Some of the scrapped images:

Image Scrapper from scratch to proudction

Table of Contents

Preface

This book is intended to help all the data scientists out there. It is a step by step guide for creating a web scraper, in this case, an image scrapper right from scratch and then deploying it to the heroku cloud platform. Image scrappers are extensively used in the industry today for collecting a huge number of images that are used as inputs for training the object detection/classification/identification models. This book takes a simple example of an online image search and tries to explain the concepts simply, extensively, and thoroughly to create a review scrapper right from scratch and then its deployment to a cloud environment.

Happy Learning!

Web Scraping(Images)

1. Introduction:

Web scraping is a technique using which the webpages from the internet are fetched and parsed to understand and extract specific information similar to a human being. Web scrapping consists of two parts:

- Web Crawling→ Accessing the webpages over the internet and pulling data from them.
- HTML Parsing→ Parsing the HTML content of the webpages obtained through web crawling and then extracting specific information from it.

Hence, web scrappers are applications/bots, which automatically send requests to websites and then extract the desired information from the website output.

Let's take an example:

how do we search for images online?

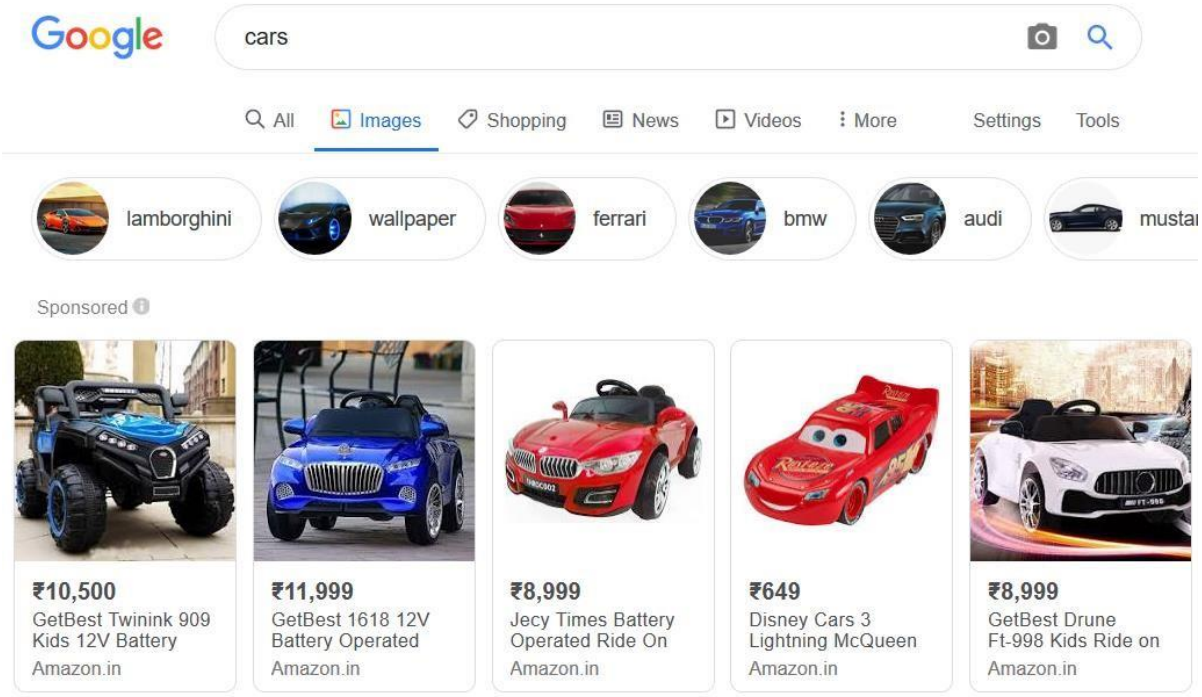
- We go to a website that shows images, and then we enter the keyword to search for the photos.
- The website shows us the images.
- We decide to download the image(s).

What if there is a computer program that can do all of these for us? That's what web scrappers necessarily do. They try to understand the webpage content as a human would do.

Let's take an example where we need thousands of images to train our object detection/classification model. It is an excellent use case for implementing an image scraper.

In this document, we'll take the example of searching for images and downloading them and take it further.

For example, if we open google.com and search for 'cars, the search result will be as follows:



Our end goal is to build a web scraper that collects the images for a keyword from the internet.

2. Prerequisites:

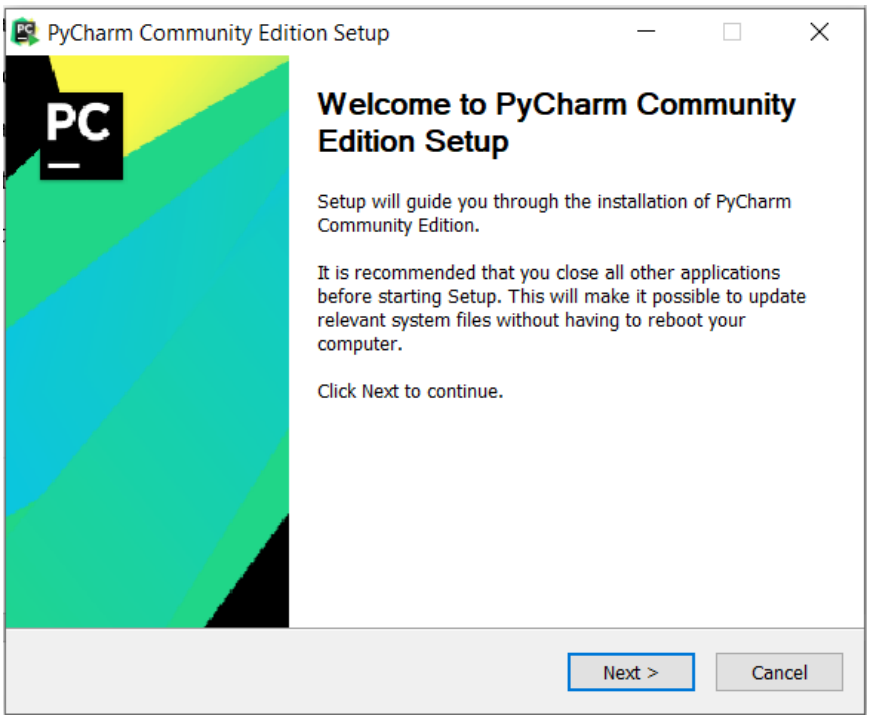
The things needed before we start building a python based web scraper are:

- Python installed.
- A Python IDE (Integrated Development Environment): like PyCharm, Spyder, or any other IDE of choice (Explained Later)
- Flask Installed. (A simple command: pip install flask)
- MongoDB installed (Explained Later).
- Basic understanding of Python and HTML.
- Basic understanding of Git (download Git CLI from https://gitforwindows.org/)

