Ruby Web Clients

Prerequisites

Intro to the Internet

What's you learn

- What are clients and servers?
- What's an API?
- Consuming APIs with HTTParty
- Web scraping with Nokogiri and XPath
- Some Chrome DevTools

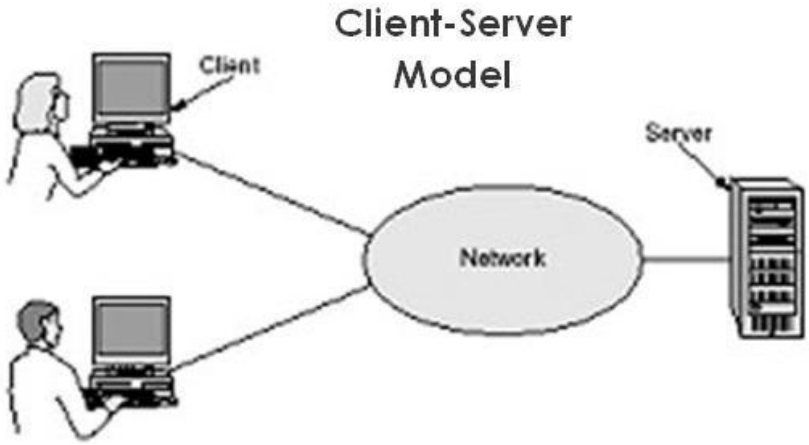
Before we begin

```
gem install httparty nokogiri
```

Now that you know Ruby, for the remainder of the course we'll be talking about specific gems. Sometimes gems can take a while to download and install, so get that started now and hopefully it'll be done by the time we need it.

Some of you will have trouble installing Nokogiri. I'll send out some more specific instructions to help those folks out later.

Client-Server Model



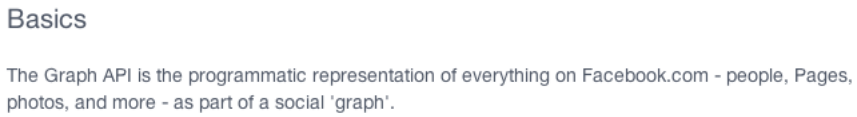
In HTTP, the client connects to and communicates with the server. “Client” can mean multiple things: the computer itself, the operating system, the browser, or something inside the browser. The “Server” can also mean multiple things: the server computer itself, the server os, the server program, or some part of the server program.

We’re going to focus on software side, so a “client” is going to be a program that initiates a connection and sends requests to a “server”. A “server” is a program that accepts that connection and processes those requests, returning a response.

Rails is a “server-side” Ruby library. It accepts connections and processes requests. Before we talk about the “server-side”, we’re going to spend some time talking about the “client-side”. We’ll begin by creating an HTTP client in Ruby, then later we’ll talk about client-side processing in a web browser.

API

The Facebook Graph API:



An API is an “application programming interface”. There’s a lot of confusion about this term because some people use it to describe any kind of interaction between two things.

For our purposes, an API specifies how two programs (rather than two people or a person and a program) should talk to each other. HTTP is an API. It allows a program (like curl) to interact with a server via HTTP methods. REST is also an API. The REST pattern extends the HTTP API with some conventions about how objects should be created, updated, and deleted.

When a server is said to be offering or presenting an API, the expectation is that the client that connects and interacts with that server is code, not a person.

Contrast an API with a UI (user interface) or GUI (graphical user interface), which are meant to be used by people. Your web browser offers you a GUI. Behind the scenes, the browser talks to the server via an API.

Facebook's Graph API

GUI:
https://www.facebook.com/eddroid

API:
https://graph.facebook.com/eddroid

Go to your Facebook profile, if you have one. If not, use mine. The URL to your profile loads a GUI in your browser. I can interact with this profile by pointing and clicking on things with my mouse pointer.

Replace the "www" in the URL with "graph". This returns your Facebook profile via the Facebook Graph API. It's presented in a format best-suited for a program to process it. It should look familiar: it's a Hash.

