The Machine learning world currently sees Data Scientists (DS) performing one or both of the following 2 prominent roles:

1. Where a DS receives a data dump, applies some Machine learning algo on the data and reports back the results in the form of some presentation or report.
2. Where the DS creates a usable piece of software for the stakeholders to consume the machine learning models.

In this blog post, I’m attempting to display an example approach to the second aspect of a DS’s job i.e., creating some software that can be used by the stakeholders. Specifically, we would create a web-service that can be queried to obtain the predictions from a machine learning model. The post is mostly intended for machine learning practitioners who would like to go beyond only developing models.

**Tech-stack**: Python, [Flask](http://flask.pocoo.org/), [Docker](https://www.docker.com/), [AWS ec2](https://aws.amazon.com/ec2/)

The workflow can be broken down into following basic steps:

1. Training a machine learning model on a local system.
2. Wrapping the inference logic into a flask application.
3. Using docker to containerize the flask application.
4. Hosting the docker container on an AWS ec2 instance and consuming the web-service.

Let’s start with the first step.

Training a machine learning model on a local system

We need **some**machine learning model that we can wrap in a web-service. For demo purpose, I chose a logistic regression model to do multiclass classification on iris dataset The model was trained on a local system using python 3.6.

Using the familiar scikit-learn, the above mentioned model can be trained quickly. For model development, refer the notebook ‘Model\_training.ipynb’ in the [github repo](https://github.com/tanujjain/deploy-ml-model) for this blog. There are only 2 important aspects of model development that I would like to highlight:

1. The model file generated after training is stored as a pickle file which is a serialized format for storing objects. (In the repo, the file is named ‘iris\_trained\_model.pkl’)
2. The inference call (.predict()) call requires 4 features per test sample in the form of a numpy array.

Wrapping the inference logic into a flask web service

Now that we have the trained model file, we are ready to query the model to get a class label for a test sample. The inference is as simple as calling a predict() function on the trained model with the test data. However, we would like to build the inference as a web-service. For this purpose, we would use Flask.

Flask is a powerful python microwebserver framework that allows us to build REST API based web-services quickly with minimum configuration hassle. Let’s dive into the code:

a. First, let’s define a simple function to load the trained model file.

Here, we define a global variable called ‘model’ and populate it within the load\_model() function. The purpose of using a global variable will become clear shortly.

b. Next, we instantiate a Flask object called ‘app’:

c. Now, we define a home endpoint, which when hit, returns a ‘Hello World!’ message.

Notice the use of app.route decorator.

d. Now, we define a ‘predict’ endpoint. The endpoint accepts a ‘POST’ request wherein the test data on which we wish to get a prediction is received by the endpoint. Keeping things simple, the function works only when a single test sample needs to be predicted

e. Finally, declare the main function:

Here, a call to the load\_model() function ensures that the variable ‘model’ is populated with the trained model attributes (and hence the need for a global model variable). So, there is no need to load the model repeatedly with every call to the predict endpoint. This allows the web-service to be quick. The response is returned as a string which is the predicted class label.

The complete flask specific code is as below:

At this point, the web-service is ready to be run locally. Let’s test this.

Execute the command python app.py from the terminal. Go to the browser and hit the url 0.0.0.0:80 to get a message Hello World! displayed. This corresponds to the home endpoint return message.

Next, let’s test if we can get predictions using this web-service using the following curl post request on the terminal:

curl -X POST \  
 0.0.0.0:80/predict \  
 -H 'Content-Type: application/json' \  
 -d '[5.9,3.0,5.1,1.8]'

The curl request posts one test sample [5.9,3.0,5.1,1.8]to our web-server and returns a single class label.

Using docker to containerize the flask service

Up to this point, we have a web-service that runs locally. Our ultimate intention is to be able to run this piece of code on a cloud virtual machine.

In the Software Development world, there is a famous justification given by a developer whose code was found to be broken by a tester: ‘*But it worked on my machine!*’. The problem portrayed here can usually be attributed to a lack of consistent environment that runs the software across different machines. Ideally, our code itself should be independent of the underlying machine/OS that runs it. Containerization allows developers to provide such isolation.