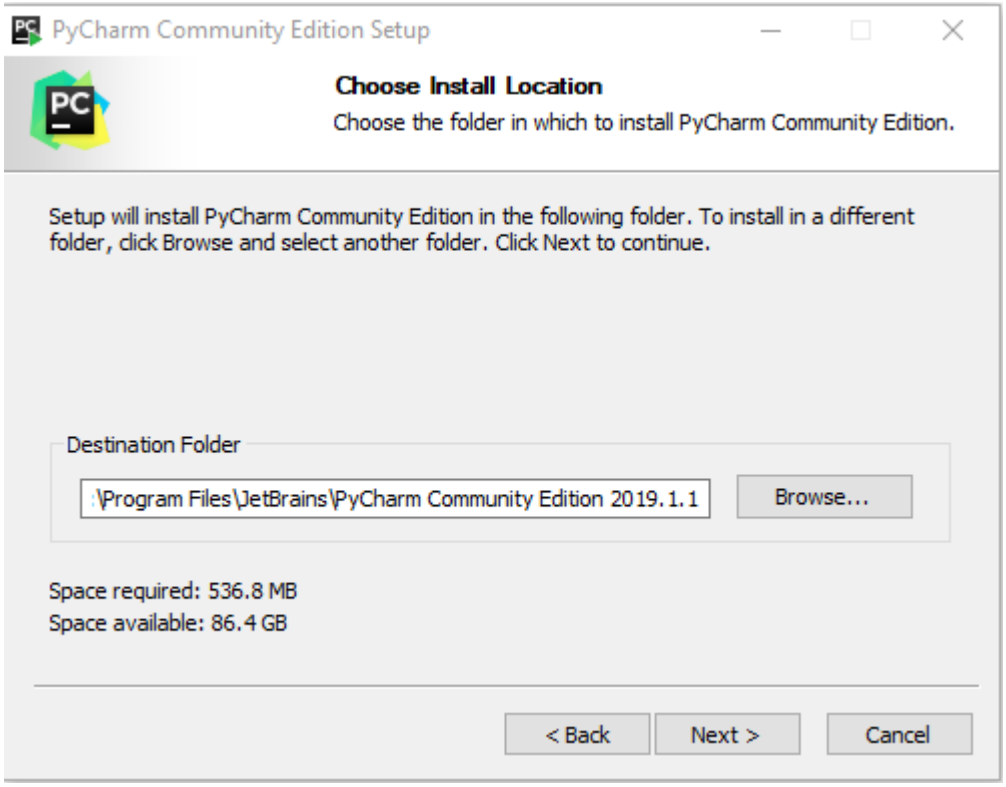
2.1 PyCharm Installation:

a) Go to the link https://www.jetbrains.com/pycharm/download/#section=windows and download the community edition.

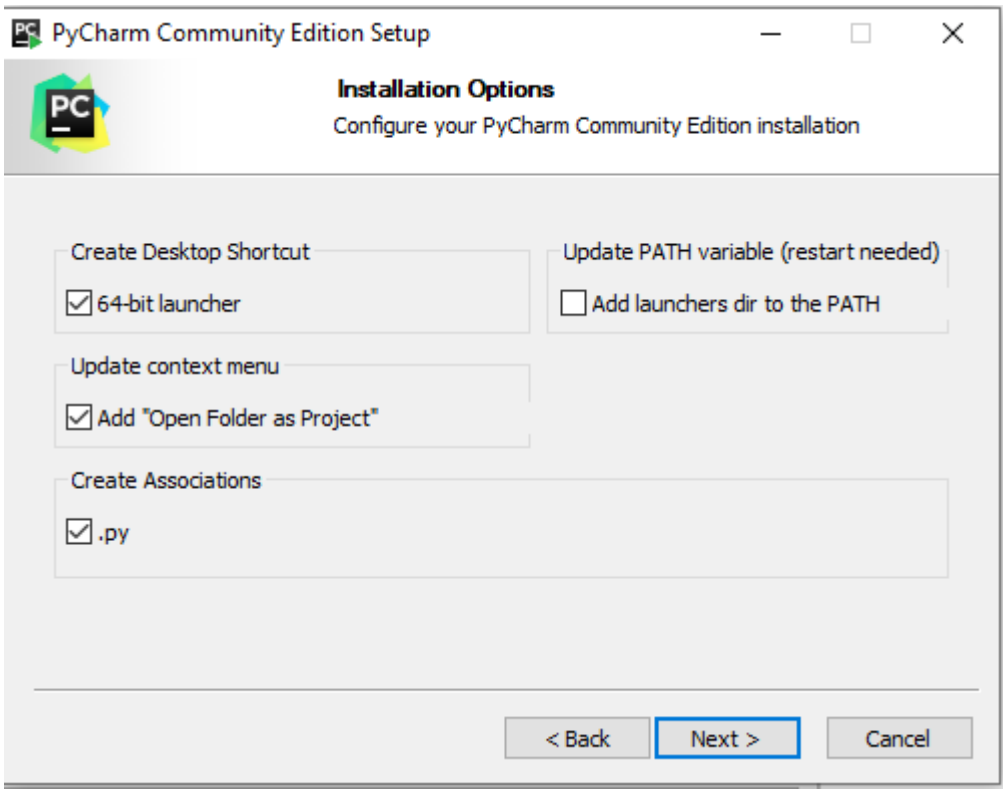
b) Double click on the installation file to start the installation process and click next to continue.