HTTParty

```
require 'httparty'
response = HTTParty.get('http://graph.
facebook.com/eddroid')
puts response.body, response.code, response.
message, response.headers.inspect
```

Let's have an HTTParty! Hopefully the gem is installed by now. To use it, we first require it. That gives us access to the HTTParty class. The class has a number of class-scoped methods, one for each of the HTTP methods. So we can use the "get" method to issue an HTTP GET to the Facebook graph API.

The "get" method returns an HTTP response object containing a body, an HTTP status code and message, and the response headers. We've seen all of this stuff before in the HTTP lecture, only now we have the results in Ruby objects rather than as just text in the Terminal window. Now we can actually do something useful with them.

Response Headers

```
puts response.class
```
=> HTTParty::Response

```
puts response.headers.class
```
=> HTTParty::Response::Headers

Note that HTTParty returns the response as a custom class called HTTParty::Response. The headers are also a custom class, HTTParty::Response::Headers. Both of these custom classes take advantage of module namespacing. While it's likely someone may create a "Response" class in Ruby, prefixing it with the module name HTTParty helps make it unique, avoiding name clashes.

Response Headers

```
puts response.headers.inspect

p response.headers
```

To display the Response Headers from my request, I need to call the “inspect” method to first convert the custom class into a String that I can pass to puts. Alternatively, I can use the “p” method, which calls “inspect” automatically.

Response Headers

{"etag"=>["\"13c7f0554589d7c4f15a3c258060416a97101d3e\""], "content-type"
=>["text/javascript; charset=UTF-8"], "pragma"=>["no-cache"], "access-control-
allow-origin"=>["*"], "x-fb-rev"=>["1224624"], "cache-control"=>["private, no-
cache, no-store, must-revalidate"], "expires"=>["Sat, 01 Jan 2000 00:00:00
GMT"], "x-fb-debug"=>
["HNQG1hjvYFHnrHr8PSCevKCDCTDubtvSKb6gXIxFKS8="], "date"=>["Mon,
28 Apr 2014 05:23:47 GMT"], "connection"=>["close"], "content-length"=>
["134"]}

In either case, my headers look like this. Take a close look at the content-type.

Content-Type

```
puts response.headers["content-type"]
```

=> text/javascript; charset=UTF-8

Since it “quacks like a Duck”, let’s treat it like a Duck, even though it isn’t. In other words, since the headers seem to behave like a Hash, even though it isn’t a Hash object, let’s treat it like a Hash and try to access the value stored in by the “content-type” key.

Here we see that the content type is text/javascript. We’ve seen text/html before, when we used curl to get the Google homepage. But HTML is meant for people. Since an API is meant for a computer, the response is javascript (or you may see application/json, which is the same).

text/javascript

```
puts response.body.class
```
=> String

```
puts response.body
```
=> {"id":"646374019","first_name":"Edward","gender":"male","last_name":"
Toro","locale":"en_US","name":"Edward Toro","username":"eddroid"}

So Facebook is telling us in the response header that we should expect to receive JavaScript (or JSON, JavaScript object notation). But if we introspect the response body, we see that it's a String. We need to treat this String as if it were JavaScript.

JSON

```
require 'json'
body = JSON.parse response.body
puts body.class
```

=> Hash

```
puts body
```

=> {"id"=>"646374019", "first_name"=>"Edward", ...

Lucky for us, the Ruby standard library comes with a json module. That means Ruby can understand the JavaScript Object Notation and turn it into a Ruby object for us. Using the class-scoped parse method in the JSON class, we can turn a JavaScript String object into a Ruby Hash object.

JSON

```
puts "My Facebook id is #{body['id']}."
```
=> My Facebook id is 646374019.

```
puts "My name is #{body['first_name']}."
```
=> My name is Edward.

Once we have the body of the response in a useful format, we can get to work. For example, we can output some human friendly sentences containing the public details of my Facebook profile.

What we've learned is that the Facebook Graph API returns JSON in response to GET requests. Using HTTParty, we were able to get that JSON String and convert it into a Ruby Hash, which we are then able to write programs with.

