



(Established under Galgotias University Uttar Pradesh Act No. 14 of 2011)

A Project Report

on

## **Web Scraping using python**

*Submitted in partial fulfilment of the  
requirement for the award of the degree of*

Bachelors of Technology in Computer Science and  
Engineering

**Under The Supervision  
of MR. Arjun KP**

**Assistant Professor  
Department of Computer Science and Engineering**

**Submitted by**

18SCSE1010197 - Devanshu

18SCSE1010050 – Abhishek Mishra



**SCHOOL OF COMPUTING SCIENCE  
AND ENGINEERING,  
GALGOTIAS UNIVERSITY, GREATER NOIDA**

**CANDIDATE'S DECLARATION**

I/We hereby certify that the work which is being presented in the thesis/project/dissertation, entitled “ **Web Scraping using python** ” in partial fulfilment of the requirements for the award of the **Bachelors of Technology in Computer Science and Engineering** submitted in the School of Computing Science and Engineering of Galgotias University, Greater Noida, is an original work carried out during the period of **July-2021 to December-2021**, under the supervision of **Mr. Arjun KP , Assistant Professor , Department of Computer Science and Engineering** of School of Computing Science and Engineering , Galgotias University, Greater Noida

The matter presented in the thesis/project/dissertation has not been submitted by me/us for the award of any other degree of this or any other places.

18SCSE1010197 -Devanshu

18SCSE1010050-AbhishekMishra

This is to certify that the above statement made by the candidates is correct to the best of my knowledge.

Supervisor

(Mr. Arjun KP, Assistant Professor)

## **CERTIFICATE**

The Final Thesis/Project/ Dissertation Viva-Voce examination of 18SCSE1010197 - Devanshu, 18SCSE1010050 – Abhishek Mishra has been held on \_\_\_\_\_ and his/her work is recommended for the award of Bachelors of Technology in Computer Science and Engineering:-

**Signature of Examiner(s)**

**Signature of Supervisor(s)**

**Signature of Project Coordinator**

**Signature of Dean**

Date:

Place:

## **ACKNOWLEDGEMENT**

The feeling of gratitude when we expressed a holy acknowledgement and it's with deep sense of gratitude that we acknowledge the able guidance.

We express our grateful thanks to Mr. Arjun K P, Associate professor, Department of Computer Science and Engineering, Galgotias University for providing us an opportunity for the research report on “ **Web Scraping using python** ” and for his keen interest and the encouragement, which was required for the fulfilment of our capstone project report. We would also like to thank him for giving us valuable guidance at all levels, help and suggestions, which prove to be valuable for preparation of the report.

Finally, I would also like to thank all our friends for their cooperation and interest, which was necessary for completing our project report.

Date:

Devanshu & Abhishek Mishra  
School of Computing Science &  
Engineering, Galgotias University, Greater  
Noida, Uttar Pradesh

# ABSTRACT

The purpose of this thesis is to evaluate state of the art web scraping tools. To support the process, an evaluation framework to compare web scraping tools is developed and utilised, based on previous work and established software comparison metrics. Twelve tools from different programming languages are initially considered. These twelve tools are then reduced to six, based on factors such as similarity and popularity. Nightmare.js, Puppeteer, Selenium, Scrapy, HtmlUnit and rvest are kept and then evaluated. The evaluation framework includes performance, features, reliability and ease of use. Performance is measured in terms of run time, CPU usage and memory usage. The feature evaluation is based on implementing and completing tasks, with each feature in mind. In order to reason about reliability, statistics regarding code quality and GitHub repository statistics are used. The ease of use evaluation considers the installation process, official tutorials and the documentation.

While all tools are useful and viable, results showed that Puppeteer is the most complete tool. It had the best ease of use and feature results, while staying among the top in terms of performance and reliability. If speed is of the essence, HtmlUnit is the fastest. It does however use the most overall resources. Selenium with Java is the slowest and uses the most amount of memory, but is the second best performer in terms of features. Selenium with Python uses the least amount of memory and the second least CPU power. If JavaScript pages are to be accessed, Nightmare.js, Puppeteer, Selenium and HtmlUnit can be used.

## Table of Contents

<b>Title</b>	<b>Page No.</b>
<b>Candidates Declaration</b>	<b>I</b>
<b>Certificate</b>	<b>II</b>
<b>Acknowledgement</b>	<b>III</b>
<b>Abstract</b>	<b>IV</b>
<b>Table of Contents</b>	<b>V</b>
<b>List of Table</b>	<b>VII</b>
<b>List of Figures</b>	<b>VIII</b>
<b>Acronyms</b>	<b>IX</b>
<b>Chapter 1: Getting Started with Scraping</b>	

---

Introduction
Setting up a Python development environment
Getting ready
How to do it...
Scraping Python.org with Requests and BeautifulSoup
Getting ready...
How to do it...
How it works...
Scraping Python.org in urllib3 and BeautifulSoup
Getting ready...
How to do it...
How it works
There's more...
Scraping Python.org with Scrapy
Getting ready...
How to do it...
How it works
Scraping Python.org with Selenium and PhantomJS
Getting ready
How to do it...
How it works
There's more...

### **Chapter 2: Data Acquisition and Extraction**

---

Introduction
How to parse websites and navigate the DOM using BeautifulSoup
Getting ready
How to do it...
How it works
There's more...
Searching the DOM with BeautifulSoup's find methods
Getting ready

## Chapter 3: Processing Data

---

- Introduction
- Working with CSV and JSON data
- Getting ready
- How to do it
- How it works
- There's more...
- Storing data using AWS S3
- Getting ready
- How to do it
- How it works
- There's more...

## Chapter 4: Creating a Simple Data API

- Introduction
- Creating a REST API with Flask-RESTful
- Getting ready
- How to do it
- How it works
- There's more...

## Chapter 05: Creating Scraper Microservices with Docker Introduction

- Installing Docker
- Getting ready
- How to do it
  
- Getting ready
- How to do it
- Running a Docker container(RabbitMQ)
  
- Getting ready
- Creating and running an Elasticsearch container
  
- There's more...
- Stopping/restarting a container and removing the image
- Creating a generic microservice with Nameko
- Creating a scraping microservice

## Chapter 06: Making the Scraper as a Service Real

- Creating and configuring an Elastic Cloud trial account
- Accessing the Elastic Cloud cluster with curl
- Performing an Elasticsearch query with the Python API
- Modifying the API to search for jobs by skill

## List of Table

<b>S.No.</b>	<b>Caption</b>	<b>Page No.</b>
5.1	Unit Testing	29
5.2	Integration Testing	30
5.3	System Testing	31



## List of Figures/Images

<b>S.No.</b>	<b>Title</b>	<b>Page No.</b>
2.1	Existing Projects	10
4.1	Use Case Diagram	24
4.2	Activity Diagram	25
4.3	Class Diagram	26
4.4	State Chart Diagram	27
6.1	Home Page Screen	32
6.2	Title Suggestions	33
6.3	Movie Details	34
6.4	Movie Cast	34
6.5	Actor Details	35
6.6	Recommended Movies	35
6.7	Reviews with Sentiments	36

## Acronyms

B.Tech.	Bachelor of Technology
R.S.	Recommendation Systems
NLP	Natural Language Processing
API	Application Programming Interface
C.F.	Collaborative Filtering
SCSE	School of Computing Science and Engineering

# CHAPTER-1

## Introduction

### 1.1 General Introduction

**The act of going through web pages and extracting selected text or images. An excellent tool for getting new data or enriching your current data.**

**Usually the first step of a data science project which requires a lot of data. An alternative to API calls for data retrieval. Meaning, if you don't have an API or if it's limited in some way.**

