JAVA WEB SCRAPING HANDBOOK

Learn advanced Web Scraping techniques

Kevin Sahin

Java Web Scraping Handbook

Kevin Sahin

This book is for sale at http://leanpub.com/webscrapinghandbook

This version was published on 2018-07-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction to Web scraping

Web scraping or crawling is the act of fetching data from a third party website by downloading and parsing the HTML code to extract the data you want. It can be done manually, but generally this term refers to the automated process of downloading the HTML content of a page, parsing/extracting the data, and saving it into a database for further analysis or use.

Web Scraping VS APIs

When a website wants to expose data/features to the developper community, they will generally build an API(Application Programming Interface[1]). The API consists of a set of HTTP requests, and generally responds with JSON or XML format. For example, let's say you want to know the real time price of the Ethereum cryptocurrency in your code. There is no need to scrape a website to fetch this information since there are lots of APIs that can give you a well formated data :

```
curl https://api.coinmarketcap.com/v1/ticker/ethereum/?convert=EUR
```

and the response :

[1] https://en.wikipedia.org/wiki/Application_programming_interface

Coinmarketcap JSON response


```
{
    id: "ethereum",
    name: "Ethereum",
    symbol: "ETH",
    rank: "2",
    price_usd: "414.447",
    price_btc: "0.0507206",
    24h_volume_usd: "1679960000.0",
    market_cap_usd: "39748509988.0",
    available_supply: "95907342.0",
    total_supply: "95907342.0",
    max_supply: null,
    percent_change_1h: "0.64",
    percent_change_24h: "13.38",
    percent_change_7d: "25.56",
    last_updated: "1511456952",
    price_eur: "349.847560557",
    24h_volume_eur: "1418106314.76",
    market_cap_eur: "33552949485.0"
}
```


We could also imagine that an E-commerce website has an API that lists every product through this endpoint :

```
curl https://api.e-commerce.com/products
```

It could also expose a product detail (with “123” as id) through :

```
curl https://api.e-commerce.com/products/123
```

Since not every website offers a clean API, or an API at all, web scraping can be the only solution when it comes to extracting website informations.

APIs are generally easier to use, the problem is that lots of websites don't offer any API. Building an API can be a huge cost for companies, you have to ship it, test it, handle versioning, create the documentation, there are infrastructure costs, engineering costs etc. The second issue with APIs is that sometimes there are rate limits (you are only allowed to call a certain endpoint X times per day/hour), and the third issue is that the data can be incomplete.

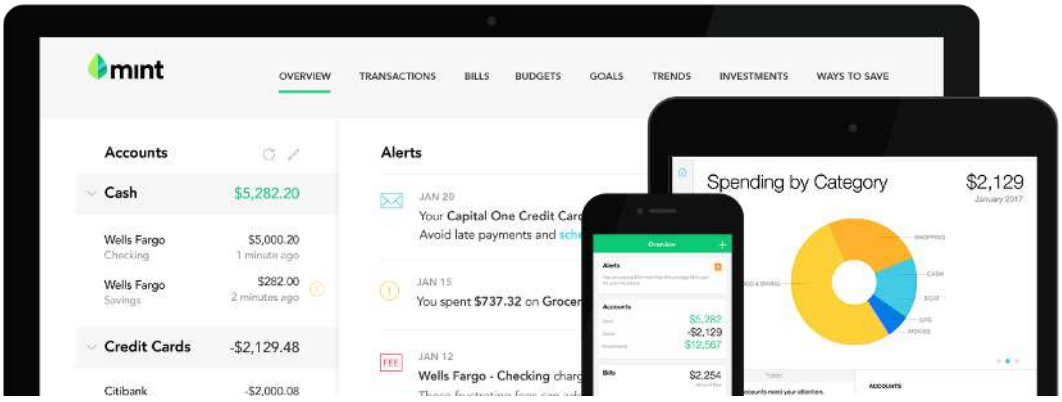
The good news is : **almost everything that you can see in your browser can be scraped**.

Web scraping use cases

Almost everything can be extracted from HTML, the only information that is "difficult" to extract is inside images or other medias. Here are some industries where webscraping is being used :

- News portals : to aggregate articles from different datasources : Reddit / Forums / Twitter / specific news websites
- Real Estate Agencies.
- Search Engine
- The travel industry (flight/hotels prices comparators)
- E-commerce, to monitor competitor prices
- Banks : bank account aggregation (like Mint and other similar apps)
- Journalism : also called "Data-journalism"
- SEO
- Data analysis
- "Data-driven" online marketing
- Market research
- Lead generation ...

As you can see, there are many use cases to web scraping.



Mint.com screenshot

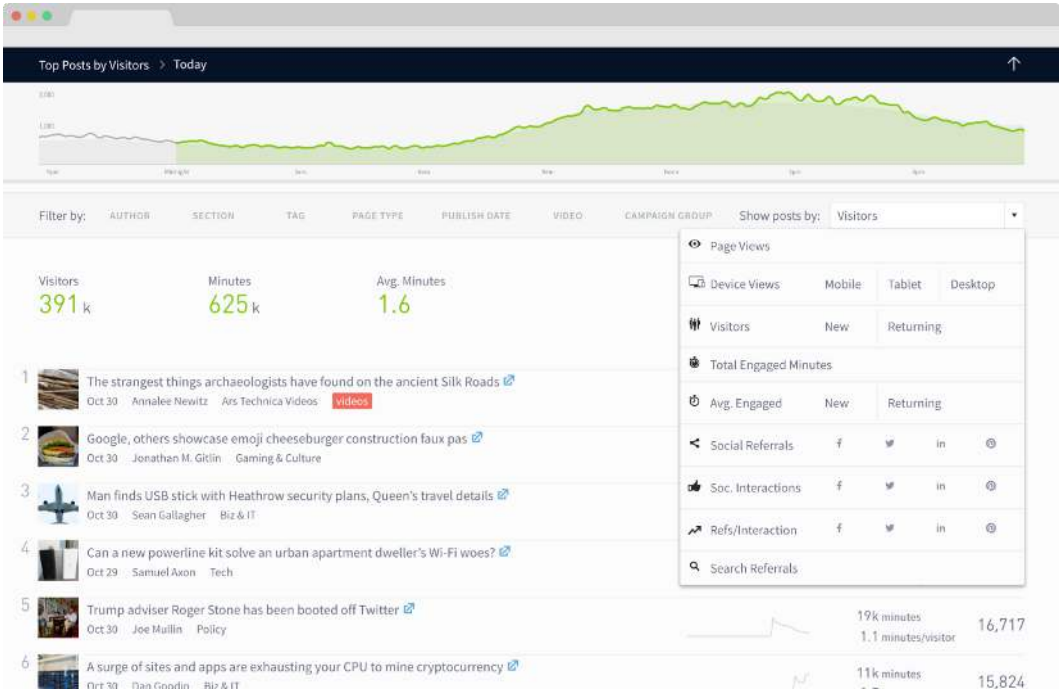
Mint.com is a personnal finance management service, it allows you to track the bank accounts you have in different banks in a centralized way, and many different things. Mint.com uses web scraping to perform bank account aggregation for its clients. It's a classic problem we discussed earlier, some banks have an API for this, others do not. So when an API is not available, Mint.com is still able to extract the bank account operations.

A client provides his bank account credentials (user ID and password), and Mint robots use web scraping to do several things :

- Go to the banking website
- Fill the login form with the user's credentials
- Go to the bank account overview
- For each bank account, extract all the bank account operations and save it intothe Mint back-end.
- Logout

With this process, Mint is able to support any bank, regardless of the existance of an API, and no matter what backend/frontend technology the bank uses. That's a good example of how useful and powerful web scraping is. The

drawback of course, is that each time a bank changes its website (even a simple change in the HTML), the robots will have to be changed as well.



Parsely Dashboard

Parse.ly is a startup providing analytics for publishers. Its plateform crawls the entire publisher website to extract all posts (text, meta-data…) and perform Natural Language Processing to categorize the key topics/metrics. It allows publishers to understand what underlying topics the audiance likes or dislikes.

Jobijoba meta-search engine

Jobijoba.com is a French/European startup running a plateform that aggregates job listing from multiple job search engines like Monster, CareerBuilder and multiple "local" job websites. The value proposition here is clear, there are hundreds if not thousands of job plateforms, applicants need to create as many profiles on these websites for their job search and Jobijoba provides an easy way to visualize everything in one place. This aggregation problem is common to lots of other industries, as we saw before, Real-estate, Travel, News, Data-analysis...

As soon as an industry has a web presence and is really fragmentated into tens or hundreds of websites, there is an "aggregation opportunity" in it.

What you will learn in this book

In 2017, web scraping is becoming more and more important, to deal with the huge amount of data the web has to offer. In this book you will learn how

to collect data with web scraping, how to inspect websites with Chrome dev tools, parse HTML and store the data. You will learn how to handle javascript heavy websites, find hidden APIs, break captchas and how to avoid the classic traps and anti-scraping techniques.

Learning web scraping can be challenging, this is why I aim at explaining just enough theory to understand the concepts, and immediatly apply this theory with practical and down to earth examples. We will focus on Java, but all the techniques we will see can be implemented in many other languages, like Python, Javascript, or Go.

Web fundamentals

The internet is **really complex** : there are many underlying techologies and concepts involved to view a simple web page in your browser. I don't have the pretention to explain everything, but I will show you the most important things you have to understand to extract data from the web.

HyperText Transfer Protocol

From Wikipedia :

> The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, and hypermedia information systems.[1] HTTP is the foundation of data communication for the World Wide Web. Hypertext is structured text that uses logical links (hyperlinks) between nodes containing text. HTTP is the protocol to exchange or transfer hypertext.

So basically, as in many network protocols, HTTP uses a **client/server** model, where an HTTP client (A browser, your Java program, curl, wget…) opens a connection and *sends a message* ("I want to see that page : /product")to an HTTP server (Nginx, Apache…). Then the server answers with a **response** (The HTML code for exemple) and closes the connection. HTTP is called a **stateless** protocol, because each transaction (request/response) is independant. FTP for example, is **stateful**.

Structure of HTTP request

When you type a website adress in your browser, it sends and HTTP request like this one :

Http request

```
GET /how-to-log-in-to-almost-any-websites/ HTTP/1.1
Host: ksah.in
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/web\
p,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch, br
Connection: keep-alive
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit\
/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36
```

In the first line of this request, you can see the `GET` verb or method being used, meaning we request data from the specific path : `/how-to-log-in-to-almost-any-websites/` There are other HTTP verbs, you can see the full list here[2]. Then you can see the version of the HTTP protocol, in this book we will focus on HTTP 1. Note that as of Q4 2017, only 20% of the top 10 million websites supports HTTP/2. And finally, there is a key-value list called **headers** Here is the most important header fields :

- **Host** : The domain name of the server, if no port number is given, is assumed to be 80.
- **User-Agent** : Contains information about the client originating the request, including the OS information. In this case, it is my web-browser (Chrome), on OSX. This header is important because it is either used for statistics (How many users visit my website on Mobile vs Desktop) or to prevent any violations by bots. Because these headers are sent by the clients, it can be modified (it is called "**Header Spoofing**"), and that is exactly what we will do with our scrapers, to make our scrapers look like a normal web browser.
- **Accept** : The content types that are acceptable as a response. There are lots of different content types and sub-types: ***text/plain***, ***text/html***, ***image/jpeg***, ***application/json*** ...

[2] https://www.w3schools.com/tags/ref_httpmethods.asp

- **Cookie** : name1=value1;name2=value2... This header field contains a list of name-value pairs. It is called **session cookies**, these are used to store data. Cookies are what websites use to authenticate users, and/or store data in your browser. For example, when you fill a login form, the server will check if the credentials you entered are correct, if so, it will redirect you and inject a session cookie in your browser. Your browser will then send this cookie with every subsequent request to that server.
- **Referer** : The Referer header contains the URL from which the actual URL has been requested. This header is important because websites use this header to change their behavior based on where the user came from. For example, lots of news websites have a paying subscription and let you view only 10% of a post, but if the user came from a news aggregator like Reddit, they let you view the full content. They use the referer to check this. Sometimes we will have to spoof this header to get to the content we want to extract.

And the list goes on...you can find the full header list here[3]

The server responds with a message like this :

Http response

```
HTTP/1.1 200 OK
Server: nginx/1.4.6 (Ubuntu)
Content-Type: text/html; charset=utf-8
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    ...[HTML CODE]
```

On the first line, we have a new piece of information, the HTTP code `200 OK`. It means the request has succeeded. As for the request headers, there are lots of HTTP codes, split in four common classes :

[3] https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

- **2XX** : Successful, understood, and accepted requests
- **3XX** : This class of status code requires the client to take action to fulfill the request (i.e generally request a new URL, found in the response header **Location**)
- **4XX** : Client Error : `400 Bad Request` is due to a malformed synthax in the request, `403 Forbidden` the server is refusing to fulfill the request, `404 Not Found` The most famous HTTP code, the server did not find the resource requested.
- **5XX** : Server errors

Then, in case you are sending this HTTP request with your web browser, the browser will parse the HTML code, fetch all the eventual assets (Javascript files, CSS files, images...) and it will **render** the result into the main window.

HTML and the Document Object Model

I am going to assume you already know HTML, so this is just a small reminder.

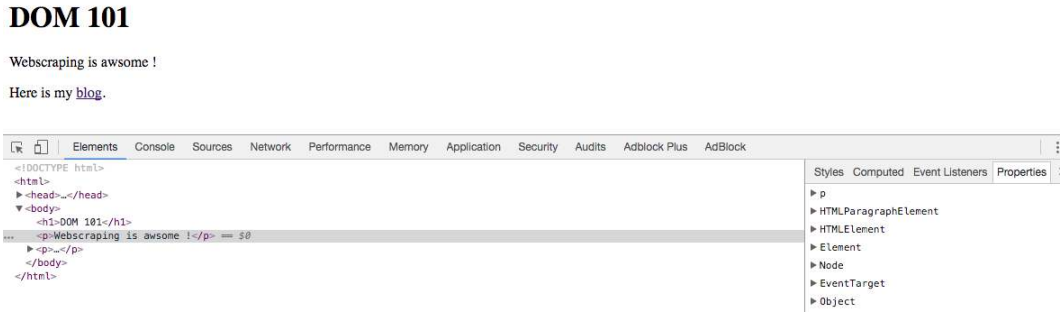
- HyperText Markup Language (HTML) is used to add "meaning" to raw content.
- Cascading Style Sheet (CSS) is used to format this marked up content.
- Javascript makes this whole thing interractive.

As you already know, a web page is a document containing text within tags, that add meaning to the document by describing elements like titles, paragraphs, lists, links etc. Let's see a basic HTML page, to understand what the Document Object Model is.

HTML page

```
<!doctype html>
<html>

<head>
    <meta charset="utf-8">
    <title>What is the DOM ?</title>
</head>

<body>
    <h1>DOM 101</h1>
    <p>Websraping is awsome !</p>
    <p>Here is my <a href="https://ksah.in">blog</a></p>
</body>
</html>
```

This HTML code is basically HTML content encapsulated inside other HTML content. The HTML hierarchy can be viewed as a tree. We can already see this hierarchy through the indentation in the HTML code. When your web browser parses this code, it will create a tree which is an object representation of the HTML document. It is called the Document Oject Model. Below is the internal tree structure inside Google Chrome inspector :

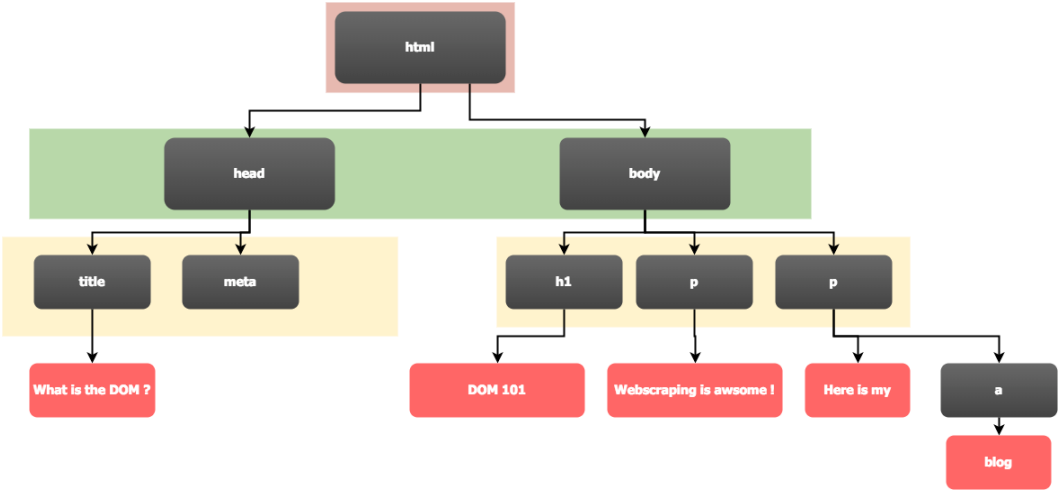


Chrome Inspector

On the left we can see the HTML tree, and on the right we have the Javascript

object representing the currently selected element (in this case, the `<p>` tag), with all its attributes. And here is the tree structure for this HTML code :

The important thing to remember is that the DOM you see in your browser, when you right click + inspect can be really different from the actual HTML that was sent. Maybe some Javascript code was executed and dynamically changed the DOM ! For example, when you scroll on your twitter account, a request is sent by your browser to fetch new tweets, and some Javascript code is dynamically adding those new tweets to the DOM.



Dom Diagram

The **root node** of this tree is the `<html>` tag. It contains two **children** : `<head>`and `<body>`. There are lots of node types in the DOM specification[4] but here is the most important one :

- ELEMENT_NODE the most important one, example : `<html>`, `<body>`, `<a>`, it can have a child node.
- TEXT_NODE like the red ones in the diagram, it **cannot** have any child node.

[4] https://www.w3.org/TR/dom/

The Document Object Model provides a programmatic way (API) to add, remove, modify, or attach any event to a HTML document using Javascript. The **Node** object has many interesting properties and methods to do this, like :

- `Node.childNodes` returns a list of all the children for this node.
- `Node.nodeType`returns an ***unsigned short*** representing the type of the node (1 for ELEMENT_NODE, 3 for TEXT_NODE ...).
- `Node.appendChild()` adds the child node argument to the last child of the current node.
- `Node.removeChild()` removes the child node argument from the current node.