POST

http://requestb.in/

Let's say we want to use HTTParty to POST or PUT data. Now usually we'd have to create a profile on some service in order to POST or PUT data. After all, where would it be stored?

But, since we're just learning, we can use a site called RequestBin (URL spelled this way) to quickly create a test server that will display any data we POST or PUT.

POST

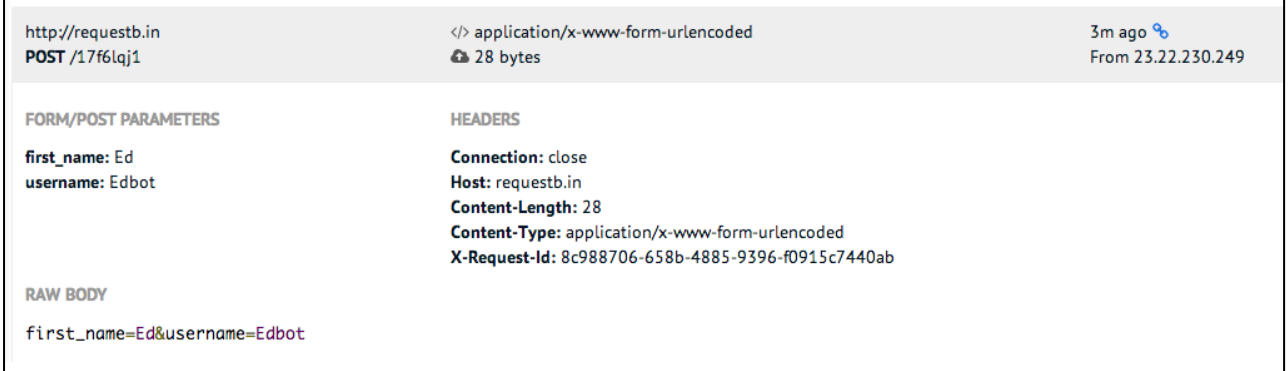
```
data = {first_name: 'Ed', username: 'Edbot'}

response = HTTParty.post('http://requestb.
in/17f6lqj1', { body: data })

puts response.body
```
=> ok

Since POST and PUT both involve sending data to a server, I need to define some data first. Let's say I want to pretend I'm updating my Facebook profile. I want to change my first_name to "Ed" and my username to "Edbot". So I can construct a Hash containing the data I'd like to update, with keys for the fields and values for the new values.

Then, using HTTParty, I can POST my data to my RequestBin. According to the documentation for HTTParty, the data is accepted as part of an options Hash with the key "data".

In response, I get a simple "ok".

POST



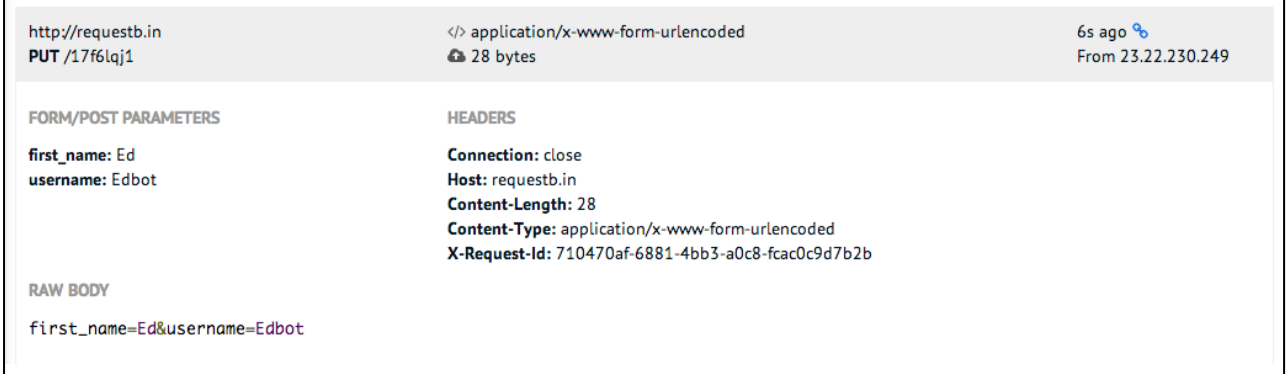
When I refresh my RequestBin, I can see that it has received a POST request containing data that was encoded as “application/x-www-form-urlencoded”, which is the default encoding for sending HTTP data to a server. The body of my request was a String containing my key/value pairs, which RequestBin was able to parse into a Hash correctly.