**Web scraping is not initially developed for research of social science, as a effect, analysts using this method may incorporate unknown suppositions into their own, because web scraping will not usually require direct contact among the analyst and those who were formerly collecting the information and inserting it online, data analysis issues may simply arise. Research teams using web scraping techniques as an information gathering method still have to be acquainted with the accuracy and correct analysis of the details retrieved from the website. One final problem analysts must address is the potential effect of web scraping on a publication's functionality, as certain web scraping actions unintentionally overpowered and close down a webpage. A web scraper which is appropriately intended and executed, could assist analysts prevail over obstacle to data access, gather online information more resourcefully, and eventually respond investigation queries that cannot be answered by conventional means of assortment and examination. The below figure 1 shows the overview of how web scraping is done.**

## **1.2 Problem Definition**

This paper depends on R.S. that prescribes various things to users. This system will prescribe movies to users. This system will give more. exact outcomes when contrasted with the current systems. The current system chips away at individual users' appraising. This might be some of the time futile for the users who have various preferences from the recommendations shown by the system as each client might have various preferences. This system ascertains the likenesses between various users and then prescribes movies to them according to the evaluations given by the various users of comparable preferences. This will give an exact recommendation to the client. This is an electronic just an android system where there is a film web administration which offers types of assistance to users to rate movies, see recommendations, put remarks and see comparable movies. There are systems that manage the self-recommendation rather than: considering the preferences of users, we thereby assemble a system that admits the user's wishes and then suggest a watch-rundown of movies which depends on their chosen kind. And along these lines makes the watch more ideal and pleasant to the client. Given a bunch of users with their past appraisals for a bunch of movies, would we be able to foresee the rating they will allocate to a film they have not recently evaluated? Ex. Which film would you like given that you have seen "The Avengers", "Avenger Age of Ultron", "Avengers Endgame" and users who saw these movies also liked "Avengers Infinity war"?

## **1.3 Problem Purpose**

R.S. is data filtering devices that try to foresee the rating for users and things, dominantly from huge information to suggest their preferences. Film R.S. gives a component to help users in ordering users with comparable interests. The motivation behind a R.S. essentially is to look for: content that would be fascinating to a person. Additionally, it includes various elements to make customised arrangements of valuable and intriguing substance explicit to

## CHAPTER-2

### Literature Survey

#### 2.1 Literature Review

1 Renita Crystal Pereira et. al., provided web scraping summary and techniques and tools that face several complexities as data extraction isn't that simple. These strategies guarantee

2 that the data collected is correct, consistent and has better integrity,

because there is a large amount of data present which is hard to handle and retain. Although there are a few problems faced by functional techniques that can be such as the elevated amount of web scraping be able to cause rigid harm to the websites. The measurement level of the web scraper will vary with the measurement units of the original source file, making it very difficult to interpret the data.

3 Using social networking sites and internet is amplifying day by day like facebook, twitter, linked-in and some other, user knowledge is also high in the internet available from everywhere. This as well offers hackers an advantage in stealing information. Where the

4 concept of rising income comes into being, social networking is important from a view of business point. Like with online shopping, it will also assist consumers in getting fast shopping and also save time. On the other hand, there is advantage in supporting the

5 company and profiting from it.

6 Kaushal Parikh et. al., [2] proposed a web scraping detection with the

help of machine learning It is valuable for research dependent companies. Web scraping has forever been a difficult preventive attack. Every time a company places its data on internet, it is probable that it could be copied and pasted and then utilized in the other point of view without the corporation knowing itself about it. A lot of protection mechanisms have already been in place but some of them continue to be ignored. The significance of machine learning

7 therefore steps in. Machine learning is quite effective on pattern detection. Therefore if we succeed in making the machine understand a cadence of intruder then it will avoid these types

8 of threats from occurring.

Web scraping solutions are aimed primarily at translating complex data obtained through networks into structured data that could be stored and examined in a central database. Web scraping solutions thus have a significant impact on the result of the cause.

Sameer Padghan et. al., [3] projected an approach where data extraction is done from web pages in assistance with web scraping easily. This method would enable the data to be scrapped from numerous websites that will minimize human intervention, save time and also enhance the quality of data relevance. It will also support the user in gathering data from the site and to save the data to their intent and use it as the individual wishes. The scraped information may be used for database development or for research purposes and also for different similar activities. The scraping used would increase significantly and will often encroach on the framework to obtain the details. However the scraping can be stopped by using effective and safe-web scraping methods.

Web scraping is the process of using bots to extract content and data from a website. Unlike screen scraping, which only copies pixels displayed onscreen, web scraping extracts underlying HTML code and, with it, data stored in a database. The scraper can then replicate entire website content elsewhere. Web scraping is used in a variety of digital businesses that rely on data harvesting.

Legitimate use cases include:

- Search engine bots crawling a site, analysing its content and then ranking it. Price comparison sites deploying bots to auto-fetch prices and product descriptions for allied seller websites.

- Market research companies using scrapers to pull data from forums and social media (e.g., for sentiment analysis).

Since all scraping bots have the same purpose—to access site data—it can be difficult to distinguish between legitimate and malicious bots. That said, several key differences help distinguish between the two.

- Legitimate bots are identified with the organization for which they scrape. For example, Googlebot identifies itself in its HTTP header as belonging to Google. Malicious bots, conversely, impersonate legitimate traffic by creating a false HTTP user agent.

# Data Acquisition and Extraction

## How to parse websites and navigate the DOM using BeautifulSoup

When the browser displays a web page it builds a model of the content of the page in a representation known as the document object model (DOM). The DOM is a hierarchical representation of the page's entire content, as well as structural information, style information, scripts, and links to other content.

It is critical to understand this structure to be able to effectively scrape data from web pages. We will look at an example web page, its DOM, and examine how to navigate the DOM with BeautifulSoup.





## Getting ready

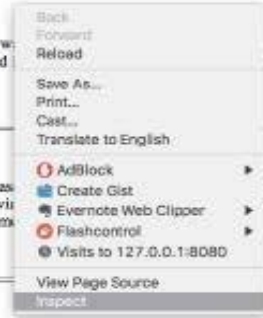
We will use a small web site that is included in the `www` folder of the sample code. To follow along, start a web server from within the `www` folder. This can be done with Python 3 as follows:

```
www $ python3 -m http.server 8080
```

Serving HTTP on 0.0.0.0 port 8080 (<http://0.0.0.0:8080/>) ...

The DOM of a web page can be examined in Chrome by right-clicking the page and selecting Inspect. This opens the Chrome Developer Tools. Open a browser page to <http://localhost:8080/planets.html>. Within chrome you can right click and select 'inspect' to open developer tools (other browsers have similar tools).

	Mercury	0.330	4879	Named Mercurius by the Romans because it appears to move so swiftly.
	Venus	4.87	12104	Roman name for the goddess of love. This planet was also the most beautiful planet of the Roman star in the heavens. Other civilizations have named it differently.
	Earth	5.97	12756	The name Earth comes from the Indo-European base word 'er', which produced the Germanic noun 'ertha', and ultimately German 'erde', Dutch 'aarde', Scandinavian 'jord', and English 'earth'. Related forms include Greek 'gē', meaning 'on the ground,' and Welsh 'erw', meaning 'a piece of land.'
	Mars	0.642	6792	Named by the Romans for their god of war because of its red, bloodlike color. Other civilizations also named this planet from this attribute; for example, the Egyptians named it 'Her Desher,' meaning "the red one."



Selecting Inspect on the Page

This opens the developer tools and the inspector. The DOM can be examined in the Elements tab.