You can see the full list here[5]. Now let's write some Javascript code to understand all of this :

First let's see how many child nodes our `<head>` element has, and show the list. To do so, we will write some Javascript code inside the Chrome console. The `document` object in Javascript is the owner of all other objects in the web page (including every DOM nodes.)
We want to make sure that we have two child nodes for our `head` element. It's simple :

How many childnodes ?

```
document.head.childNodes.length
```

And then show the list :

[5] https://developer.mozilla.org/en-US/docs/Web/API/Node

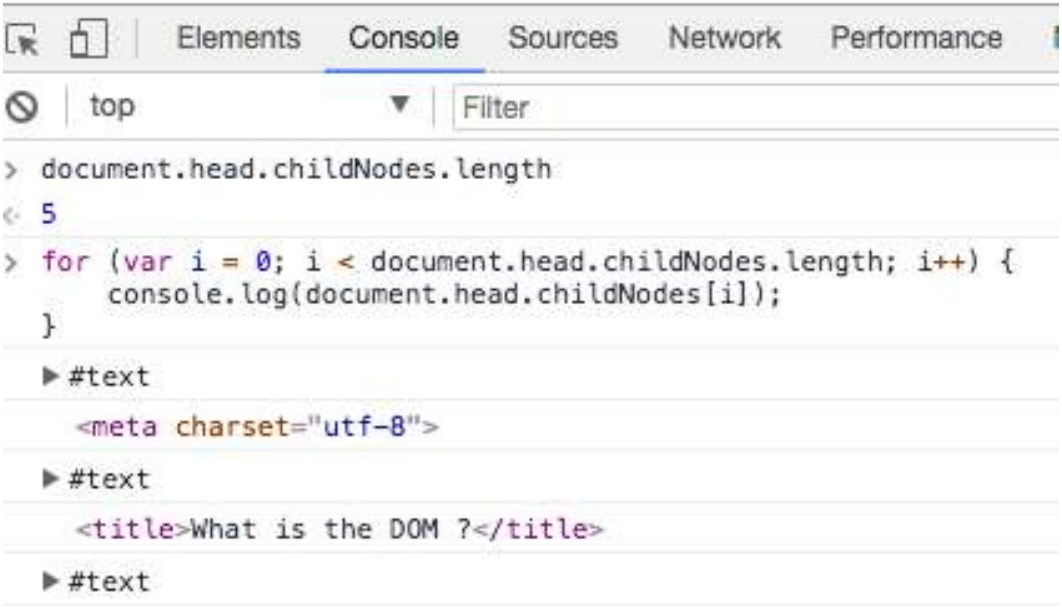
head's childnode list

```
for (var i = 0; i < document.head.childNodes.length; i++) {
    console.log(document.head.childNodes[i]);
}
```

DOM 101

Webscraping is awsome !

Here is my blog.



Javascript example

What an **unexpected result** ! It shows five nodes instead of the expected two.

We can see with the for loop that three text nodes were added. If you click on the this text nodes in the console, you will see that the text content is either a linebreak or tabulation (`\n` or `\t`). In most modern browsers, a text node is created for each whitespace outside a HTML tags.

This is something really important to remember when you use the DOM API. So the previous DOM diagram is not exactly true, in reality, **there are lots of text nodes containing whitespaces everywhere**. For more information on this subject, I suggest you to read this article from Mozilla : Whitespace in the DOM[6]

In the next chapters, we will not use directly the Javascript API to manipulate the DOM, but a similar API directly in Java. I think it is important to know how things works in Javascript before doing it with other languages.

Web extraction process

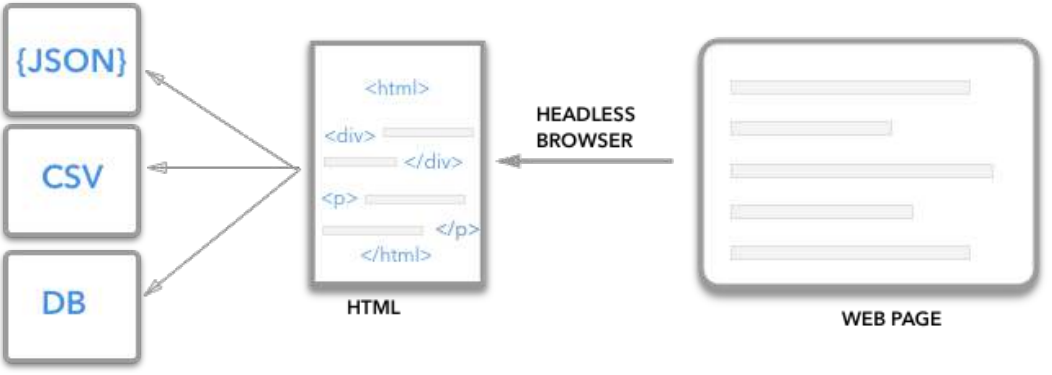
In order to go to a URL in your code, fetch the HTML code and parse it to extract the date we can use different things :

- Headless browser
- Do things more "manually" : Use an HTTP library to perform the GET request, then use a library like Jsoup[7] to parse the HTML and extract the data you want

Each option has its pros and cons. A headless browser is like a normal web browser, without the Graphical User Interface. It is often used for QA reasons, to perform automated testing on websites. There are lots of different headless browsers, like Headless Chrome[8], PhantomJS[9], HtmlUnit[10], we will see this

[6] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Whitespace_in_the_DOM
[7] https://jsoup.org/
[8] https://chromium.googlesource.com/chromium/src/+/lkgr/headless/README.md
[9] http://phantomjs.org/
[10] http://htmlunit.sourceforge.net/

later. The good thing about a headless browser is that it can take care of lots of things : Parsing the HTML, dealing with authentication cookies, fill in forms, execute Javascript functions, access iFrames… The drawback is that there is of course some overhead compared to using a plain HTTP library and a parsing library.



In the next three sections we will see how to select and extract data inside HTML pages, with Xpath, CSS selectors and regular expressions.

Xpath

Xpath is a technology that uses path expressions to select nodes or node-sets in an XML document (or HTML document). As with the Document Object Model, Xpath is a W3C standard since 1999. Even if Xpath is not a programming language in itself, it allows you to write expression that can access directly to a specific node, or a specific node set, without having to go through the entire HTML tree (or XML tree).

Entire books has been written on Xpath, and as I said before I don't have the pretention to explain everything in depth, this is an introduction to Xpath and we will see through real examples how you can use it for your web scraping needs.

We will use the following HTML document for the examples below:

HTML example

```
<!doctype html>
<html>

<head>
    <meta charset="utf-8">
    <title>Xpath 101</title>
</head>

<body>
   <div class="product">

        <header>
            <hgroup>
                <h1>Amazing product #1</h1>
                <h3>The best product ever made</h4>
            </hgroup>
        </header>

        <figure>
            <img src="http://lorempixel.com/400/200">
        </figure>

    <section>
        <p>Text text text</p>
        <details>
            <summary>Product Features</summary>
                <ul>
                    <li>Feature 1</li>
                    <li class="best-feature">Feature 2</li>
                    <li id="best-id">Feature 3</li>
                </ul>
        </details>
        <button>Buy Now</button>
    </section>
```

```
    </div>

</body>
</html>
```

First let's look at some Xpath vocabulary :

- In Xpath terminology, as with HTML, there are different types of nodes : root nodes, element nodes, attribute nodes, and so called **atomic values** which is a synonym for text nodes in an HTML document.
- Each element node has one **parent**. in this example, the `section` element is the parent of `p`, `details` and `button`.
- Element nodes can have any number of **children**. In our example, `li` elements are all children of the `ul` element.
- **Siblings** are nodes that have the same parents. `p`, `details` and `button` are siblings.
- **Ancestors** a node's parent and parent's parent…
- **Descendants** a node's children and children's children…

Xpath Syntax

There are different types of expressions to select a node in an HTML document, here are the most important ones :

Xpath Expression	Description
`nodename`	This is the simplest one, it select all nodes with this ***nodename***
`/`	Selects from the root node (useful for writing **absolute path**)
`//`	Selects nodes from the current node that matches
`.`	Selects the current node
`..`	Selects the current node's parent

Xpath Expression	Description
`@`	Selects attribute
`*`	Matches any node
`@*`	Matches any attribute node

You can also use **predicates** to find a node that contains a specific value. Predicate are always in square brackets : `[predicate]` Here are some examples :

Xpath Expression	Description
`//li[last()]`	Selects the last `li` element
`//div[@class='product']`	Selects all `div` elements that have the class attribute with the product value.
`//li[3]`	Selects the third `li` element (the index starts at 1)
`//div[@class='product']`	Selects all `div` elements that have the class attribute with the product value.

Now we will see some example of Xpath expressions. We can test XPath expressions inside Chrome Dev tools, so it is time to fire up Chrome. To do so, right click on the web page -> inspect and then `cmd + f` on a Mac or `ctrl + f` on other systems, then you can enter an Xpath expression, and the match will be highlighted in the Dev tool.

Amazing product #1

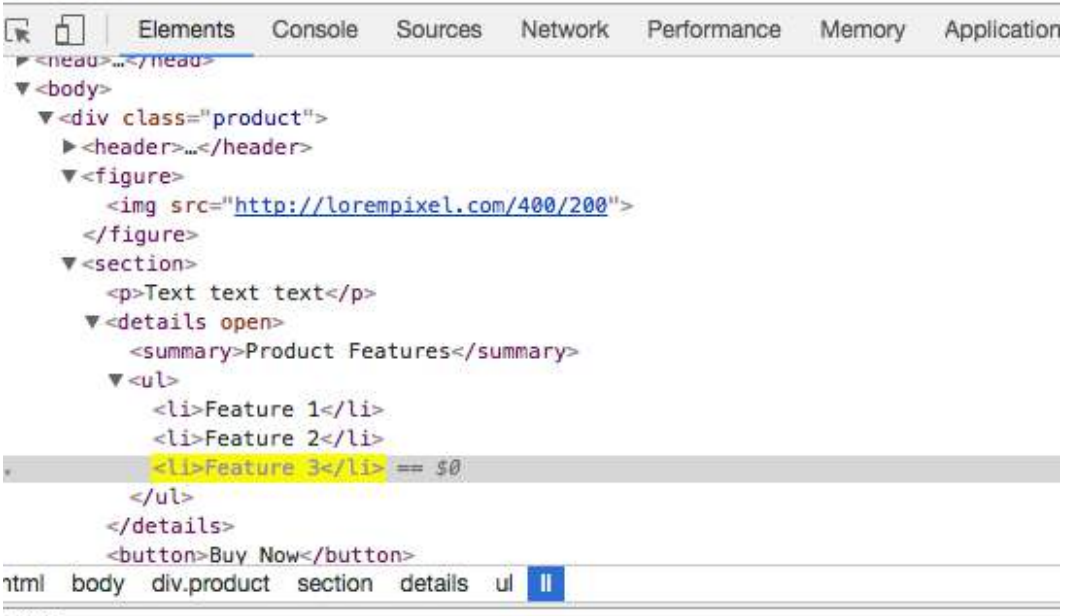
The best product ever made

Text text text

▼ Product Features

- Feature 1
- Feature 2
- Feature 3

Buy Now



In the dev tools, you can right click on any DOM node, and show its full Xpath expression, that you can later factorize. There is a lot more that we could discuss about Xpath, but it is out of this book's scope, I suggest you to read this great W3School tutorial[11] if you want to learn more.

In the next chapter we will see how to use Xpath expression inside our Java scraper to select HTML nodes containing the data we want to extract.

Regular Expression

A regular expression (RE, or Regex) is a search pattern for strings. With regex, you can search for a particular character/word inside a bigger body of text. For example you could identify all phone numbers inside a web page. You can also replace items, for example you could replace all upercase tag in a poorly formatted HTML by lowercase ones. You can also validate some inputs …

The pattern used by the regex is applied from left to right. Each source character is only used once. For example, this regex : `oco` will match the string `ococo` only once, because there is only one distinct sub-string that matches.

Here are some common matchings symbols, quantifiers and meta-characters :

Regex	Description
Hello World	Matches exactly "Hello World"
.	Matches any character
[]	Matches a range of character within the brackets, for example [a-h] matches any character between a and h. [1-5] matches any digit between 1 and 5
[^xyz]	Negation of the previous pattern, matches any character except x or y or z
*	Matches zero or more of the preceeding item
+	Matches one or more of the preceeding item

[11] https://www.w3schools.com/xml/xpath_intro.asp

Regex	Description
?	Matches zero or one of the preceeding item
{x}	Matches exactly x of the preceeding item
\d	Matches any digit
D	Matches any non digit
\s	Matches any whitespace character
S	Matches any non-whitespace character
(expression)	Capture the group matched inside the parenthesis

You may be wondering why it is important to know about regular expressions when doing web scraping ? We saw before that we could select HTML nodes with the DOM API, and Xpath, so why would we need regular expressions ? In an ideal semantic world[12], data is easily machine readable, the information is embedded inside relevant HTML element, with meaningful attributes.

But the real world is messy, you will often find huge amounts of text inside a `p` element. When you want to extract a specific data inside this huge text, for example a price, a date, a name... you will have to use regular expressions.

For example, regular expression can be useful when you have this kind of data :

```
<p>Price : 19.99$</p>
```

We could select this text node with an Xpath expression, and then use this kind a regex to extract the price :

```
^Price\s:\s(\d+\.\d{2})\$
```

This was a short introduction to the wonderful world of regular expressions, you should now be able to understand this :

[12] https://en.wikipedia.org/wiki/Semantic_Web

```
(?:[a-z0-9!#$%&'*+\/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+\/=?^_`{|}~-]+)*|"(\
?:[\x01-
\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e\x\
7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[\
a-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]\
|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\\
x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

I am kidding :) ! This one tries to validate an email address, according to RFC 2822[13]. There is a lot to learn about regular expressions, you can find more information in this great Princeton tutorial[14]

If you want to improve your regex skills or experiment, I suggest you to use this website : Regex101.com[15] . This website is really interesting because it allows you not only to test your regular expressions, but explains each step of the process.

[13] https://tools.ietf.org/html/rfc2822#section-3.4.1
[14] https://www.princeton.edu/~mlovett/reference/Regular-Expressions.pdf
[15] https://regex101.com/

Extracting the data you want

For our first exemple, we are going to fetch items from Hacker News, although they offer a nice API, let's pretend they don't.

Tools

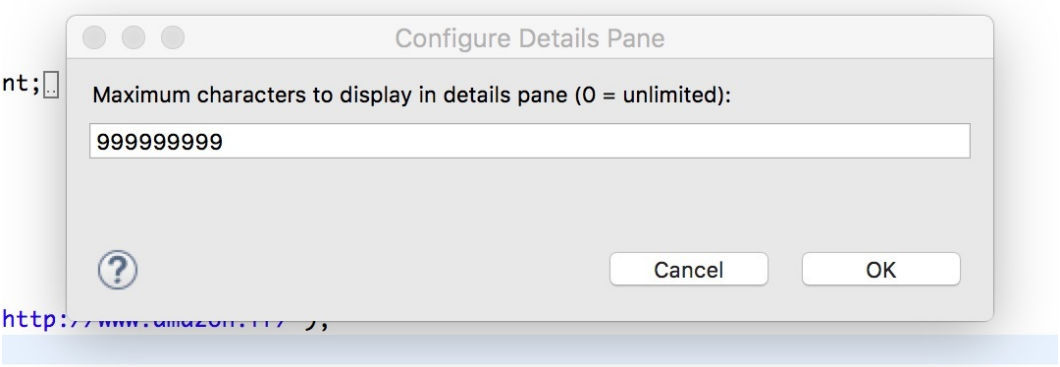
You will need Java 8 with HtmlUnit[16]. HtmlUnit is a Java headless browser, it is this library that will allow you to perform HTTP requests on websites, and parse the HTML content.

pom.xml

```
<dependency>
  <groupId>net.sourceforge.htmlunit</groupId>
  <artifactId>htmlunit</artifactId>
  <version>2.28</version>
</dependency>
```

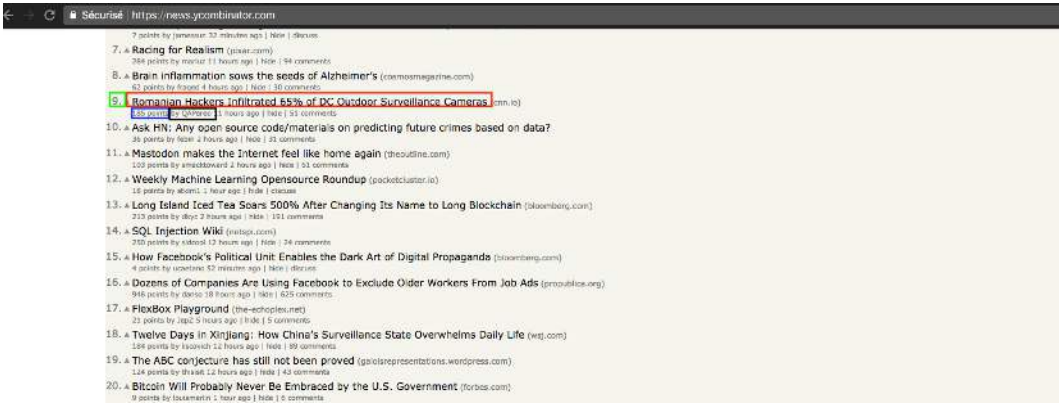
If you are using Eclipse, I suggest you configure the max length in the detail pane (when you click in the variables tab) so that you will see the entire HTML of your current page.

[16] http://htmlunit.sourceforge.net



Let's scrape Hacker News

The goal here is to collect the titles, number of upvotes, number of comments on the first page. We will see how to handle pagination later.



The base URL is : `https://news.ycombinator.com/`

Now you can open your favorite IDE, it is time to code. HtmlUnit needs a WebClient to make a request. There are many options (Proxy settings, browser, redirect enabled ...) We are going to disable Javascript since it's not required for our example, and disabling Javascript makes the page load faster in general (in this specific case, it does not matter). Then we perform a GET request to

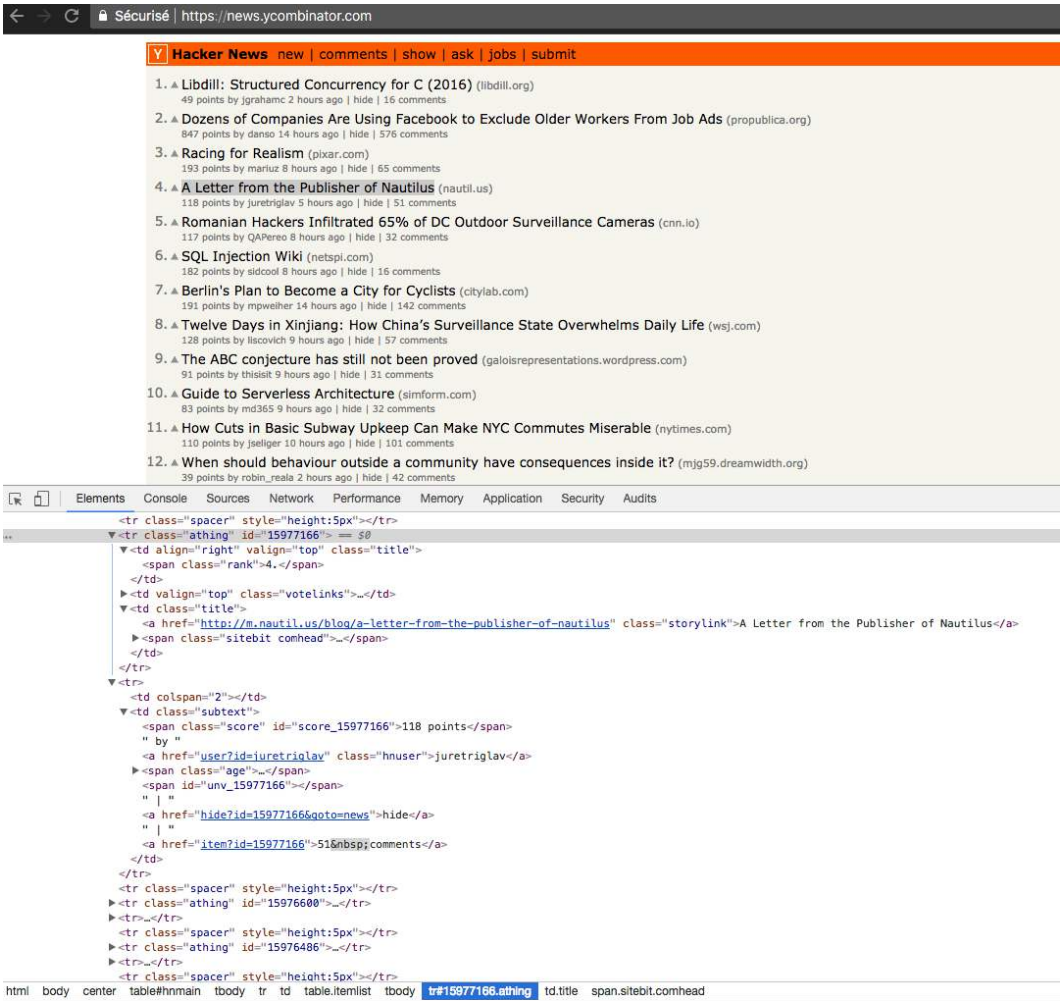
the hacker news's URL, and print the HTML content we received from the server.

Simple GET request

