An Intro to Web Scraping

A simplifed overview of how data gets to and from the web browser

Times Square snow cleanup. Photo by Dan Nguyen

Chapter outline

Previous Chapter: Regular Expressions
Next Chapter: Meet Your Web Inspector

For journalists and researchers, being able to web-scrape is perhaps one of the most compelling reasons to learn programming. Agencies and organizations don't always release their information in nicely-formatted databases. Learning to web scrape allows you to gather, in an automated fashion, freely available data in virtually any kind of online format.

It's also one of the best ways to practice programming because the end goal is clear: either you have the data or you don't. And there's little harm to be done, provided your program scrapes at a reasonable rate.

This chapter introduces the strategy of web scraping with a very non-technical overview of how websites work. We won't cover any code. But knowing about give you the insight to immediately know how to traverse and parse any given website.

Scraping: The big picture

Rather than cover the concepts of GET, POST, HTTP requests, here's a broad overview of what happens when you interact with a website and submit a request.

I will demonstrate each step as it works on Wikipedia. Please make sure your Javascript is enabled for your browser in order to see all the action.

1. The User Acts

Precondition	The browser has loaded a webpage and interpreted the webpage's Javascript code
What Happens	The user interacts with some element on a webpage, or some other a time-sensitive event is triggered while the webpage is open.
Examples	Clicking a link, typing into a search field, hovering the mouse over an element, scrolling to the end of a page

This step consists of anything **you**, the user, does after a webpage has loaded. In the olden days, this usually meant clicking a link to go to the next page. Or, filling out a form and clicking the "Submit" button. Today, just moving your mouse or scrolling down the page will trigger an event that downloads more information onto the current webpage.

Interacting with Wikipedia

Open up http://en.wikipedia.org

Nothing too fancy here. You basically have the option of clicking on a link to get to an article or entering a term in the search box on the top right.

2. The Browser Reacts

Precondition	Some kind of event has been triggered, by time elapsed or user interaction
What Happens	The browser acts according to the webpage's Javascript instructions. For our purposes, the action is formulating a request to send to the server.
Examples	If the user has filled out a form, the browser collects the information in the fields, parses it, and creates an appropriate request for the webpage server. Or, every 30 seconds that the browser is open, it formulates a request to ask the web server if any new tweets/notifications have happened since the previous request for updates.

Again, in the old days, the browser wouldn't have to process anything until you clicked on a link or button. Then it would form a request to ask the server for a new webpage.

Just as the variety of ways of users interact with a website has increased, your web browser does calculations and work beyond just fetching the next webpage.

This is what's referred to as **client-side processing**. Your browser is the client. When a website serves a page to the browser, besides sending the text and images that make up the page's content, it will send **Javascript** files which contain code that tells the browser what to do when certain events occur.

These **events** include user interaction during step 1, such as hovering your mouse over a hotspot or typing into a form. The Javascript code governs how the browser reacts to your input. Thus, disabling Javascript prevents this kind of client-side processing, letting you browse the web before it became "2.0".

About Javascript

Javascript is the language for making the Web more than just a collection of text, links and images. It allows developers to do everything from making boxes slide smoothly on a webpage to creating entire applications, such as Google Maps, all inside the web browser. This is why Javascript is the most popular language among web developers.

To most casual web users, Javascript is most noticeable in features like dynamic updating seen in the Facebook notifications popup and the Wikipedia suggested terms in the search box. Without it, you would have to manually refresh the page to see the changes. This ability to gracefully refresh parts of a webpage with new data is what people mean when they say AJAX. The J stands for Javascript.

A sidenote: Not all client-side processing involves doing AJAXy actions, where the browser a request to the website's server.

For example, **click the following button**:

PUSH ME!

Thu Apr 12 2018 11:16:21 GMT+0200 (CEST)

Pushing that button executes a Javascript program that prints out *your* (or more accurately, your browser's) local time. Your browser has already downloaded the Javascript program. There's no need for the browser to reach across the Internet to get the date. To remind you of the term again, this is a case of **client-side processing**. If you shut off the Internet and pressed that button, it would still print a date. All the work is done by the browser.

Read more about Javascript at w3schools.com and Wikipedia.

You essentially have total access to the Javascript used to manipulate your browser because your browser has to download it in the first place before the Javascript can do anything. But if you're solely interested in a website's content, there is rarely any point in trying to decipher the raw Javascript.

