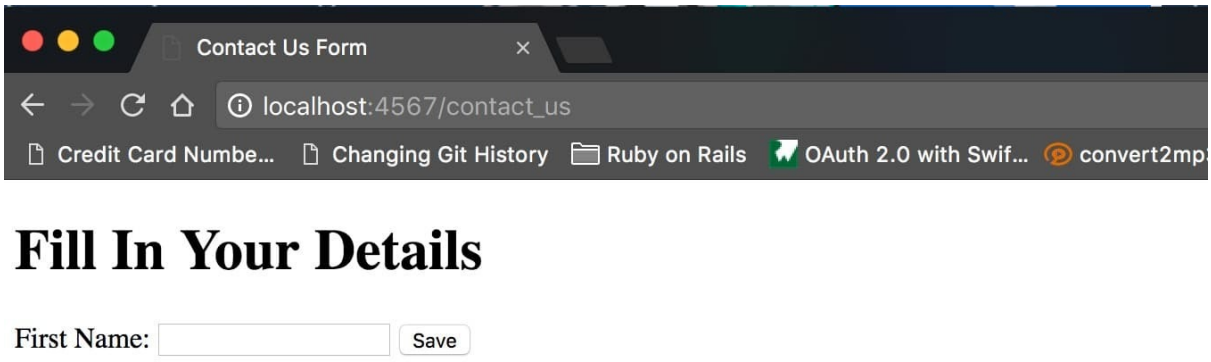


ACM CRM

Contact us

ACM CRM Welcome Page

And if you click on the link `Contact Us`, you will see this:



ACM CRM - Contact Us

Cool! We have just created two endpoints in our Sinatra application. Wasn't that easy?

1. The first endpoint, `GET '/'`, is displaying a welcome page. With a link to visit the other endpoint.
2. The second endpoint, `GET '/contact_us'`, is displaying a form for the user to submit their details.

Try a `POST` Endpoint

Let's enhance our application to include the endpoint `POST '/contact_us'`. This will be handled by a route handler that would take the data posted by the user using the form.

This is the new version of our `main.rb` file:

```ruby
# File: main.rb
#
require 'sinatra'

def welcome
  File.read('views/welcome.html')
end

def contact_us
  File.read('views/contact_us.html')
end

def process_contact_us_form
  first_name = params['first_name']
  last_name = params['last_name']
  message = params['message']
  File.open('messages.txt', 'a') do |file|
    file.write("New message from #{first_name} #{last_name}. Message is: #{messag\
e}\n")
  end
  redirect '/'
end

get '/' do
  welcome
end

get '/contact_us' do
  contact_us
end

post '/contact_us' do
  process_contact_us_form
end
```

(the above code snippet online)

As you can see above, we have added the `post '/contact_us'` route handler. This is implemented with the help of the method `process_contact_us_form`. Before we see the implementation of this method, let's see the new version of the `views/contact_us.html` that has all the fields and details for the form to be properly submitted to the server.

```
<!DOCTYPE html>
<!-- File: views/contact_us.html -->
<!--                             -->
<html>
  <head>
    <meta charset="utf-8">
    <title>Contact Us Form</title>
  </head>
  <body>
    <h1>Fill In Your Details And Your Message To Us</h1>
    <small>Or <a href="/">Go Back</a></small>
    <br/>
    <br/>

    <form action="/contact_us" method="post">
      <label for="first_name">First Name:</label>
      <input type="text" name="first_name" id="first_name"/>
      <br/>

      <label for="last_name">Last Name:</label>
      <input type="text" name="last_name" id="last_name"/>
      <br/>

      <label for="message">Message:</label>
      <textarea name="message" id="message"></textarea>
      <br/>

      <button type="submit">Save</button>
    </form>
  </body>
</html>
```

(the above code snippet online)

Besides the fact that we have added two more fields, the `last_name` and the `message`, the most important detail here is `action` and `method` attributes of the `form` tag. The `action` specifies at which endpoint the data will be submitted. And the `method` specifies the verb that will be used for the submission. In our case, the `action` has the value `/contact_us` and the `method` has the value `post`. And this is how we bind the submission of this form to the end point `POST /contact_us` and the route handler `post '/contact_us'` in our code.

With regards to the implementation of the route handler, the method `process_contact_us_form` is as follows:

```
1  ...
2
3  def process_contact_us_form
4    first_name = params['first_name']
5    last_name = params['last_name']
6    message = params['message']
7    File.open('messages.txt', 'a') do |file|
8      file.write("New message from #{first_name} #{last_name}. Message is: #{messag\
9  e}")
10   end
11   redirect '/'
12 end
13
14 ...
15 ...
```

(the above code snippet online)

There is a method called `params` that returns a `Hash` and contains the keys and values of the parameters submitted by the form. For example, `params['first_name']` has the `first_name` submitted. This is offered by Sinatra for free.

On lines 17 till 19 we just save the details of the message submitted into a text file.

Another important piece of information here is the `redirect '/'`. This sends back to the browser a `302` response and tells browser where it should take user to. In other words, the browser will issue a `GET` request to the endpoint `/`. Our implementation displays the `welcome.html` page for the `GET '/'` requests.

Restart your server and revisit the `http://localhost:4567` and try to submit a message. Look what is happening. The following video demonstrates our current implementation.

Sinatra App - Contact Us Form In Action

Reloading Of Changed Files While On Development

You might have noticed, until now, that whenever you did a change in your app code, you had to restart your server in order for it to reload your pages and take into account your changes. This is really annoying while doing development.

There is a way you can tell Sinatra to reload the files on every request so that it takes into account any changes that might have been implemented in between.

First, we need to use the gem `sinatra-contrib`. Amend your Gemfile:

```
1  # File: Gemfile
2  #
3  source 'https://rubygems.org'
4
5  gem 'sinatra'
6  gem 'sinatra-contrib'
```

(the above code snippet online)

and do `bundle`. You will see the `sinatra-contrib` gem to be installed (alongside its dependencies).