The following shows the selection of the first row in the table:

The screenshot shows the browser's developer tools. The 'Elements' tab is active, displaying the DOM tree. The following HTML structure is visible:

```

<p>
  <table id="planetsList" border="1">
    <tbody>
      <tr>...</tr>
      <tr id="planet1" class="planet" name="Mercury">...</tr>
      <tr id="planet2" class="planet" name="Venus">...</tr>
      <tr id="planet3" class="planet" name="Earth"> == $0
        <td>...</td>
        <td>Earth</td>
        <td>5.97</td>
        <td>12756</td>
        <td>
          "The name Earth comes from the Indo-European base
          'er', which produced the Germanic noun 'ertha', and
          ultimately German 'erde', Dutch 'aarde', Scandinavian 'jord', and
          English 'earth'. Related forms include Greek 'gē', meaning 'on the
          ground,' and Welsh 'erw', meaning 'a piece of land.'
        </td>
      </tr>
    </tbody>
  </table>
</p>
  
```

The 'Styles' panel on the right shows the computed styles for the selected `tr` element:

```

tr {
  display: table-row;
  vertical-align: inherit;
  border-color: inherit;
}
  
```

The 'Inherited from' section shows styles from the parent `table` element:

```

table {
  white-space: normal;
  line-height: normal;
}
  
```



## How it works

Beautiful Soup converts the HTML from the page into its own internal representation. This model has an identical representation to the DOM that would be created by a browser. But Beautiful Soup also provides many powerful capabilities for navigating the elements in the DOM, such as what we have seen when using the tag names as properties. These are great for finding things when we know a fixed path through the HTML with the tag names.

## Querying data with XPath and CSS selectors

CSS selectors are patterns used for selecting elements and are often used to define the elements that styles should be applied to. They can also be used with lxml to select nodes in the DOM. CSS selectors are commonly used as they are more compact than XPath and generally can be more reusable in code. Examples of common selectors which may be used are as follows:

## Scraping Challenges and Solution

Retrying failed page  
downloads Supporting page  
redirects  
Waiting for content to be available in Selenium Limiting crawling to a single domain Processing  
infinitely scrolling pages Controlling the depth of a crawl  
Controlling the length of a crawl  
Handling paginated websites  
Handling forms and form-based authorization Handling basic authorization Preventing bans by  
scraping via proxies Randomizing useragents  
Caching responses

## Retrying failed page downloads

Failed page requests can be easily handled by Scrapy using retry middleware. When installed, Scrapy will attempt retries when receiving the following HTTP error codes:  
[500, 502, 503, 504, 408]

## Supporting page redirects

Page redirects in Scrapy are handled using redirect middleware, which is enabled by default. The process can be further configured using the following parameters:

REDIRECT\_ENABLED: (True/False - default is True)

REDIRECT\_MAX\_TIMES: (The maximum number of redirections to follow for any single request - default is 20)

### How it works

The spider is defined as the following:

```
class Spider(scrapy.spiders.SitemapSpider):
```

```
    name = 'spider'
```

```
    sitemap_urls =
```

```
    ['https://www.nasa.gov/sitemap.xml'] def
```

```
    parse(self, response):
```

```
        print("Parsing: ", response)
```

```
        print (response.request.meta.get('redirect_urls'))
```

This is identical to our previous NASA sitemap based crawler, with the addition of one line printing the redirect\_urls. In any call to parse, this metadata will contain all redirects

that occurred to get to this page.

The crawling process is configured with the

following code: process = CrawlerProcess({

```
'LOG_LEVEL': 'DEBUG',
```

```
'DOWNLOADER_MIDDLEWARES':
```

```
{
```

```
    "scrapy.downloadermiddlewares.redirect.RedirectMiddleware": 500 },
```

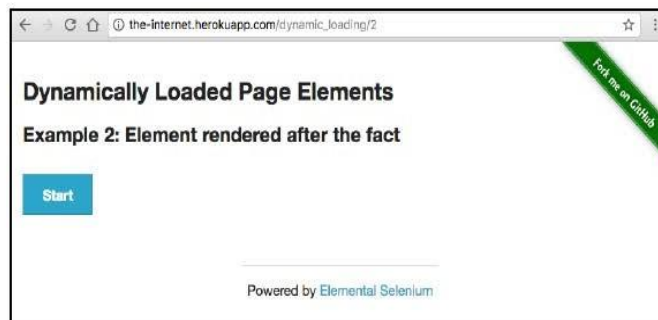
```
'REDIRECT_ENABLED': True,
```

```
'REDIRECT_MAX_TIMES': 2
```

## Waiting for content to be available in Selenium

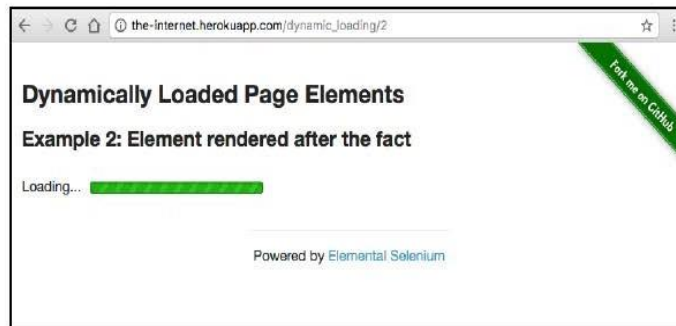
A common problem with dynamic web pages is that even after the whole page has loaded, and hence the `get()` method in Selenium has returned, there still may be content that we need to access later as there are outstanding Ajax requests from the page that are still pending completion. An example of this is needing to click a button, but the button not being enabled until all data has been loaded asynchronously to the page after loading.

Take the following page as an example: [http://the-internet.herokuapp.com/dynamic\\_loading/2](http://the-internet.herokuapp.com/dynamic_loading/2). This page finishes loading very quickly and presents us with a Start button:



The Start button presented on screen

When pressing the button, we are presented with a progress bar for five seconds:



The status bar while waiting

And when this is completed, we are presented with **Hello World!**



After the page is completely rendered

## How it works

Let us break down the explanation:

1. We start by importing the required items from

```
Selenium: from selenium import webdriver
```

```
from selenium.webdriver.support import ui
```

2. Now we load the driver and the page:

```
driver = webdriver.PhantomJS()
```

```
driver.get("http://the-internet.herokuapp.com/dynamic_loading/2")
```

3. With the page loaded, we can retrieve the button:

```
button =
```

```
driver.find_element_by_xpath("//*[@div[@id='start']/button")
```

4. And then we can click the button:

```
button.click()
```

```
print("clicked")
```

5. Next we create a WebDriverWait object:

```
wait = ui.WebDriverWait(driver, 10)
```

6. With this object, we can request Selenium's UI wait for certain events. This also sets a maximum wait of 10 seconds. Now using this, we can wait until we meet a criterion; that an element is identifiable using the following XPath:

```
wait.until(lambda driver:
```

```
driver.find_element_by_xpath("//*[@div[@id='finish']"))
```

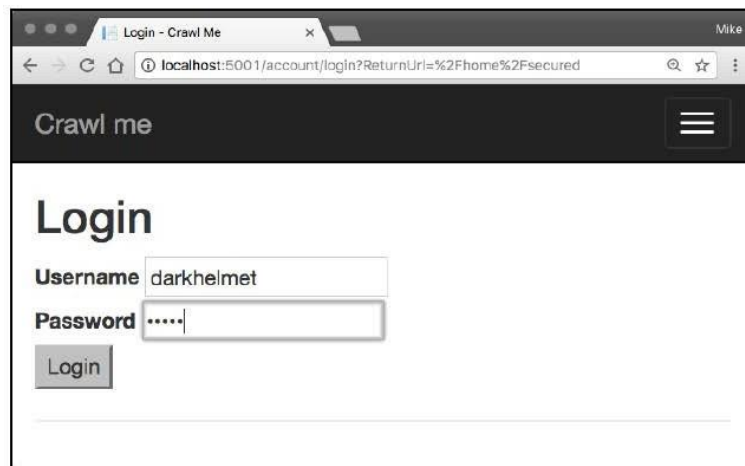
7. When this completes, we can retrieve the h4 element and get its enclosing text: finish\_element=driver.find\_element\_by\_xpath("//\*[@div[@id='finish']/h4")

```
print(finish_element.text)
```

## Handling forms and forms-based authorization

We are often required to log into a site before we can crawl its content. This is usually done through a form where we enter a user name and password, press Enter, and then granted access to previously hidden content. This type of form authentication is often called cookie authorization, as when we authorize, the server creates a cookie that it can use to verify that you have signed in. Scrapy respects these cookies, so all we need to do is somehow automate the form during our crawl.

