Ruby Web Servers

Prerequisites

Ruby Web Clients

Yesterday we talked about writing code that accesses the internet as a client. Today we'll be creating code for the other side of that relationship: the server.

What you'll learn

- How to build an HTTP server with Sinatra.
- How to run your server.
- How to deploy your server to Heroku.

Before we begin

https://toolbelt.heroku.com/

```
heroku login

gem install sinatra bundler
brew unlink ruby && brew link ruby
```

Before we begin, make sure you have installed the Heroku Toolbelt (from the prework). That allows you to use the "`heroku`" Command Line command. You can confirm that things are working correctly with the "`heroku login`" command in the Terminal.

You'll also want to install the sinatra and bundler gems. Bundle installs a new Command Line command, so for those of you using Homebrew, you'll have to unlink and relink to get that command to work.

Project Initialization

`cd` to your project directory, then

```
mkdir server
cd server
git init
```

You've learned a lot about the web at this point. You know how it works. You know how to write code that accesses it. The only thing left is to start building websites. So let's get started.

First, let's officially initialize a brand new project. We'll call this one "server". We'll init a new git repo in this folder as well.

server.rb

```
require 'sinatra'

get '/' do
  "Hello world"
end
```

Let’s create a file named “server.rb”. In it, all we need to implement a “Hello world” webpage is these lines of code. We first require the ‘sinatra’ gem that we installed. Then call the “get” method, which takes a path and a block. What we’re saying with these lines of code is that when someone issues an HTTP GET method to our server with no path, let’s return the String “Hello world”.

Sinatra

Sinatra is a free and open source library for creating web applications. It was created in 2007 by Blake Mizerany. It is one of the best examples of a “domain-specific language”, meaning using Sinatra is almost like using a new programming language dedicated specifically to the task of creating web sites.

Sinatra DSL

```
get '/' do
   .. show something ..
end

post '/' do
   .. create something ..
end

put '/' do
   .. replace something ..
end

delete '/' do
   .. annihilate something ..
end
```

Imagine for a moment that you didn't even know Ruby. If you understand how the internet works, reading a Sinatra app is very straightforward. You have the name of an HTTP method, followed by the URL path (which is everything after the domain), followed by a block of code that gets run when someone sends that method to that path.

Since you know Ruby, you can understand what Sinatra is actually doing. It's defining new methods in the current scope, one for each HTTP method. These methods each take a String and a block. By calling these methods, you associating blocks of code with web requests.

ruby server.rb

```
[2014-04-30 15:46:31] INFO  WEBrick 1.3.1
[2014-04-30 15:46:31] INFO  ruby 2.1.1 (2014-02-24)
[x86_64-darwin13.0]
== Sinatra/1.4.5 has taken the stage on 4567 for
development with backup from WEBrick
[2014-04-30 15:46:31] INFO  WEBrick::HTTPServer#start:
pid=35720 port=4567
```

With just that one file and those few lines of code we saw earlier, we can start a web server. From the Terminal, execute the server.rb file using the ruby command. Now our web server is running, or, as Sinatra puts it, “Sinatra has taken the stage”.

As the output states, Sinatra is running on port 4567. So, to access the server, we need to access our laptop over the internet using port 4567.

Do you remember how to access your own computer over the internet?

curl to localhost

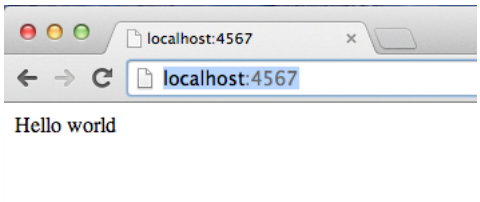
```
curl http://localhost:4567
```

Hello world

Localhost (or 127.0.0.1) is the loopback address. Instead of sending our request out over the internet, the request comes right back to our computers.

In a new Terminal tab, if we curl this address at port 4567, we get our “Hello World” output (without a newline).

In a web browser



We can even access this URL in a browser and see our output appear in the page.

Congratulations, you just created your first website!

Web Logs

::1 - - [30/Apr/2014 15:55:07] "GET / HTTP/1.1" 200 11 0.0003
localhost - - [30/Apr/2014:15:55:07 EDT] "GET / HTTP/1.1" 200 11
- -> /

In the Terminal tab where Sinatra is running, you can see all the web requests being logged. You can see the date and time the request was made, the HTTP method, the URL path requested, the HTTP version, the HTTP response status code, and the time elapsed and/or the size of the response (depending on which line you're looking at).

404 Page

Sinatra doesn't know this ditty.

Try this:

```
get '/blah' do
  "Hello World"
end
```

If you try to access a path that doesn't exist, like /blah, Sinatra returns this classy 404 error page suggesting some lines of code you can use to make this page work.