Then, what you only have to do is to require the file `sinatra/reloader`. So, let's change the first lines of our `crm/main.rb` file:

```
1  # File: main.rb
2  #
3  require 'sinatra'
4  require 'sinatra/reloader' if development?
5
6  def welcome
7  ...
```

(the above code snippet online)

You can see the line 4 that requires the file `sinatra/reloader`. Note also the `if development?` statement. Sinatra distinguishes the environment it is running on and you can have access to this information using the `development?` or `production?` methods. The default environment is `development`. But when we push our application to a production server we will make sure that the environment that Sinatra application is bound to is the `production` environment. In that case, the `if development?` will return `false` and hence the `require 'sinatra/reloader'` will not be executed. This is the required behaviour for production environments. We do not want the files to be reloaded on every request.

Now that you have done that, you can restart your server to take into account the Sinatra reloader. Then on every change in your files, you do not have to restart your server in order to continue your development with the changes in. The changes will be picked up automatically on every request to the server.

Setting the Environment

As we said earlier, you can set the environment to different values, with the default environment being `development` and the method `development?` returning `true`.

The Sinatra application takes its environment from the `RACK_ENV` environment variable. Let's change our `crm/main.rb` as follows:

```
# File: main.rb
#
require 'sinatra'

puts "RACK_ENV = #{ENV['RACK_ENV']}"
puts "development? = #{development?}"
puts "production? = #{production?}"

require 'sinatra/reloader' if development?
...
```

(the above code snippet online)

We have changed the first lines of our `crm/main.rb` file. We print the value of the environment variable `RACK_ENV`. If you restart your server like below, you will see that `RACK_ENV` is not set and the `development?` returns `true`, whereas the `production?` returns `false`.

```
crm > $ bundle exec ruby main.rb
RACK_ENV =
development? = true
production? = false
[2017-05-11 08:54:34] INFO  WEBrick 1.3.1
[2017-05-11 08:54:34] INFO  ruby 2.2.4 (2015-12-16) [x86_64-darwin14]
== Sinatra (v2.0.0) has taken the stage on 4567 for development with backup from \
WEBrick
[2017-05-11 08:54:34] INFO  WEBrick::HTTPServer#start: pid=28674 port=4567
```

(the above code snippet online)

Now, stop your server and restart it with the command `RACK_ENV=production bundle exec ruby main.rb`. You will see this:

```
crm > $ RACK_ENV=production bundle exec ruby main.rb
RACK_ENV = production
development? = false
production? = true
[2017-05-11 08:55:36] INFO  WEBrick 1.3.1
[2017-05-11 08:55:36] INFO  ruby 2.2.4 (2015-12-16) [x86_64-darwin14]
== Sinatra (v2.0.0) has taken the stage on 4567 for production with backup from W\
EBrick
[2017-05-11 08:55:36] INFO  WEBrick::HTTPServer#start: pid=28695 port=4567
```

(the above code snippet online)

You can see that when we have set the `RACK_ENV` to the specific value of `production`, then the corresponding environment check method, `production?`, returns `true`. Whereas `development?` returns `false`.

This is how we set the environment that our Sinatra application will live in. And with branching statements (like `if development?`) we might change its behaviour according to the environment.

Stop your server and start it again without specifying the `RACK_ENV`, so that it starts in `development` environment.

Views and Templates

In our crm example application, the views files that reside inside the `views` folder, they are simple `html` files. Although this is enough for static HTML content, it is not enough for our Web application that might need to generate content dynamically. Content that depends on the value of various data variables. For example, let's suppose that we want to redirect to a *Thank You* page after the user submits their message, and on that *Thank You* page we want to address the user with their details, first name and last name. In that case, we need a page like this:

```
<!DOCTYPE html>
<!-- File: views/contact_us_thank_you.html -->
<!-- ------------------------------------ -->
<html>
  <head>
    <meta charset="utf-8">
    <title>Thank You for Your message</title>
  </head>
  <body>
    <h1>Thank you for your message Panos Matsinopoulos!</h1>
    <a href="/">Home</a>
  </body>
</html>
```

(the above code snippet online)

But with the first_name/last_name part of it being dynamically generated.

```
 1. <!DOCTYPE html>
 2. <!-- File: views/contact_us_thank_you.html -->
 3. <!-- ------------------------------------ -->
 4. <html>
 5.   <head>
 6.     <meta charset="utf-8">
 7.     <title>Thank You for Your message</title>
 8.   </head>
 9.   <body>
10.     <h1>Thank you for your message Panos Matsinopoulos </h1>
11.     <a href="/">Home</a>
12.   </body>
13. </html>
```

This needs to be dyncamically built and not hard coded to a specific value.

Part Of Page Needs To Be Dynamically Generated

In order to achieve this, we need to use templates and a template language. Templates are like HTML files but they have dynamic parts inside. The language we use to write the dynamic parts is the template language. There are many template languages out there and Sinatra supports plenty of them. We are going to use ERB, Embedded Ruby which comes with Ruby as part of the standard Ruby library. Sinatra supports ERB with the call to method `erb` from within the route handlers.

But before we call this method, the `erb`, let's write the above HTML page as an ERB template.

Create the file `views/contact_us_thank_you.erb` (watch out for the `.erb` suffix) with the following content:

```
1  <!DOCTYPE html>
2  <!-- File: views/contact_us_thank_you.erb -->
3  <!-- ----------------------------------- -->
4  <html>
5    <head>
6      <meta charset="utf-8">
7      <title>Thank You for Your message</title>
8    </head>
9    <body>
10     <h1>Thank you for your message <%= "#{@first_name} #{@last_name}" %>!</h1>
11     <a href="/">Home</a>
12   </body>
13 </html>
```

(the above code snippet online)

The `.erb` files are HTML files that can contain embedded Ruby code. The Ruby code is embedded within blocks that start with the `<%` and end with the `%>`. On line 10 above we see one Ruby embedded code block. It references the instance variables `@first_name` and `@last_name`. The `=` that follows the opening `<%` tells ERB rendering engine to embed the output of the right-hand side into the HTML content. Hence, the ERB rendering engine will take the value of the Ruby expression `"#{@first_name} #{@last_name}"` and will embedded it into the HTML content.