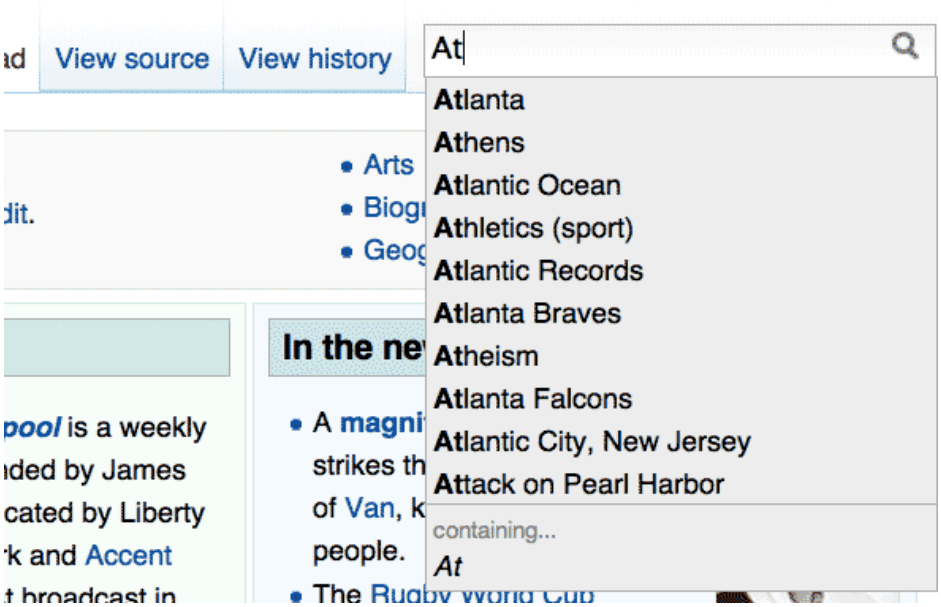
Instead, it's usually enough to examine the **effects** of the Javascript and treat the Javascript itself as a black box. In the next step, you can see exactly what the Javascript (and HTML) has the browser send to the server. There's no need to figure out how the Javascript actually packaged that request.

Wikipedia's search form

On Wikipedia's front page, type in a single letter, such as '**a**'

A dropdown list of topics that begin with the letter '**A**' should appear:

Add a second letter, such as '**t**', and that dropdown list should update with only topics that begin with '**At**'.



In the Javascript that Wikipedia sent to your browser, your browser is instructed to create and submit a new web request – the details of which we cover in step 3 – as soon as you type into the search field. This is done in the background silently while the user is still reading the Wikipedia front page.

3. The Browser Sends Request to Server

Precondition	The **browser** has formulated a **request** to send to the **server**
What Happens	The **browser** sends the request
Examples	Asking for the webpage at a URL that was clicked on by the user. Checking for any new Facebook notifications.

The simplest manifestation of this is when you click on an ordinary link to another page. The browser sends a request to the server to get that webpage. The new page then loads in the browser.

However, most modern webpages use Javascript to send requests to the server without any action by you, the user. You see this in effect on your Facebook and Twitter homepages when they auto-refresh with the latest updates and notifications

How does the browser know when, where, and how to check for the latest updates? From the Javascript files it downloaded in step 2. The code in those files give the browser all the logic and instructions needed to send periodic requests to Facebook or Twitter for any new notifications.

Wikipedia's search form's suggested topics

In step 2, the browser reacts to your typing into the search field by creating the following web request:

```
http://en.wikipedia.org/w/api.php?action=opensearch&search=At&namespace=0
```