```
String baseUrl = "https://news.ycombinator.com/" ;
WebClient client = new WebClient();
client.getOptions().setCssEnabled(false);
client.getOptions().setJavaScriptEnabled(false);
try{
  HtmlPage page = client.getPage(baseUrl);
  System.out.println(page.asXml());
catch(Exception e){
  e.printStackTrace();
}
```

The HtmlPage object will contain the HTML code, you can access it with the `asXml()` method.

Now for each item, we are going to extract the title, URL, author etc. First let's take a look at what happens when you inspect a Hacker news post (right click on the element + inspect on Chrome)



With HtmlUnit you have several options to select an html tag :

- `getHtmlElementById(String id)`
- `getFirstByXPath(String Xpath)`
- `getByXPath(String XPath)` which returns a List
- Many more can be found in the HtmlUnit Documentation

Since there isn't any ID we could use, we have to use an Xpath expression to select the tags we want. We can see that for each item, we have two lines of

text. In the first line, there is the position, the title, the URL and the ID. And on the second, the score, author and comments. In the DOM structure, each text line is inside a `<tr>` tag, so the first thing we need to do is get the full `<tr class="athing">`list. Then we will iterate through this list, and for each item select title, the URL, author etc with a relative Xpath and then print the text content or value.

HackerNewsScraper.java

Selecting nodes with Xpath

```
HtmlPage page = client.getPage(baseUrl);
List<HtmlElement> itemList =  page.getByXPath("//tr[@class='athing']");
if(itemList.isEmpty()){
  System.out.println("No item found");
}else{
  for(HtmlElement htmlItem : itemList){
    int position = Integer.parseInt(
      ((HtmlElement) htmlItem.getFirstByXPath("./td/span"))
      .asText()
      .replace(".", ""));
    int id = Integer.parseInt(htmlItem.getAttribute("id"));
    String title =  ((HtmlElement) htmlItem
      .getFirstByXPath("./td[not(@valign='top')][@class='title']"))
      .asText();
    String url = ((HtmlAnchor) htmlItem
      .getFirstByXPath("./td[not(@valign='top')][@class='title']/a"))
      .getHrefAttribute();
    String author =  ((HtmlElement) htmlItem
      .getFirstByXPath("./following-sibling::tr/td[@class='subtext']/a[\
@class='hnuser']"))
      .asText();
    int score = Integer.parseInt(
      ((HtmlElement) htmlItem
      .getFirstByXPath("./following-sibling::tr/td[@class='subtext']/sp\
an[@class='score']"))
      .asText().replace(" points", ""));
```

```
    HackerNewsItem hnItem = new HackerNewsItem(title, url, author, scor\
e, position, id);

    ObjectMapper mapper = new ObjectMapper();
    String jsonString = mapper.writeValueAsString(hnItem) ;

    System.out.println(jsonString);
  }
}
```

Printing the result in your IDE is cool, but exporting to JSON or another well formated/reusable format is better. We will use JSON, with the Jackson[17] library, to map items in JSON format.

First we need a POJO (plain old java object) to represent the Hacker News items :

HackerNewsItem.java

POJO

```
public class HackerNewsItem {
        private String title;

        private String url ;
        private String author;
        private int score;
        private int position ;
        private int id ;

        public HackerNewsItem(String title, String url, String author, int sco\
re, int position, int id) {
                super();
                this.title = title;
                this.url = url;
```

[17] https://github.com/FasterXML/jackson

```
                this.author = author;
                this.score = score;
                this.position = position;
                this.id = id;
        }
//getters and setters
}
```

Then add the Jackson dependency to your pom.xml : **pom.xml**

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.7.0</version>
</dependency>
```

Now all we have to do is create an HackerNewsItem, set its attributes, and convert it to JSON string (or a file ...). Replace the old `System.out.println()` by this :

HackerNewsScraper.java

```
HackerNewsItem hnItem = new HackerNewsItem(title, url, author, score, p\
osition, id);
ObjectMapper mapper = new ObjectMapper();
String jsonString = mapper.writeValueAsString(hnItem) ;
// print or save to a file
System.out.println(jsonString);
```

And that's it. You should have a nice list of JSON formatted items.

Go further

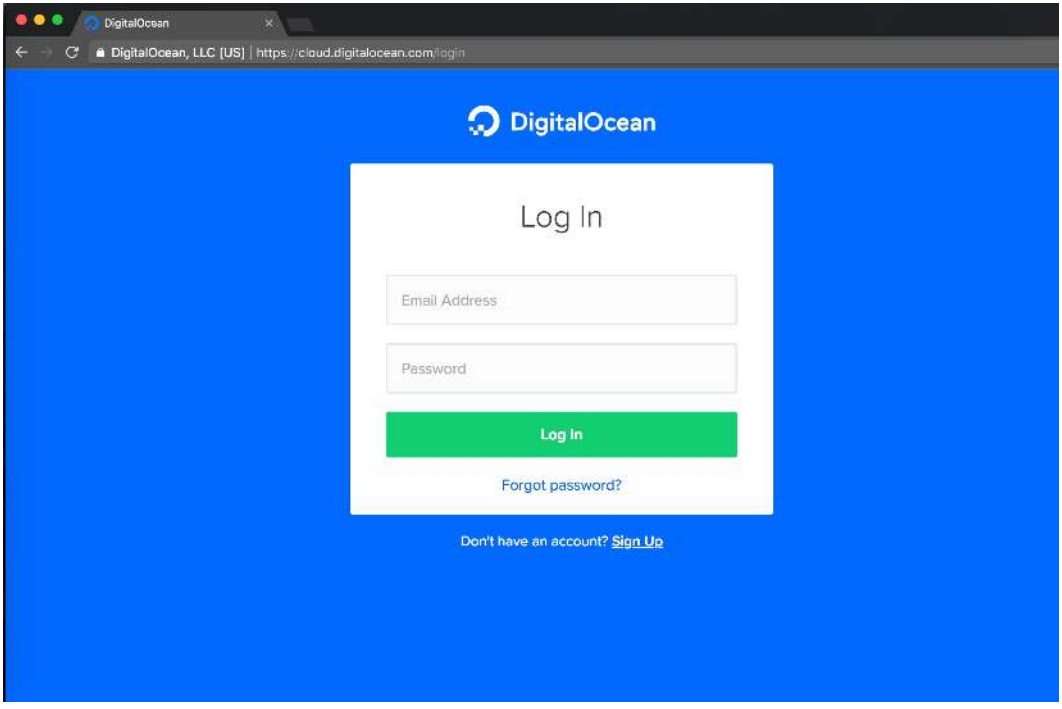
This example is not perfect, there are many things that can be done :

- Saving the result in a database.
- Handling pagination.
- Validating the extracted data using regular expressions instead of doing dirty `replace()`.

You can find the full code in this Github repository[18].

[18] https://github.com/ksahin/javawebscrapinghandbook_code

Handling forms



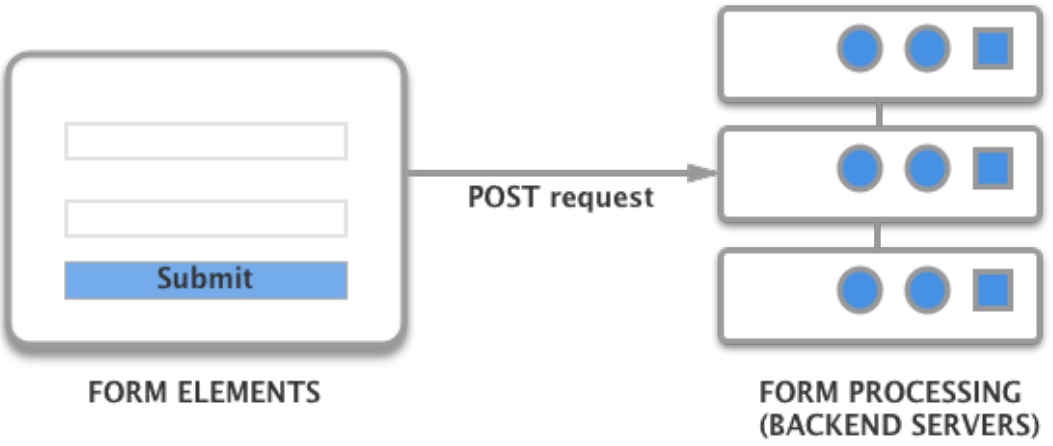
Typical login form from Digital Ocean website

In this chapter, we are going to see how to handle forms on the web. Knowing how to submit forms can be critical to extract information behind a login form, or to perform actions that require to be authenticated. Here are some examples of actions that require to submit a form :

- Create an account
- Authentication
- Post a comment on a blog
- Upload an image or a file

- Search and Filtering on a website
- Collecting a user email
- Collecting payment information from a user
- Any user-generated content !

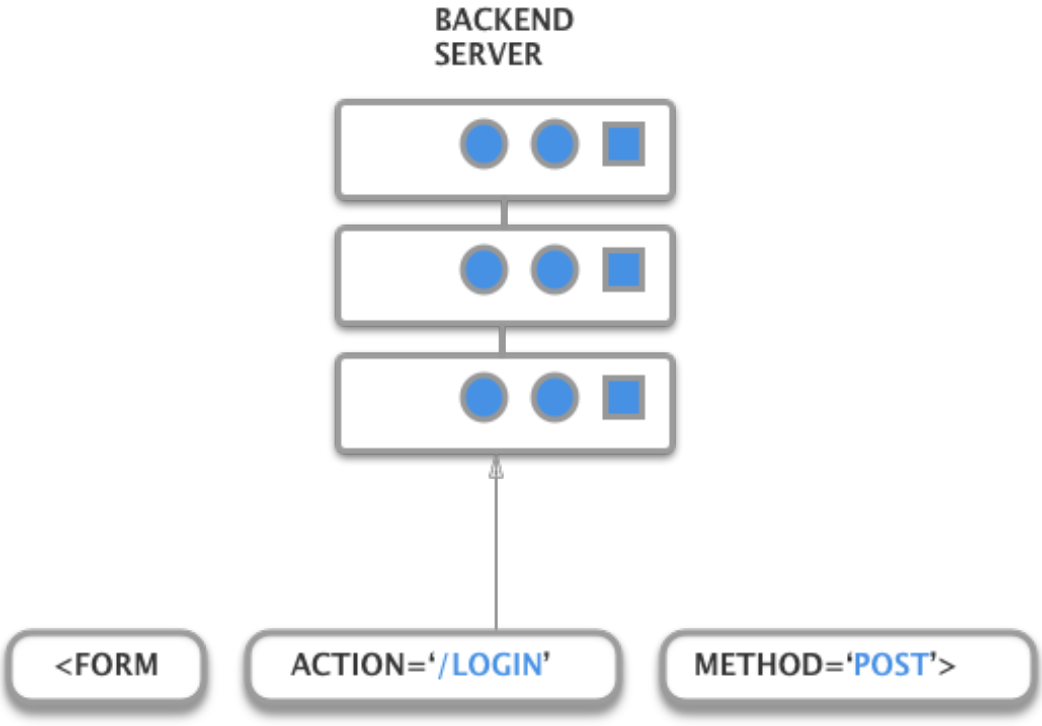
Form Theory



Form diagram

There are two parts of a functional HTML form: the user interface (defined by its HTML code and CSS) with different inputs and the backend code, which is going to process the different values the user entered, for example by storing it in a database, or charging the credit card in case of a payment form.

Form tag



Form diagram 2

HTML forms begins with a `<form>` tag. There are many attributes[19]. The most important ones are the `action` and `method` attribute.

The `action` attribute represents the URL where the HTTP request will be sent, and the `method` attribute specifies which HTTP method to use.

Generally, `POST` methods are used when you ***create or modify*** something, for example:

- Login forms
- Account creation
- Add a comment to a blog

[19] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form

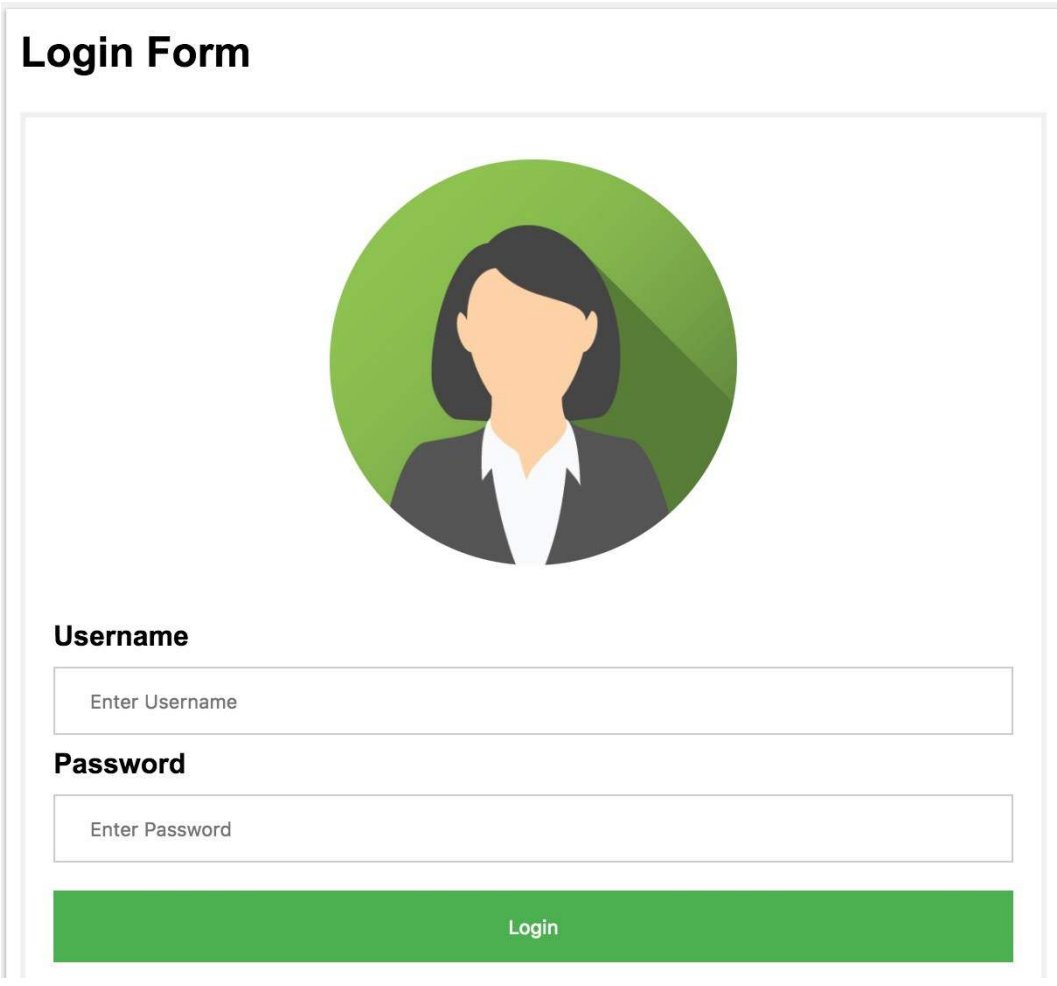
Form inputs

In order to collect user inputs, the `<input>` element is used. It is this element that makes the text field appear. The `<input>` element has different attributes :

- type: email, text, radio, file, date…
- name: the name associated with the value that will be sent
- many more[20]

Let's take an example of a typical login form :

[20] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input



Classic login form

And here is the corresponding HTML code (CSS code is not included):

Form's HTML code

```
<form action="login" method="POST">
  <div class="imgcontainer">
    <img src="img_avatar2.png" alt="Avatar" class="avatar">
  </div>

  <div class="container">
    <label for="uname"><b>Username</b></label>
    <input type="text" placeholder="Enter Username" name="uname" requir\
ed>

    <label for="psw"><b>Password</b></label>
    <input type="password" placeholder="Enter Password" name="psw" requ\
ired>

    <button type="submit">Login</button>

  </div>

</form>
```

When a user fills the form with his credentials, let's say `username`and `my_great_password` and click the submit button, the request sent by the browser will look like this :

Http response

```
POST /login HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
uname=username&psw=my_great_password
```

Cookies

After the POST request is made, if the credentials are valid the server will generally set ***cookies*** in the response headers, to allow the user to navigate.