PUT

```
data = {first_name: 'Ed', username: 'Edbot'}

response = HTTParty.put('http://requestb.
in/17f6lqj1', { body: data })

puts response.body
```
=> ok

Since I'm pretending to update my profile, the RESTful convention says that I should be using a PUT instead of a POST. So I can simply change my method to a PUT and try again.

PUT



Now everything is the same, except RequestBin recorded a PUT instead of a POST.

JSON

```
body = {first_name: 'Ed', username: 'Edbot'}
headers = {'Content-Type' =>
'application/json'}

response = HTTParty.post('http://requestb.
in/17f6lqj1', { body: JSON.dump(body),
headers: headers})
```

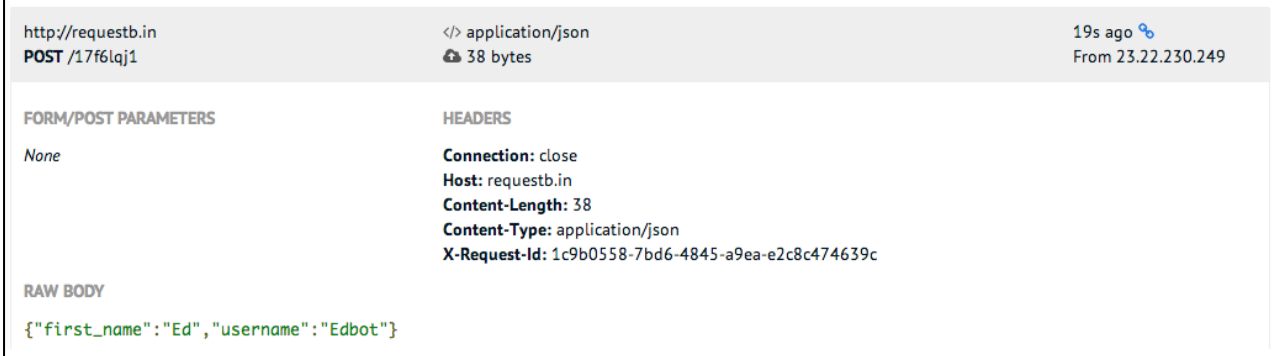
The Facebook API sent me data in the JSON format. What if it wants me to send it back data in the same format? How do I do that?

Up to now I've been using the default format. I need to change it to JSON.

Just like a server can tell me what format to expect via a Response Header, I can tell the server what format to expect my data using a Request Header. In fact, I can use the exact same header, the "Content-Type" header.

HTTParty accepts the headers in the headers key of the options Hash. Since I'm sending JSON data this time, I need to use the JSON.dump method to turn my Hash into a JSON String. JSON.dump does the opposite of JSON.parse.

JSON



My RequestBin received my POST request. The body of the request contains my JSON representation of my Hash. But RequestBin doesn't seem to have been been able to parse my JSON String back into a Hash of parameters. I guess that's not the format it wanted.

Web as API

```
require 'httparty'

response = HTTParty.get('http://finance.
yahoo.com/q?s=AAPL')

puts response.body
```

The web itself is an API. Your browser is making a request to a server to get a String. The String is encoded as HTML (rather than, say, JavaScript or JSON). Your browser knows how to turn that HTML into a document you can scroll around and click on. But, like a String of JavaScript, you can parse and process that String yourself using Ruby.

As an example, let's grab the String representing the Yahoo Finance site for Apple's stock.