We will crawl a page in the containers web site at the following URL: <http://localhost:5001/home/secured>. On this page, and links from that page, there is content we would like to scrape. However, this page is blocked by a login. When opening the page in a browser, we are presented with the following login form, where we can enter darkhelmet as the user name and vespa as the password

A screenshot of a web browser window titled "Login - Crawl Me". The address bar shows the URL "localhost:5001/account/login?ReturnUrl=%2Fhome%2Fsecured". The page content includes a dark header with the text "Crawl me" and a hamburger menu icon. Below the header, the word "Login" is displayed in a large font. There are two input fields: "Username" with the value "darkhelmet" and "Password" with masked characters ".....". A "Login" button is positioned below the password field. The browser's address bar and navigation icons are visible at the top.

Username and password credentials are entered

Upon pressing Enter we are authenticated and taken to our originally desired page. There's not a great deal of content there, but the message is enough to verify that we have logged in, and our scraper knows that too.

## How to do it

We proceed with the recipe as follows:

1. If you examine the HTML for the sign-in page, you will have noticed the following form code:

```
<form action="/Account/Login" method="post"><div>
<label for="Username">Username</label>
<input type="text" id="Username" name="Username" value="" />
<span class="field-validation-valid" data-valmsg-
for="Username" data-valmsg-replace="true"></span></div>
<div>
<label for="Password">Password</label>
<input type="password" id="Password" name="Password" />
<span class="field-validation-valid" data-valmsg-
for="Password" data-valmsg-replace="true"></span>
</div>
<input name="submit" type="submit" value="Login"/>
<input name="__RequestVerificationToken" type="hidden"
value="CfDJ8CqzjGWzUMJKkKCmxuBIgZf3UkeXZnVKBwRV_Wu4qUkprH8b_2jno5-
1SGSNjFqIFgLie84xI2ZBkhHDzwgUXpz6bbBwER0v_-
fP5iTITiZi2VfyXzLD_beXUp5cgjCS5AtkIayWThJSI36InzBqj2A" /></form>
```

2. To get the form processors in Scrapy to work, we will need the IDs of the username and password fields in this form. They are Username and Password respectively. Now we can create a spider using this information. This spider is in the script file, 06/09\_forms\_auth.py. The spider definition starts with the following:

```
class Spider(scrapy.Spider):
name = 'spider'
start_urls = ['http://localhost:5001/home/secured']
login_user = 'darkhelmet'
login_pass = 'vespa'
```

3. We define two fields in the class, login\_user and login\_pass, to hold the username we want to use. The crawl will also start at the specified URL.

4. The parse method is then changed to examine if the page contains a login form. This is done by using XPath to see if there is an input form of type password and with an id of Password:

```
def parse(self, response):
print("Parsing: ",
response)
count_of_password_field
s =
int(float(response.xpath("count(//*[input[@type='password'
and @id='Password']]").extract()[0]))
if count_of_password_fields > 0:
print("Got a password page")
```

5. If that field is found, we then return a FormRequest to Scrapy, generated using its from\_response method:

```
return scrapy.FormRequest.from_response(  
response,  
formdata={'Username': self.login_user,  
'Password': self.login_pass},  
callback=self.after_login)
```

## Searching, Mining and Visualizing Data

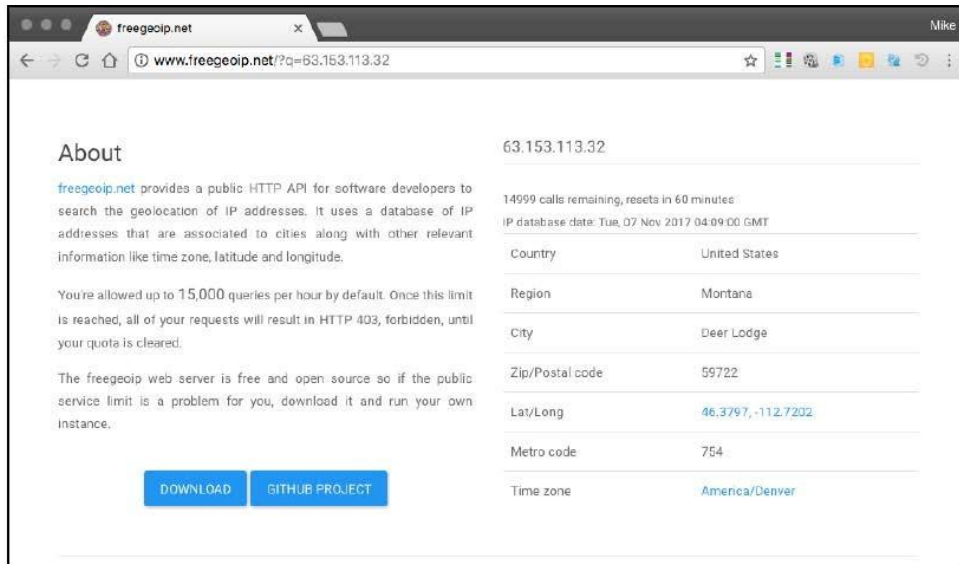
### Geocoding an IP address

Geocoding is the process of converting an address into geographic coordinates. These addresses can be actual street addresses, which can be geocoded with various tools such as the Google maps geocoding API ([https:// developers. google. com/ maps/ documentation/ geocoding/ intro](https://developers.google.com/maps/documentation/geocoding/intro)). IP addresses can be, and often are, geocoded by various applications to determine where computers, and their users, are located. A very common and valuable use is analyzing web server logs to determine the source of users of your website. This is possible because an IP address does not only represent an address of the computer in terms of being able to communicate with that computer, but often can also be converted into an approximate physical location by looking it up in IP address / location databases.

There are many of these databases available, all of which are maintained by various registrars (such as ICANN). There are also other tools that can report geographic locations for public IP addresses.

There are a number of free services for IP geolocation. We will examine one that is quite easy to use, [freegeoip.net](http://freegeoip.net).