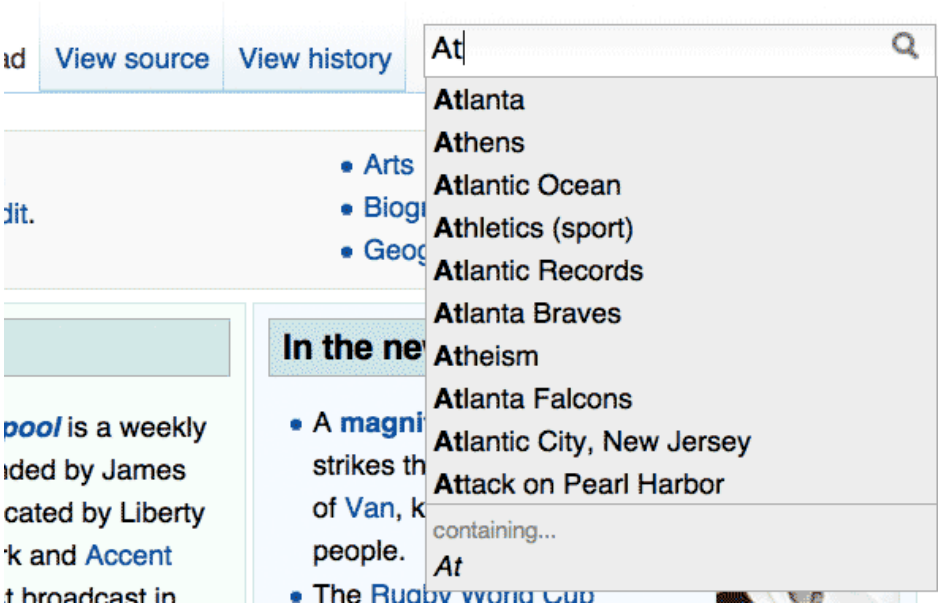
This is the request formed when I typed in "**At**"; I've emphasized the parameter that is affected by this. And this is the request send to the Wikipedia server.

In fact, you can visit the address of this request in your browser as you would any other URL: click here.

You should see a page of plain text (or be asked to download **api.php**, which contains the plain text). This is the server's response, as we'll cover in Step 5:

```
["At",["Atlanta","Athens","Atlantic Ocean","Athletics (sport)",
"Atlantic Records","Atlanta Braves","Atheism","Atlanta Falcons",
"Atlantic City, New Jersey","Attack on Pearl Harbor"]]
```

Type "**At**" in the search field as normal. Do the topics in the dropdown list match up?

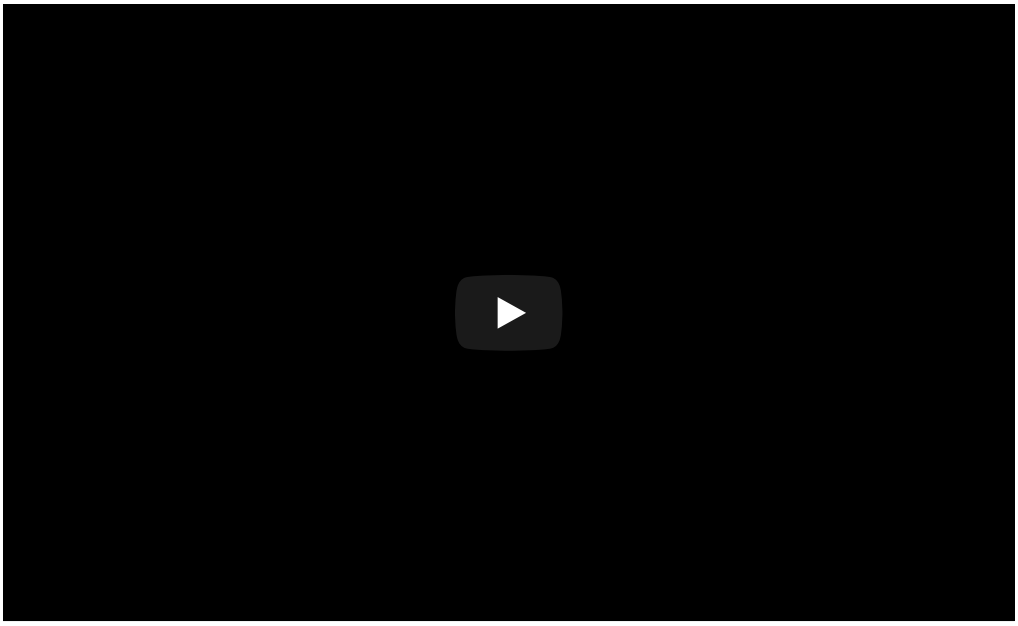


4. The Server Computes

Precondition	The **server** has received the **browser's request**
What Happens	The **server** processes the request.
Examples	Seeing if the requested account name and password are correct and retrieving the account information from the database. Performing a search based on the user's query.

After your browser sends the request (Step 3) you have to wait for the request to go across the Internet to get to the web server. Then you have to wait as the web server does the appropriate data crunching and retrieval. And after all that, you have to wait for the server's response (Step 5) to cross back over the Internet.