This cookie is often named (the name depends on the technology/framework used by the website's backend):

- session_id
- session
- JSESSION_ID
- PHPSESSID

This cookie will be sent for each subsequent requests by the browser, and the website's backend will check its presence and validity to authorize requests. Cookies are not only used for login, but for lots of different use cases:

- Shopping carts
- User preferences
- Tracking user behavior

Cookies are small key/value pairs stored in the browser, or in an HTTP client, that looks like this:

```
cookie_name=cookie_value
```

An HTTP response that sets a cookie looks like this:

Http response

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: cookie_name=cookie_value
```

An HTTP request with a cookie looks like this:

Http request

```
GET /sample_page.html HTTP/1.1
Host: www.example.org
Cookie: cookie_name=cookie_value
```

A cookie can have different attributes :

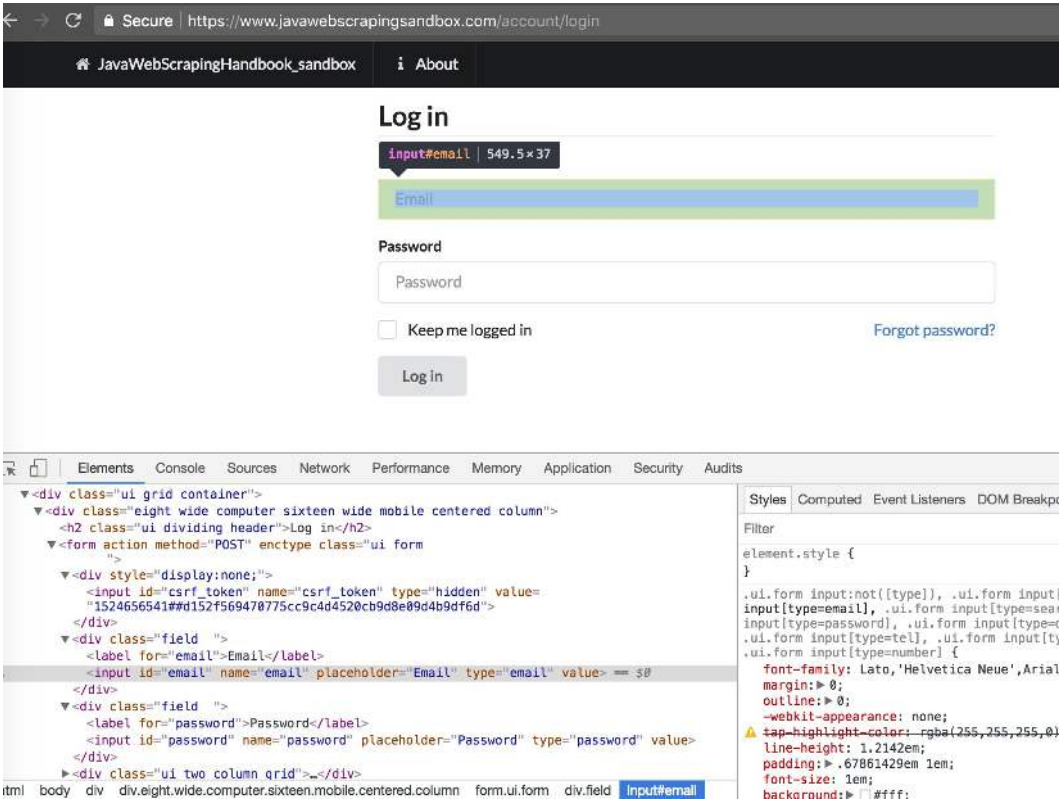
- ***Expires***: Expiration date, by default, cookies expire when the client closes the connection.
- ***Secure***: only sent to HTTPS URLs
- ***HttpOnly***: Inaccessible to Javascript `Document.cookie`, to prevent session hijacking and XSS attack[21]
- ***Domain***: Specifies which host is allowed to receive the cookie

Login forms

To study login forms, let me introduce you the website I made to apply some example in this book : https://www.javawebscrapingsandbox.com[22]

This website will serve for the rest of the book for lots of different examples, starting with the authentication example. Let's take a look at the login form HTML :

[21] https://developer.mozilla.org/en-US/docs/Glossary/Cross-site_scripting
[22] https://www.javawebscrapingsandbox.com



Basically, our scraper needs to :

- Get to the login page
- Fills the input with the right credentials
- Submit the form
- Check if there is an error message or if we are logged in.

There are two "difficult" thing here, the XPath expressions to select the different inputs, and how to submit the form.

To select the email input, it is quite simple, we have to select the first input inside a form, which name attribute is equal to email, so this XPath attribute should be ok: `//form//input[@name='email']`.

Same for the password input : `//form//input[@name='password']`

To submit the form, HtmlUnit provides a great method to select a form : `HtmlForm loginForm = input.getEnclosingForm();`.

Once you have the form object, you can generate the POST request for this form using: `loginForm.getWebRequest(null)` that's all you have to do :)

Let's take a look at the full code:

Login example

```
public class Authentication {

static final String baseUrl = "https://www.javawebscrapingsandbox.com/"\
 ;
static final String loginUrl = "account/login" ;
static final String email = "test@test.com" ;
static final String password = "test" ;


public static void main(String[] args) throws FailingHttpStatusCodeExce\
ption,
 MalformedURLException, IOException, InterruptedException {
        WebClient client = new WebClient();
        client.getOptions().setJavaScriptEnabled(true);
        client.getOptions().setCssEnabled(false);
        client.getOptions().setUseInsecureSSL(true);
        java.util.logging.Logger.getLogger("com.gargoylesoftware").setLevel(Le\
vel.OFF);

        // Get the login page
        HtmlPage page = client.getPage(String.
                format("%s%s", baseUrl, loginUrl)) ;

        // Select the email input
        HtmlInput inputEmail = page.getFirstByXPath(
                "//form//input[@name='email']");
```

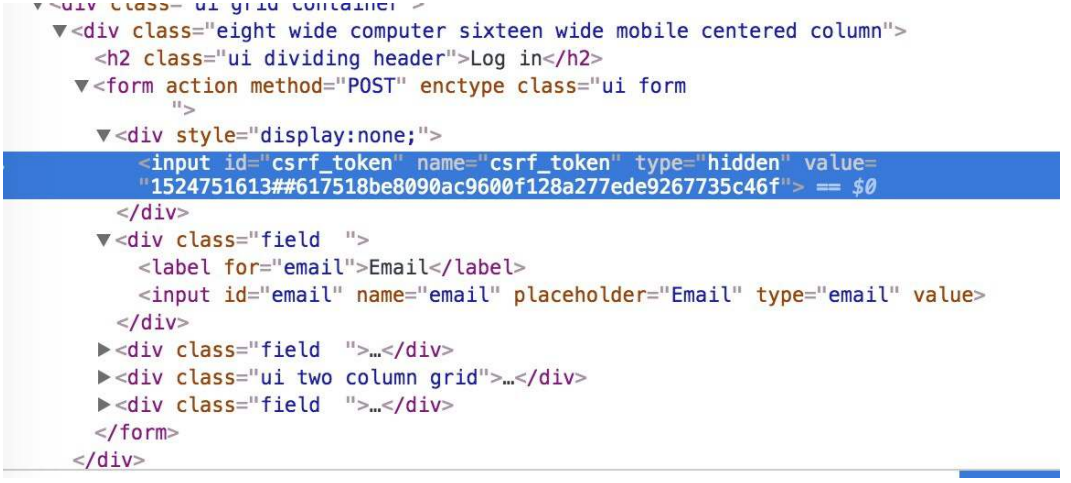
```
        // Select the password input
        HtmlInput inputPassword = page.getFirstByXPath(
                "//form//input[@name='password']");

        // Set the value for both inputs
        inputEmail.setValueAttribute(email);
        inputPassword.setValueAttribute(password);

        // Select the form
        HtmlForm loginForm = inputPassword.getEnclosingForm() ;

        // Generate the POST request with the form
        page = client.getPage(loginForm.getWebRequest(null));

        if(!page.asText().contains("You are now logged in")){
                System.err.println("Error: Authentication failed");
        }else{
                System.out.println("Success ! Logged in");
        }

}
}
```

This method works for almost every websites. Sometimes if the website uses a Javascript framework, HtmlUnit will not be able to execute the Javascript code (even with `setJavaScriptEnabled(true)`) and you will have to either 1) inspect the `HTTP POST` request in Chrome Dev Tools and recreate it, or use Headless Chrome which I will cover in the next chapter.

Let's take a look at the POST request created by HtmlUnit when we call `loginForm.getWebRequest(null)`. To view this, launch the main method in debug mode, and inspect the content (ctrl/cmd + MAJ + D in eclipse) :

```
WebRequest[<url="https://www.javawebscrapingsandbox.com/account/login",
POST, EncodingType[name=application/x-www-form-urlencoded],
[csrf_token=1524752332##6997dd9d5ed448484131add18b41a4263541b5c2,
 email=test@test.com,
 password=test],
 {Origin=https://www.javawebscrapingsandbox.com/account/login,
 Accept=text/html,application/xhtml+xml,application/xml;q=0.9,image/web\
p,image/apng,*/*;q=0.8,
 Cache-Control=max-age=0,
 Referer=https://www.javawebscrapingsandbox.com/account/login,
 Accept-Encoding=gzip, deflate}, null>]
```

We have a lot going one here. You can see that instead of just having two parameters sent to the server (email and password), we also have a `csrf_token` parameter, and its value changes everytime we submit the form. This parameter is hidden, as you can see in the form's HTML :



CSRF token

CSRF stands for Cross Site Request Forgery. The token is generated by the server and is required in every form submissions / POST requests. Almost every website use this mechanism to prevent CSRF attack. You can learn

more about CSRF attack here[23]. Now let's create our own POST request with HtmlUnit.

The first thing we need is to create a `WebRequest` object. Then we need to set the URL, the HTTP method, headers, and parameters. Adding request header to a `WebRequest` object is quite simple, all you need to to is to call the `setAdditionalHeader` method. Adding parameters to your request must me done with the `setRequestParameters`method, which takes a list of `NameValuePair`. As discussed earlier, we have to add the `csrf_token` to the parameters, which can be selected easily with this XPath expression : `//form//input[@name='csrf_token']`

Forging the request manually

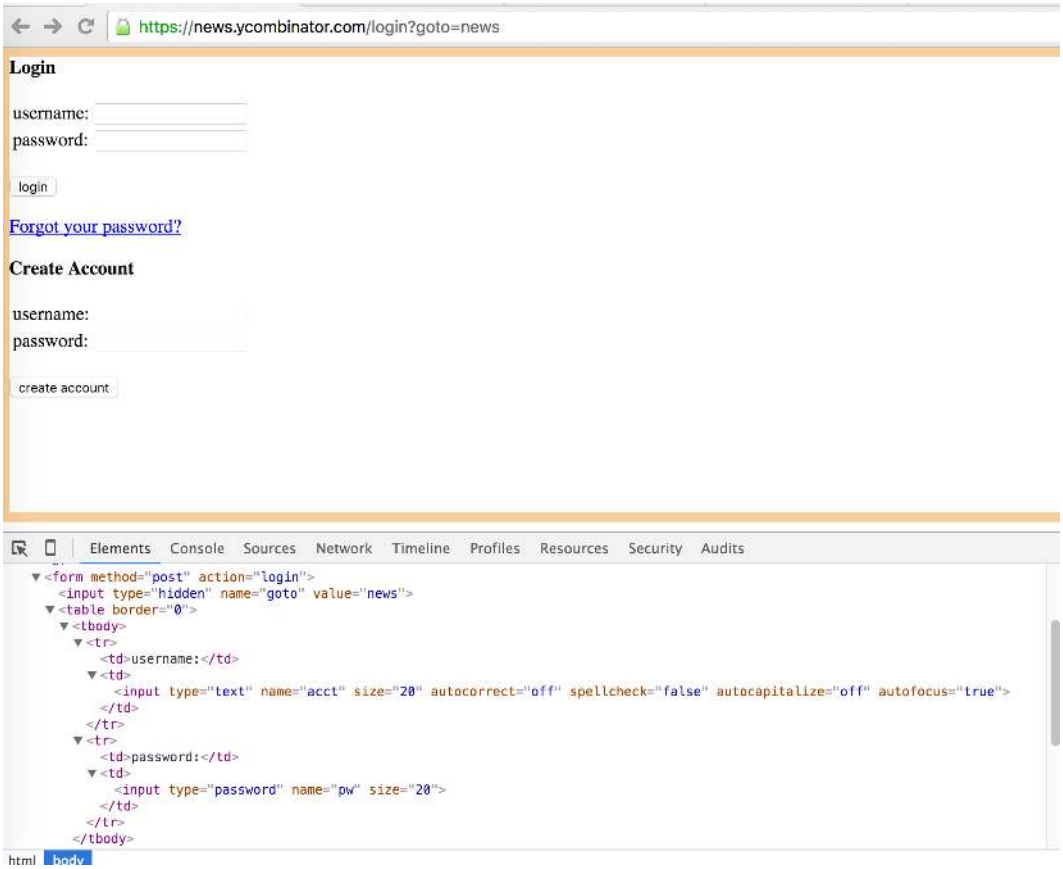
```
HtmlInput csrfToken = page.getFirstByXPath("//form//input[@name='csrf_t\
oken']") ;
WebRequest request = new WebRequest(
                new URL("http://www.javawebscrapingsandbox.com/account/login"), Htt
ethod.POST);
List<NameValuePair> params = new ArrayList<NameValuePair>();
params.add(new NameValuePair("csrf_token", csrfToken.getValueAttribute(\
)));
params.add(new NameValuePair("email", email));
params.add(new NameValuePair("password", password));

request.setRequestParameters(params);
request.setAdditionalHeader("Content-Type", "application/x-www-form-url\
encoded");
request.setAdditionalHeader("Accept-Encoding", "gzip, deflate");

page = client.getPage(request);
```

[23] https://en.wikipedia.org/wiki/Cross-site_request_forgery

Case study: Hacker News authentication

Let's say you want to create a bot that logs into a website (to submit a link or perform an action that requires being authenticated) :

Here is the login form and the associated DOM :



Now we can implement the login algorithm

Login algorithm

```
public static WebClient autoLogin(String loginUrl, String login, String\
 password)
throws FailingHttpStatusCodeException, MalformedURLException, IOExcepti\
on{
        WebClient client = new WebClient();
        client.getOptions().setCssEnabled(false);
        client.getOptions().setJavaScriptEnabled(false);

        HtmlPage page = client.getPage(loginUrl);

        HtmlInput inputPassword = page.getFirstByXPath("
                //input[@type='password']");
        //The first preceding input that is not hidden
        HtmlInput inputLogin = inputPassword.getFirstByXPath("
                .//preceding::input[not(@type='hidden')]");

        inputLogin.setValueAttribute(login);
        inputPassword.setValueAttribute(password);

        //get the enclosing form
        HtmlForm loginForm = inputPassword.getEnclosingForm() ;

        //submit the form
        page = client.getPage(loginForm.getWebRequest(null));

        //returns the cookie filled client :)
        return client;
}
```

Then the main method, which :

- calls `autoLogin` with the right parameters
- Go to `https://news.ycombinator.com`

- Check the logout link presence to verify we're logged
- Prints the cookie to the console

Hacker News login

```
public static void main(String[] args) {

        String baseUrl = "https://news.ycombinator.com" ;
        String loginUrl = baseUrl + "/login?goto=news" ;
        String login = "login";
        String password = "password" ;

        try {
                System.out.println("Starting autoLogin on " + loginUrl);
                WebClient client = autoLogin(loginUrl, login, password);
                HtmlPage page = client.getPage(baseUrl) ;

                HtmlAnchor logoutLink = page
                        .getFirstByXPath(String.format(
                                "//a[@href='user?id=%s']", login)) ;

                if(logoutLink != null ){
                        System.out.println("Successfuly logged in !");
                        // printing the cookies
                        for(Cookie cookie : client.
                                        getCookieManager().getCookies()){
                                        System.out.println(cookie.toString());
                        }
                }else{
                        System.err.println("Wrong credentials");
                }

        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

You can find the code in this Github repo[24]

Go further

There are many cases where this method will not work: Amazon, DropBox… and all other two-steps/captcha-protected login forms.

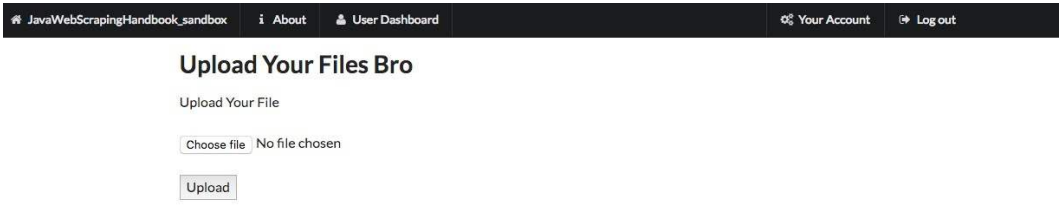
Things that can be improved with this code :

- Handle the check for the logout link inside `autoLogin`
- Check for `null` inputs/form and throw an appropriate exception

File Upload

File upload is not something often used in web scraping. But it can be interesting to know how to upload files, for example if you want to test your own website or to automate some tasks on websites.

There is nothing complicated, here is a little form on the sandbox website[25] (you need to be authenticated):



Here is the HTML code for the form :

[24] https://github.com/ksahin/introWebScraping
[25] https://www.javawebscrapingsandbox.com/upload_file

Form example

```
<div class="ui text container">
        <h1>Upload Your Files Bro</h1>

        <form action="/upload_file" method="POST" enctype="multipart/form-data\
">

                <label for="user_file">Upload Your File</label>
                <br></br>
                <input type="file" name="user_file">
                <br></br>
                <button type="submit">Upload</button>

        </form>
</div>
```

As usual, the goal here is to select the form, if there is a `name` attribute you can use the method `getFormByBame()` but in this case there isn't, so we will use a good old XPath expression. Then we have to select the input for the file and set our file name to this input. Note that you have to be authenticated to post this form.

File upload example

```
fileName = "file.png" ;
page = client.getPage(baseUrl + "upload_file") ;
HtmlForm uploadFileForm = page.getFirstByXPath("//form[@action='/upload\
_file']");
HtmlFileInput fileInput = uploadFileForm.getInputByName("user_file");

fileInput.setValueAttribute(fileName);
fileInput.setContentType("image/png");

HtmlElement button = page.getFirstByXPath("//button");
page = button.click();
```

```
if(page.asText().contains("Your file was successful uploaded")){
        System.out.println("File successfully uploaded");
}else{
        System.out.println("Error uploading the file");
}
```

Other forms

Search Forms

Another common need when doing web scraping is to submit search forms. Websites having a large database, like marketplaces often provide a search form to look for a specific set of items.
There is generally three different ways search forms are implemented :

- When you submit the form, a POST request is sent to the server
- A GET request is sent with query parameters
- An AJAX call is made to the server

As an example, I've set up a search form on the sandbox website :

Search Form

It is a simple form, there is nothing complicated. As usual, we have to select the inputs field, fill it with the values we want, and submit the form. We could also reproduce the POST request manually, as we saw in the beginning of the chapter. When the server sends the response back, I chose to loop over the result, and print it in the console (The whole code is available in the repo as usual.)