The freegeoip.net home page

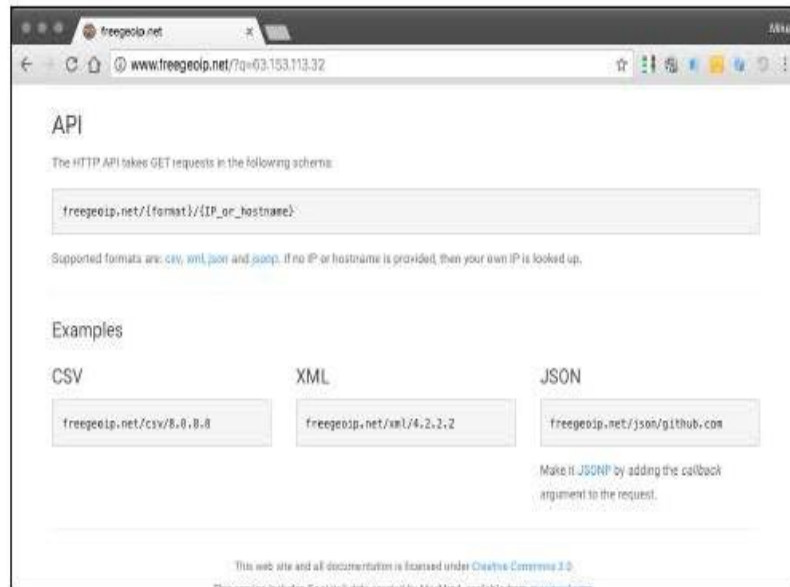
The default page reports your public IP address, and also gives you the geolocation of the IP address according to their database. This isn't accurate to the actual address of my house, and is actually quite a few miles off, but the general location in the world is fairly accurate. We can do important things with data that is at this resolution and even lower. Often just knowing the country origin for web requests is enough for many purposes.



Freegeoip lets you make 15000 calls per hour. Each page load counts as one call, and as we will see, each API call also counts as one.

## How to do it

We could scrape this page to get this information but fortunately, freegeoip.net gives us a convenient REST API to use. Scrolling further down the page, we can see the API documentation:



The freegeoip.net API documentation

We can simply use the requests library to make a GET request using the properly formatted URL. As an example, just entering the following URL in the browser returns a JSON representation of the geocoded data for the given IP address:



Sample JSON for an IP address

A Python script to demonstrate this is available in 08/01\_geocode\_address.py. The is simple and consists of the following:

```
import json
import
requests
raw_json = requests.get("http://www.freegeoip.net/json/63.153.113.92").text
parsed = json.loads(raw_json)
print(json.dumps(parsed, indent=4,
sort_keys=True)) This has the following output:
```

```
{
"city": "Deer Lodge",
"country_code": "US",
"country_name": "United
States", "ip": "63.153.113.92",
"latitude": 46.3797,
"longitude": -112.7202,
"metro_code": 754,
"region_code": "MT",
"region_name": "Montana",
"time_zone":
"America/Denver", "zip_code":
"59722"
}
```

Note that your output for this IP address may vary, and surely will with different IP addresses.

## Creating a REST API with Flask-RESTful

We start with the creation of a simple REST API using Flask-RESTful. This initial API will consist of a single method that lets the caller pass an integer value and which returns a JSON blob. In this recipe, the parameters and their values, as well as the return value, are not important at this time as we want to first simply get an API up and running using Flask- RESTful.

Getting ready

Flask is a web microframework that makes creating simple web application functionality incredibly easy. Flask-RESTful is an extension to Flask which does the same for making REST APIs just as simple. You can get Flask and read more about it at [flask.pocoo.org](http://flask.pocoo.org).

Flask-RESTful can be read about

at <https://flask-restful.readthedocs.io/en/latest/>. Flask can be installed into

your Python environment using pip install flask. and Flask-RESTful can also be installed with pip install flask-restful.

The remainder of the recipes in the book will be in a subfolder of the chapter's directory. This is because most of these recipes either require multiple files to operate, or use the same filename (ie: apy.py).

## How to do it

The initial API is implemented in 09/01/api.py. The API itself and the logic of the API is implemented in this single file: api.py. The API can be run in two manners, the first of which is by simply executing the file as a Python script.

The API can then be launched with the following:

When run, you will initially see output similar to the following:

Starting the job listing API

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

\* Restarting with stat

Starting the job listing

API

\* Debugger is active!

\* Debugger pin code: 362-310-034

This program exposes a REST API on 127.0.0.1:5000, and we can make requests for job listings using a GET request to the path /joblisting/<joblistingid>. We can try this with curl:

```
curl localhost:5000/joblisting/1
```

The result of this command will be the following:

```
{
  "YouRequestedJobWithId": "1"
}
```

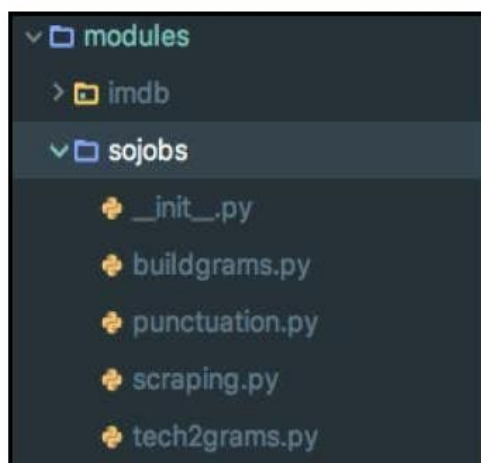
And just like that, we have a REST API up and running. Now let's see how it is implemented.

# Integrating the REST API with scraping code

In this recipe, we will integrate code that we wrote for scraping and getting a clean job listing from StackOverflow with our API. This will result in a reusable API that can be used to perform on-demand scrapes without the client needing any knowledge of the scraping process. Essentially, we will have created a scraper as a service, a concept we will spend much time with in the remaining recipes of the book.

## Getting ready

The first part of this process is to create a module out of our preexisting code that was written in Chapter 7, Text Wrangling and Analysis so that we can reuse it. We will reuse this code in several recipes throughout the remainder of the book. Let's briefly examine the structure and contents of this module before going and integrating it with the API. The code for the module is in the sojobs (for StackOverflow Jobs) module in the project's modules folder.



The sojobs folder

# Storing data in Elasticsearch as the result of a scraping request

In this recipe, we extend our API to save the data we received from the scraper into Elasticsearch. We will use this later (in the next recipe) to be able to optimize requests by using the content in Elasticsearch as a cache so that we do not repeat the scraping process for jobs listings already scraped. Therefore, we can play nice with StackOverflows servers.

Make sure you have Elasticsearch running locally, as the code will access Elasticsearch at localhost:9200. There a good quick-start available at [https://www.elastic.co/guide/en/elasticsearch/reference/current/\\_installation.html](https://www.elastic.co/guide/en/elasticsearch/reference/current/_installation.html), or you can check out the docker Elasticsearch recipe in Chapter 10, Creating Scraper Microservices with Docker if you'd like to run it in Docker.

Once installed, you can check proper installation with the following curl: `curl 127.0.0.1:9200?pretty`

If installed properly, you will get output similar to the following:

```
{
  "name": "KHhxNlz",
  "cluster_name": "elasticsearch",
  "cluster_uuid":
  "fA1qyp78TB623C8IKXgT4g", "version": {
  "number": "6.1.1",
  "build_hash":
  "bd92e7f",
  "build_date": "2017-12-17T20:23:25.338Z",
  "build_snapshot": false,
  "lucene_version": "7.1.0",
  "minimum_wire_compatibility_version": "5.6.0",
```

## How to do it

We will make a few small changes to our API code. The code from the previous recipe has been copied into 09/04/api.py, with the few modifications made.

**1.** First, we add an import for elasticsearch-py:

```
from elasticsearch import Elasticsearch.
```

**2.** Now we make a quick modification to the get method of the JobListing class (I've done the same in JobListingSkills, but it's omitted here for brevity):

```
class JobListing(Resource):
    def get(self, job_listing_id):
        print("Request for job listing with id: " + job_listing_id)
        listing = get_job_listing_info(job_listing_id)
        es = Elasticsearch()
        es.index(index='joblistings', doc_type='job-listing',
            id=job_listing_id, body=listing)
        print("Got the following listing as a response: " +
            listing)
        return listing.
```

**3.** The two new lines create an Elasticsearch object, and then insert the resulting document into ElasticSearch. Before the first time of calling the API, we can see that there is no content, nor a 'joblistings' index, by using the following curl: curl localhost:9200/joblistings.