*How is it important here?*

Our intention is to run our web-service on a cloud VM. The cloud VM itself may run any OS. Containerization of our web-server allows us to avoid the trouble of running into environment related issues. If the containerized code works on one machine, it will surely run on another irrespective of the characteristics of the machine. Docker is the most famous containerized technology out there at this point and we will be using the same here.

Let’s dive into the Dockerfile that comprises a set of instructions for docker daemon to build the docker image.

We pull the base docker image from python dockerhub repo on which our specific build instructions are executed. The COPY commands are simply taking specific files from the current folder and copying them over to a folder called ‘deploy’ within the docker image we are trying to build. In addition to app.py and model file, we also need a requirements file that lists specific versions of python packages we use to run our code. The WORKDIR command changes the working directory to ‘deploy/’ within the image. We then issue a RUN command to install specific python packages using the requirements file. The EXPOSE command makes the port 80 accessible to the outside world (our flask service runs on port 80; we need this port inside the container to be accessible outside the container).

Issue the build command to end up with a docker image:

docker build -t app-iris .

(Don’t forget the period at the end of the command).

Use command ‘docker images’ to see a docker image with a docker repository named ‘app-iris’ created. (Another repository named python will also be seen since it is the base image on top of which we build our custom image.)

Now, the image is built and ready to be run. We can do this using the command:

docker run -p 80:80 app-iris .

The above commands uses -p flag to map port 80 of the local system to the port 80 of the docker container for the redirection of traffic on local HTTP port 80 to port 80 of the container. (If you are using local port 5000 instead of port 80, change the port mapping part of the command to 5000:80).

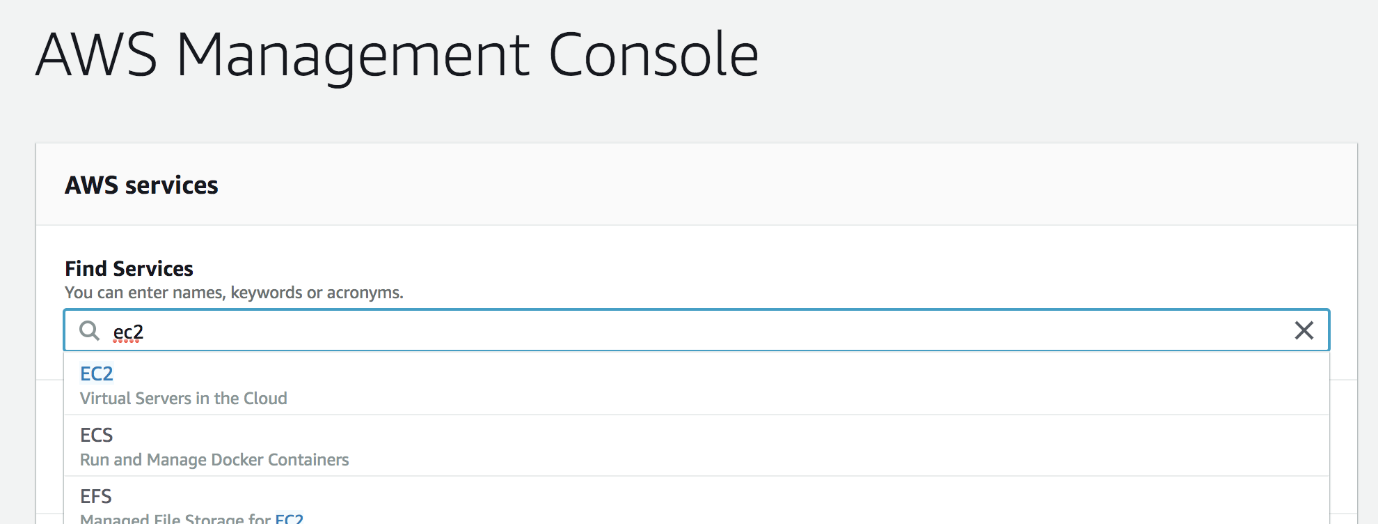
Hosting the docker container on an AWS ec2 instance

We already have a containerized application that works on our local system. Now, what if someone else wishes to consume the service? What happens if we need to build an architectural ecosystem around the service that needs to be available, automated and scalable? It’s easy to see that having a web-service running locally would be a very bad idea. So, we wish to host the web-service somewhere on the internet to fulfil the requirements we listed.