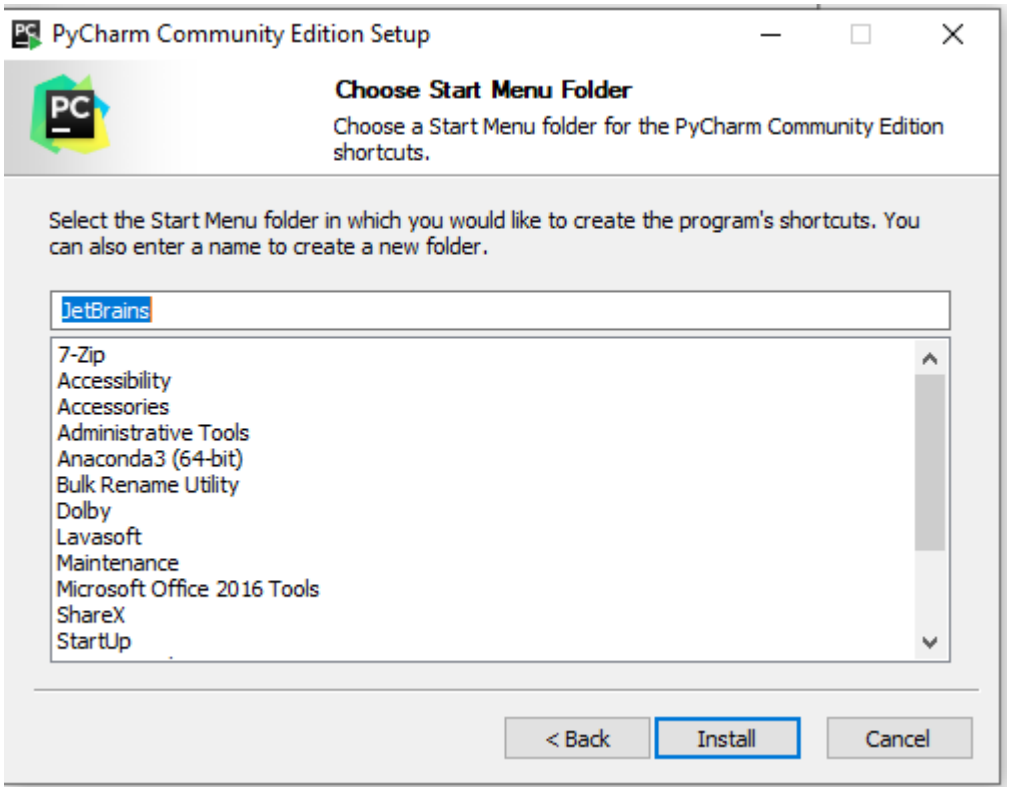
c) Select the directory to install PyCharm and then click next.



d) Check the appropriate checkboxes and then click next.

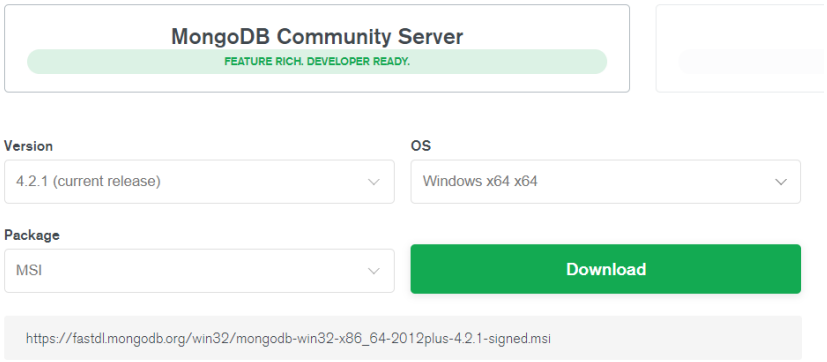


e) Choose the name of the start menu folder and then click on install to finish the installation.



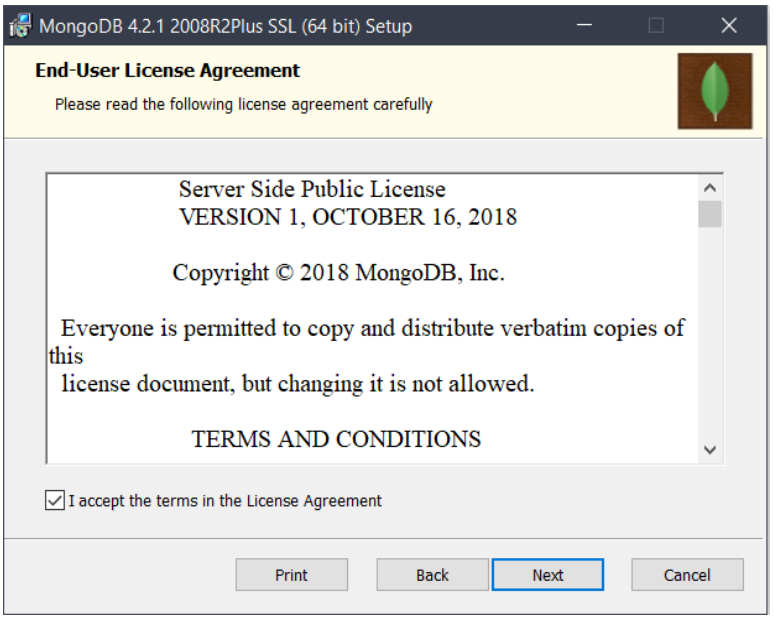
2.2 MongoDB Installation:

1. Go to the page: https://www.mongodb.com/download-center/community and select the MongoDB installation to download based on your operating system.