**4.** Given we just installed Elasticsearch, this will result in the following error.

```
{"error":{"root_cause":[{"type":"index_not_found_exception","reason":"no such index","resource.type":"index_or_alias","resource.id":"joblistings","index_uuid":"_na_","index":"joblistings"}],"type":"index_not_found_exception","reason":"no such index","resource.type":"index_or_alias","resource.id":"joblistings","index_uuid":"_na_","index":"joblistings"},"status":404}.
```

**5.** Now start up the API by using python api.py. Then issue the curl to get the job listing (curl localhost:5000/joblisting/122517). This will result in output similar to the previous recipes. The difference now is that this document will be stored inElasticsearch.

**6.** Now reissue the previous curl for the index:

```
curl localhost:9200/joblistings
```

**7. And now you will get the following result (only the first few lines shown):**

```
{
"joblistings
": {
"aliases":
},
"mappings
": {
"job-listing": {
"properties": {
"CleanedWords
" { "type":
"text",
"fields": {
"keyword": {
"type":
"keyword",
"ignore_above": 256
},
"ID
": {
"type":
"text",
"fields": {
"keyword": {
"type":
"keyword",
"ignore_above": 256
},

```

**8. The specific document that we just stored can be retrieved by using the following curl:**

```
curl localhost:9200/joblistings/job-listing/122517
```

**9. Which will give us the following result (again, just the beginning of the content shown):**

```
{
"_index": "joblistings",
"_type": "job-listing",
"_id": "122517",
"_version": 1,
"found": true,
"_source": {
"ID":
"122517",
"JSON": {
"@context": "http://schema.org",
"@type": "JobPosting",
```



```
"title": "SpaceX Enterprise Software Engineer, Full Stack", "skills": [
  "c#",
  "sql",
  "javascri
pt",
  "asp.net
",
  "angular
js"
],
"description": "<h2>About this job</h2>\r\n<p><span>Location
options: <strong>Paid relocation</strong></span><br/><span>Job
type: <strong>Permanent</strong></span><br/><span>Experience
level:
<strong>Mid-Level,
And just like that, with two lines of code, we have the document stored in our
Elasticsearch database. Now let's briefly examine how this worked.
```

# Checking Elasticsearch for a listing before scraping

Now lets leverage Elasticsearch as a cache by checking to see if we already have stored a job listing and hence do not need to hit StackOverflow again. We extend the API for performing a scrape of a job listing to first search Elasticsearch, and if the result is found there we return that data. Hence, we optimize the process by making Elasticsearch a job listings cache.

## How to do it

We proceed with the recipe as follows:

The code for this recipe is within 09/05/api.py. The JobListing class now has the following implementation:

```
class JobListing(Resource):
def get(self, job_listing_id):
print("Request for job listing with id: " + job_listing_id)
es = Elasticsearch()
if(es.exists(index='joblistings', doc_type='job-listing',
id=job_listing_id)):
print('Found the document in ElasticSearch')
doc = es.get(index='joblistings', doc_type='job-listing',
id=job_listing_id)
return doc['_source']
listing = get_job_listing_info(job_listing_id)
es.index(index='joblistings', doc_type='job-listing',
id=job_listing_id, body=listing)
print("Got the following listing as a response: " + listing)
return listing
```

Before calling the scraper code, the API checks to see if the document already exists in Elasticsearch. This is performed by the appropriately named 'exists' method, which we pass the index, doc type and ID we are trying to get.

There's more...

The JobListingSkills API implementation follows a slightly different pattern. The following is its code:

```
class JobListingSkills(Resource):
def get(self, job_listing_id):
print("Request for job listing's skills with id: " +
job_listing_id)
es = Elasticsearch()
if(es.exists(index='joblistings', doc_type='job-listing',
id=job_listing_id)):
print('Found the document in ElasticSearch')
doc = es.get(index='joblistings', doc_type='job-listing',
id=job_listing_id)
return doc['_source']['JSON']['skills']
skills = get_job_listing_skills(job_listing_id)
print("Got the following skills as a response: " + skills)
```

# Making the Scraper as a service Real

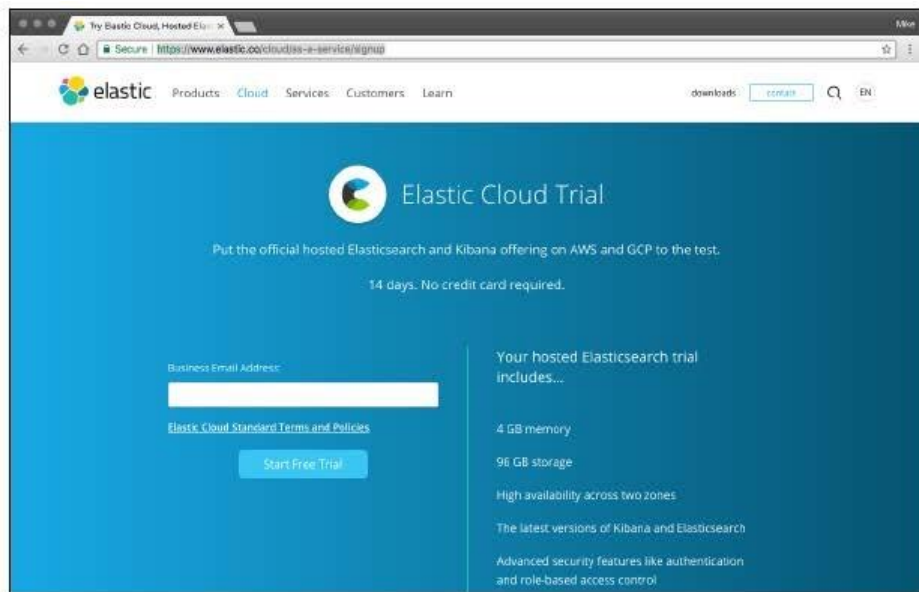
In this recipe we will create and configure an Elastic Cloud trial account so that we can use Elasticsearch as a hosted service. Elastic Cloud is a cloud service offered by the creators of Elasticsearch, and provides a completely managed implementation of Elasticsearch.

While we have examined putting Elasticsearch in a Docker container, actually running a container with Elasticsearch within AWS is very difficult due to a number of memory requirements and other system configurations that are complicated to get working within ECS. Therefore, for a cloud solution, we will use ElasticCloud

## How to do it

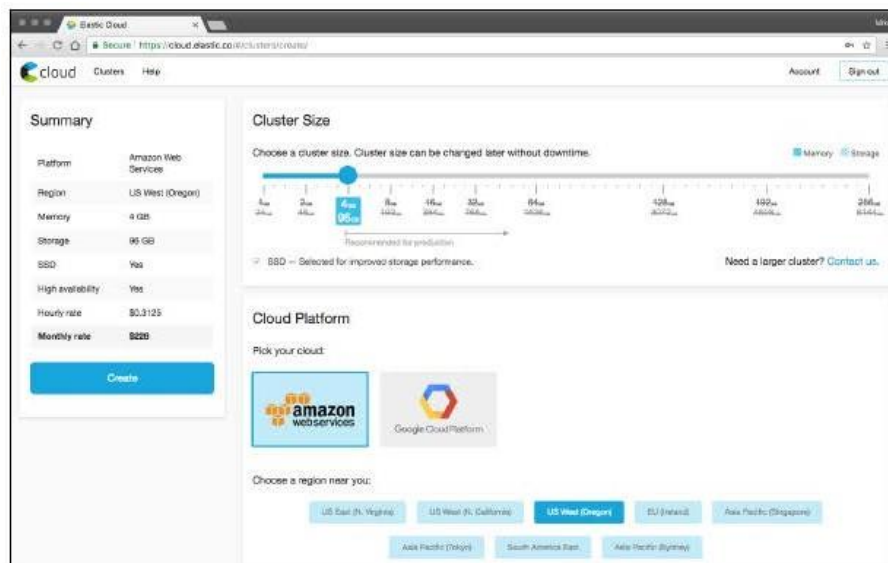
We'll proceed with the recipe as follows:

1. Open your browser and navigate to <https://www.elastic.co/cloud/as-a-service/signup>. You will see a page similar to the following:



The Elastic Cloud signup page

1. Enter your email and press the Start Free Trial button. When the email arrives, verify yourself. You will be taken to a page to create your cluster:



Cluster creation page

2. I'll be using AWS (not Google) in the Oregon (us-west-2) region in other examples, so I'll pick both of those for this cluster. You can pick a cloud and region that works for you. You can leave the other options as it is, and just press create. You will then be presented with your username and password. Jot those down. The following screenshot gives an idea of how it displays the username and password:

# Your New Elastic Cluster

Copy down the generated password for the `elastic` user and keep it somewhere safe. We can show you this password only once. If you lose the password, you need to reset it on the security page.

Username	<code>elastic</code>
Password	<code>tduhdZxu</code>
Cloud ID	<code>7dc EGF jYs</code> <a href="#">Copy</a>

Get started with Beats and Logstash quickly. The Cloud ID simplifies sending data to your cluster on Elastic Cloud. [Learn more ...](#)

OK

The credentials info for the Elastic Cloud account



We won't use the Cloud ID in any recipes.

- Next, you will be presented with your endpoints. The Elasticsearch URL is what's important to us:

Endpoints		
Elasticsearch	<code>https://7dc72c</code>	<code>a27.us-west-2.aws.found.io:9243</code>
Kibana	<code>https://96eet</code>	<code>37f.us-west-2.aws.found.io:9243</code>

- And that's it - you are ready to go (at least for 14 days)!

# Connecting to the Elastic Cloud cluster with Python

Now let's look at how to connect to Elastic Cloud using the Elasticsearch Python library.

## Getting ready

The code for this recipe is in the 11/01/elasticcloud\_starwars.py script. This script will scrape Star Wars character data from the swapi.co API/website and put it into the Elastic Cloud.

### How to do it

We proceed with the recipe as follows:

1. Execute the file as a Python script:

```
$ python elasticcloud_starwars.py
```

2. This will loop through up to 20 characters and drop them into the sw index with a document type of people. The code is straightforward (replace the URL with

yours):

```
from elasticsearch import
```

```
Elasticsearch import requests
```

```
import json
```

```
if __name__ == '__main__':
```

```
    es =
```

```
    Elasticsearch([
```

```
        "https://elastic:tduhdExunhEWPjSuH73O6yLS@d7c72d3327076cc4daf552
```

```
        8103c46a27.us-west-2.aws.found.io:9243"
```

```
    ])
```

```
    i = 1
```

```
    while i < 20:
```

```
        r = requests.get('http://swapi.co/api/people/' +
```

```
        str(i)) if r.status_code is not 200:
```

```
            print("Got a " + str(r.status_code) + " so stopping")
```

```
            break
```

```
        j =
```

```
        json.loads(r.content)
```

```
        print(i, j)
```

```
        #es.index(index='sw', doc_type='people', id=i,
```

```
        body=json.loads(r.content))
```

```
        i = i + 1
```

3. The connection is made using the URL with the username and password added to it. The data is pulled from swapi.co using a GET request and then with a call to .index() on the Elasticsearch object. You'll see output similar to the

# Configuring Docker to authenticate with ECR

In this recipe, we will configure docker to be able to push our containers to the Elastic Container Repository (ECR).

## Getting ready

A key element of Docker is docker container repositories. We have previously used Docker Hub to pull containers. But we can also push our containers to Docker Hub, or any Dockercompatible container repository, such as ECR. But this is not without its troubles. The docker CLI does not naturally know how to authenticate with ECR, so we have to jump through a few hoops to get it to work.

Make sure that the AWS command line tools are installed. These are required to get Docker authenticated

to work with ECR. Good instructions are found at [https:// docs. aws. amazon. com/ cli/ latest/ userguide/](https://docs.aws.amazon.com/cli/latest/userguide/)

installing. html. Once the install is verified, you will need to configure the CLI to use the account created

in the previous recipe. This can be done using the aws configure command, which will prompt you for four items:

```
$ aws configure
```

```
AWS Access Key ID [None]: AKIA ----- QKCVQAA
```

```
AWS Secret Access Key [None]: KEuSaLgn4dpyXe ----- VmEKdhV
```

```
Default region name [None]: us-west-2
```

```
Default output format [None]: json
```

Swap the keys to be the ones you retrieved earlier, and set your default region and data type.

## How to do it

We proceed with the recipe as follows:

1. Execute the following command. This returns a command to authenticate Docker with ECR:

```
$ aws ecr get-login --no-include-email --region us-west-2 docker
```

```
login -u AWS -p
```

```
eyJwYXlsb2FkIjoiN3BZVWY4Q2JoZkFwYUNKOUUp6c1BkRy80VmRYN0Y2LzQ0  
Y2pVNFJ  
KZTA5alBrUEdSMHINuk9TMytsTFVURGtxb3Q5VTZqV0xxNmRCVHJnL1Fib2IG  
bEF0dV  
ZhNFpEOUkxb1FxUTNwcUluaVhqS1FCZmU2WTRLNlQrbjE4VHdiOEpbmtwWjJ  
Jek8xR  
IR2Y2Y5S3NGRIQrbDZhcktUNXZJbjNkb1czVGQ2TXZPUlg5cE5Ea2w4S29vamt6S  
E10  
Ym8rOW5mLzBvVkRRSDlaY3hqRG45d0FzNVA5Z1BPVUU5OVFrTEZGeENPUHJRZm  
lTeHFqaEVPcGo3ZVAGM0WEIKdy83bG4wSGMwMERNZW5s2R0V4SEniWTRSS  
XBUTUNJNThJblV3QUFBSDR3ZkFZSkvWklodmNOQVFjR29HOHdiUUICQURC  
b0Jna3Fo  
a2lHOXcwQkJ3RXdIZ1IKWUlaSUFxVURCQUV1TUJFRURQdTFQVXQwRDFkN3c3Ry  
s3Z0l  
CRUIBN21Xay9EZnNOM3R5MS9iRFdRYIZtZjdOOURST2xhQWFFbTBFQVFndy9  
JYIBjTz  
hLc0RINDBCLzhOVnR0YmlFK1FXSDBCaTZmemtCbzNxTke9IiwidmVyc2lvbiI6IjJlL  
CJ0eXBIIjoiREFUQV9LRVkiLCJleHBpcmF0aW9uIjoxNTE1NjA2NzM0fQ==  
https://270157190882.dkr.ecr.us-west-2.amazonaws.com
```

## Creating a task to run our containers

In this recipe, we will create an ECSTask. A task tells the ECR cluster manager which containers to run. A task is a description of which containers in ECR to run and the parameters required for each. The task description will feel a lot like that which we have done with Docker Compose.