Web Logs

::1 - - [30/Apr/2014 15:58:36] "GET /blah HTTP/1.1" 404 441 0.0009
localhost - - [30/Apr/2014:15:58:36 EDT] "GET /blah HTTP/1.1" 404 441
- -> /blah

You can confirm via the logs that you attempted a GET to the /blah path and received a 404 in response.

Killing Sinatra

^C== Sinatra has ended his set (crowd applauds)
[2014-04-30 16:03:38] INFO going to shutdown ...
[2014-04-30 16:03:38] INFO WEBrick::HTTPServer#start done.

To stop the Sinatra server, hit Ctrl+C in the Sinatra tab. Sinatra “ends his set” to “crowd applause”, then the server shuts down. Classy to the end.

server.rb

```
get '/' do
   "Hello world"
end

get "/sinatra" do
   "Hello Sinatra"
end
```

Start Sinatra back up again, then go to the server.rb file and add these lines to add a “sinatra” path that returns “Hello Sinatra”.

Checking for updates

```
curl -i http://localhost:
4567/sinatra
```

HTTP/1.1 404 Not Found

...

Uh-oh. We're still getting a 404 on our new path. What happened?

Bounce Sinatra

^C== Sinatra has ended his set (crowd applauds)

```
ruby server.rb
```

== Sinatra/1.4.5 has taken the stage on 4567 for development with backup from WEBrick

You need to bounce the Sinatra server to get it to see the new code. Stop it with ctrl-c, then restart it again.

Check Again

```
curl http://localhost:4567/sinatra
```

Hello Sinatra

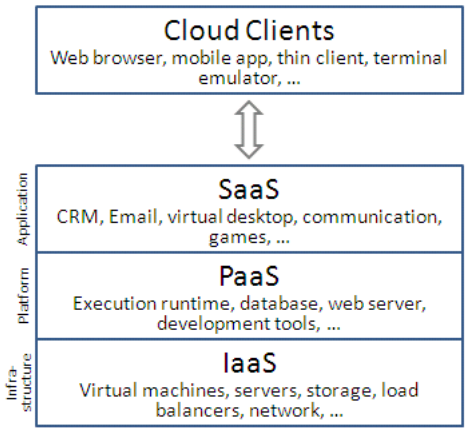
Now our new /sinatra path works.

Heroku (PaaS)

So we've got a website running on our computer. Now it's time to show the world.

Heroku is a cloud-based PAAS (platform-as-a-service). The AAS (as-a-service) part basically means it's something that's running in a web browser (aka "in the cloud") rather than on your computer. The "platform" part is a bit more complicated.

*aaS



First there was SAAS, software-as-a-service. That was your web-based email, online games, Google Docs, etc. Those services directly replace applications that you'd download and run on your computer.

Then, even though everything is technically "software", we came up with new terms to describe different types of services. So web sites that allow you to manage infrastructure (servers, disks, networks) became knows as IaaS, Infrastructure as a Service. Those are things like Amazon Web Services and Digital Ocean. They replace you having to drive over a datacenter to setup (and reboot) hardware for your website.

But Infrastructure is complicated. You need to be a system administrator (or sysadmin) to know how to work all that stuff. You need to be really good with the Command Line. If you don't have time for that, there's PaaS (platform-as-a-service). PaaS makes it easy for you to deploy a SaaS because you don't have to worry about the IaaS. Basically, Heroku is a platform where you can deploy your website without having to know a lot about servers. Which makes it perfect for us because we don't have a lot of time to get into setting up and configuring servers.

Gemfile

```
source "https://rubygems.org"
ruby "2.1.3"
gem "sinatra"
```

There are two files you need to include in your project to let Heroku know how to deploy your website. The first is Gemfile (capital-G Gemfile. Not Gemfile.rb, just Gemfile).

A Gemfile is written in Ruby syntax.

The source method accepts a String letting Heroku know where to download gems. This is almost always RubyGems.org. We use https to make the NSA's job more difficult.

The "ruby" method accepts a String letting Heroku know which version of Ruby you'd like to use to run your website. I just copy-and-pasted here the result from from running the "ruby -v" command (without all the stuff after the p).