As a prerequisite, one needs to have an AWS account for using the ec2 instance. For new users, there are several AWS resources that are available for free for a period of 1 year (usually up to some limit). In this blog, I would be using a ‘*t2.micro*’ ec2 instance type which is free tier eligible. For users who have exhausted their AWS free-tier period, this instance costs around 1 cent(USD) per hour at the time of writing this blog; a super negligible amount to pay.

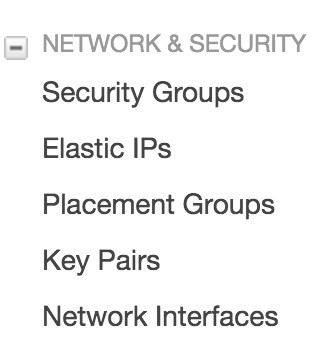
Let’s start with the process.

Log into the AWS management console and search for ec2 in the search bar to navigate to EC2 dashboard.



Search for ec2 on the aws management console

Look for the below pane, select ‘Key Pairs’ and create one.



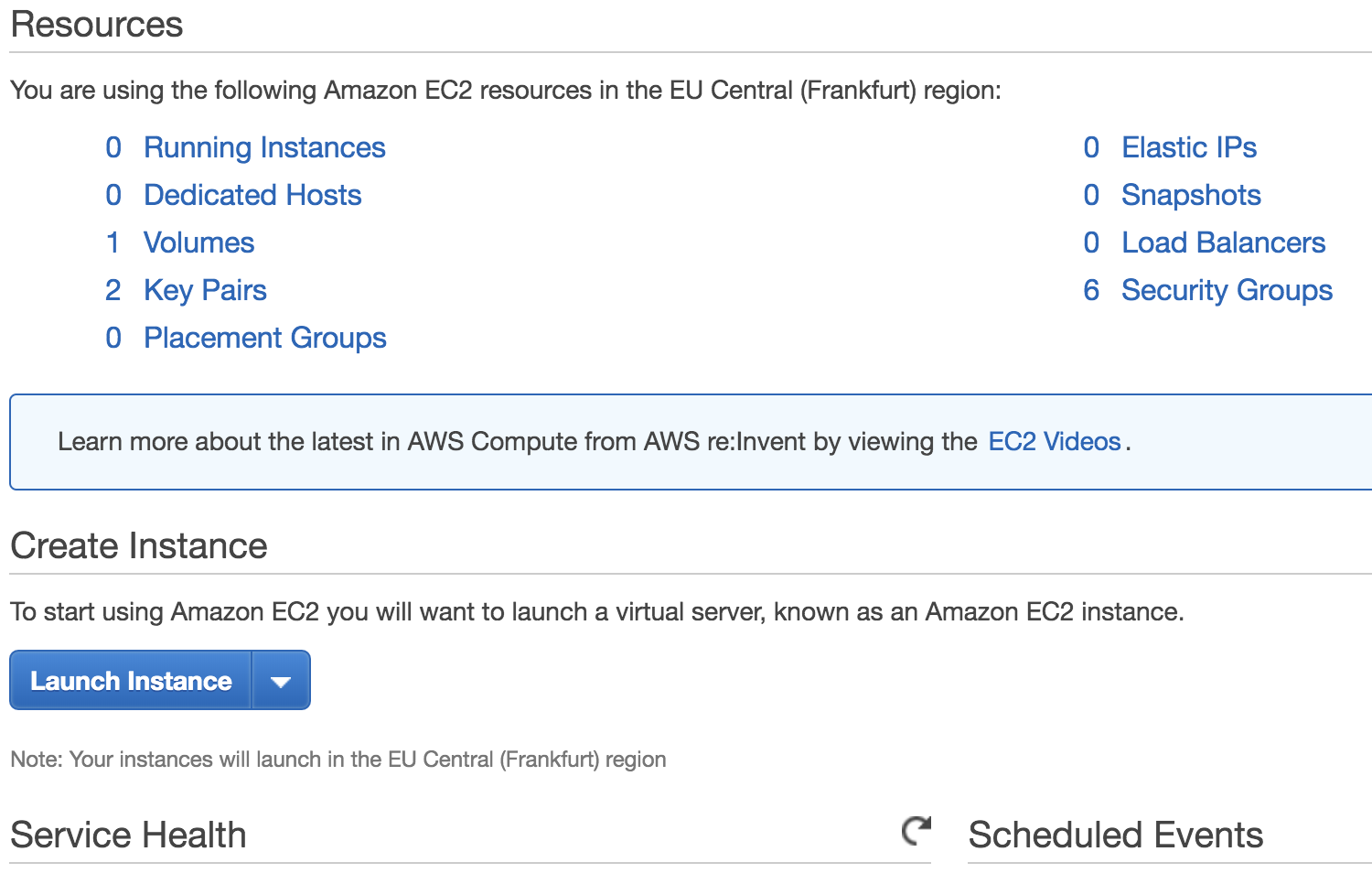
Select Key Pairs for looking at existing key pairs and creating new ones