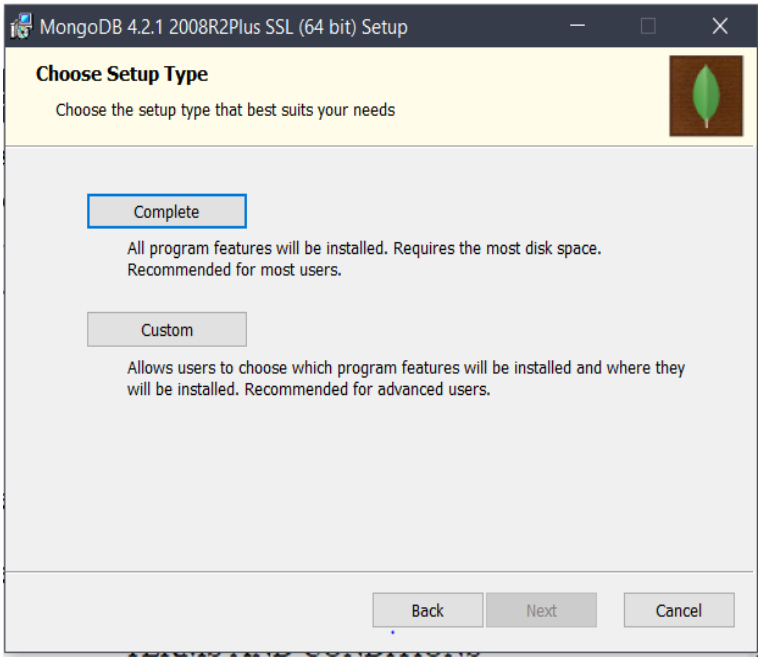


2. After the installer gets downloaded, double click on the installer file to start installing the application.

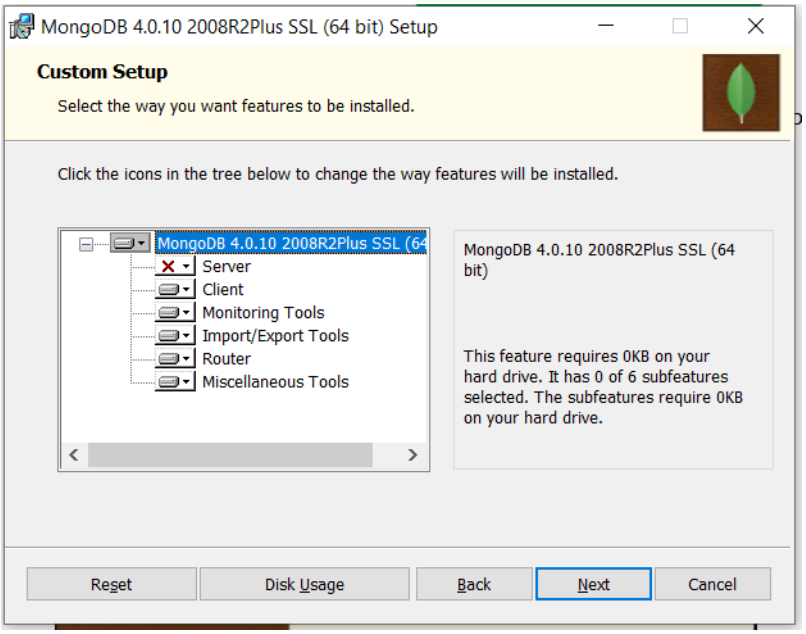
3. Click on the next button to move to the next step and accept the agreement.



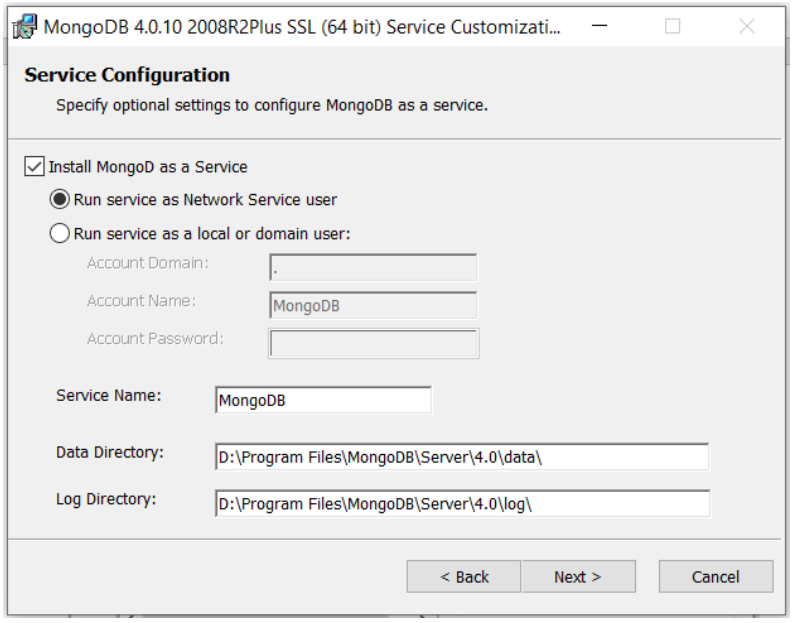
4. Select the type of installation:



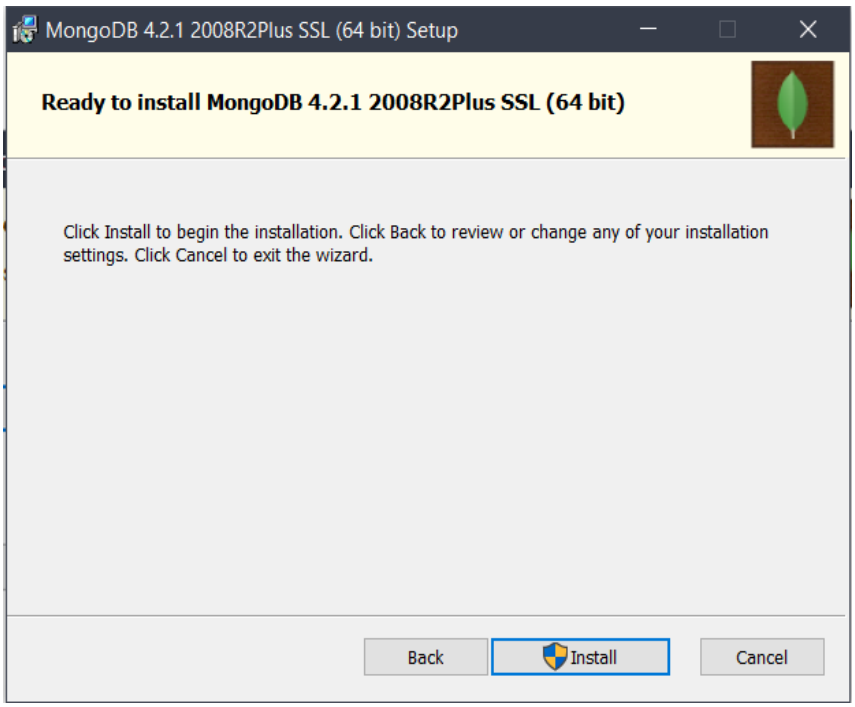
5. Select the features to install.