It'd be much faster if it all happened, well, in your browser, right? The owner of whatever service you're hitting up probably would agree. But for any website that's worth a damn, your browser and computer can't cut it. For example, here's <a href="http://www.youtube.com/watch?feature=player_detailpage&v=BNHR6IQJGZs#t=48s" title="How Search Works - YouTube">a description of the computational power to serve an everyday Google search:



Fulfilling your request also requires proprietary algorithms (PageRank, in the case of a Google Search) and credentials (the password to the site's database). Those are things a web site would rather not expose to the browser.

This kind of work is called **server-side** (or "back-end") **processing/scripting**. It typically is inaccessible to you and your browser*. So for all you know, the back-end script gathers the data in a millisecond and then sleeps for a few seconds just to make you appreciate it more.

Wikipedia's backend

Remember that URL your browser sent in Step 3?

```
http://en.wikipedia.org/w/api.php?action=opensearch&search=At&namespace=0
```

The server-side program that actually does something is:

```
http://en.wikipedia.org/w/api.php
```

How does it actually work? Well, if Wikipedia is functioning correctly, you won't know when you visit the script (api.php). In fact, the script has been configured to send you documentation on how to use it. But it doesn't tell you what actually happens at the back-end.

We can take a guess, though. Wikipedia has a list of topics in its database, and when we send it some letters, its database software retrieves a pre-set number of appropriate topics.

How it actually does that – including what password is used to gain access to the database – is kept secret* from us outside users. We only get to see the results.

"C'mon, I'd like to see the server-side code myself"

No really, *you can't*. Just forget about it and move on to Step 5 and content yourself with scraping the resulting data. So move along.

...*Unless* there is a **catastrophic f*ck-up** on the server's end, such as a bad configuration file or an insecure server-side script. That would effectively allow outsiders to see the raw code of a server-side script or send requests that would give direct access to a web site's database.

For the purposes of this book's lessons, we will never assume that this is the case, because a) it wouldn't really involve actual programming as much as just taking advantage of someone else's massive stupidity/bad luck and b) exploiting such security holes *may* put you on the wrong side of the law.

5. The Server Responds

Precondition	The **server** has finished processing the http://gawker.com/5855664/judge-abuse-of-disabled-daughter-not-as-bad-as-it-looks-on-tape
What Happens	The **server sends** the requested data, or some other message, such as an error code.
Examples	The webpage at the requested URL. Search results for the requested query. A 404 error: "Page not found"

After the web server has finished crunching the numbers and reading from the database, it sends back a response. Ideally, it's the data that your browser requested. Sometimes it may be an error message or an explanation of why the request was rejected. Either way, the response is typically a structured text-file, such as HTML, XML, or JSON.

This is where the actual web-scraping will happen. This is when your Ruby program grabs the response and parses it, rather than needing it to actually load in your web browser. Your Ruby program doesn't need to see a nicely-formatted webpage, it just needs the server's response in its raw form. And this step is where it gets that.

Wikipedia's response

We previewed Wikipedia's response in Step 3: the plaintext file it sends after we type in "At" into the search field.

```
["At",["Atlanta","Athens","Atlantic Ocean","Athletics (sport)",
"Atlantic Records","Atlanta Braves","Atheism","Atlanta Falcons",
"Atlantic City, New Jersey","Attack on Pearl Harbor"]]
```

This textfile is in JSON format, which in Step 6 is rendered by the browser as the HTML that forms the dropdown list of suggested items.

But we don't need the browser to see this. Once you know the URL and parameters, you can run the following Ruby script to download it and parse the parameters:

```
require 'rubygems'
require 'crack'
require 'open-uri'
require 'rest-client'

url='http://en.wikipedia.org/w/api.php?action=opensearch&search=At&namespace=0'

puts Crack::JSON.parse(RestClient.get(url))
```

This nets the following input:

```
At
Atlanta
Athens
Atlantic Ocean
Athletics (sport)
Atlantic Records
Atlanta Braves
Atheism
Atlanta Falcons
```

```
Atlantic City, New Jersey
Attack on Pearl Harbor
```

6. The Browser Makes It Pretty

Precondition	The **browser** has received the **server**'s response
What Happens	The **browser** formats the response to show to the **user**.
Examples	Displaying the requested webpage. Showing the number in a little red balloon of new notifications waiting for you. Showing "Page Not Found" in giant 50-point font.