This will download a ‘.pem’ file that is the key. Save this file somewhere safely. Now navigate to the location of this file on your system and issue the below command with key file name replaced by yours:

chmod 400 key-file-name.pem

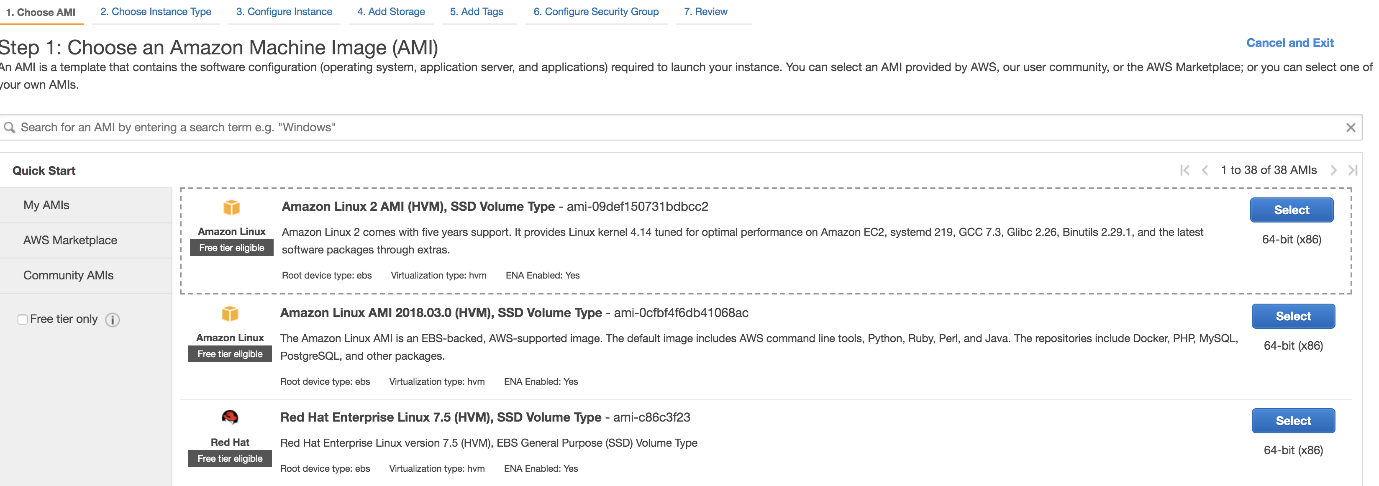
This commands changes permissions on your key pair file to [private](https://chmodcommand.com/chmod-400/). The use of key pairs will be explained later.

Next, click ‘Launch Instance’ on the EC2 dashboard:



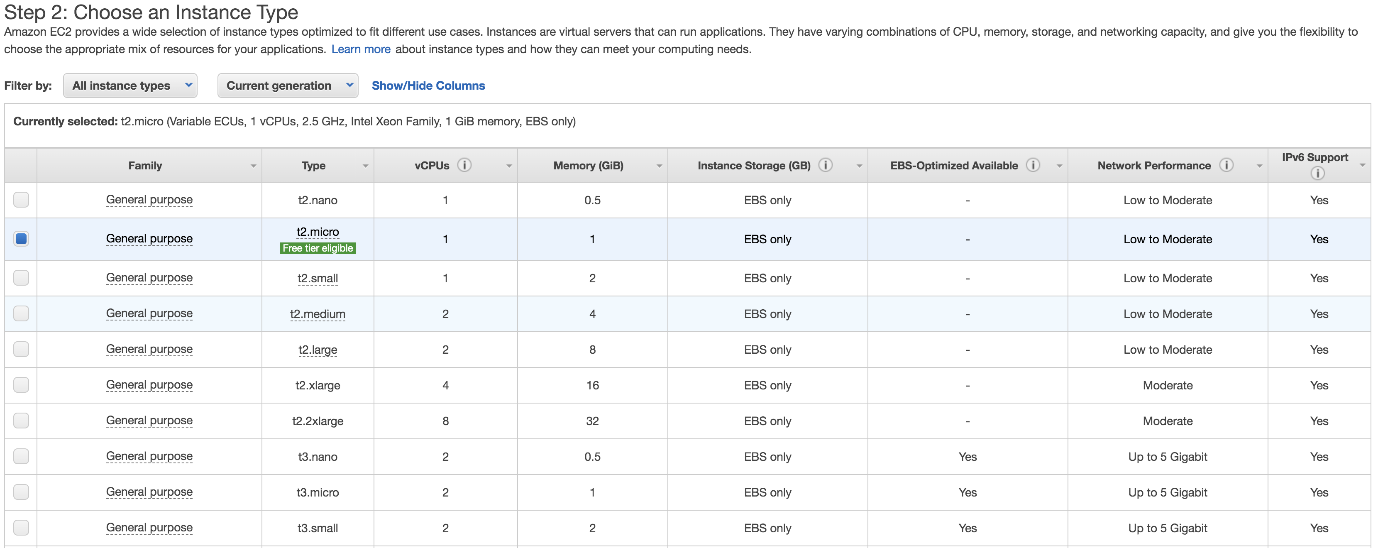
Launch ec2 instance

Choose the Amazon Machine Instance (AMI) from the list of options. An AMI determines the OS that the VM will be running (plus some other stuff we don’t care about at this point). For this blog, I chose ‘Amazon Linux 2 AMI’ which was the default selection.



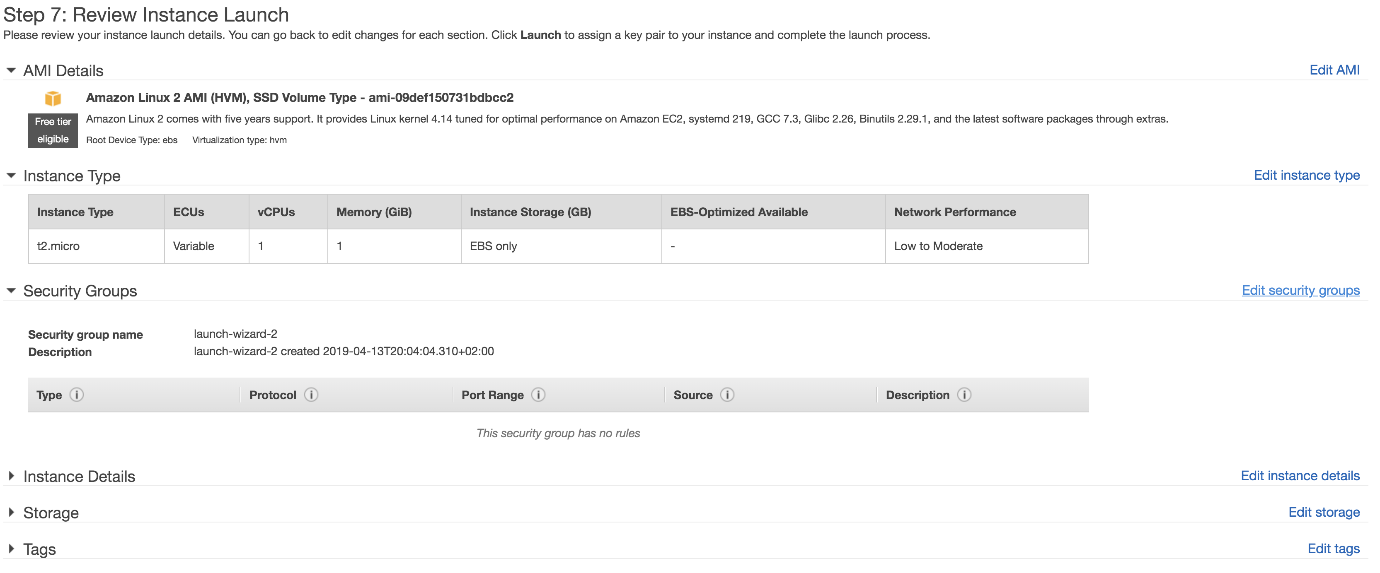
Choosing AMI

The next screen allows you to select the instance type. This is where the hardware part of the VM can be selected. As mentioned previously, we will work with ‘t2.micro’ instance.