6. Click on next and then configure/customize the way you want the application to be installed.

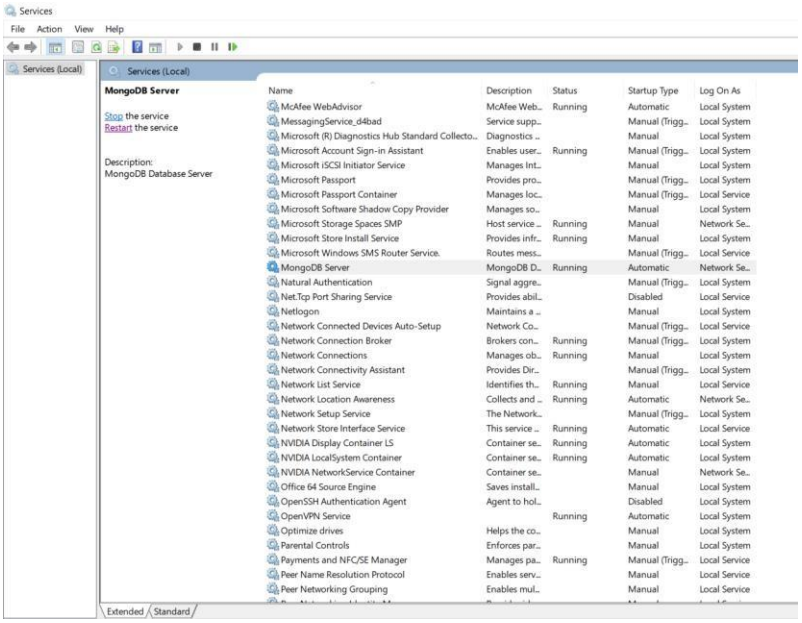


7. Click next and then click on install to start the MongoDB installation.

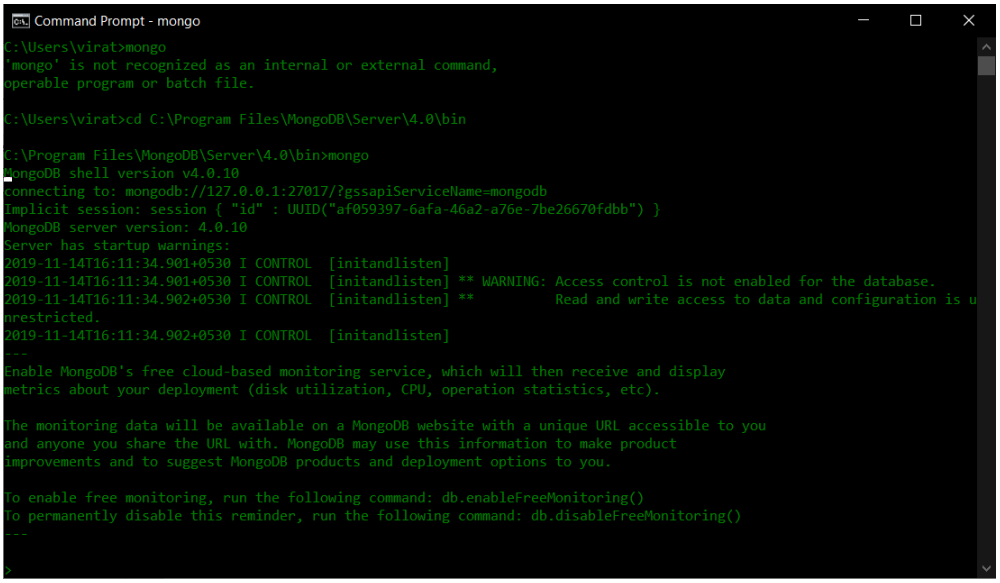


2.3 Starting MongoDB:

1. Go the services section and then start the MongoDB service if not already started.



2. Now, to check whether the database server is up or not, go to the bin directory of the MongoDB installation and run the 'mongo' command as shown. If the command runs successfully, it means that the server is up and running, and we can proceed.

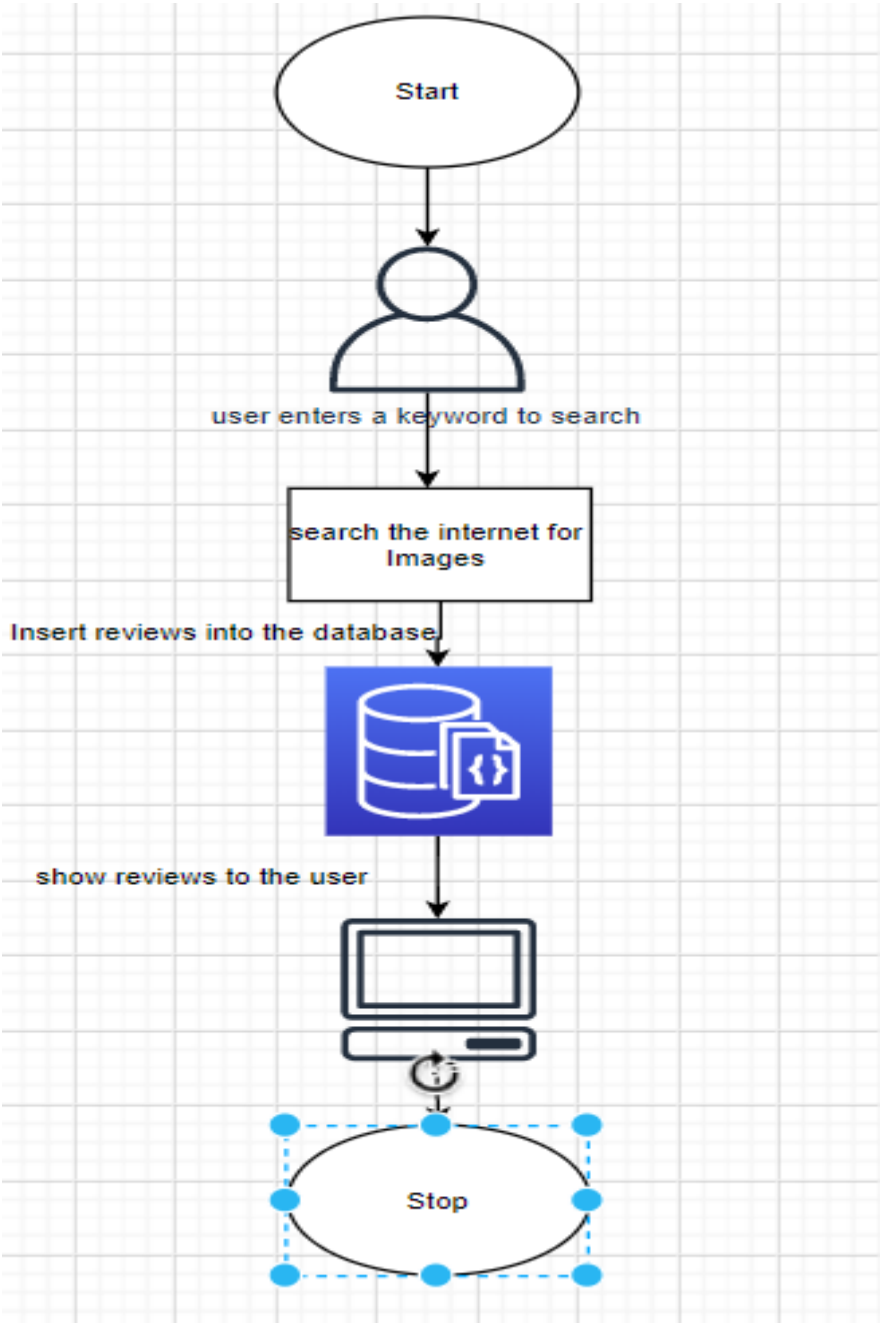

Command Prompt - mongo
C:\Users\virat>mongo
'mongo' is not recognized as an internal or external command,
operable program or batch file.
C:\Users\virat>cd C:\Program Files\MongoDB\Server\4.0\bin
C:\Program Files\MongoDB\Server\4.0\bin>mongo
MongoDB shell version v4.0.10
connecting to: mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("af059397-6afa-46a2-a76e-7be26670fdbb") }
MongoDB server version: 4.0.10
Server has startup warnings:
2019-11-14T16:11:34.901+0530 I CONTROL [initandlisten]
2019-11-14T16:11:34.901+0530 I CONTROL [initandlisten] ** WARNING: Access control is not enabled for the database.
2019-11-14T16:11:34.902+0530 I CONTROL [initandlisten] ** Read and write access to data and configuration is u
nrestricted.
2019-11-14T16:11:34.902+0530 I CONTROL [initandlisten]

Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).
The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.
To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()

>

3. Application Architecture:

The architecture of the application is:



4. Implementation in Python:

Note: I have used PyCharm as an IDE for this documentation

1. Let's create a folder called 'ImageScrapper' on our local machines.
2. Inside that folder, let's create two more folders called 'imagescrapper','imagescrapperservice','imagescrapperutils', 'static', and 'templates'

to hold the code for the UI of our application. Inside 'static', let's create a folder called 'css' for keeping the stylesheets for our UI.