Web as API

```
puts response.headers['content-type']
```

=> text/html; charset=utf-8

Like before, let's check the Content-Type to make sure we're getting HTML. We are. So now we need to find a method that turn this String of HTML into something useful.

Nokogiri

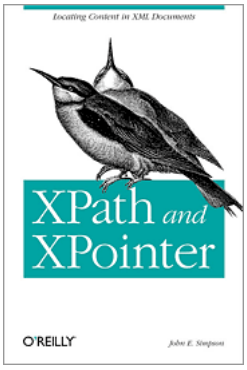
```
require 'nokogiri'

dom = Nokogiri::HTML(response.body)
puts dom.class
```

=> Nokogiri::HTML::Document

That's where Nokogiri comes in. The Nokogiri library has an HTML parser that takes an HTML String and turns it into a custom Class object called a Document.

According to the documentation, a Document provides a few different methods for finding things in an HTML page. One of those methods accepts a line of CSS code. Another accepts a line of XPath code. We'll learn about CSS later, so let's take a look at the XPath first.

XPath



XPath is a language used to navigate through XML (the Extensible Markup Language), a type of structured document format. HTML (the HyperText Markup Language, the language of websites) is closely related to XML. So, to parse the HTML String, Nokogiri transformed it into XML.

XPath

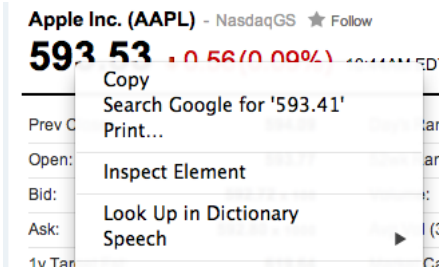
`dom.xpath("/")`
The entire page.

`dom.xpath("//*")`
All the individual elements of the page.

To get the entire webpage in a single object, use the forward-slash. To get a list of all the individual elements in the page, use a double-slash followed by an asterisk.

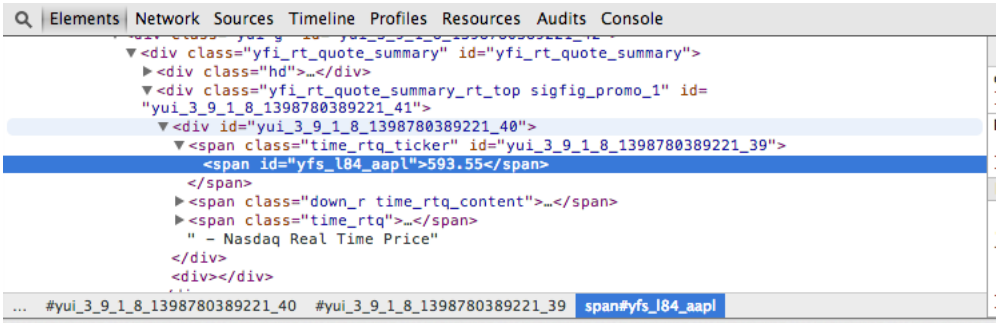
I'm interested in Apple's current stock price. So I need something between these two results: not the whole page, and not every individual element.

Chrome DevTools



To find out what I need, I need to find out how to describe the part of the page containing the stock ticker price. So, in Google Chrome, I can right click the price and select “Inspect Element” to open up the Chrome DevTools.

Chrome DevTools



In the DevTools, Chrome will highlight the specific line of HTML corresponding to where I clicked on the page. From this HTML, I can construct an XPath to find the stock price.

Chrome Dev Tools

```
<span class="time_rtq_ticker">
  <span id="yfs_l84_aapl">
    593.55
  </span>
</span>
```

The area around the ticker price looks like this. There's a SPAN element in the outside. Within it is another SPAN element. And within that is the price that I want.

XPath

```
dom.xpath("//*").size
```
=> 889

```
dom.xpath("//span").size
```
=> 94

There are 889 elements in the page. The element I want is in there somewhere.