### Getting ready

The task definition can be built with the GUI or started by submitting a task definition JSON file. We will use the latter technique and examine the structure of the file, `td.json`, which describes how to run our containers together. This file is in the 11/07 recipe folder. **How to do it** The following command registers the task with ECS:

```
$ aws ecs register-task-definition --cli-input-json file://td.json
```

```
{
  "taskDefinition":
  : { "volumes":
  [
  ],
  "family": "scraper",
  "memory": "4096",
  "placementConstraints": [
  ],
  "cpu": "1024",
  "containerDefinitions": [
  {
    "name": "rabbitmq",
    "cpu": 0,
    "volumesFrom": [
    ],
    "mountPoints": [
    ],
    "portMappings": [
    {
      "hostPort": 15672,
      "protocol": "tcp",
      "containerPort": 15672
    },
    {
      "hostPort": 5672,
      "protocol": "tcp",
      "containerPort": 5672
    }
    ],
    "environment": [
    "hostPort": 5672,
    "protocol": "tcp",
    "containerPort": 567
    "environment": [
    ],
    "image": "414704166289.dkr.ecr.us-west-2.amazonaws.com/rabbitmq", "memory":
    256,
    "essential": true
```



```

microservice",
"memory": 256,
"links": [
"rabbitmq"
]
},
{
"name": "api",
"cpu": 0,
"essential":
true,
"volumesFrom"
: [
],
"mountPoints": [
],
"portMappings": [
{
"hostPort": 80,
"protocol": "tcp",
"containerPort": 8080
}
],
"environment": [
{
"name": "AMQP_URI",
"value": "pyamqp://guest:guest@rabbitmq"
},
{
"name": "ES_HOST",
"value":
"https://elastic:tduhdExunhEWPjSuH73O6yLS@7dc72d3327076cc4daf5528103c46a27 . us-
west-2.aws.found.io:9243"
}
],
"image": "414704166289.dkr.ecr.us-west-2.amazonaws.com/scrapper-restapi",
"memory": 128,
"links": [
"rabbitmq"
]
}
],
"requiresCompatibilities": [
"EC2"
],
"status": "ACTIVE",
"taskDefinitionArn": "arn:aws:ecs:us-west-
2:414704166289:taskdefinition/ scraper:7",
"requiresAttributes": [
"name": "com.amazonaws.ecs.capability.ecr-auth"
}
],
"revision": 7,
"compatibilities": [

```

# Starting and accessing the containers in AWS

In this recipe, we will start our scraper as a service by telling ECS to run our task definition. Then we will check that it is running by issuing a curl to get contents of a job listing.

## Getting ready

We need to do one quick thing before running the task. Tasks in ECS go through revisions. Each time you register a task definition with the same name ("family"), ECS defines a new revision number. You can run any of the revisions.

To run the most recent one, we need to list the task definitions for that family and find the most recent revision number. The following lists all of the task definitions in the cluster.

At this point we only have one:

```
$ aws ecs list-task-definitions
{
  "taskDefinitionArns": [
    "arn:aws:ecs:us-west-2:414704166289:task-definition/scraper-as-a-service:17"
  ]
}
```

Notice my revision number is at 17. While this is my only currently registered version of this task, I have registered (and unregistered) 16 previous revisions.

How to do it

We proceed with the recipe as follows:

1. Now we can run our task. We do this with the following command:

```
$ aws ecs run-task --cluster scraper-cluster --task-definition scraper-as-a-service:17 --count 1
{
  "tasks": [
    {
      "taskArn": "arn:aws:ecs:uswest-2:414704166289:task/00d7b868-1b99-4b54-9f2a-0d5d0ae75197",
      "version": 1,
      "group": "family:scraper-as-a-service",
      "containerInstanceArn": "arn:aws:ecs:uswest-2:414704166289:container-instance/5959fd63-7fd6-4f0e-92aeea136dabd762",
      "taskDefinitionArn": "arn:aws:ecs:uswest-2:414704166289:task-definition/scraper-as-a-service:17",
      "containers": [
        {
          "name": "rabbitmq",
          "containerArn": "arn:aws:ecs:uswest-2:414704166289:container/4b14d4d5-422c-4ffaa64c-476a983ec43b",
          "lastStatus": "PENDING",
          "taskArn": "arn:aws:ecs:uswest-2:414704166289:task/00d7b868-1b99-4b54-9f2a-0d5d0ae75197",
          "networkInterfaces": [
```

```

{
  "name": "scraper-microservice",
  "containerArn": "arn:aws:ecs:uswest-
2:414704166289:container/511b39d2-5104-4962- a859-
86fdd46568a9",
  "lastStatus": "PENDING",
  "taskArn": "arn:aws:ecs:uswest-
2:414704166289:task/00d7b868-1b99-4b54-9f2a-0d5d0ae75197",
  "networkInterfaces": [
  ]
},
{
  "name": "api",
  "containerArn": "arn:aws:ecs:uswest-
2:414704166289:container/0e660af7-
e2e8-4707-b04bb8df18bc335b", "lastStatus": "PENDING",
  "taskArn": "arn:aws:ecs:uswest-
2:414704166289:task/00d7b868-1b99-4b54-9f2a-0d5d0ae75197",
  "networkInterfaces": [
  ]
},
],
"launchType": "EC2",
"overrides": {
  "containerOverrides": [
  {
    "name": "rabbitmq"
  },
  {
    "name": "scraper-microservice"
  },
  {
    "name": "api"
  }
  ]
},
"lastStatus": "PENDING",
"createdAt": 1515739041.287,
"clusterArn": "arn:aws:ecs:uswest-
2:414704166289:cluster/scraper-cluster",
"memory": "4096",
"cpu": "1024",
"desiredStatus": "RUNNING",
"attachments": [
]
],
"failures": [
]
}

```

The output gives us a current status of the task. The very first time this is run, it will take a little time to get going, as the containers are being copied over to the EC2 instance. The main culprit of that delay is the scraper-microservice container with all of the **136K** data.

2. You can check the status of the task with the following command:

```
$ aws ecs describe-tasks --cluster scraper-cluster --  
task 00d7b868-1b99-4b54-9f2a-0d5d0ae75197
```

You will need to change the task GUID to match guid in the "taskArn" property of the output from running the task. When all the containers are running, we are ready to test the API.

3. To call our service, we will need to find the IP address or DNS name for our cluster instance. you can get this from the output when we created the cluster, through the portal, or with the following commands. First, describe the cluster instances:

```
$ aws ecs list-container-instances --cluster scraper-cluster  
{  
  "containerInstanceArns": [  
    "arn:aws:ecs:us-west-  
2:414704166289:containerinstance/ 5959fd63-7fd6-  
4f0e-92aa-ea136dabd762"  
  ]  
}
```

4. With the GUID for our EC2 instance, we can query its info and pull the EC2 instance ID with the following:

```
$ aws ecs describe-container-instances --cluster scraper-cluster  
-- container-instances 5959fd63-7fd6-4f0e-92aa-ea136dabd762  
| grep "ec2InstanceId"  
"ec2InstanceId": "i-08614daf41a9ab8a2",
```

5. With that instance ID, we can get the DNS name:

```
$ aws ec2 describe-instances --instance-ids i-  
08614daf41a9ab8a2 | grep "PublicDnsName"  
"PublicDnsName": "ec2-52-27-26-  
220.uswest-2.compute.amazonaws.com",  
"PublicDnsName":  
"ec2-52-27-26-220.us-west-2.compute.amazonaws.com"  
"PublicDnsName":  
"ec2-52-27-26-220.us-west-2.compute.amazonaws.com"
```

6. And with that DNS name, we can make a curl to get a job listing:

```
$ curl ec2-52-27-26-220.uswest-  
2.compute.amazonaws.com/joblisting/122517 | head -n 6
```

And we get the following familiar result!

```
{  
  "ID": "122517",  
  "JSON": {  
    "@context": "http://schema.org",  
    "@type": "JobPosting",  
    "title": "SpaceX Enterprise Software Engineer, Full Stack",
```

**Our scraper is now running in the cloud!**