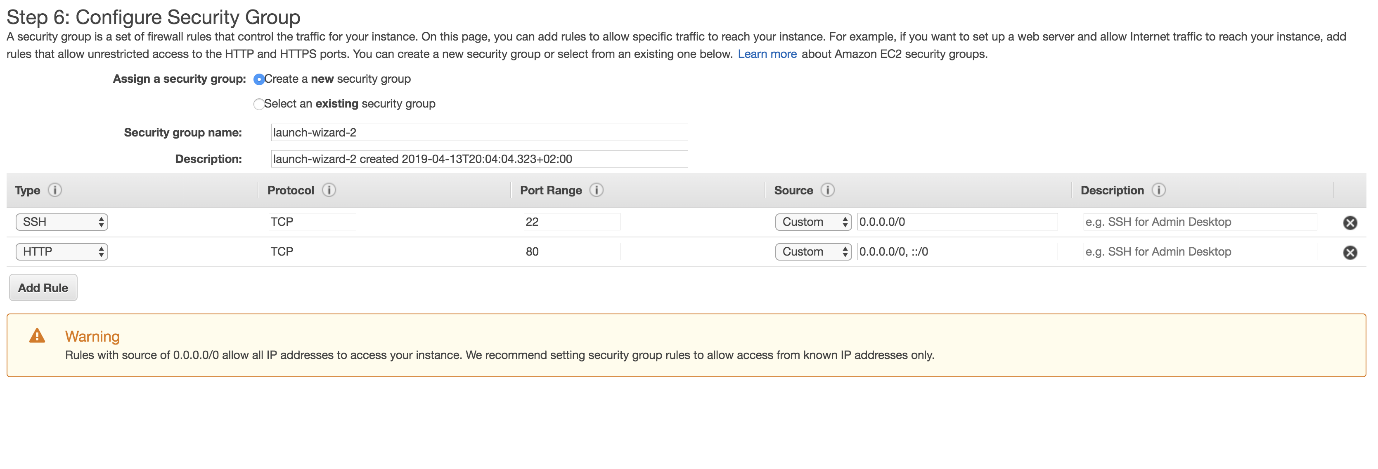
Selecting instance type

You can select ‘Review and Launch’ that takes you to ‘Step 7: Review Instance Launch’ screen. Here, you need to click the ‘Edit Security Groups’ link:



Security Groups

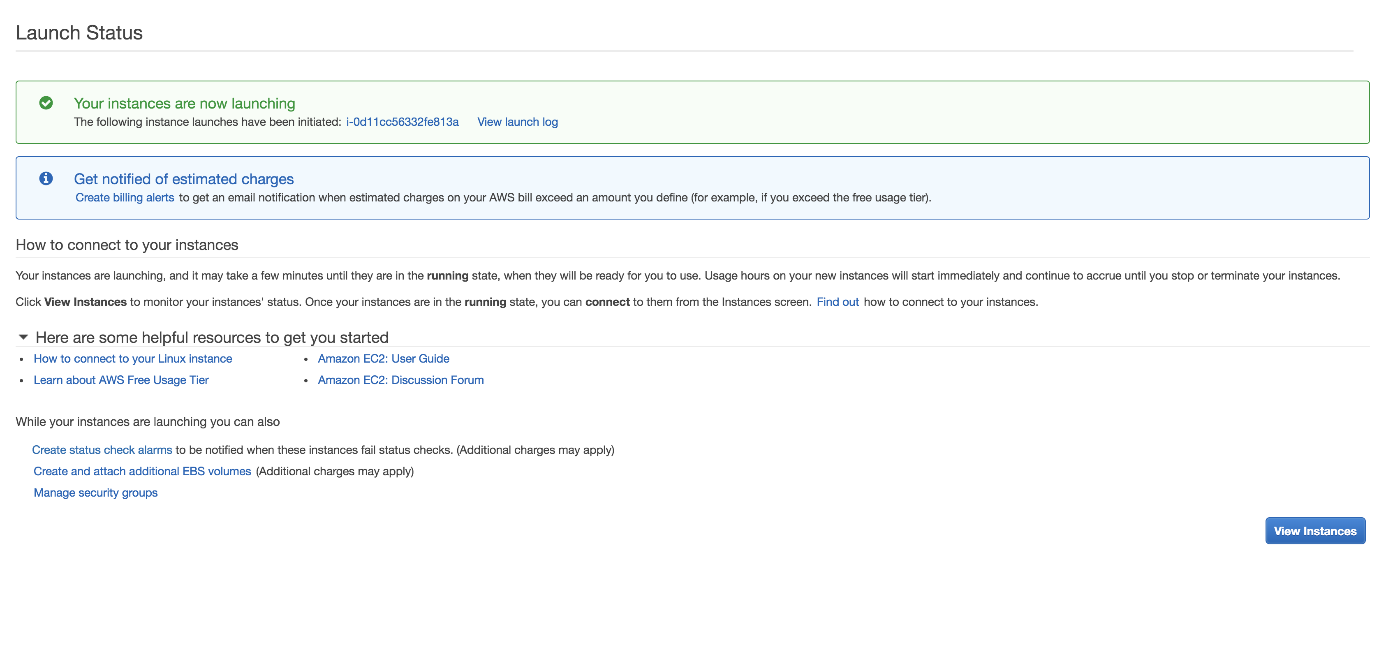
You now have to modify the security group to allow HTTP traffic on port 80 of your instance to be accessible by the outside world. This can be done by creating a rule. At the end, you should end up with such a screen:



Adding HTTP rule to security group

In the absence of this rule, your web-service will never be reachable.Clicking on the ‘Launch’ icon will lead to a pop up seeking a confirmation on having a key-pair. Use the name of the key pair that was generated earlier and launch the VM.

You would be redirected to a Launch screen:



Launch Status of ec2 instance

Use the ‘View Instance’ button to navigate to a screen that displays the ec2 instance being launched. When the instance state turns to ‘running’, then it is ready to be used.

We will now ssh into the ec2 machine from our local system terminal using the command with the field public-dns-name replaced with your ec2 instance name (of the form: [ec2–x–x–x–x.compute-1.amazonaws.com](mailto:ec2-user@ec2-198-51-100-1.compute-1.amazonaws.com)) and the path of the key pair pem file you saved earlier.

ssh -i /path/my-key-pair.pem [ec2-user@](mailto:ec2-user@ec2-198-51-100-1.compute-1.amazonaws.com)public-dns-name

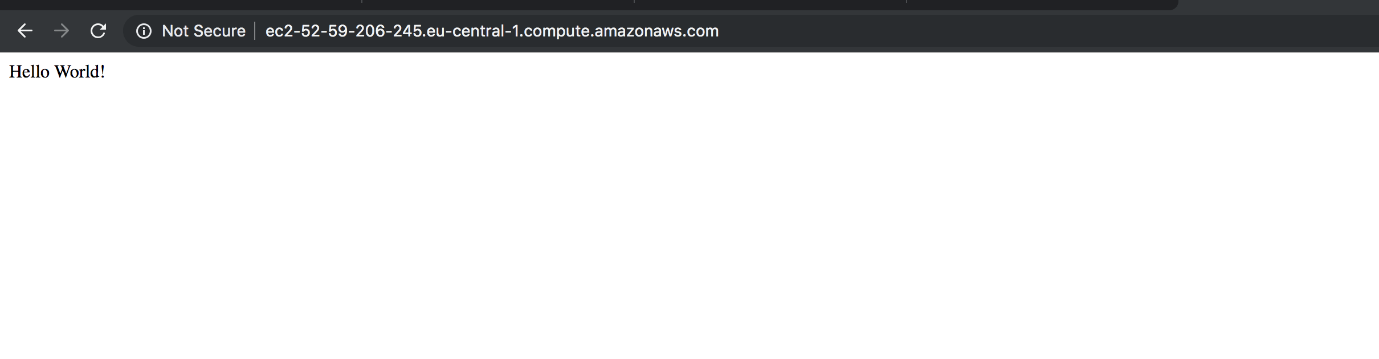
Log out of the ec2 instance using the ‘exit’ command and log back in again using the ssh command. Check if docker works by issuing the ‘docker info’ command. Log out again or open another terminal window.