3. Inside 'imagescrapper','imagescrapperservice', and 'imagescrapperutils' create the files 'ImageScrapper.py', 'ImageScrapperService.py' and 'ImageScrapperUtils.py' respectively. The files are attached here for reference.

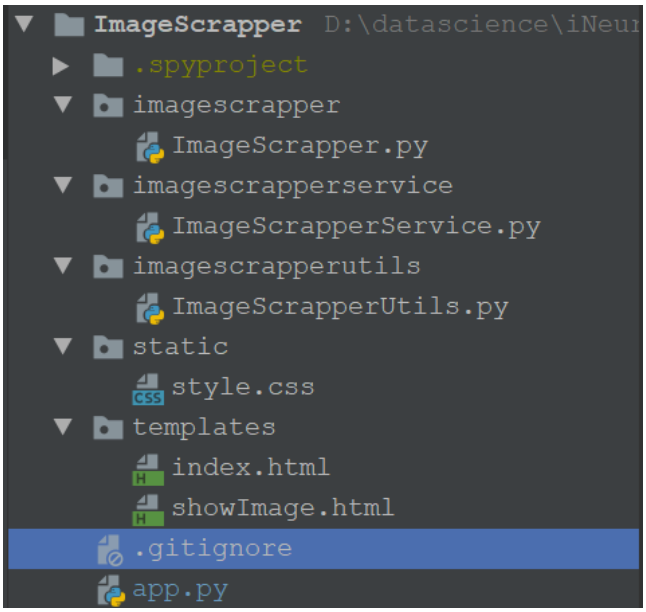
4. Let's create a file called 'app.py' inside the 'ImageScrapper' folder. The file is attached here for reference.

5. Inside the folder 'css', create the file: 'style.css'. The file is attached here for reference.

6. Inside the folder 'templates', create three HTML files called: 'index.html', and 'showImage.html'. The files are attached here for reference.

- index.html→ Home page of our application.
- showImage.html→ Page to show the images for the searched keyword.

7. Now, the folder structure should look like:



8. Now, let's understand the flow:

a) When the application starts, the user sees the page called 'index.html'.
b) The user enters the search keyword into the search box and presses the submit button.
c) The application now searches for images and shows the result on the 'showImage.html' page.

9. Understanding 'app.py'.
   a) Import the necessary libraries:

```
from flask_cors import CORS,cross_origin
from imagescrapperservice.ImageScrapperService import
ImageScrapperService
from imagescrapper.ImageScrapper import ImageScrapper
from flask import Flask, render_template, request,jsonify
```

   b) Initialize the flask app

```
app = Flask(__name__)  # initialising the flask app with the name
'app'
```

   c) Creating the routes to redirect the control inside the application itself. Based on the route path, the control gets transferred inside the application. Let's understand the various routes:

      i. The route for redirecting to home page

```
@app.route('/')  # route for redirecting to the home
page
@cross_origin()
def home():
    return render_template('index.html')
```

      ii. Showing the images on the screen once our parser successfully gives the list of images.

```
@app.route('/showImages') # route to show the images on
a webpage
@cross_origin()
def show_images():
    scraper_object=ImageScrapper() #Instantiating the
object of class ImageScrapper

list_of_jpg_files=scraper_object.list_only_jpg_files('st
atic') # obtaining the list of image files from the
static folder
    print(list_of_jpg_files)
    try:
        if(len(list_of_jpg_files)>0): # if there are
images present, show them on a wen UI
            return
render_template("showImage.html",user_images =
list_of_jpg_files)
        else:
            return "Please try with a different string" #
show this error message if no images are present in
```

```
the static folder
    except Exception as e:
        print('no Images found ', e)
        return "Please try with a different string"
```

iii. The route to actually scrape the web for images and then preparing the list of images and then calling the method in step (ii) to show the images to the user.

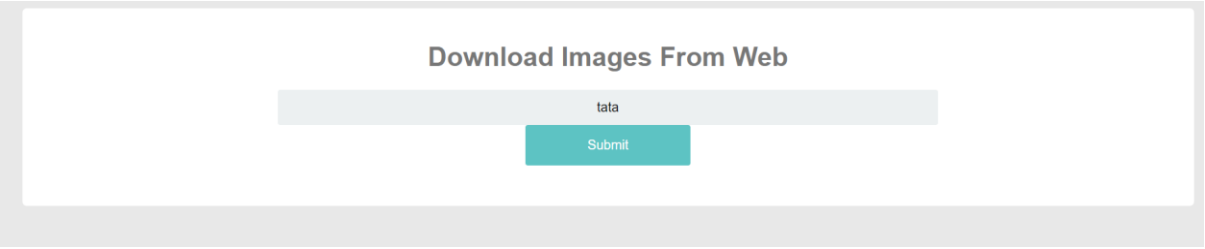
```
@app.route('/searchImages', methods=['GET','POST'])
def searchImages():
    if request.method == 'POST':
        print("entered post")
        keyWord = request.form['keyword'] # assigning
the value of the input keyword to the variable keyword

    else:
        print("did not enter post")
    print('printing = ' + keyWord)

    scraper_object = ImageScrapper() # instantiating the
class
    list_of_jpg_files =
scraper_object.list_only_jpg_files('static') # obtaining
the list of image files from the static folder

scraper_object.delete_existing_image(list_of_jpg_files) #
deleting the old image files stored from the previous
search
    # splitting and combining the keyword for a string
containing multiple words
    image_name = keyWord.split()
    image_name = '+'.join(image_name)
    # adding the header metadata
    header = {
        'User-Agent': "Mozilla/5.0 (Windows NT 6.1;
WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/43.0.2357.134 Safari/537.36"}

    service = ImageScrapperService  # instantiating the
object of class ImageScrapperService
    # (imageURLList, header, keyWord, fileLoc)
    masterListOfImages = service.downloadImages(keyWord,
header) # getting the master list from keyword
    imageList = masterListOfImages[0] # extracting the
list of images from the master list
    imageTypeList = masterListOfImages[1] # extracting
the list of type of images from the masterlist

    response = "We have downloaded ", len(imageList),
"images of " + image_name + " for you"

    return show_images() # redirect the control to the
show images method
```

iv. The route to send the list of image URLs when the API is not called from a web browser.