Now, the point is that we have to change our `crm/main.rb` file to take advantage of this new view. Let's do that:

```
1  # File: main.rb
2  #
3  require 'sinatra'
4
5  puts "RACK_ENV = #{ENV['RACK_ENV']}"
6  puts "development? = #{development?}"
7  puts "production? = #{production?}"
8
9  require 'sinatra/reloader' if development?
10
```

```
11  def welcome
12    File.read('views/welcome.html')
13  end
14
15  def contact_us
16    File.read('views/contact_us.html')
17  end
18
19  def process_contact_us_form
20    @first_name = params['first_name']
21    @last_name = params['last_name']
22    message = params['message']
23    File.open('messages.txt', 'a') do |file|
24      file.write("New message from #{@first_name} #{@last_name}. Message is: #{mess\
25  age}\n")
26    end
27    erb :contact_us_thank_you
28  end
29
30  get '/' do
31    welcome
32  end
33
34  get '/contact_us' do
35    contact_us
36  end
37
38  post '/contact_us' do
39    process_contact_us_form
40  end
```

(the above code snippet online)

The only changes are for the method `process_contact_us_form`:

```
1   ...
2   def process_contact_us_form
3     @first_name = params['first_name']
4     @last_name = params['last_name']
5     message = params['message']
6     File.open('messages.txt', 'a') do |file|
7       file.write("New message from #{@first_name} #{@last_name}. Message is: #{mess\
8   age}\n")
9     end
10    erb :contact_us_thank_you
11  end
12  ...
```

(the above code snippet online)

On line 20 and 21 we assign the first name and last name to the corresponding *instance* variables, `@first_name` and `@last_name`. Assigning to instance variables and not just local variables, makes these variable being available at the view template level. That's why we can later use code like `"#{@first_name} #{@last_name}"` in the view template.

Then on line 24 we use the instance variables instead of the local variables. And finally ...

The other big change is the invocation of the method `erb` which basically tells Sinatra to take parse as an ERB template the file with name `contact_us_thank_you.erb` inside the `views` folder. Again, when we invoke `erb` with a symbol as argument, e.g. `erb :foo`, we instruct Sinatra to render the file `views/foo.erb` using the ERB template engine.

The following video shows how the *Thank You* page is displayed with the name of the person that submitted the message dynamically rendered.

Sinatra App - Thank You Page With Dynamic Content

ERB File Does not Have To Contain ERB Code

Now that we have learned about the ERB files, let me just tell you that you don't have to have ERB code in order to use ERB templates.

Our crm application now has two more files, the `views/contact_us.html` and the `views/welcome.html` that we can turn to ERB templates and then simplify the `crm/main.rb` code that requests their rendering.

Hence, rename the files `views/contact_us.html` and `views/welcome.html` to `views/contact_us.erb` and `views/welcome.erb` respectively. Then, simplify the implementation of the `get '/'` and `get '/contact_us'` endpoints by just calling `erb` with the correct symbols:

```
1  # File: main.rb
2  #
3  require 'sinatra'
4
5  puts "RACK_ENV = #{ENV['RACK_ENV']}"
6  puts "development? = #{development?}"
7  puts "production? = #{production?}"
8
9  require 'sinatra/reloader' if development?
10
11 def process_contact_us_form
12   @first_name = params['first_name']
13   @last_name = params['last_name']
14   message = params['message']
15   File.open('messages.txt', 'a') do |file|
16     file.write("New message from #{@first_name} #{@last_name}. Message is: #{mess\
17 age}\n")
18   end
```

```
19    erb :contact_us_thank_you
20  end
21
22  get '/' do
23    erb :welcome
24  end
25
26  get '/contact_us' do
27    erb :contact_us
28  end
29
30  post '/contact_us' do
31    process_contact_us_form
32  end
```

(the above code snippet online)

Look at lines 22 and 26. We are calling `erb` with the view template symbol that we want to render. And we got rid off of the methods `welcome` and `contact_us`, cause we didn't need them any more.

Try your application again on your browser and you will see that it works without any problem, like before.

Layouts

Currently, we have 3 view templates. And they both have some code that it is repeated in all templates. For example the head part. How can we have common pieces of all the view templates reside in one file and being reused? This is the idea of the *layout*.

A Sinatra application uses a default layout template and *yields* the view templates within it. The default layout is the file `views/layout.erb`.

So, let's create the following file, the `views/layout.erb`:

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1">
7      <!-- The above 3 meta tags *must* come first in the head; any other head cont\
8  ent must come *after* these tags -->
9      <meta name="description" content="">
10     <meta name="author" content="">
11     <link rel="icon" href="../../favicon.ico">
12
13     <title><%= @page %></title>
```

```
14
15      <!-- Latest compiled and minified CSS -->
16      <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/\
17  css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RY\
18  dg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
19
20      <!-- Custom CSS -->
21      <link rel="stylesheet" href="assets/stylesheets/main.css">
22
23      <!-- Latest jQuery -->
24      <script src="https://code.jquery.com/jquery-2.2.4.min.js" integrity="sha256-B\
25  bhdlvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44=" crossorigin="anonymous"></script>
26
27      <!-- Latest compiled and minified JavaScript -->
28      <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min\
29  .js" integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIc\
30  PD7Txa" crossorigin="anonymous"></script>
31
32      <!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media qu\
33  eries -->
34      <!--[if lt IE 9]>
35      <script src="https://oss.maxcdn.com/html5shiv/3.7.3/html5shiv.min.js"></scrip\
36  t>
37      <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
38      <![endif]-->
39    </head>
40
41    <body>
42
43    <nav class="navbar navbar-inverse navbar-fixed-top">
44      <div class="container">
45        <div class="navbar-header">
46          <button type="button" class="navbar-toggle collapsed" data-toggle="collap\
47  se" data-target="#navbar" aria-expanded="false" aria-controls="navbar">
48            <span class="sr-only">Toggle navigation</span>
49            <span class="icon-bar"></span>
50            <span class="icon-bar"></span>
51            <span class="icon-bar"></span>
52          </button>
53          <a class="navbar-brand" href="/">CRM</a>
54        </div>
55        <div id="navbar" class="collapse navbar-collapse">
56          <ul class="nav navbar-nav">
57            <li class="active"><a href="/">Home</a></li>
58            <li><a href="/contact_us">Contact</a></li>
59          </ul>
```