The rest of the file is a list of gems you're using in your site. The only one we're using for this "Hello World" web site is sinatra. Most sites will have multiple lines of gems specified.

bundle install

```
Fetching gem metadata from https://rubygems.org/.....
Fetching additional metadata from https://rubygems.org/..
Resolving dependencies...
Using rack 1.5.2
Using rack-protection 1.5.3
Using tilt 1.4.1
Using sinatra 1.4.5
Using bundler 1.6.2
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

The Gemfile is used to tell Heroku which gems you would like them to install on the server. And Heroku uses bundler to install those gems. So, to make sure everything is working ok, run the “bundle install” command, which will read the Gemfile and install all the gems listed. This is redundant for you because you’ve already installed the Sinatra gem, but it verifies that the Gemfile is correct.

The output of “`bundle install`” is a file named Gemfile.lock, which lists the specific versions of each gem you need. That helps protect you in case a new version of a gem is released that breaks your site. With Gemfile.lock, you lock the version to the ones you’re using now.

Procfile

```
web: ruby server.rb -p $PORT
```

The next file you need to configure Heroku is the Procfile (capital-P Procfile. Not Procfile.rb, just Procfile). This is not Ruby code.

The first part, up to the colon, tells Heroku what kind of site this is. We're deploying a web site. Another kind of site is a "worker", which is a program that runs on a server, but doesn't have a web interface. So, for example, you can deploy a program to Heroku that pulls stock prices from Yahoo every few seconds and stores them somewhere.

After the colon comes the command you use to run the server - the same command you'd enter into the Command Line. In this case, we're using a command line variable to let Heroku decide which port to run the server on.

heroku create

```
Creating aqueous-inlet-2159... done, stack is
cedar
http://aqueous-inlet-2159.herokuapp.com/ |
git@heroku.com:aqueous-inlet-2159.git
Git remote heroku added
```

Once the config files are ready, run the "`heroku create`" command, which reserves a spot for your app on a Heroku server. You may notice references to Git in the output. The Heroku deployment process is built on Git. If you know how to push to GitHub, you already know how to deploy to Heroku.

git commit

```
git add .
git commit -m "first Heroku commit"
```

Since Heroku is based on Git, we need to make our first Git commit. Let’s add all the files we’ve created so far, then commit.

Push to deploy

```
git push heroku master
```

If you get a publickey error:

```
heroku keys:add
```

and try again.

Once we have some commits in Git, it's time to push our master branch up to Heroku, similar to the way we'd push our changes up to GitHub.

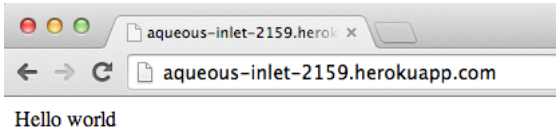
If you get an error that says something about "Permission denied (publickey)", then upload your keys to Heroku and try again. Not just anyone can upload code to your Heroku server, so some extra protection may be called for.

git push heroku master

-----> Compressing... done, 13.0MB
-----> Launching... done, v4
http://aqueous-inlet-2159.herokuapp.com/
deployed to Heroku

You'll see a lot of Terminal output while the site is deploying. The lines at the end are the important ones. They tell you the URL to your newly deployed website.

And we're live!



And there you go. Your first website, live online. You can share it with your friends if you'd like.

Public Static Files

```
mkdir public

echo 'Hello world!' > public/hello.txt

ruby server.rb
```

During the next few lectures we're going to take a break from Ruby and talk about HTML, CSS, and JavaScript for a while. These "front-end" technologies run inside of browsers. Ruby, a "back-end" technology, runs on the server.

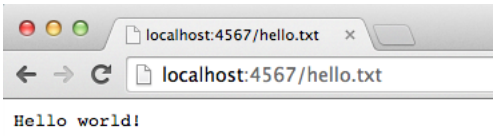
Ruby isn't a requirement for running a website, and, in fact, most websites don't rely on any kind of back-end programming at all. When you make a request to a website, you usually just get a plain old file. Usually it's an HTML file, but it doesn't have to be.

These plain files are called static files because they don't change. Every time you go to the website, the file is the same. There's nothing a website user can do to change it. It only updates when the developer updates it.

In contrast, there are dynamic websites, where the content on a page can change depending on, for example, whether or not you're logged in or the data being pulled from an API. We'll talk about dynamic websites, also known as web applications, later. That's where Rails comes in.

In the meantime, let's setup our Heroku application to act like a simple static website server. To do that, all we need to do is create a directory named "public" in our project. Let's create a file named "hello.txt", add the lines "Hello world" to it, and add it to that folder. Then let's start the server up and see what happens.

Public Static Files



Excellent! As you can see, Sinatra just returns the file we uploaded. We didn't have to define a new route.

Note that I didn't even have to use the word "public" in the URL. Sinatra just knows that, if it can't find a route matching the URL path, then it should check the public folder to see if any files match.

This is how we're going to deploy our static websites. We're going to create files that end in html, css, and js and add them to our public folder. Then we can test them out via localhost. When we're ready, we can push to Heroku and our site will be live!