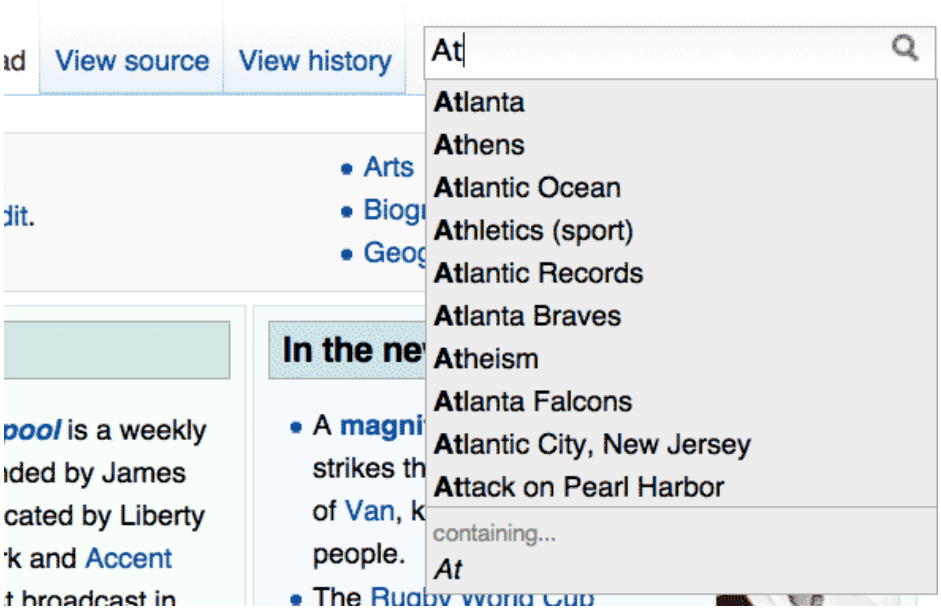
This last step is how you've experienced the Internet for most of your life. Web developers and designers have built out a design and scripted animation so that the raw data sent by the server, whether it be HTML or a JSON-array of new data, is formatted for your viewing and interactive pleasure.

In the context of automated webscraping, this step is essentially irrelevant to the a webscraper script. It doesn't need to see a pretty version of the data. In fact, the data made fit for human viewing and interaction is almost always **less structured** than it is in its raw form.

However, you – the human programmer – will find it helpful to actually look at the resulting page in order to figure out how the structure and context of the webpage and its data.

Wikipedia's formatted search terms

The browser, after receiving the Javascript (JSON) array of suggested search terms, iterates through it and outputs them in a nicely-formatted, clickable dropdown list:



Return to chapter outline

The scraping roadmap

From here on out, our discussion of web scraping will focus on two things:

1. Examining what the webpage expects from the user and what it shows the user: 1 and 6
2. Processing the data being sent or received by the browser: Steps 3 and 5

The next four chapters in this book are meant to be read in order and are devoted to the theory, technique, and programming needed to write web-scrapers.

I realize that makes this whole enterprise sound ridiculously complex. But it covers some Web basics for those who aren't web designer/developer-types and I wanted to break the techniques down into discrete steps. I also hope to provide enough examples so that you can figure out virtually every kind of publicly-facing website:

- Meet Your Web Inspector – This built-in tool makes it easy to read the structure of web pages.
- Inspecting a Webpage's Traffic – Use the web inspector to see the incoming and outgoing traffic from a webpage.
- Parsing HTML with Nokogiri – Quickly process a webpage's HTML and extract its data with the Nokogiri gem
- Writing a Web Crawler – Use your knowledge of HTML parsing and web inspection to programmatically navigate and scrape websites.

The first two chapters do not involve Ruby code. If you aren't into programming, these two chapters still may be very useful to you.

- **Next Chapter**
  Meet Your Web Inspector
- **Previous Chapter**
  Regular Expressions

- Chapter Outline
- Table of Contents

The Book

Version: 0.1

- Home
- About
- Contents
- Resources
- Blog

- Twitter: @bastardsbook
- Facebook

Author Info

Dan Nguyen is a journalist in New York

- Email: dan@danwin.com
- Blog: danwin.com
- Twitter: @dancow
- Tumblr: eyeheartnewyork
- Flickr