Search Form example

```
HtmlPage page = client.getPage(baseUrl + "product/search");

HtmlInput minPrice = page.getHtmlElementById("min_price");
HtmlInput maxPrice = page.getHtmlElementById("max_price");

// set the min/max values
minPrice.setValueAttribute(MINPRICE);
maxPrice.setValueAttribute(MAXPRICE);
HtmlForm form = minPrice.getEnclosingForm();

page = client.getPage(form.getWebRequest(null));

HtmlTable table = page.getFirstByXPath("//table");
for(HtmlTableRow elem : table.getBodies().get(0).getRows()){
        System.out.println(String.format("Name : %s Price: %s", elem.getCell(0\
).asText(), elem.getCell(2).asText()));
}
```

And here is the result:

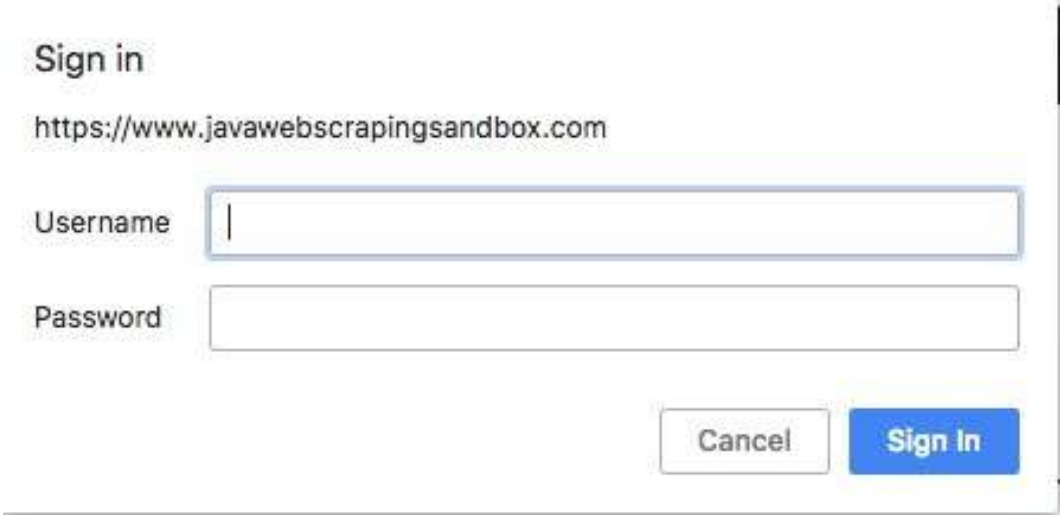
Ouput

```
Name : ClosetMaid 1937440 SuiteS Price: 319.89 $
Name : RWS Model 34 .22 Caliber Price: 314.97 $
Name : Neato Botvac D5 Connected Price: 549.00 $
Name : Junghans Men's 'Max Bill' Price: 495.00 $
```

Basic Authentication

In the 90s, basic authentication was everywhere. Nowadays, it's rare, but you can still find it on corporate websites. It's one of the simplest forms of authentication. The server will check the credentials in the `Authorization` header sent by the client, or issue a prompt in case of a web browser.

If the credentials are not correct, the server will respond with a `401 (Unauthorized)` response status.



Here is the URL on the sandbox website : https://www.javawebscrapingsandbox.com/basi
auth

The Username is : `basic`

The password is : `auth`

It's really simple to use basic auth with HtmlUnit, all you have to do is format your URL with this pattern : `https://username:password@www.example.com`

Basic auth example

```
HtmlPage page = client.getPage(String.format("https://%s:%s@www.javaweb\
scrapingsandbox.com/basic_auth", username, password));
System.out.println(page.asText());
```

Dealing with Javascript

Dealing with a website that uses lots of Javascript to render their content can be tricky. These days, more and more sites are using frameworks like Angular, React, Vue.js for their frontend. These frontend frameworks are complicated to deal with because there are often using the newest features of the HTML5 API, and HtmlUnit and other headless browsers do not commonly support these features.

So basically the problem that you will encounter is that your headless browser will download the HTML code, and the Javascript code, but will not be able to execute the full Javascript code, and the webpage will not be totally rendered.

There are some solutions to these problems. The first one is to use a better headless browser. And the second one is to inspect the API calls that are made by the Javascript frontend and to reproduce them.

Javascript 101

Javascript is an interpreted scripting language. It's more and more used to build "Web applications" and "Single Page Applications".

The goal of this chapter is not to teach you Javascript, to be honest, I'm a terrible Javascript developer, but I want you to understand how it is used on the web, with some examples.

The Javascript syntax is similar to C or Java, supporting common data types, like Boolean, Number, String, Arrays, Object… Javascript is loosely typed, meaning there is no need to declare the data type explicitly.

Here is some code examples:

Plus one function

```
function plusOne(number) {
    return number + 1 ;
}
var a = 4 ;
var b = plusOne(a) ;
console.log(b);
// will print 5 in the console
```

As we saw in chapter 2, Javascript is mainly used on the web to modify the DOM dynamically and perform HTTP requests. Here is a sample code that use a stock API to retrieve the latest Apple stock price when clicking a button:

Apple stock price vanilla Javascript

```
<!DOCTYPE html>
<html>
<head>
    <script>
    function refreshAppleStock(){
      fetch("https://api.iextrading.com/1.0/stock/aapl/batch?types=quot\
e,news,chart&range=1m&last=10")
        .then(function(response){
          return response.json();
        }).then(function(data){
           document.getElementById('my_cell').innerHTML = '$' + data.qu\
ote.latestPrice ;
        });
    }
  </script>
</head>
<body>
  <div>
    <h2>Apple stock price:</h2>
    <div id="my_cell">
    </div>
```

```
    <button id="refresh" onclick="refreshAppleStock()">Refresh</button>
  </div>
</body>
</html>
```

Jquery

jQuery[26] is one of the most used Javascript libraries. It's really old, the first version was written in 2006, and it is used for lots of things such as:

- DOM manipulation
- AJAX calls
- Event handling
- Animation
- Plugins (Datepicker etc.)

Here is a jQuery version of the same apple stock code (you can note that the jQuery version is not necessarily clearer than the vanilla Javascript one...) :

Apple stock price

```
<!DOCTYPE html>
<html>
<head>
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquer\
y.min.js"></script>
    <script>
    function refreshAppleStock(){
      $.get("https://api.iextrading.com/1.0/stock/aapl/batch?types=quot\
e,news,chart&range=1m&last=10", function(data, status) {
          $('#my_cell').html('$' + data.quote.latestPrice);
      });
```

[26] https://jquery.com/

```
    }

    $(document).ready(function(){
      $("#refresh").click(function(){
          refreshAppleStock();
      });
    });

  </script>
</head>
<body>
  <div>
    <h2>Apple stock price:</h2>
    <div id="my_cell">
    </div>
    <button id="refresh">Refresh</button>
  </div>

</body>
</html>
```

If you want to know more about Javascript, I suggest you this excellent book: Eloquent Javascript[27]

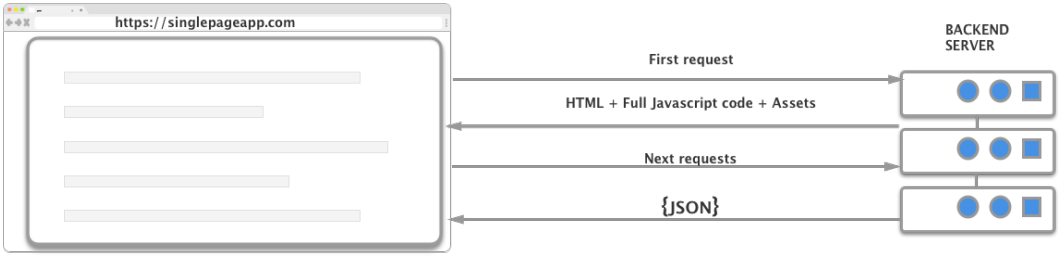
Modern Javascript frameworks

There are several problems with jQuery. It is extremely difficult to write clean/maintainable code with it as the Javascript application growths. Most of the time, the codebase becomes full of "glue code", and you have to be careful with each id or class name changes. The other big concern is that it can be complicated to implement data-binding between Javascript models and the DOM.

[27] https://eloquentjavascript.net/

The other problem with the traditional server-side rendering is that it can be inefficient. Let's say you are browsing a table on an old website. When you request the next page, the server is going to render the entire HTML page, with all the assets and send it back to your browser. With an SPA, only one HTTP request would have been made, the server would have sent back a JSON containing the data, and the Javascript framework would have filled the HTML model it already has with the new values!

Here is a diagram to better understand how it works :



Single Page Application

In theory, SPAs are faster, have better scalability and lots of other benefits compared to server-side rendering.

That's why Javascript frameworks were created. There are lots of different Javascript frameworks :

- AngularJS[28] made by Google
- EmberJS[29] by Yehuda Katz (ex Jquery team)
- ReactJS[30] by Facebook
- VueJS[31] by Evan You (ex AngularJS team)

These frameworks are often used to create so-called "Single Page Applications". There are lots of differences between these, but it is out of this book scope to dive into it.

[28] https://angularjs.org/
[29] https://www.emberjs.com/
[30] https://reactjs.org/
[31] https://vuejs.org/

It can be challenging to scrape these SPAs because there are often lots of Ajax calls and websockets[32] connections involved. If performance is an issue, you should always try to reproduce the Javascript code, meaning manually inspecting all the network calls with your browser inspector, and replicating the AJAX calls containing interesting data.

So depending on what you want to do, there are several ways to scrape these websites. For example, if you need to take a screenshot, you will need a real browser, capable of interpreting and executing all the Javascript code, that is what the next part is about.

Headless Chrome

We are going to introduce a new feature from Chrome, the ***headless*** mode. There was a rumor going around, that Google used a special version of Chrome for their crawling needs. I don't know if this is true, but Google launched the headless mode for Chrome with Chrome 59 several months ago.

PhantomJS was the leader in this space, it was (and still is) heavy used for browser automation and testing. After hearing the news about Headless Chrome, the PhantomJS maintainer said that he was stepping down as maintainer, because I quote *"Google Chrome is faster and more stable than PhantomJS [...]"* It looks like Chrome headless is becoming the way to go when it comes to browser automation and dealing with Javascript-heavy websites.

HtmlUnit, PhantomJS, and the other headless browsers are very useful tools, the problem is they are not as stable as Chrome, and sometimes you will encounter Javascript errors that would not have happened with Chrome.

Prerequisites

- Google Chrome > 59

[32] https://en.wikipedia.org/wiki/WebSocket

- Chromedriver[33]
- Selenium
- In your ***pom.xml*** add a recent version of Selenium :

pom.xml

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.8.1</version>
</dependency>
```

If you don't have Google Chrome installed, you can download it here[34] To install Chromedriver you can use brew on MacOS :

```
brew install chromedriver
```

You can also install Chrome driver with npm:

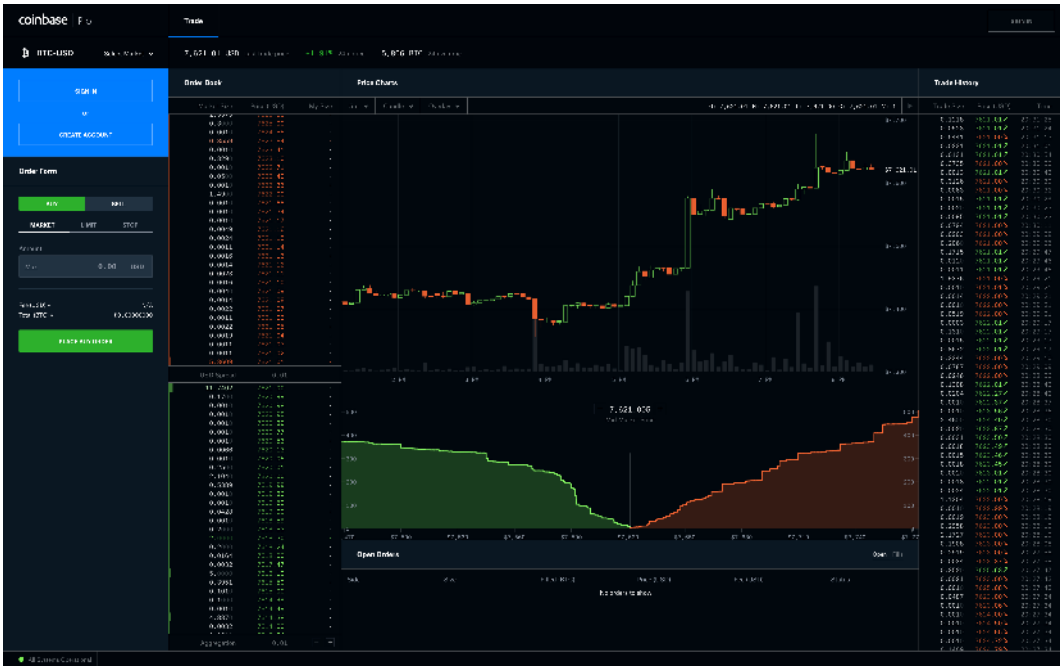
```
npm install chromedriver
```

Or download it using the link below. There are a lot of versions, I suggest you to use the last version of Chrome and chromedriver.

Let's take a screenshot of a real SPA

We are going to take a screenshot of the Coinbase[35] website, which is a cryptocurrency exchange, made with React framework, and full of API calls and websocket !

[33] https://sites.google.com/a/chromium.org/chromedriver/downloads
[34] https://www.google.com/chrome/browser/desktop/index.html
[35] https://pro.coinbase.com/trade/BTC-USD



Coinbase screenshot

We are going to manipulate Chrome in headless mode using the Selenium API. The first thing we have to do is to create a WebDriver object, whose role is similar the toe `WebClient` object with HtmlUnit, and set the chromedriver path and some arguments :

Chrome driver

```
// Init chromedriver
String chromeDriverPath = "/Path/To/Chromedriver" ;
System.setProperty("webdriver.chrome.driver", chromeDriverPath);
ChromeOptions options = new ChromeOptions();
options.addArguments("--headless", "--disable-gpu", "--window-size=1920\
,1200","--ignore-certificate-errors");
WebDriver driver = new ChromeDriver(options);
```

The `--disable-gpu` option is needed on Windows systems, according to the

documentation[36] Chromedriver should automatically find the Google Chrome executable path, if you have a special installation, or if you want to use a different version of Chrome, you can do it with :

```
options.setBinary("/Path/to/specific/version/of/Google Chrome");
```

If you want to learn more about the different options, here is the Chromedriver documentation[37]

The next step is to perform a GET request to the Coinbase website, wait for the page to load and then take a screenshot.

We have done this in a previous article, here is the full code :

GDAX Screenshot example

```
public class ChromeHeadlessTest {
    private static String userName = "" ;
    private static String password = "" ;

    public static void main(String[] args) throws IOException{
              String chromeDriverPath = "/path/to/chromedriver" ;
                System.setProperty("webdriver.chrome.driver", chromeDriverPath);
                ChromeOptions options = new ChromeOptions();
        options.addArguments("--headless", "--disable-gpu", "--window-s\
ize=1920,1200","--ignore-certificate-errors", "--silent");
        WebDriver driver = new ChromeDriver(options);

        // Get the login page
        driver.get("https://pro.coinbase.com/trade/BTC-USD");
        Thread.sleep(10000);

        // Take a screenshot of the current page
        File screenshot = ((TakesScreenshot) driver).getScreenshotAs(Ou\
tputType.FILE);
```

[36] https://developers.google.com/web/updates/2017/04/headless-chrome
[37] https://sites.google.com/a/chromium.org/chromedriver/capabilities

```
        FileUtils.copyFile(screenshot, new File("screenshot.png"));
        driver.close();
            driver.quit();
    }
}
```

You should now have a nice screenshot of the Coinbase homepage.

Several things are going on here. The line with the `Thread.sleep(10000)` allows the browser to wait for the entire page to load. This is not necessarily the best method, because maybe we are waiting too long, or too little depending on multiple factors (your own internet connection, the target website speed etc.).

This is a common problem when scraping SPAs, and one way I like to solve this is by using the `WebDriverWait` object:

WebDriverWait usage