```
60        </div><!--/.nav-collapse -->
61      </div>
62    </nav>
63
64    <div class="container">
65
66      <%= yield %>
67
68    </div><!-- /.container -->
69
70    </body>
71  </html>
```

(the above code snippet online)

The above is the layout and it has a `<%= yield %>` command on line 57. This is where the view template files are going to be embedded. Having said that, we need to change the view template files now, in order to include only the necessary HTML snippet that would live inside the layout template, and not having a whole HTML page.

Here is the new version of the `views/welcome.erb`:

```
1  <!-- File: views/welcome.html -->
2  <!--                          -->
3  <h1>ACM CRM</h1>
4  <a href="/contact_us">Contact us</a>
```

(the above code snippet online)

This is a very simple file. Its content is going to appear where the `<%= yield %>` statement inside the `views/layout` appears.

Similar content we now see for the other view templates. The `views/contact_us.erb`:

```
1  <!-- File: views/contact_us.html -->
2  <!--                              -->
3  <h1>Fill In Your Details And Your Message To Us</h1>
4  <small>Or <a href="/">Go Back</a></small>
5  <br/>
6  <br/>
7
8  <form action="/contact_us" method="post">
9    <div class="form-group">
10     <label for="first_name">First Name:</label>
11     <input type="text" name="first_name" id="first_name" class="form-control" pla\
12 ceholder="Give your first name"/>
13   </div>
14
```

```
15    <div class="form-group">
16      <label for="last_name">Last Name:</label>
17      <input type="text" name="last_name" id="last_name" class="form-control" place\
18  holder="Give your last name"/>
19    </div>
20
21    <div class="form-group">
22      <label for="message">Message:</label>
23      <textarea name="message" id="message" class="form-control" placeholder="Place\
24   your message here"></textarea>
25    </div>
26
27    <button type="submit" class="btn btn-default">Save</button>
28  </form>
```

(the above code snippet online)

And the `views/contact_us_thank_you.erb`:

```
1  <!-- File: views/contact_us_thank_you.html.erb -->
2  <!-- ----------------------------------------- -->
3  <h1>Thank you for your message <%= @first_name %> <%= @last_name %>!</h1>
4  <a href="/">Home</a>
```

(the above code snippet online)

Before you browse your application again, make sure that you have the file `public/assets/stylesheets/main.cs` in your project:

```
1  /* File: crm/public/assets/stylesheets/main.css */
2
3  body {
4      margin-top: 70px;
5  }
```

(the above code snippet online)

This file is referenced by the `views/layout.erb` and it has a very simple rule so that the content of the page is not obscured by the navigation bar.

Note that the reference of static files like local CSS files and local JavaScript files from the view templates or layouts is done using the relative path of the file that needs to be placed inside the `public` folder of the application, excluding the `public` folder itself. Hence the file `public/assets/stylesheets/main.css` is referenced as `assets/stylesheets/main.css` on line 18 in `views/layout.erb`:

```
...
   <!-- Custom CSS -->
   <link rel="stylesheet" href="assets/stylesheets/main.css">
...
```

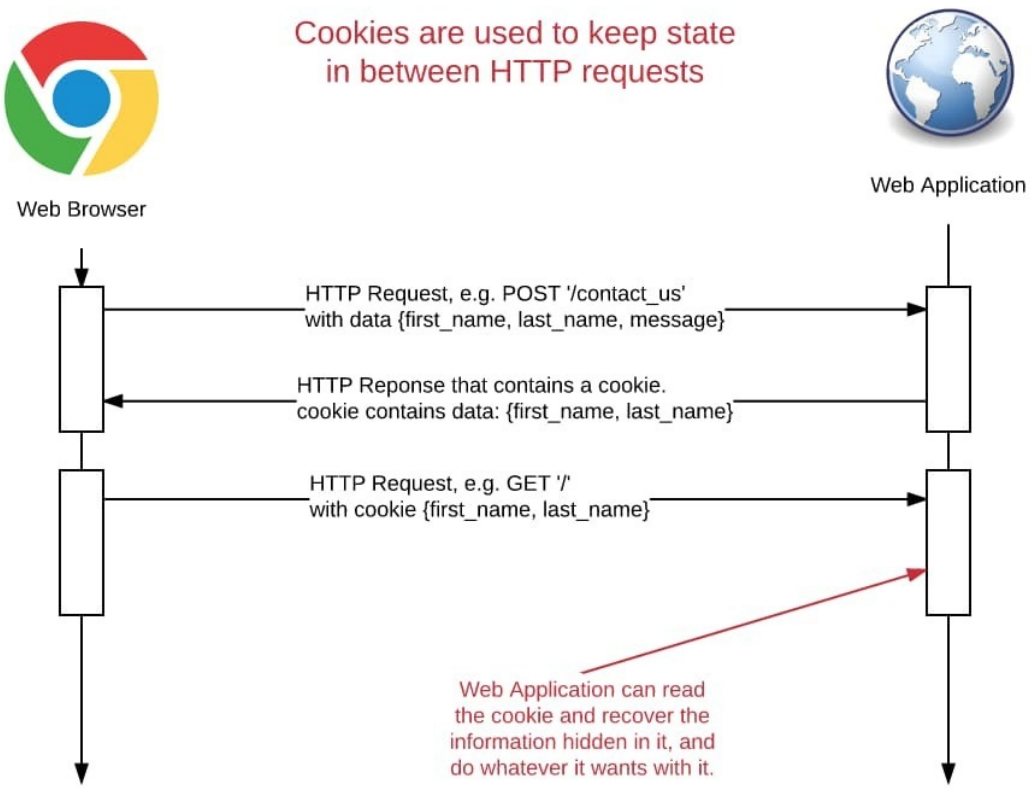
(the above code snippet online)

Now everything is set up. Let's navigate to our new styled application. Here is a video that demonstrates the navigation:

 <a href="https://player.vimeo.com/v

Cookies

We have learned that HTTP is a stateless protocol. But Web applications need to keep some kind of state in between requests. For example, a Web application might want to know which user is signed in. A mechanism that Web applications usually use in order to keep state in between HTTP requests is the mechanism of a *cookie*. The *cookie* is saved at the browser side file system and it is being sent back to the server on subsequent requests. Hence, server can read the information stored in the cookie. And this works as a state storage in between HTTP requests.



Cookies Are Store on Browser Side And They Are Sent Over on Every Request

As we are trying to depict in the above picture, when the server decides to create a cookie with some information inside, then the browser is going to be sending this cookie back within subsequent requests. Hence server keeps track of things in between HTTP requests and this works like a state-keeping mechanism.

Storing a Cookie

Let's use a cookie with our crm application. We will use a cookie to store the first name and last name of the user sending a message.

First, let's save the first name and last name in a new cookie. See how the `crm/main.rb` method `process_contact_us_form` is amended to save the information in a cookie:

```
1  def process_contact_us_form
2    @first_name = params['first_name']
3    @last_name = params['last_name']
4   message = params['message']
5   File.open('messages.txt', 'a') do |file|
6     file.write("New message from #{@first_name} #{@last_name}. Message is: #{messa\
7  ge}\n")
8   end
9   cookies[:user] = {first_name: @first_name, last_name: @last_name}.to_json
10  erb :contact_us_thank_you
11 end
```

(the above code snippet online)

It is the line 18 that was added. The `cookies` works like a `Hash`. Before leaving the route handler, we set the cookie with key `:user` and we attach to it the information `{first_name: @first_name, last_name: @last_name}`. Note that we convert the `Hash` value to a string with a call to `#to_json` method. This is because we need to be able to save strings that can then be deserialized when we will later on read the cookie value back.

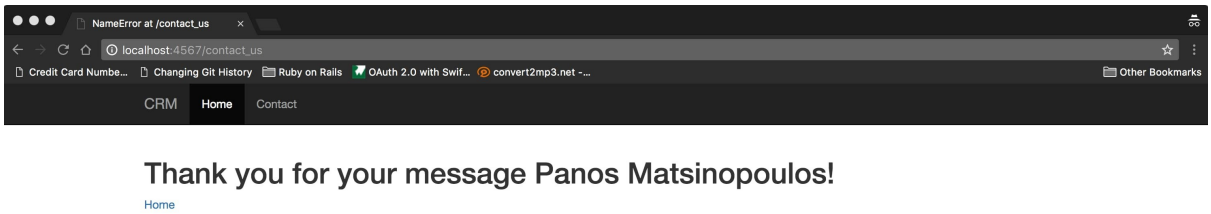
Note that you have to `require 'sinatra/cookies'` at the top of `crm/main.rb` file in order to make sure that `cookies` are available:

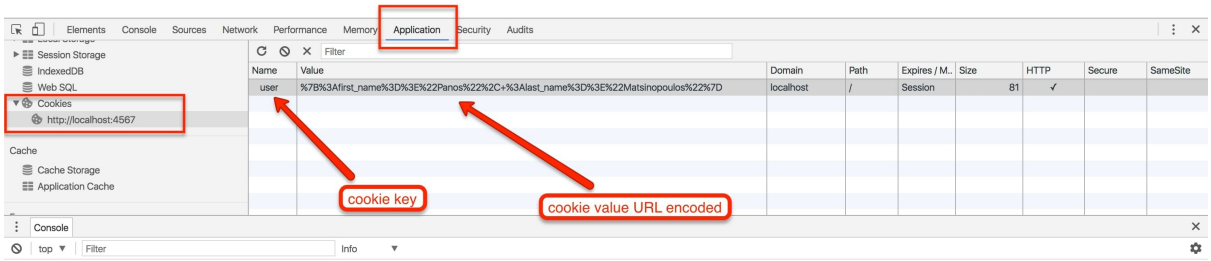
```
1  # File: main.rb
2  #
3  require 'sinatra'
4
5  puts "RACK_ENV = #{ENV['RACK_ENV']}"
6  puts "development? = #{development?}"
7  puts "production? = #{production?}"
8
9  require 'sinatra/reloader' if development?
10 require 'sinatra/cookies'
11 ...
```

(the above code snippet online)

Now, every time the user posts data using the contact us form, a cookie with key `user` is set in the browser and saved on client side.

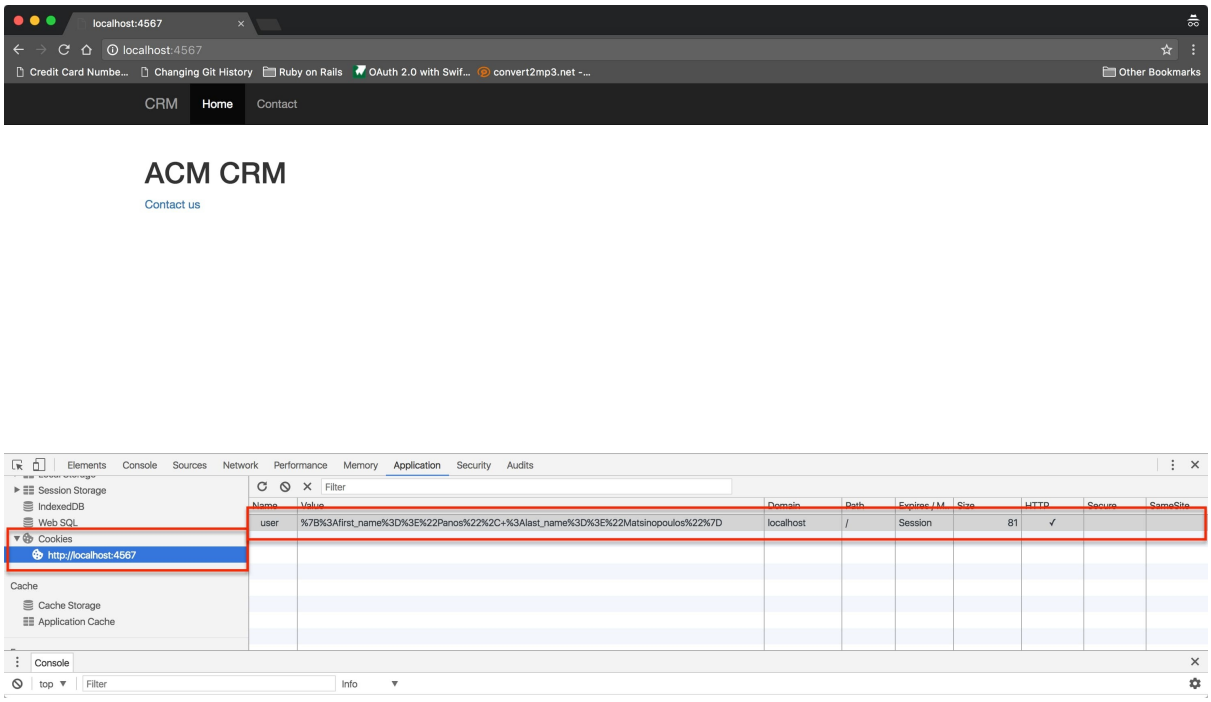
Let's see that with the help of Google Developer Tools.





Cookie Inspection with Developer Tools

And as we said earlier, the cookie persists when user goes to another page. You can still see that if you go to the home page:



Cookie Reserved Even If We Visit Other Page

Retrieving a Cookie

But writing a cookie is the one side of the story. Reading back closes the loop and allows us to use cookies as a state storage. We can use the `cookies` `Hash` to read the cookies and act accordingly.

See how we amend the `views/welcome.erb` to display a greeting message if there is a user stored in the cookie:

```
1  <!-- File: views/welcome.erb -->
2  <!--                          -->
3  <h1>ACM CRM</h1>
4  <br/>
5  <a href="/contact_us">Contact us</a>
6
7  <% require 'json' %>
8  <% if cookies[:user] %>
9      <% details = JSON.parse(cookies[:user]) %>
10     <h2>Hi! <%= "#{details['first_name']} #{details['last_name']}" %></h2>
11 <% end %>
```

(the above code snippet online) On line 7 we require the `json` library that will allow us to call `JSON.parse`. The `JSON.parse` is called on `cookies[:user]`. We know that this value stores a JSON string and with the `JSON.parse` we convert it to a `Hash`. Then we use the `Hash` values to display a greeting to the user.

The following video shows how cookie is updated every time we use the contact us form.

Sinatra App - Contact Us Form In Action

Helpers

Before we move on to another subject, let's just make this piece of code a little bit more clean with the use of *helpers*. Helpers are methods that can be used inside route handlers and templates and they are define using `helpers` method in the main part of the application.

First, let's clean the `views/welcome.erb` code:

```
1  <!-- File: views/welcome.erb -->
2  <!--                         -->
3  <h1>ACM CRM</h1>
4  <br/>
5  <a href="/contact_us">Contact us</a>
6
7  <% if user? %>
8      <h2>Hi! <%= "#{user.first_name} #{user.last_name}" %></h2>
9  <% end %>
```

(the above code snippet online)

You can now see lines 7 to 9 which are much cleaner if compared to the previous version. Here we are now calling the methods `user?` and `user`. The first returns `true` or `false` depending on whether there is a user in cookie or not. The second returns a `User` instance that responds to `#first_name` and `#last_name`.

The `user?` and `user` methods are defined as `helpers` inside the main file:

```
1  # File: main.rb
2  #
3  require 'sinatra'
4
5  puts "RACK_ENV = #{ENV['RACK_ENV']}"
6  puts "development? = #{development?}"
7  puts "production? = #{production?}"
8
9  require 'sinatra/reloader' if development?
10 require 'sinatra/cookies'
11 require 'json'
12 require_relative './user'
13
14 helpers do
15   # Returns true if there is a user stored in the cookie
16   def user?
17     !cookies[:user].nil?
18   end
19
20   # Returns an instance of the User class. It's values are populated from the coo\
21 kie
```

```
22    def user
23      details = JSON.parse(cookies[:user])
24      User.new(details['first_name'], details['last_name'])
25    end
26  end
27
28  def process_contact_us_form
29    @first_name = params['first_name']
30    @last_name = params['last_name']
31    message = params['message']
32    File.open('messages.txt', 'a') do |file|
33      file.write("New message from #{@first_name} #{@last_name}. Message is: #{mess\
34  age}\n")
35    end
36    cookies[:user] = {first_name: @first_name, last_name: @last_name}.to_json
37    erb :contact_us_thank_you
38  end
39
40  get '/' do
41    erb :welcome
42  end
43
44  get '/contact_us' do
45    erb :contact_us
46  end
47
48  post '/contact_us' do
49    process_contact_us_form
50  end
```

(the above code snippet online)

See lines 11 till 25. This is where we define the two helper methods. Their implementation is quite simple and easy to understand.

And the class `User` lives inside the file `user.rb`:

```
1  # File: crm/user.rb
2  #
3  class User
4    def initialize(first_name, last_name)
5      @first_name = first_name
6      @last_name = last_name
7    end
8
9    attr_accessor :first_name, :last_name
10 end
```

(the above code snippet online)

If you play with the pages of your application, you will see that it works as before, but the code is much cleaner now.

Sessions

Another tool that can help you keep track of state in between HTTP requests is the *session*. When you start your browser, and visit a specific page, then you start a new *session*. How does your code can take advantage of it?

With Rack and Sinatra, you have access to the `session Hash`. You can use it to store data that would last as long as the user browser session lasts. The `session Hash` in Sinatra is usually implemented using some kind of cookie. That's why is called cookie-based session technique. This means that the state of the session is saved client-side, at the browser-side. However, you can use tools and techniques to implement a session saved at server-side, maybe inside a database.

Here, let's convert the previous implementation of keeping the state with cookies, to a session based state. We need to change the `crm/main.rb` file to stop using `cookies[:user]`, but use the `session Hash` instead.

Here is this new version that is using `session Hash`:

```
# File: main.rb
#
require 'sinatra'

puts "RACK_ENV = #{ENV['RACK_ENV']}"
puts "development? = #{development?}"
puts "production? = #{production?}"

require 'sinatra/reloader' if development?
require 'json'
require_relative './user'

helpers do
  # Returns true if there is a user stored in the session
  def user?
    !session[:user].nil?
  end

  # Returns an instance of the User class. It's values are populated from the ses\
sion
  def user
    details = JSON.parse(session[:user])
    User.new(details['first_name'], details['last_name'])
  end
end
```