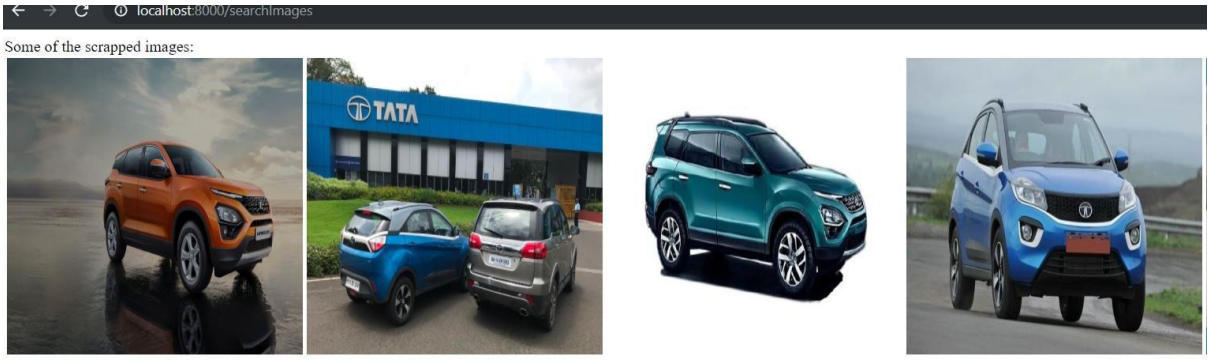
```
@app.route('/api/showImages', methods=["GET",'POST'])
# route to return the list of file locations for API
calls
@cross_origin()
def get_image_url():
    if request.method == 'POST':
        print("entered post")
        keyWord =  request.json['keyword'] # assigning
the value of the input keyword to the variable keyword

    else:
        print("Did not enter post")
    print('printing = ' + keyWord)
    # splitting and combining the keyword for a string
containing multiple words
    image_name = keyWord.split()
    image_name = '+'.join(image_name)
    # adding the header metadata
    header = {
        'User-Agent': "Mozilla/5.0 (Windows NT 6.1;
WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/43.0.2357.134 Safari/537.36"}

    service = ImageScrapperService # instantiating the
object of class ImageScrapperService
    url_enum = service.get_image_urls(keyWord, header)
# getting the URL enumeration
    url_list=[] # initializing and empty url list
    for i, (img, Type) in enumerate(url_enum[0:5]):
        # creating key value pairs of image URLs to be
sent as json
        dict={'image_url':img}
        url_list.append(dict)
    return jsonify(url_list) # send the url list in
JSON format
```

d) After this, we'll just run our python app on our local system, and it'll start scraping for images as shown below:

Home Page:



Search Results:

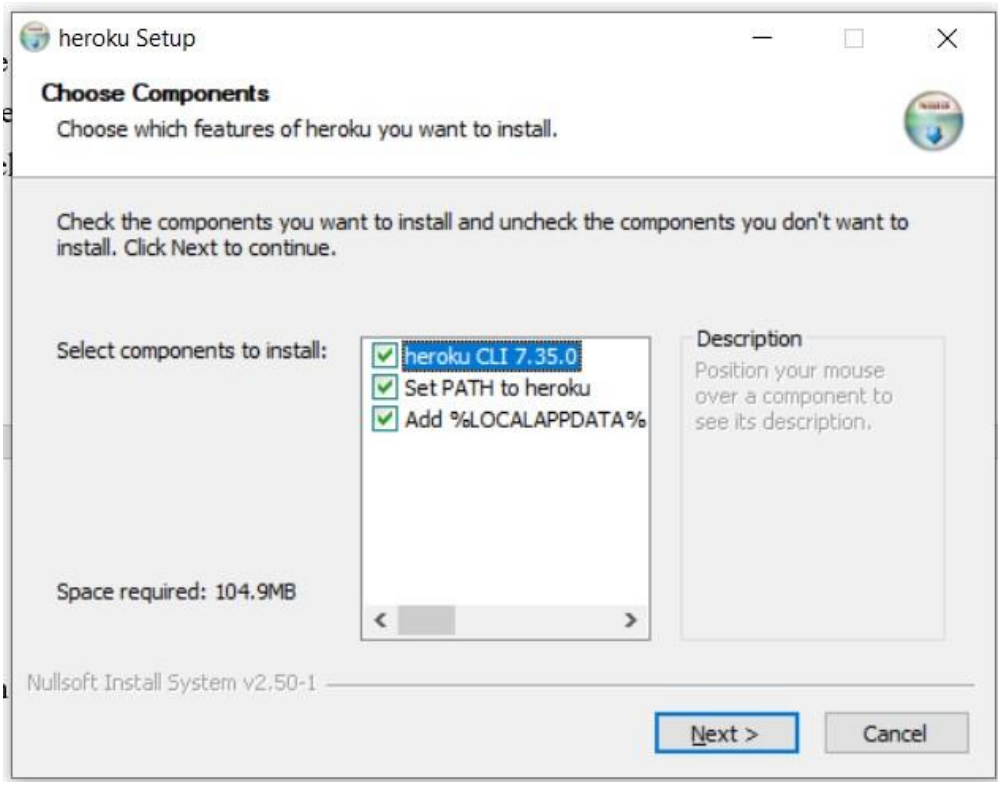


5. Heroku:

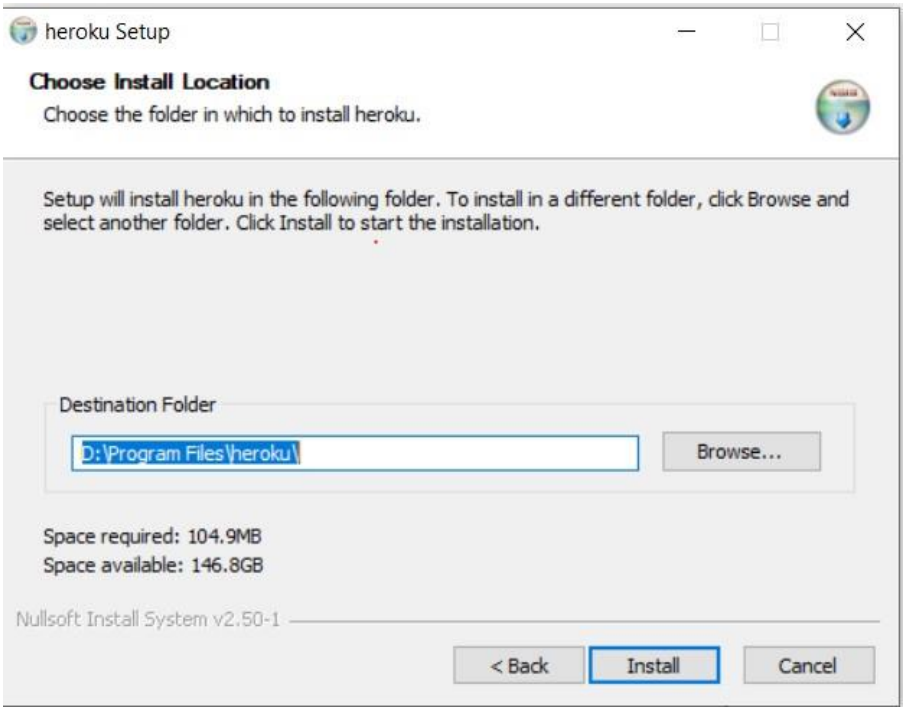
The Python app that we have developed is residing on our local machine. But to make it available to end-users, we need to deploy it to either an on-premise server or to a cloud service. Heroku is one such cloud service provider. It is free to use(till 5 applications). We'll deploy this application to the Heroku cloud, and then anybody with the URL can then consume our app.

6. Heroku Basics:

- We'll first go to heroku.com, and we'll create a new account if we already don't have one.
- We'll download and install the Heroku CLI from the Heroku website: https://devcenter.heroku.com/articles/heroku-cli.
- Double-click the installation file and the following window shall appear:



- Click on next and select the installation directory for the CLI.



- Click on install to complete the installation.

7. Steps before cloud deployment:

We need to change our code a bit so that it works unhindered on the cloud, as well.

a) Add a file called 'gitignore' inside the 'reviewScrapper' folder. This folder contains the list of the files which we don't want to include in the git repository. My gitignore file looks like:

```
.idea
```

As I am using PyCharm as an IDE, and it's provided by the Intellij Idea community, it automatically adds the .idea folder containing some metadata. We need not include them in our cloud app.

b) Add a file called 'Procfile' inside the 'reviewScrapper' folder. This folder contains the command to run the flask application once deployed on the server:

```
web: gunicorn app:app
```

Here, the keyword 'web' specifies that the application is a web application. And the part 'app:app' instructs the program to look for a flask application called 'app' inside the 'app.py' file. Gunicorn is a Web Server Gateway Interface (WSGI) HTTP server for Python.

c) Open a command prompt window and navigate to your 'reviewScrapper' folder. Enter the command 'pip freeze > requirements.txt'. This command generates the 'requirements.txt' file. My requirements.txt looks like:

```
beautifulsoup4==4.8.1
bs4==0.0.1
certifi==2019.9.11
Click==7.0
Flask==1.1.1
Flask-Cors==3.0.8
gunicorn==20.0.4
itsdangerous==1.1.0
Jinja2==2.10.3
MarkupSafe==1.1.1
numpy==1.17.4
opencv-python==4.1.2.30
Pillow==6.2.1
pymongo==3.9.0
requests==2.21.0
requests-oauthlib==1.2.0
six==1.13.0
soupsieve==1.9.5
Werkzeug==0.16.0
```

requirements.txt helps the Heroku cloud app to install all the dependencies before starting the webserver.

d) A change has been made to app.py. The part where the port number for local machine was provided has been commented and the part without port number has been uncommented to run on the cloud.

```
if __name__== "__main__":
    #app.run(host='127.0.0.1', port=8000) # port to run on local
machine
    app.run(debug=True) # to run on cloud
```

8. Heroku app creation and deploying the app to cloud:

a. After installing the Heroku CLI, Open a command prompt window and navigate to your 'reviewScrapper' folder.

b. Type the command `'heroku login'` to login to your heroku account as shown below:

```
D:\datascience\iNeuron\docs\reviewScrapper>heroku login
heroku: Press any key to open up the browser to login or q to exit:
```

c. After logging in to Heroku, enter the command `'heroku create'` to create a heroku app. It will give you the URL of your Heroku app after successful creation.

d. Before deploying the code to the Heroku cloud, we need to commit the changes to the local git repository.

e. Type the command `'git init` to initialize a local git repository as shown below:

```
D:\datascience\iNeuron\docs\reviewScrapper>git init
```

f. Enter the command `'git status'` to see the uncommitted changes

g. Enter the command `'git add .'` to add the uncommitted changes to the local repository.

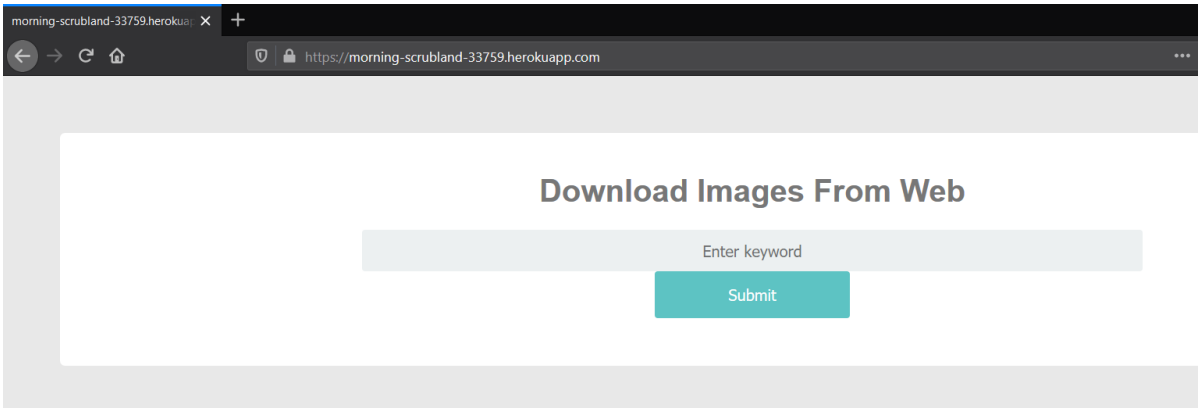
h. Enter the command `'git commit -am "make it better"'` to commit the changes to the local repository.

i. Enter the command `'git push heroku master'` to push the code to the heroku cloud.

j. After deployment, heroku gives you the URL to hit the web API.

k. Once your application is deployed successfully, enter the command `'heroku logs --tail'` to see the logs.

Final Result:



Thank You!