```
WebDriverWait wait = new WebDriverWait(driver, 20);
wait.until(ExpectedConditions.
   presenceOfElementLocated(By.xpath("/path/to/element")));
```

There are lots of different `ExpectedConditions` you can find the documentation here[38] I often use `ExpectedConditions.visibilityOfAllElementsLocatedBy(locator)` because the element can be present, but hidden until the asynchronous HTTP call is completed.

This was a brief introduction to headless Chrome and Selenium, now let's see some common and useful Selenium objects and methods!

[38] https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/support/ui/ExpectedConditions.html

Selenium API

In the Selenium API, almost everything is based around two interfaces : * `WebDriver` which is the HTTP client * `WebElement` which represents a DOM object

The WebDriver[39] can be initialized with almost every browser, and with different options (and of course, browser-specific options) such as the window size, the logs file's path etc.

Here are some useful methods :

Method	Description
`driver.get(URL)`	performs a GET request to the specified URL
`driver.getCurrentUrl()`	returns the current URL
`driver.getPageSource()`	returns the full HTML code for the current page
`driver.navigate().back()`	navigate one step back in the history, works with forward too
`driver.switchTo().frame(frameElement)`	switch to the specified iFrame
`driver.manage().getCookies()`	returns all cookies, lots of other cookie related methods exists
`driver.quit()`	quits the driver, and closes all associated windows
`driver.findElement(by)`	returns a `WebElement` located by the specified locator

The `findElement()` method is one of the most interesting for our scraping needs.

You can locate elements with different ways :

- `findElement(By.Xpath('/xpath/expression'))`

[39] https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/WebDriver.html

- `findElement(By.className(className))`
- `findElement(By.cssSelector(selector))`

Once you have a WebElement object, there are several useful methods you can use:

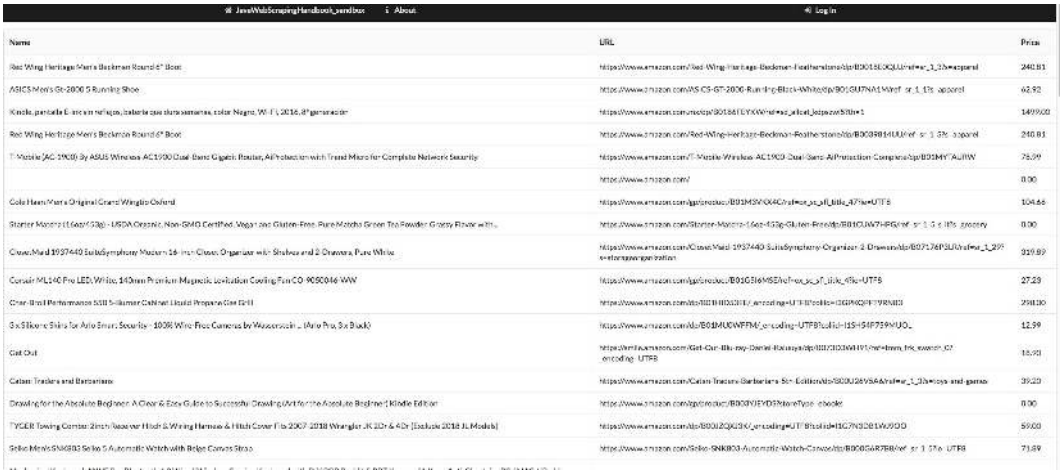
Method	Description
`findElement(By)`	you can again use this method, using a relative selector
`click()`	clicks on the element, like a button
`getText()`	returns the inner text (meaning the text that is inside the element)
`sendKeys('some string')`	enters some text in an input field
`getAttribute('href')`	returns the attribute's value(in this example, the href attribute)

Infinite scroll

Infinite scroll is heavily used in social websites, news websites, or when dealing with a lot of information. We are going to see three different ways to scrape infinite scroll.

I've set up a basic infinite scroll here: Infinite Scroll[40] Basically, each time you scroll ***near*** the bottom of the page, an AJAX call is made to an API and more elements are added to the table.

[40] https://www.javawebscrapingsandbox.com/product/infinite_scroll



Infinite table

Scrolling to the bottom

The first way of scraping this page is to make our headless browser scroll to the bottom of the page. There is a nice method we can use on the Window[41] object, called ScrollTo()[42]. It is really simple to use, you give it an X and Y coordinate, and it will scroll to that location.

In order to execute this Javascript code, we are going to use a Javascript executor. It allows us to execute any Javascript code in the context of the current web page (or more specifically, the current tab). It means we have access to every Javascript function and variables defined in the current page.

In this example, note that the webpage is showing a fixed 20 rows in the table on the first load. So if our browser window is too big, we won't be able to scroll. This "mistake" was made on purpose. To deal with this, we must tell our headless Chrome instance to open with a small window size !

[41] https://developer.mozilla.org/en-US/docs/Web/API/Window
[42] https://developer.mozilla.org/en-US/docs/Web/API/Window/scrollTo

Infinite scroll with headless Chrome

```
String chromeDriverPath = "/path/to/chromedriver" ;
System.setProperty("webdriver.chrome.driver", chromeDriverPath);
ChromeOptions options = new ChromeOptions();
options.addArguments("--headless" ,"--disable-gpu", "--ignore-certifica\
te-errors", "--silent");
// REALLY important option here, you must specify a small window size t\
o be able to scroll
options.addArguments("window-size=600,400");

WebDriver driver = new ChromeDriver(options);
JavascriptExecutor js = (JavascriptExecutor) driver;
int pageNumber = 5 ;

driver.get("https://www.javawebscrapingsandbox.com/product/infinite_scr\
oll");
for(int i = 0; i < pageNumber; i++){
    js.executeScript("window.scrollTo(0, document.body.scrollHeight);");
    // There are better ways to wait, like using the WebDriverWait obje\
ct
    Thread.sleep(1200);
}
List<WebElement> rows = driver.findElements(By.xpath("//tr"));

// do something with the row list
processLines(rows);

driver.quit();
```

Executing a Javascript function

The second way of doing this, is inspecting the Javascript code to understand how the infinite scroll is built, to do this, as usual, right click + inspect to open

the Chrome Dev tools, and find the `<script>` tag that contains the Javascript code:

Javascript code

```
$(document).ready(function() {
var win = $(window);
var page = 1 ;
var apiUrl = '/product/api/' + page ;

// Each time the user scrolls

var updatePage = function(){
    apiUrl = apiUrl.replace(String(page), "");
    page = page + 1;
    apiUrl = apiUrl + page;
}
var drawNextLines = function(url){
    win.data('ajaxready', false);
    $.ajax({
            url: url,
            dataType: 'json',
            success: function(json) {
                for(var i = 0; i < json.length; i++){
                    var tr = document.createElement('tr');
                    var tdName = document.createElement('td');
                    var tdUrl = document.createElement('td');
                    var tdPrice = document.createElement('td');

                    tdName.innerText = json[i].name;
                    tdUrl.innerText = json[i].url ;
                    tdPrice.innerText = json[i].price;

                    tr.appendChild(tdName);
                    tr.appendChild(tdUrl);
                    tr.appendChild(tdPrice);
```

```
                    var table = document.getElementById('table');
                    table.appendChild(tr);

                }
                win.data('ajaxready', true);
                if(url !== '/product/api/1' && url !== '/product/api/2'\
){
                    updatePage();
                }

                $('#loading').hide();
            }
        });
}

drawNextLines('/product/api/1');
drawNextLines('/product/api/2');

page = 3 ;
apiUrl = '/product/api/3';

// need to update the "ajaxready" variable not to fire multiple ajax ca\
lls when scrolling like crazy
win.data('ajaxready', true).scroll(function() {
    // End of the document reached?
    if (win.data('ajaxready') == false) return;

    // fire the ajax call when we are about to "touch" the bottom of th\
e page
    // no more data past 20 pages
    if (win.scrollTop() + win.height() > $(document).height() - 100 && \
page < 20) {
        $('#loading').show();
        drawNextLines(apiUrl);
```

```
    }
});
});
```

You don't have to understand everything there, the only information that is interesting is that each time we scroll near the bottom of the page (100 pixels to be precise) the `drawNextLines()` function is called. It takes one argument, a URL with this pattern `/product/api/:id` which will return 10 more rows.

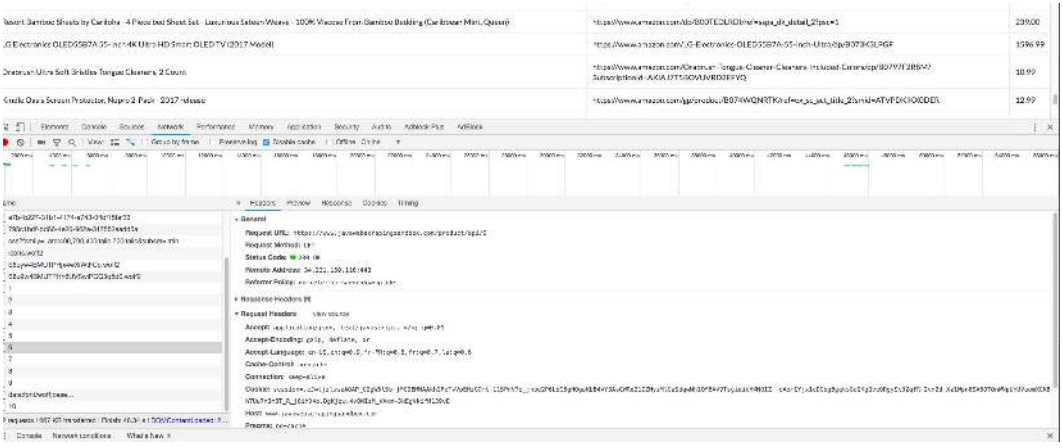
Let's say we want 50 more rows on our table. Basically we only have to make a loop and call `drawNextLines()` five times. If you look closely at the Javascript code, when the AJAX call is loading, we set the variable `ajaxready` to false. So we could check the status of this variable, and wait until it is set to `true`.

Calling a Javascript function

```
JavascriptExecutor js = (JavascriptExecutor) driver;
int pageNumber = 5 ;

driver.get("https://www.javawebscrapingsandbox.com/product/infinite_scr\
oll");
// we start at i=3 because on the first load, /product/api/1 and /produ\
ct/api/2 have already been called.
for(int i = 3; i < pageNumber + 3; i++){
    js.executeScript("drawNextLines('/product/api/" + i +"');");
    while((Boolean)js.executeScript("return win.data('ajaxready');") ==\
 false){
        Thread.sleep(100);
    }
}
List<WebElement> rows = driver.findElements(By.xpath("//tr"));

// do something with the rows
processLines(rows);
```

The “best” way

My favorite way of scraping website using AJAX is to make the HTTP calls to the REST API endpoint directly. In this case, it’s pretty easy to understand what API to call, because the Javascript code is straightforward, but sometimes it can be more complicated. A good method is to open the Chrome Dev tools, and look what’s happening in the “network” tab.



Network Tab on Chrome Dev Tools

We can clearly see the API URl being called, and what the response looks like. Then we can use HtmlUnit or any other HTTP client to perform the requests we want, and parse the JSON response with the Jackson library for example.

Let’s say we want the 50 first rows :

Direct HTTP calls to the API

```
WebClient client = new WebClient();
client.getOptions().setJavaScriptEnabled(false);
client.getOptions().setCssEnabled(false);
client.getOptions().setUseInsecureSSL(true);
java.util.logging.Logger.getLogger("com.gargoylesoftware").setLevel(Lev\
el.OFF);

for(int i = 1; i < 5; i++){
    Page json = client.getPage("https://www.javawebscrapingsandbox.com/\
product/api/" + i );
    parseJson(json.getWebResponse().getContentAsString());
}
```

The API responds with a JSON array, like this one:

JSON response

```
[
    {
        id: 31,
        name: "Marmot Drop Line Men's Jacket, Lightweight 100-Weight Sw\
eater Fleece",
        price: "74.96",
        url: "https://www.amazon.com/gp/product/B075LC96R2/ref=ox_sc_sf\
l_title_39?ie=UTF8"
    },
    {
        id: 32,
        name: "ASUS ZenPad 3S 10 9.7" (2048x1536), 4GB RAM, 64GB eMMC, \
5MP Front / 8MP Rear Camera, Android 6.0, Tablet, Titanium Gray (Z500M-\
C1-GR)",
        price: "296.07",
        url: "https://www.amazon.com/dp/B01MATMXZV?tag=thewire06-20"
    },
    {
```

```
        id: 33,
        name: "LG Electronics OLED65C7P 65-Inch 4K Ultra HD Smart OLED \
TV (2017 Model)",
        price: "2596.99",
        url: "https://www.amazon.com/gp/product/B01NAYM1TP/ref=ox_sc_sf\
l_title_35?ie=UTF8"
    },
    ...
]
```

Here is a simple way to parse this JSON array, loop over every element and print it to the console. In general, we don't want to do this, maybe you want to export it to a CSV file, or save it into a database...

Parsing the JSON response

```
public static void parseJson(String jsonString) throws JsonProcessingEx\
ception, IOException{
    ObjectMapper mapper = new ObjectMapper();
    JsonNode rootNode = mapper.readTree(jsonString);
    Iterator<JsonNode> elements = rootNode.elements();
    while(elements.hasNext()){
        JsonNode node = elements.next();
        Long id = node.get("id").asLong();
        String name = node.get("name").asText();
        String price = node.get("price").asText();
        System.out.println(String.format("Id: %s - Name: %s - Price: %s\
", id, name, price));
    }
}
```

Here are some tips when working with JS rendered web pages:

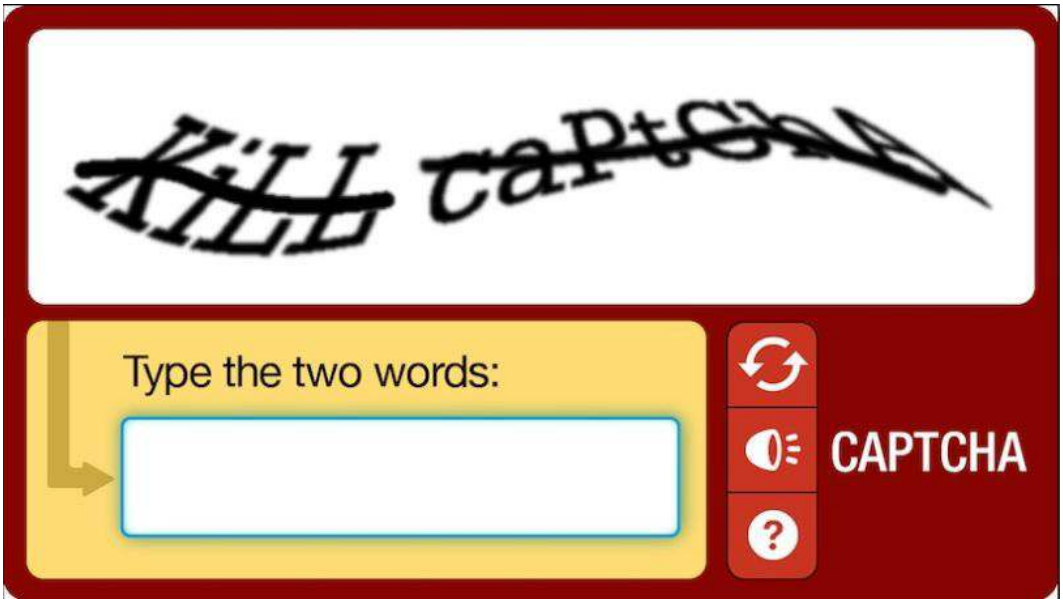
- Try to find the hidden API using the network pane in Chrome Dev Tools
- Try to disable Javascript in your web browser, some websites switch to a server-side rendering in this case.
- Look for a mobile version of the target website, the UI is generally easier to scrape. You can check this using your own phone. If it works without redirecting to a mobile URL (like https://m.example.com or https://mobile.example.com) try to spoof the "User-Agent" request header in your request
- If the UI is tough to scrape, with lots of edge cases, look for Javascript variable in the code, and access the data directly using the Selenium Javascript Executor to evaluate this variable, as we saw earlier.

Captcha solving, PDF parsing, and OCR

In this chapter we are going to see several things, that can block you from scraping websites / extracting information such as Captchas, data inside PDF and images.

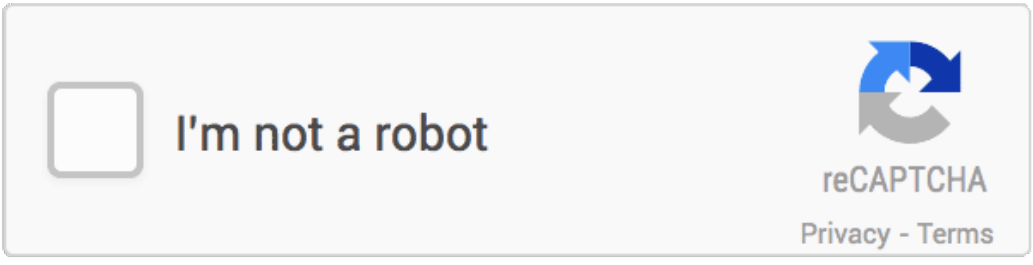
Captcha solving

Completely Automated Public Turing test to tell Computers and Humans Apart is what captcha stands for. Captchas are used to prevent bots/scripts from accessing and performing actions on website or applications. There are dozens of different captcha types, but you should have seen at least these two:



Old Captcha

And this one:



Gooogle ReCaptcha v2

The last one is the most used captcha mechanism, Google ReCaptcha v2. That's why we are going to see how to "break" these captchas.

The only thing the user has to do is to click inside the checkbox. The service will then analyze lots of factors to determine if it a real user, or a bot. We don't know exactly how it is done, Google didn't disclose this for obvious reasons, but a lot of speculations has been made:

- Clicking behavior analysis: where did the user click ? Cursor acceleration etc.
- Browser fingerprinting
- Click location history (do you always click straight on the center, or is it random, like a normal user)
- Browser history and cookies

For old captchas like the first one, Optical Caracter Recognition and recent machine-learning frameworks offer an excellent solving accuracy (sometimes better than Humans…) but for Recaptcha v2 the easiest and more accurate way is to use third-party services.

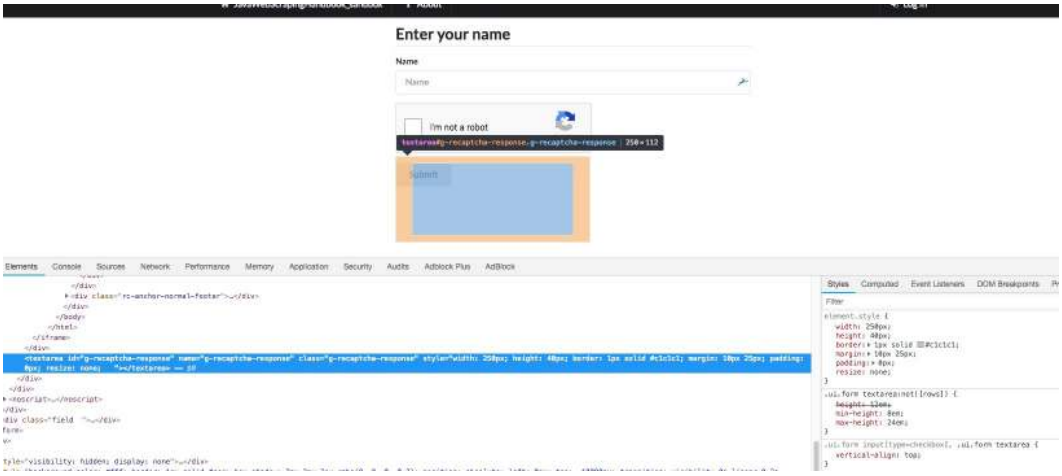
Many companies are offering Captcha Solving API that uses real human operators to solve captchas, I don't recommend one in particular, but I have found 2captcha.com[43] easy to use, reliable and cheap (it is $2.99 for 1000 captchas).

Under the hood, 2captcha and other similar APIs need the specific site-key and the target website URL, with this information they are able to get a human operator to solve the captcha.