```
26
27 enable :sessions
28
29 def process_contact_us_form
30   @first_name = params['first_name']
31   @last_name = params['last_name']
32   message = params['message']
33   File.open('messages.txt', 'a') do |file|
34     file.write("New message from #{@first_name} #{@last_name}. Message is: #{mess\
35 age}\n")
36   end
37   session[:user] = {first_name: @first_name, last_name: @last_name}.to_json
38   erb :contact_us_thank_you
39 end
40
41 get '/' do
42   erb :welcome
43 end
44
45 get '/contact_us' do
46   erb :contact_us
47 end
48
49 post '/contact_us' do
50   process_contact_us_form
51 end
```

(the above code snippet online)

The differences to the cookies version are the following.

1. We don't require `sinatra/cookies` any more.
2. We call `enable :sessions` which enables the session management.
3. Then, wherever we had `cookies`, we now call `sessions`.

Restart your server in order to enable sessions and play with the pages of your application. You will see how the session keeps track of the first name and last name of the person that submits the contact us form.

> *Important:* Usually, we don't want data to be stored in clear text format inside cookies (or session that relies on cookies). This is for security reasons. When we go to production, we make sure that data stored in cookies and sessions are encrypted in order to avoid a hacker changing the data inside them. We will not see this aspect of the cookies security today. More on this, in the Ruby on Rails section.

Sinatra is a Rack Based Framework

Your Sinatra application is also a Rack application. This means that we can create a `config.ru` file and be able to start the server using `bundle exec rackup` command. Let's do that:

```
# File: crm/config.ru
#
require_relative './main'

run Sinatra::Application
```

(the above code snippet online)

The `crm/config.ru` file is very simple, as you can see above. It requires the `main.rb` file and then it just calls `run` with the standard `Sinatra::Application` class as an argument.

Stop your server, if it is running, and issue the following command:

```
crm > $ bundle exec rackup
RACK_ENV = development
development? = true
production? = false
[2017-05-11 22:45:11] INFO  WEBrick 1.3.1
[2017-05-11 22:45:11] INFO  ruby 2.2.4 (2015-12-16) [x86_64-darwin14]
[2017-05-11 22:45:11] INFO  WEBrick::HTTPServer#start: pid=34858 port=9292
```

(the above code snippet online)

Your Web application starts as before. The only difference is that the `rackup` default port is `9292` and not `4567`.

Deploying to Heroku

Now we have our first primitive Web application based on Sinatra. Heroku allows you to deploy your Rack-based applications with no cost, if they don't consume too many resources. Let's do that then. We will deploy our primitive crm application to Heroku.

Create a Heroku Account

If you have not done that in the past, please create a Heroku account by following the instructions for signing up for free.

Download and install Heroku toolkit

After that, make sure that you install Heroku toolkit.

Make Sure You Login to Heroku From CLI

Using the `heroku` executable, login to your Heroku account. Issue the command `heroku login` at your shell prompt and follow the instructions.

Integrate with Git

Run the following command from your CRM project folder to initialize it with `git`:

```
crm > $ git init
Initialized empty Git repository in /..../crm/.git/
crm > $
```

(the above code snippet online)

Then add all the files and commit your current work.

```
crm > $ git add .
crm > $ git commit -m "Initial Commit"
create ....
....
crm > $
```

(the above code snippet online)

Create Heroku Application

Then create a new Heroku application under your account:

> *Important:* replace the `panosm-crm` with whatever name you want your application to have.

```
crm > $ heroku create panosm-crm
Creating □ panosm-crm... done
https://panosm-crm.herokuapp.com/ | https://git.heroku.com/panosm-crm.git
crm > $
```

(the above code snippet online)

The above command creates the application on your Heroku account (just log in to your dashboard and you will see your new application). Also, it registers a new `git` remote repository. You can confirm that with the following command (note that you will see `fetch` and `push` URLs specific to your project):

```
crm > $ git remote -v
heroku  https://git.heroku.com/panosm-crm.git (fetch)
heroku  https://git.heroku.com/panosm-crm.git (push)
crm > $
```

(the above code snippet online)

Push Your Code to Heroku

The deployment of your latest `master` branch code to your production Heroku machine is done with the following command:

```
crm > $ git push heroku master
Counting objects: 19, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), 3.77 KiB | 0 bytes/s, done.
Total 19 (delta 0), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Ruby app detected
remote: -----> Compiling Ruby/Rack
remote: -----> Using Ruby version: ruby-2.3.4
remote: -----> Installing dependencies using bundler 1.13.7
remote:        Running: bundle install --without development:test --path vendor/b\
undle --binstubs vendor/bundle/bin -j4 --deployment
remote:        Warning: the running version of Bundler (1.13.7) is older than the\
 version that created the lockfile (1.14.6). We suggest you upgrade to the latest\
 version of Bundler by running `gem install bundler`.
remote:        Fetching gem metadata from https://rubygems.org/.........
remote:        Fetching version metadata from https://rubygems.org/.
remote:        Installing multi_json 1.12.1
remote:        Installing mustermann 1.0.0
remote:        Installing backports 3.8.0
remote:        Installing rack 2.0.2
remote:        Installing tilt 2.0.7
remote:        Using bundler 1.13.7
remote:        Installing rack-protection 2.0.0
remote:        Installing sinatra 2.0.0
remote:        Installing sinatra-contrib 2.0.0
remote:        Bundle complete! 2 Gemfile dependencies, 9 gems now installed.
remote:        Gems in the groups development and test were not installed.
remote:        Bundled gems are installed into ./vendor/bundle.
remote:        Bundle completed (2.14s)
remote:        Cleaning up the bundler cache.
```

```
remote: -----> Detecting rake tasks
remote:
remote: ###### WARNING:
remote:        You have not declared a Ruby version in your Gemfile.
remote:        To set your Ruby version add this line to your Gemfile:
remote:        ruby '2.3.4'
remote:        # See https://devcenter.heroku.com/articles/ruby-versions for more\
 information.
remote:
remote: ###### WARNING:
remote:        No Procfile detected, using the default web server.
remote:        We recommend explicitly declaring how to boot your server process \
via a Procfile.
remote:        https://devcenter.heroku.com/articles/ruby-default-web-server
remote:
remote: -----> Discovering process types
remote:        Procfile declares types     -> (none)
remote:        Default types for buildpack -> console, rake, web
remote:
remote: -----> Compressing...
remote:        Done: 17.9M
remote: -----> Launching...
remote:        Released v4
remote:        https://panosm-crm.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/panosm-crm.git
 * [new branch]      master -> master
crm > $
```

(the above code snippet online)

The above is the output of the `git push heroku master`. As you can see from the last messages, you can visit the URL `https://panosm-crm.herokuapp.com/` in order to see your application running.

Or you can just do:

```
crm > $ heroku open --app panosm-crm
crm > $
```

(the above code snippet online)

and you will see your default browser travelling to the root page of your Web application.

Bingo!!!! You should see your Web application in action. And you can share the Web application URL to the rest of the World! It's publicly accessible by anyone. This is your first Web application deployed in the wild world and you can be very proud about it.

> Every time you do a change to your local machine code, you need to commit and push to `heroku` remote in order for your changes to be deployed to production.

Closing Note

Sinatra is used in many production Web applications. And, within the learning process experience, it is a very good start to prepare the ground for the next section, which is totally devoted to the best Ruby Web Framework, Ruby on Rails.

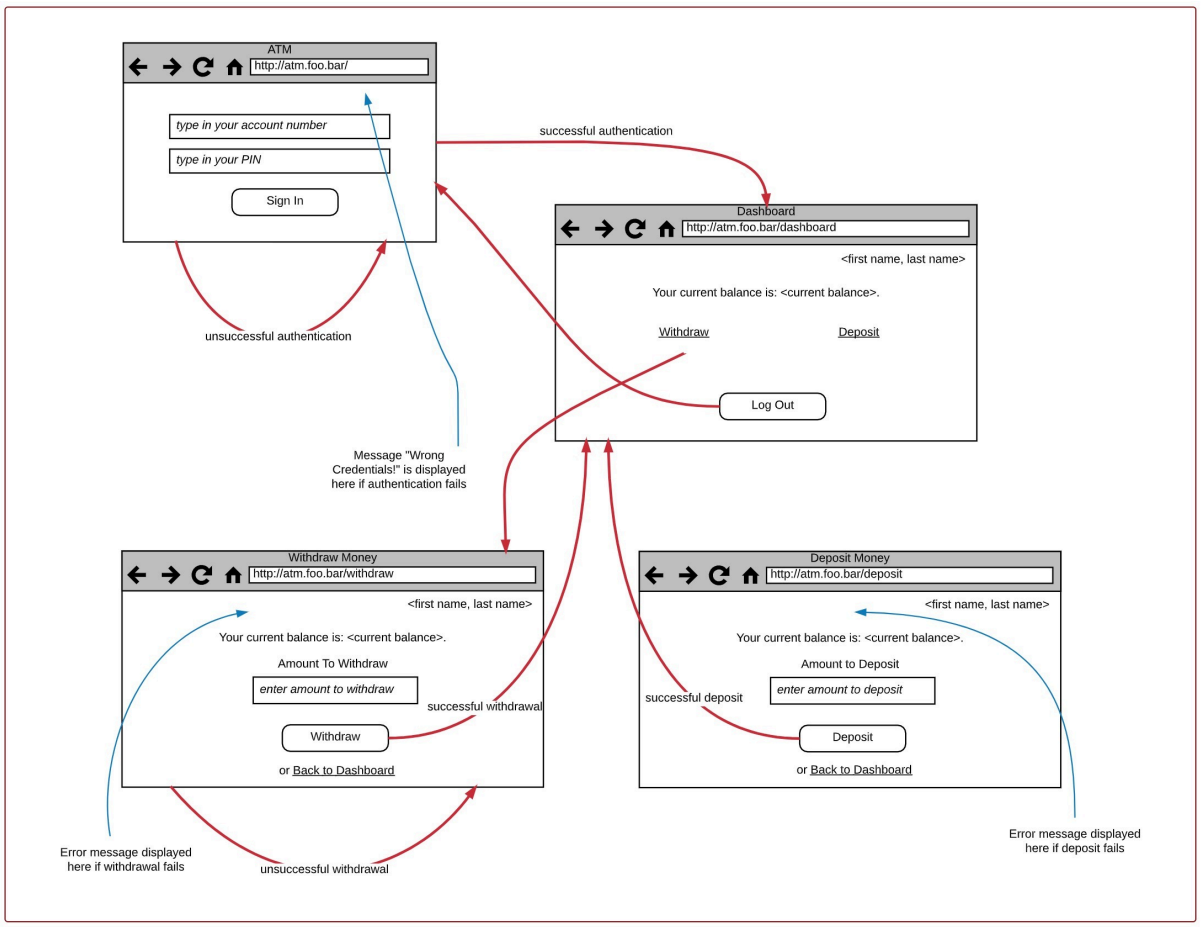
Tasks and Quizzes

> **Before you continue**, **you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Task Details

You are required to implement a Web application that resembles an ATM, i.e. a machine to withdraw and deposit money.

Here is a drawing with the flow of the application:



Task For Sinatra Application

1. When the user visits the `/` path of the application, he is presented with a sign in page. They have to give their account number and PIN and then click on *Sign In* button.
2. On successful authentication, the user is presented with the dashboard page, on path `/dashboard`.
3. Dashboard presents the name of the user.
4. Dashboard presents the current balance of the user.
5. There are three options there. 1) to withdraw money. 2) to deposit money and 3) to log out.
6. When the user clicks on withdraw link, they are taken to the page `/withdraw`.
7. When the user clicks on deposit link, they are taken to the page `/deposit`.
8. When the user clicks on button log out, they are taken back to the `/` authentication page.
9. When user is on `/withdraw` page, they need to give the amount to withdraw and click on *Withdraw* button.
10. When the withdrawal is successful, user is taken to their dashboard.
11. When user is on `/deposit` page, they need to give the amount to deposit and click on the *Deposit* button.
12. When the deposit is successful, user is taken to their dashboard.

Here are some more things that you need to take into account:

1. When the user visits the `/` page but he is already signed in, then they are redirected to the `/dashboard` page.
2. When the user tries to visit any of the pages that require authentication (`/dashboard`, `/withdraw`, `/deposit`) but they are not signed in, then they are being redirected to the `/` page.
3. When authentication, or the withdrawal, or the deposit fails, then the user stays on same page but a message appears giving them more information about the problem.
4. The Web application needs to save the account data persistently. You an use a YML or a JSON file to do that. Assume that you already have a file like that with some accounts inside.
5. Application should be using Twitter Bootstrap.

On the following video you can see the final result in action:

Sinatra App - Task - ATM Application

Important

1. You need to upload your code to your Github account.
2. You need to put your application to production onto Heroku.