Now let’s copy the files we need to build the docker image within the ec2 instance. Issue the command from your local terminal (not from within ec2):

scp -i /path/my-key-pair.pem file-to-copy [ec2-user@](mailto:ec2-user@ec2-52-59-206-245.eu-central-1.compute.amazonaws.com)public-dns-name:/home/ec2-user

We would need to copy requirements.txt, app.py, trained model file and Dockerfile to build the docker image as was done earlier. Log back into the ec2 instance and issue ‘*ls*’ command to see if the copied files exist. Next, build and run the docker image using the exact same commands that were used in the local system (Use port 80 at all locations in the code/commands this time).

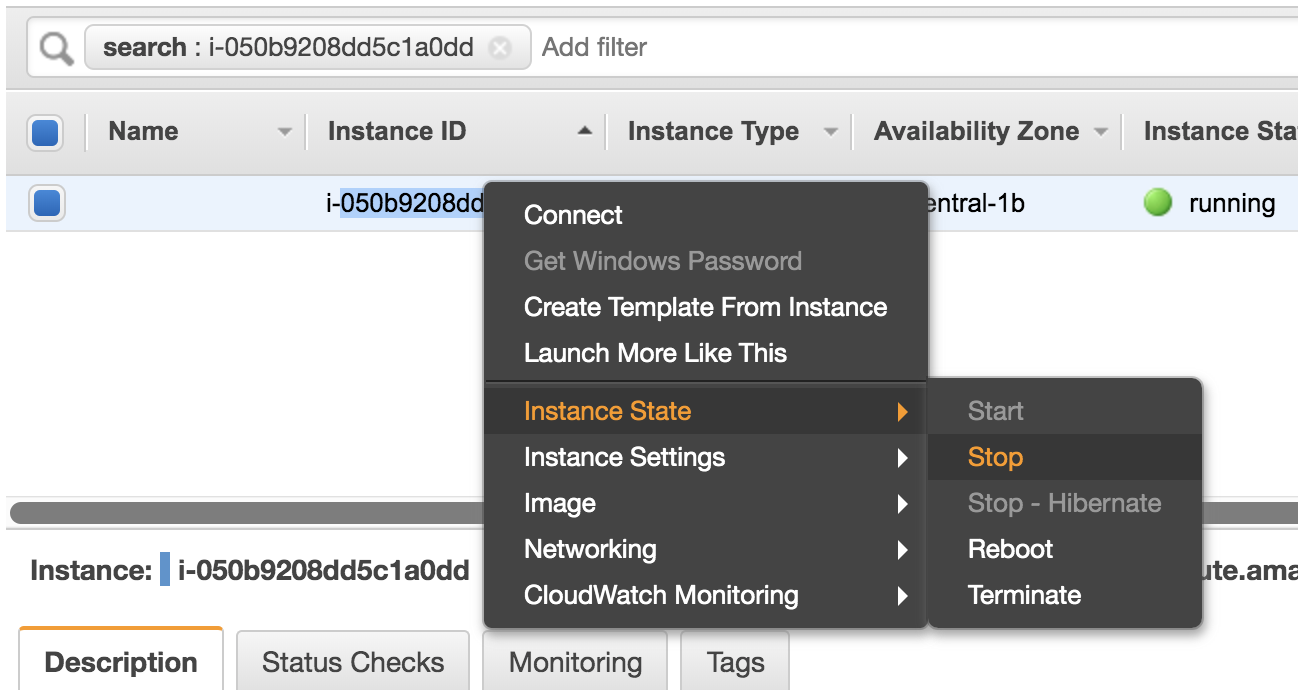
Hit the home endpoint from your browser using the public dns name to see the familiar ‘*Hello World!*’ message:



Home endpoint works from the browser (I used my ec2 public-dns-name in the address bar)

Now send a curl request to your web-service from local terminal with your test sample data after replacing the public-dns-name by yours:

curl -X POST \  
public-dns-name:80/predict \  
-H 'Content-Type: application/json' \  
-d '[5.9,3.0,5.1,1.8]'



Stop or terminate the ec2 instance to avoid getting charged