```
▶<div class="field  ">…</div>
▼<div class="field  ">
   <label for="recaptcha"></label>
   <script src="https://www.google.com/recaptcha/api.js"></script>
  ▼<div class="g-recaptcha" data-sitekey="6LdegF8UAAAAAJnA5rkZxitaXzdO1OmRJluYZnXZ"> == $0
    ▼<div style="width: 304px; height: 78px;">
      ▼<div>
        ▼<iframe src="https://www.google.com/recaptcha/api2/anchor?ar=1&k=6LdegF8UAAAAAJnA5…W5nc2FuZ
        "78" role="presentation" frameborder="0" scrolling="no" sandbox="allow-forms allow-popups all
        sandbox" kwframeid="1">
          ▼#document
```

Technically the Recaptcha challenge is an iFrame with some magical Javascript code and some hidden input. When you "solve" the challenge, by clicking or solving an image problem, the hidden input is filled with a valid token.

[43] https://2captcha.com?from=6028997



Hidden input with modified visibility

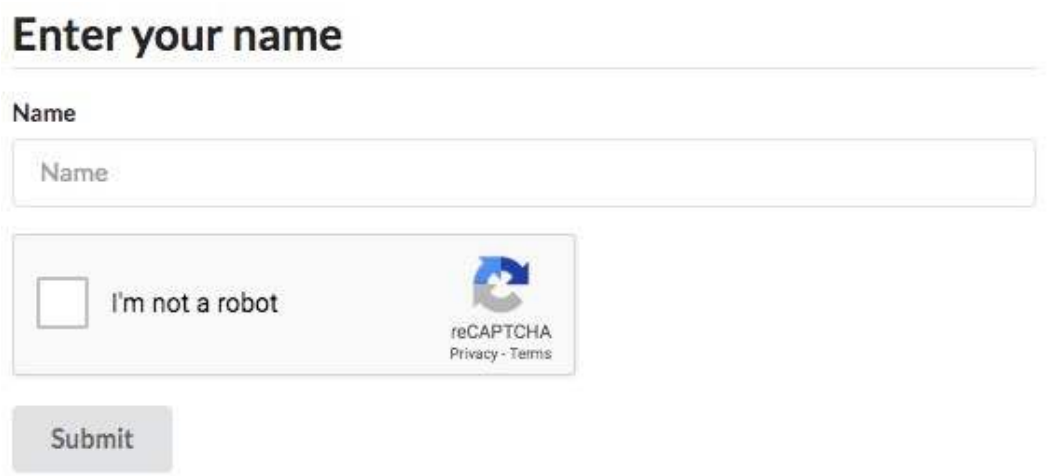
It is this token that interests us, and 2captcha API will send it back. Then we will need to fill the hidden input with this token and submit the form.

The first thing you will need to do is to create an account on 2captcha.com[44] and add some fund.

You will then find your API key on the main dashboard.

As usual, I have set up an example webpage[45] with a simple form with one input and a Recaptcha to solve:

[44] https://2captcha.com?from=6028997
[45] https://www.javawebscrapingsandbox.com/captcha



Form + captcha

We are going to use Chrome in headless mode to post this form and HtmlUnit to make the API calls to 2captcha (we could use any other HTTP client for this). Now let's code.

Instanciate WebDriver and WebClient

```
final String API_KEY = "YOUR_API_KEY" ;
final String API_BASE_URL = "http://2captcha.com/" ;
final String BASE_URL = "https://www.javawebscrapingsandbox.com/captcha\
";

WebClient client = new WebClient();
client.getOptions().setJavaScriptEnabled(false);
client.getOptions().setCssEnabled(false);
client.getOptions().setUseInsecureSSL(true);
java.util.logging.Logger.getLogger("com.gargoylesoftware").setLevel(Lev\
el.OFF);

// replace with your own chromdriver path
final String chromeDriverPath = "/usr/local/bin/chromedriver" ;
System.setProperty("webdriver.chrome.driver", chromeDriverPath);
```

```
ChromeOptions options = new ChromeOptions();
options.addArguments("--headless", "--disable-gpu", "--window-size=1920\
,1200","--ignore-certificate-errors", "--silent");
options.addArguments("--user-agent=Mozilla/5.0 (X11; Linux x86_64) Appl\
eWebKit/537.36 (KHTML, like Gecko) Ubuntu Chromium/60.0.3112.113 Chrome\
/60.0.3112.113 Safari/537.36");
WebDriver driver = new ChromeDriver(options);

driver.get(BASE_URL);
```

Here is some boilerplate code to instantiate both WebDriver and WebClient, along with the API URL and key. Then we have to call the 2captcha API with the site-key, your API key, and the website URL, as documented here[46]. The API is supposed to respond with a strange format, like this one:`OK|123456`.

Finding the sitekey and getting a job ID

```
String siteId = "" ;
WebElement elem = driver.findElement(By.xpath("//div[@class='g-recaptch\
a']"));

try {
    siteId = elem.getAttribute("data-sitekey");
} catch (Exception e) {
    System.err.println("Catpcha's div cannot be found or missing attrib\
ute data-sitekey");
    e.printStackTrace();
}
String QUERY = String.format("%sin.php?key=%s&method=userrecaptcha&goog\
lekey=%s&pageurl=%s&here=now",
        API_BASE_URL, API_KEY, siteId, BASE_URL);
Page response = client.getPage(QUERY);
String stringResponse = response.getWebResponse().getContentAsString();
String jobId = "";
if(!stringResponse.contains("OK")){
```

[46] https://2captcha.com/2captcha-api#solving_recaptchav2_new

```
    throw new Exception("Error with 2captcha.com API, received : " + st\
ringResponse);
}else{
    jobId = stringResponse.split("\\|")[1];
}
```

Now that we have the job ID, we have to loop over another API route to know when the ReCaptcha is solved and get the token, as explained in the documentation. It returns `CAPCHA_NOT_READY` and still the weirdly formatted `OK|TOKEN` when it is ready:

Solving the Captcha

```
boolean captchaSolved = false ;
while(!captchaSolved){
    response = client
        .getPage(String.format("%sres.php?key=%s&action=get&id=%s", API\
_BASE_URL, API_KEY, jobId));
    if (response.getWebResponse()
        .getContentAsString().contains("CAPCHA_NOT_READY")){
        Thread.sleep(3000);
        System.out.println("Waiting for 2Captcha.com ...");
    }else{
        captchaSolved = true ;
        System.out.println("Captcha solved !");
    }
}
String captchaToken = response.getWebResponse().getContentAsString().sp\
lit("\\|")[1];
```

Note that it can take up to 1mn based on my experience. It could be a good idea to implement a safeguard/timeout in the loop because on rare occasions the captcha never gets solved. Now that we have the magic token, we just have to find the hidden input, fills it with the token, and submit the form. The selenium API cannot fill hidden input, so we have to manipulate the DOM to

make the input visible, fills it, make it hidden again so that we can click on the submit button:

Hidden input

```
JavascriptExecutor js = (JavascriptExecutor) driver ;
js.executeScript("document
    .getElementById('g-recaptcha-response').style.display = 'block';");
WebElement textarea = driver.findElement(By
    .xpath("//textarea[@id='g-recaptcha-response']"));

textarea.sendKeys(captchaToken);
js.executeScript("document
    .getElementById('g-recaptcha-response').style.display = 'none';");
driver.findElement(By.id("name")).sendKeys("Kevin");
driver.getPageSource();
driver.findElement(By.id("submit")).click();

if(driver.getPageSource().contains("your captcha was successfully submi\
tted")){
    System.out.println("Captcha successfuly submitted !");
}else{
    System.out.println("Error while submitting captcha");
}
```

And that's it :) Generally, websites don't use ReCaptcha for each HTTP requests, but only for suspicious ones, or for specific actions like account creation, etc. You should always try to figure out if the website is showing you a captcha / Recaptcha because you made too many requests with the same IP address or the same user-agent, or maybe you made too many requests per second.

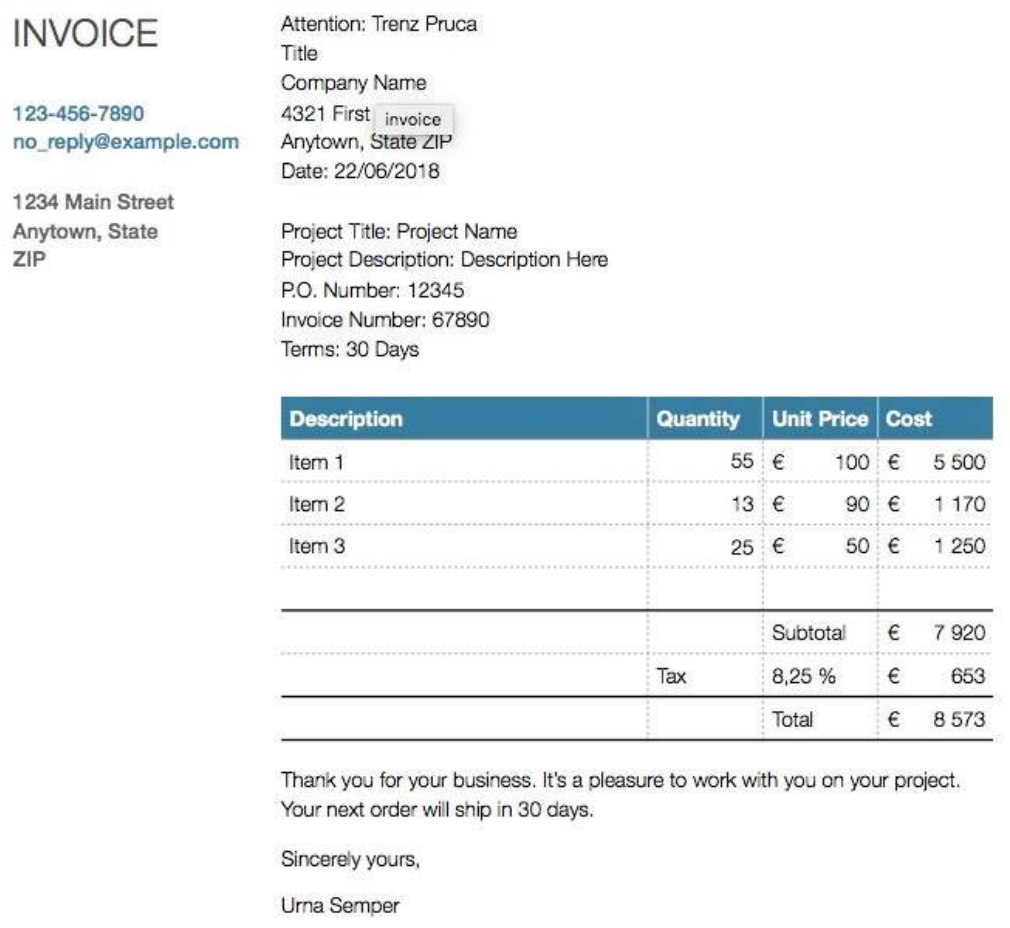
As you can see, "Recaptcha solving" is really slow, so the best way to "solve" this problem is by avoiding catpchas in the first place !

PDF parsing

Adobe created the Portable Document Format in the early 90s. It is still heavily used today for cross-platform document sharing. Lots of websites use PDF export for documents, bills, manuals... And maybe you are reading this eBook in the PDF format. It can be useful to know how to extract pieces of information from PDF files, and that is what we are going to see.

I made a simple page[47], with a link to a PDF invoice. The invoice looks like this:

[47] https://www.javawebscrapingsandbox.com/pdf



INVOICE

123-456-7890
no_reply@example.com

1234 Main Street
Anytown, State
ZIP

Attention: Trenz Pruca
Title
Company Name
4321 First invoice
Anytown, State ZIP
Date: 22/06/2018

Project Title: Project Name
Project Description: Description Here
P.O. Number: 12345
Invoice Number: 67890
Terms: 30 Days

Description	Quantity	Unit Price	Cost
Item 1	55	€ 100	€ 5 500
Item 2	13	€ 90	€ 1 170
Item 3	25	€ 50	€ 1 250
		Subtotal	€ 7 920
	Tax	8,25 %	€ 653
		Total	€ 8 573

Thank you for your business. It's a pleasure to work with you on your project.
Your next order will ship in 30 days.

Sincerely yours,

Urna Semper

Invoice

We are going to see how to download this PDF and extract information from it.

Prerequisites

We will need HtmlUnit to get the webpage and download the PDF, and PDFBox library to parse it.

pom.xml

```
<dependency>
  <groupId>org.apache.pdfbox</groupId>
  <artifactId>pdfbox</artifactId>
  <version>2.0.4</version>
</dependency>
```

Downloading the PDF

Downloading the PDF is simple, as usual: * Go to the target URL * Find the specific anchor * Extract the download URL from the anchor * Use the `Page` object to get the PDF, since it is not an HTML page * Check the content type of what we just downloaded, to make sure that it is an `application/pdf` * Copy the InputStream to a File

Here is the code:

Downloading the invoice

```
HtmlPage html = client.getPage("https://www.javawebscrapingsandbox.com/\
pdf");

// selects the first anchor which contains "pdf"
HtmlAnchor anchor = html.getFirstByXPath("//a[contains(@href, 'pdf')]");
String pdfUrl = anchor.getHrefAttribute();

Page pdf = client.getPage(pdfUrl);

if(pdf.getWebResponse().getContentType().equals("application/pdf")){
    System.out.println("Pdf downloaded");
    IOUtils.copy(pdf.getWebResponse().getContentAsStream(),
            new FileOutputStream("invoice.pdf"));
    System.out.println("Pdf file created");
}
```

Parsing the PDF

Now that we have the PDF file on disk, we can load it into PDFBox to extract the content as a `String`. We are going to extract the price from this invoice.

Once we have the text content from the PDF, it is easy to extract anything from it, using a regular expression. The text looks like this:

Downloading the invoice

```
Title
Company Name
4321 First Street
Anytown, State ZIP
Date: 22/06/2018
Project Title: Project Name
Project Description: Description Here
P.O. Number: 12345
Invoice Number: 67890
Terms: 30 Days
Thank you for your business. It's a pleasure to work with you on your p\
roject.
Your next order will ship in 30 days.
Sincerely yours,
Urna Semper
Description Quantity Unit Price Cost
Item 1 55 €         100 €         5 500
Item 2 13 €         90 €         1 170
Item 3 25 €         50 €         1 250
Subtotal €         7 920
Tax 8,25 % €         653
Total €         8 573
!1
INVOICE
123-456-7890
no_reply@example.com
1234 Main Street
```

```
Anytown, State
ZIP
COMPANY NAME
```

We just have to loop over each line, and use a regular expression with a capturing group like this one: `"Total\\s+€\\s+(.+)"` to extract the total price. We could extract everything we want with another regex, like the email address, the postal address, invoice number...

Here is the full code:

Scraping the Invoice

```
PDDocument document = null;
try{
    document = PDDocument.load(new File("invoice.pdf")) ;

    PDFTextStripperByArea stripper = new PDFTextStripperByArea();
    stripper.setSortByPosition(true);

    PDFTextStripper tStripper = new PDFTextStripper();

    String stringPdf = tStripper.getText(document);
    String lines[] = stringPdf.split("\\n");
    String pattern = "Total\\s+€\\s+(.+)";
    Pattern p = Pattern.compile(pattern);
    String price = "";
    for (String line : lines) {
        Matcher m = p.matcher(line);
        if(m.find()){
            price = m.group(1);
        }
    }

    if(!price.isEmpty()){
        System.out.println("Price found: " + price);
    }else{
```

```
            System.out.println("Price not found");
        }
    }finally{
        if(document != null){
            document.close();
        }
    }
```

There are many methods in the PDFBox library, you can work with password protected PDF, extract specific text area, and many more, here is the documentation[48].

Optical Caracter Recognition

Now that we saw how to deal with PDF, we are going to see how to handle text inside images. Using text inside images is an obfuscation technique aimed to make the extraction difficult for bots. You can often find these techniques on blogs or marketplaces to "hide" an email address/phone number.

Extracting text from an image is called "Optical Caracter Recognition" or OCR. There are many OCR library available, but we are going to use Tesseract[49] which is one of the best open source OCR library.

Installation

Installing Tesseract and all dependencies is really easy, on linux:

[48] https://pdfbox.apache.org/docs/2.0.8/javadocs/
[49] https://github.com/tesseract-ocr/

```
sudo apt install tesseract-ocr
sudo apt install libtesseract-dev
````

And on macOS:

brew install tesseract “‘

More information about installing Tesseract with specific tags can be found here[50]

Tesseract is written in C++, so we need some kind of Java bindings. We are going to use the http://bytedeco.org/ bindings:

pom.xml

```
<dependency>
    <groupId>org.bytedeco.javacpp-presets</groupId>
    <artifactId>tesseract-platform</artifactId>
    <version>3.05.01-1.4.1</version>
</dependency>
```

Tesseract example

I took a screenshot of the previous PDF:

[50] https://github.com/tesseract-ocr/tesseract/wiki

OCR example

Let's say we want to extract the invoice number.

The first thing is to locate your `tessdata` folder, it contains everything tesseract needs to recognize language specific characters. The location will vary depending on how you installed tesseract.