I want a span, so if I filter that list to only return spans, I get 94. Awesome, I just shrunk my search down to 1/10 of what it was before. But 94 is still a lot.

XPath

```
<span class="time_rtq_ticker">
  <span id="yfs_l84_aapl">
    593.55
  </span>
</span>
```

What I really want is the span with the id “yfs_l84_aapl”.

XPath

```
dom.xpath("//span[@id='yfs_l84_aapl']").size
```
=> 1

```
dom.xpath("//span[@id='yfs_l84_aapl']").class
```
=> Nokogiri::XML::NodeSet

From the list of all SPANs, to select the one with a particular id, I can use this syntax. Now I've limited my options down to 1, which is exactly what I want.

This result is an instance of the NodeSet class. But it seems to behave like a list. If it quacks like a duck...

XPath

```
dom.xpath("//span[@id='yfs_l84_aapl']").first
```

=> #<Nokogiri::XML::Element: (...)

If I get the first and only item in my list, I see that it's an Element object. Now I need to check the online documentation how to get what I want out of this class.

Nokogiri

It looks like the Element class inherits all it's methods from some class named Node, so let's check that page.

Nokogiri

Class Nokogiri::XML::Node inherits from Object

Nokogiri::XML::Node is your window to the fun filled world of dealing with XML and HTML tags. A Nokogiri::XML::Node may be treated similarly to a hash with regard to attributes. For example (from irb):

On the documentation page for the Node object, I read that Node is my “window to the fun filled worked of dealing with XML and HTML tags”. So this sounds like what I’ m looking for.

Nokogiri

content()
  Returns the content for this Node

Further down this page I see a method named “content” that sounds like it’ll give me what I want. It returns the content inside the Node.

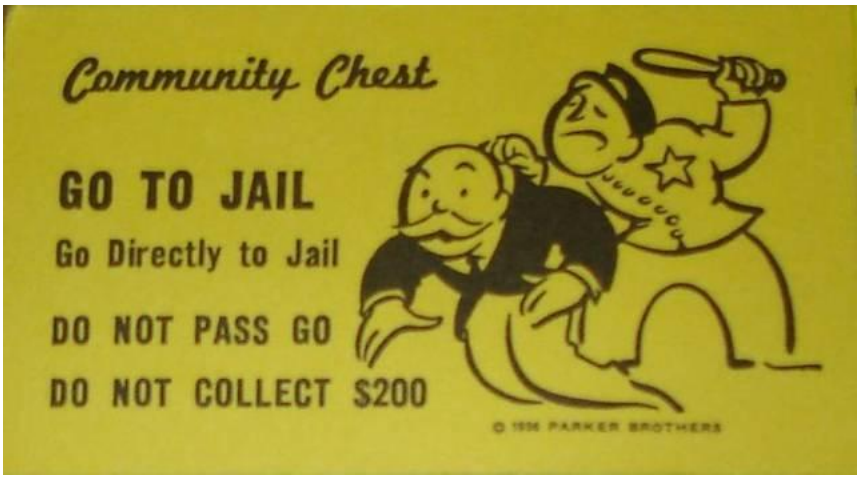
Web Scraping

```
my_span = dom.xpath("//span
[@id='yfs_l84_aapl']").first

puts my_span.content
```
=> 593.22

When I “puts” the result of calling the “content” method on the span I found, I see the result is the stock ticker price. Awesome!

This process of treating the HTML-encoded Strings of the web as APIs is called web scraping. We’re using Nokogiri and XPath to *scrape* data from a website. Then we can use that data in our own programs.

API vs. Scraping



If a website provides an API, then use it. It's much faster and easier.

But when it doesn't, consider using web scraping instead. But be careful. Sometimes copying content from someone's website and using it in your own programs violates the site's terms of service. It can be considered stealing. Don't build a business on other's people's content.

That being said, if you create something interesting by mashing together two pieces of information that hadn't been considered together before, then it could be useful for a resume, portfolio, news article, or business prototype. Interesting combinations of data from APIs or the web are called "mashups".