```
final static String TESS_DATA_PATH = "/path/to/tessdata" ;
```

Here is the full code:

OCR example

```
BytePointer outText;
TessBaseAPI api = new TessBaseAPI();

if (api.Init(TESS_DATA_PATH, "ENG") != 0) {
    System.err.println("Could not initialize tesseract.");
    System.exit(1);
}

PIX image = lept.pixRead("ocr_exemple.jpg");
api.SetImage(image);
```

```
// Get OCR result
outText = api.GetUTF8Text();
String string = outText.getString();
String invoiceNumber = "" ;
for(String lines : string.split("\\n")){
    if(lines.contains("Invoice")){
        invoiceNumber = lines.split("Invoice Number: ")[1];
        System.out.println(String.format("Invoice number found : %s", i\
nvoiceNumber));
    }
}

// Destroy used object and release memory
api.End();
outText.deallocate();
lept.pixDestroy(image);
```

This was just an example on how to use Tesseract for simple OCR, I'm not an expert on OCR and image processing, but here are some tips:

Initialize Tesseract with the right language. Image processing: image cropping, different contrasts, re-scaling, border removal... can significantly improve the quality of the Tesseract output. You can use some options like `api.SetVariable("tessedit_char_whitelist", "0123456789,")` to only include numerical characters. This will avoid confusion like `l` instead of `1` see the documentation[51] for more informations about this.

[51] https://github.com/tesseract-ocr/tesseract/wiki

Stay under cover

In this chapter, we are going to see how to make our bots look like Humans. For various reasons, there are sometimes anti-bot mechanisms implemented on websites. The most obvious reason to protect sites from bots is to prevent heavy automated traffic to impact a website's performance. Another reason is to stop bad behavior from bots like spam.

There are various protection mechanisms. Sometime your bot will be blocked if it does too many requests per second / hour / day. Sometimes there is a rate limit on how many requests per IP address. The most difficult protection is when there is a user behavior analysis. For example, the website could analyze the time between requests, if the same IP is making requests concurrently.

You won't necessarily need all the advice in this chapter, but it might help you in case your bot is not working, or things don't work in your Java code the same as it works with a real browser.

Headers

In Chapter 3 we introduced ***HTTP headers***. Your browser includes systematically 6-7 headers, as you can see by inspecting a request in your browser network inspector:

Request headers

If you don't send these headers in your requests, the target server can easily recognize that your request is not sent from a regular web browser. If the server has some kind of anti-bot mechanism, different things can happen: * The HTTP response can change * Your IP address could be blocked * Captcha * Rate limit on your requests

HtmlUnit provides a really simple way to customize our HTTP client's headers

Init WebClient with request headers

```
WebClient client = new WebClient();
client.addRequestHeader("Accept", "text/html,application/xhtml+xml,appl\
ication/xml;q=0.9,image/webp,image/apng,*/*;q=0.8");
client.addRequestHeader("Accept-Encoding", "gzip, deflate, br");
client.addRequestHeader("Accept-Language", "en-US,en;q=0.9,fr-FR;q=0.8,\
fr;q=0.7,la;q=0.6");
client.addRequestHeader("Connection", "keep-alive");
client.addRequestHeader("Host", "ksah.in");
client.addRequestHeader("User-Agent", "Mozilla/5.0 (Macintosh; Intel Ma\
c OS X 10_13_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396\
.99 Safari/537.36");
client.addRequestHeader("Pragma", "no-cache");
```

We could go even further, and assign a random User-Agent to our `WebClient`. Randomizing user-agents will help a lot to hide our bot. A good solution is to create a list of common User-Agents and pick a random one.

You can find such a list here https://developers.whatismybrowser.com/useragents/explore

We could create a file with a lot of different user agents:

user-agents.txt

```
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.1 (KHTML, like Geck\
o) Chrome/13.0.782.112 Safari/535.1
Mozilla/5.0 (Windows NT 6.0) AppleWebKit/535.1 (KHTML, like Gecko) Chro\
me/13.0.782.112 Safari/535.1
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.31 (KHTML, like Gec\
ko) Chrome/26.0.1410.64 Safari/537.31
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gec\
ko) Chrome/27.0.1453.116 Safari/537.36
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gec\
ko) Chrome/27.0.1453.110 Safari/537.36
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.1 (KHTML, like Geck\
o) Chrome/21.0.1180.89 Safari/537.1
Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) AppleWebKit/532.2 (KHTM\
L, like Gecko) Chrome/4.0.221.7 Safari/532.2
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/525.13 (KHT\
ML, like Gecko) Chrome/0.2.149.29 Safari/525.13
Mozilla/5.0 (Windows NT 5.1; rv:5.0.1) Gecko/20100101 Firefox/5.0.1
Mozilla/5.0 (Windows NT 6.1; rv:5.0) Gecko/20100101 Firefox/5.02
Mozilla/5.0 (Windows NT 6.1; WOW64; rv:5.0) Gecko/20100101 Firefox/5.0
Mozilla/5.0 (Windows NT 6.1; rv:2.0b7pre) Gecko/20100921 Firefox/4.0b7p\
re
Mozilla/5.0 (X11; U; Linux x86; fr-fr) Gecko/20100423 Ubuntu/10.04 (luc\
id) Firefox/3.6.3 AppleWebKit/532.4 Safari/532.4
Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1.9.0.11) Gecko/2009060\
215 Firefox/3.0.11
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.1.3) Gecko/20090\
824 Firefox/3.5.3 GTB5
```

And then have a little helper method that reads this file, and returns a random user agent:

Random user agent method

```
private static String getRandomUseragent(){
    List<String> userAgents = new ArrayList<String>();
    Random rand = new Random();
    try (BufferedReader br = new BufferedReader(new FileReader(FILENAME\
))) {
        String sCurrentLine;
        while ((sCurrentLine = br.readLine()) != null) {
            userAgents.add(sCurrentLine);
        }

    } catch (IOException e) {
        e.printStackTrace();
    }

    return userAgents.get(rand.nextInt(userAgents.size()));
}
```

We can then assign a random user agent to the `WebClient` instance:

Set user-agent

```
client.addRequestHeader("User-Agent", getRandomUseragent());
```

Proxies

The easiest solution to hide our scrapers is to use proxies. In combination with random user-agent, using a proxy is a powerful method to hide our scrapers, and scrape rate-limited web pages. Of course, it's better not be blocked in the first place, but sometimes websites allow only a certain amount of request per day / hour.

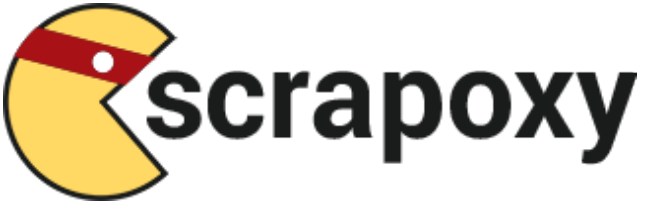
In these cases, you should use a proxy. There are lots of free proxy list, I don't recommend using these because there are often slow, unreliable, and

websites offering these lists are not always transparent about where these proxies are located. Sometimes the public proxy list is operated by a legit company, offering premium proxies, and sometimes not… What I recommend is using a paid proxy service, or you could build your own.

Setting a proxy to HtmlUnit is easy:

```
ProxyConfig proxyConfig = new ProxyConfig("host", myPort);
client.getOptions().setProxyConfig(proxyConfig);
```

Scrapoxy[52] is a great open source API, allowing you to build a proxy API on top of different cloud providers.



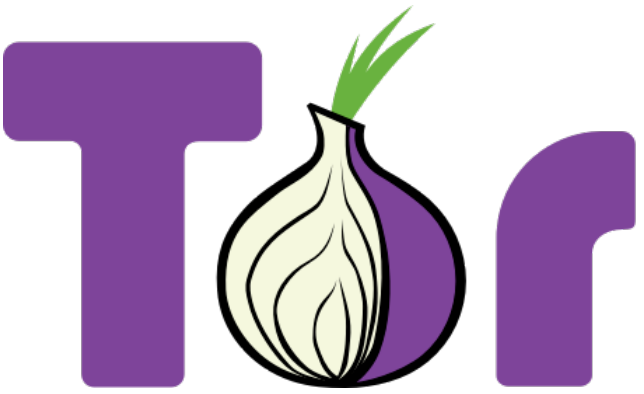
http://scrapoxy.io/

Scrapoxy creates a proxy pool by creating instances on various cloud providers (AWS, OVH, Digital Ocean). Then you will configure HtmlUnit or any HTTP client with the Scrapoxy URL, and it will automatically assign a proxy inside the proxy pool.

You can configure Scrapoxy to fit your needs, and set a minimum / maximum instance number, manage blacklisting of course, either within the configuration file, for example you could blacklist any proxy receiving a *503 HTTP response* or programmaticaly with the REST API, in case the website blocks you with a Captcha, or a special web page.

TOR: The Onion Router

[52] http://scrapoxy.io/



https://www.torproject.org/

TOR, also known as The Onion Router[53] is a worldwide computer network designed to route traffic through many different servers to hide its origin. TOR usage makes network surveillance / traffic analysis very difficult. There are a lot of use cases for TOR usage, such as privacy, freedom of speech, journalists in dictatorship regime, and of course, illegal activities.

In the context of web scraping, TOR can hide your IP address, and change your bot's IP address every 10 minutes. The TOR **exit nodes** IP addresses are public. Some websites block TOR traffic using a simple rule: if the server receives a request from one of TOR public exit node, it will block it. That's why in many cases, TOR won't help you, compared to classic proxies.

Using TOR is really easy, go to the download page, or using your package manager, on macOS:

```
brew install tor
```

Then you have to launch to TOR daemon, and set the proxy config for the `WebClient`

[53] https://www.torproject.org/

Random user agent method

```
WebClient webClient = new WebClient();
ProxyConfig prc = new ProxyConfig("localhost", 9150, true);
webClient.getOptions().setProxyConfig(prc);
```

Tips

Cookies

Cookies are used for lots of reasons, as discussed earlier. If you find that the target website is responding differently with your bots, try to analyze the cookies that are set by client-side Javascript code and inject them manually. You could also use Chrome with the headless mode for better cookie handling.

Timing

If you want to hide your scrapers, you have to behave like a human. Timing is key. Humans don't mass click on links 0.2 seconds after arriving to a web page. They don't click on each link every 5 seconds too. Add some random time between your requests to hide your scrapers.

Fast scraping is not a good practice. You will get blocked, and if you do this on small websites it will put a lot of pressure on the website's servers, it can even be illegal in some cases, as it can be considered like an attack.

Invisible elements

Invisible elements is a technique often used to detect bot accessing and crawling a website. Generally, one or more elements are hidden with CSS and there is some code that notifies the website's server if there is a click on

the element, or a request to a hidden link. Then the server will block the bot's IP address.

A good way to avoid this trap is to use the `isDisplayed()` method with the Selenium API:

Interracting with visible elements only

```
WebElement elem = driver.findElement(By.xpath("//div[@class='something'\
]"));
if(elem.isDisplayed()){
    // do something
}
```

Another technique is to include hidden inputs in a form. If you have problems submitting a form that contains hidden inputs, make sure you include those inputs in your request, and don't modify their value.

Hidden input

```
<form>
    <input type="hidden" name="itsatrap" value="value1"/>
    <input type="text" name="email"/>
    <input type="submit" value="Submit"/>
</form>
```

Cloud scraping

Serverless

In this chapter, we are going to introduce **serverless** deployment for our bots. Serverless is a term referring to the execution of code inside ephemeral containers (Function As A Service, or FaaS). It is a hot topic in 2018, after the "micro-service" hype, here come the "nano-services"!

Cloud functions can be triggered by different things such as:

- An HTTP call to a REST API
- A job in message queue
- A log
- IOT event

Cloud functions are a really good fit for web scraping for many reasons. Web Scraping is I/O bound, most of the time is spent waiting for HTTP responses, so we don't need high end CPU servers. Cloud functions are cheap and easy to setup. Cloud function are a good fit for parallel computing, we can create hundreds or thousands of function at the same time for large scale scraping.

Deploying an Azure function

We are going to deploy a scraper into Azure cloud function. I don't have any preferred vendor, AWS Lambda is a great platform too. Google Cloud doesn't support Java at the moment, only Node.js.

We are going to re-use the Hacker news scraper we built in chapter 3 and implement a little API on top of it, so that we will be able to call this API with a page parameter, and the function will return a JSON array of each hacker news item for this page number.

Prerequisites

You will need :

- JDK 8
- Maven 3+
- Azure CLI[54]

[54] https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest

- Azure function tools[55]
- Azure Account[56]

There are platform-specific instructions for each Azure component installation, I suggest you go through instructions carefully.

Once everything is installed on your system, make sure to log in with the Azure CLI:

```
az login
```

Creating, running and deploying a project

We are going to use a Maven archetype[57] to create the project structure:

Maven archetype

```
mvn archetype:generate \
    -DarchetypeGroupId=com.microsoft.azure \
    -DarchetypeArtifactId=azure-functions-archetype
```

Then Maven will ask you details about the project. The generated code is concise and straightforward:

[55] https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local#v2
[56] https://azure.microsoft.com/en-us/free/
[57] https://maven.apache.org/guides/introduction/introduction-to-archetypes.html

Auto generated function

```
public class Function {
    /**
     * This function listens at endpoint "/api/hello". Two ways to invo\
ke it using "curl" command in bash:
     * 1. curl -d "HTTP Body" {your host}/api/hello
     * 2. curl {your host}/api/hello?name=HTTP%20Query
     */
    @FunctionName("hello")
    public HttpResponseMessage<String> hello(
            @HttpTrigger(name = "req", methods = {"get"}, authLevel = A\
uthorizationLevel.ANONYMOUS) HttpRequestMessage<Optional<String>> reque\
st,
            final ExecutionContext context) {
        context.getLogger().info("Java HTTP trigger processed a request\
.");

        // Parse query parameter
        String query = request.getQueryParameters().get("name");
        String name = request.getBody().orElse(query);

        if (name == null) {
            return request.createResponse(400, "Please pass a name on t\
he query string or in the request body");
        } else {
            return request.createResponse(200, "Hello, " + name);
        }
    }
}
```

The generated code does not protect the API. The `AuthorizationLevel.ANONYMOUS` means anyone can call the route. To implement an authorization mechanism in your function, read the Azure documentation on the subject.

You can then test and run the generated code:

```
mvn clean package
mvn azure-functions:run
```

There might be some errors if you didn't correctly install the previous requirements.

Deploying your Azure Function is as easy as:

```
mvn azure-functions:deploy
```

Azure will create a new URL for your function each time you deploy your app.

The first invocation will be very slow, it can sometimes take up to one minute. This "issue" is called **cold start**. The first time you invoke a function, or when you haven't called a function for a "long" time (i.e several minutes), Azure has to :

- spin a server
- configure it
- load your function code and all the dependencies

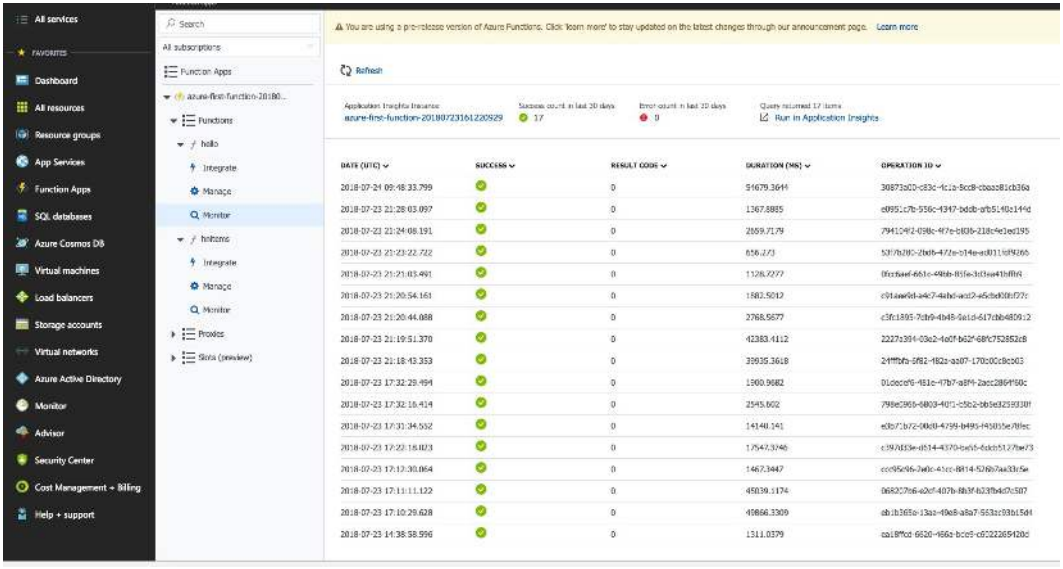
and then it can run your code.

When the app is warm, it just has to run your code, and it will be much much faster. If the cold start is an issue for you, you can use the dedicated mode.

More information about this subject can be found here[58].

You can see your function and the logs in your Azure Dashboard:

[58] https://blogs.msdn.microsoft.com/appserviceteam/2018/02/07/understanding-serverless-cold-start/



https://azure.microsoft.com/

Updating the function

We are going to rename the function to hnitems. We can remove the post method since we only need to make GET requests. Then we need to check the page number parameter, and handle the case where a non numeric value is passed.

Basically, we just change the function name from `hello` to `hnitems` and the request parameter from `name` to `pageNumber`.

The `HNScraper` class is a slightly modified version of the one in chapter 3. The method `scrape` takes a pageNumber and returns a JSON Array of all hacker news items for this page. You can find the full code in the repository.

Function hnitems

```
@FunctionName("hnitems")
public HttpResponseMessage<String> hnitems(
        @HttpTrigger(name = "req", methods = {"get"}, authLevel = Autho\
rizationLevel.ANONYMOUS) HttpRequestMessage<Optional<String>> request,
        final ExecutionContext context) {
    context.getLogger().info("Java HTTP trigger processed a request.");

    // Parse query parameter
    String pageNumber = request.getQueryParameters().get("pageNumber");

    if (pageNumber == null) {
        return request.createResponse(400, "Please pass a pageNumber on\
 the query string");
    }else if(!StringUtils.isNumeric(pageNumber)) {
        return request.createResponse(400, "Please pass a numeric pageN\
umber on the query string");
    }else {
        HNScraper scraper = new HNScraper();
        String json;
        try {
            json = scraper.scrape(pageNumber);
        } catch (JsonProcessingException e) {
            e.printStackTrace();
            return request.createResponse(500, "Internal Server Error w\
hile processing HN items: ");
        }
        return request.createResponse(200, json);
    }
}
```

You can now deploy the updated code using:

```
mvn clean package
mvn azure-functions:deploy
```

You should have your function URL in the log. It's time to test our modified API (replace `${function_url}` with your own URL)

```
curl https://${function_url}/api/hnitems?pageNumber=3
```

And it should respond with the corresponding JSON Array:

Json Response


```
[
    {
        "title": "Nvidia Can Artificially Create Slow Motion That Is Be\
tter Than a 300K FPS Camera (vice.com)",
        "url": "https://motherboard.vice.com/en_us/article/ywejmy/nvidi\
a-ai-slow-motion-better-than-a-300000-fps-camera",
        "author": "jedberg",
        "score": 27,
        "position": 121,
        "id": 17597105
    },
    {
        "title": "Why fundraising is a terrible experience for founders\
: Lessons learned (kapwing.com)",
        "url": "https://www.kapwing.com/blog/the-terrible-truths-of-fun\
draising/",
        "author": "jenthoven",
        "score": 74,
        "position": 122,
        "id": 17594807
    },
    {
        "title": "Why No HTTPS? (whynohttps.com)",
        "url": "https://whynohttps.com",
```

```
        "author": "iafrikan",
        "score": 62,
        "position": 123,
        "id": 17599022
    },
    ...
```

This is it. Instead of returning the JSON array, we could store it in the different database systems supported by Azure.

I suggest you experiment, especially around messaging queues. An interesting architecture for your scrapping project could be to send jobs into a message queue, let Azure function consume these jobs, and save the results into a database. You can read more about this subject here[59]

The possibilities of Azure and other Cloud providers like Amazon Web Service are endless and easy to implement, especially serverless architecture, and I really recommend you to experiment with these tools.

Conclusion

This is the end of this guide. I hope you enjoyed it. You should now be able to write your own scrapers, inspect the DOM and network request, deal with Javascript, reproduce AJAX calls, beat Catpcha and Recaptcha, hide your scrapers with different techniques, and deploy your code in the cloud !

This book will never be finished, as I get so much feedback from my readers. There are many chapters I would like to add. More case study, a chapter about the legal side of web scraping, a chapter about multithreaded scraping etc. If there is enough people interested, I will maybe create a full online video course :)

I made a Google Form[60] to get feedback from my readers, I would really appreciate if you could answer it !

[59] https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-storage-queue-triggered-function
[60] https://docs.google.com/forms/d/e/1FAIpQLSeis4z-NHXeFJfeRLQ6L82-YawEb6ABrOWsN0F4ZIsPZp6cug/viewform

You can send me an email at hi@ksah.in and also find me on Twitter[61] !

Kevin

[61] https://twitter.com/SahinKevin