Programming Computer Vision Awesome Guides

Deployment could be easy—A Data Scientist's Guide to deploy an Image detection FastAPI API using Amazon ec2

By Rahul Agarwal 04 August 2020

Just recently, I had written a simple tutorial on FastAPI, which was about simplifying and understanding how APIs work, and creating a simple API using the framework.

That post got quite a good response, but the most asked question was how to deploy the FastAPI API on ec2 and how to use images data rather than simple strings, integers, and floats as input to the API.

I scoured the net for this, but all I could find was some undercooked documentation and a lot of different ways people were taking to deploy using NGINX or ECS. None of those seemed particularly great or complete to me.

So, I tried to do this myself using some help from FastAPI documentation . In this post, we will look at predominantly four things:

- Setting Up an Amazon Instance
- Creating a FastAPI API for Object Detection
- Deploying FastAPI using Docker
- An End to End App with UI

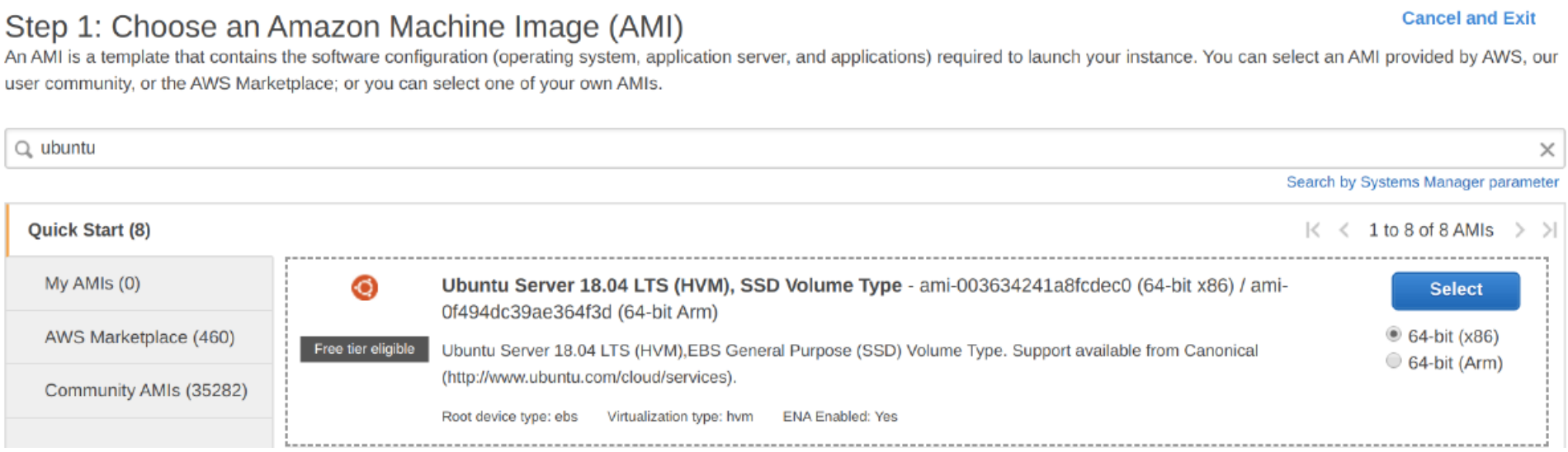
So, without further ado, let's get started.

You can skip any part you feel you are versed with though I would expect you to go through the whole post, long as it may be, as there's a lot of interconnection between concepts.

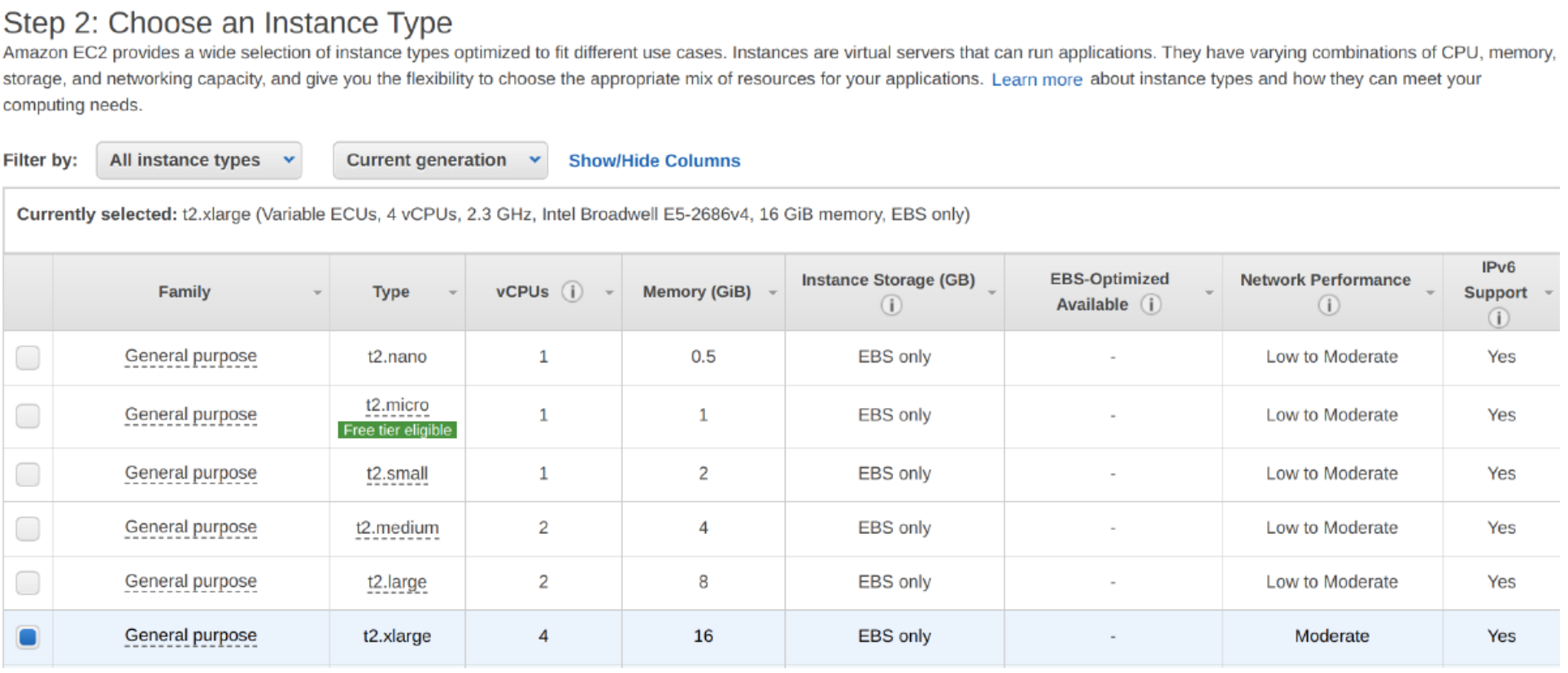
1. Setting Up Amazon Instance

Before we start with using the Amazon ec2 instance, we need to set one up. You might need to sign up with your email ID and set up the payment information on the AWS website . Works just like a single sign-on. From here, I will assume that you have an AWS account and so I am going to explain the next essential parts so you can follow through.

- Go to AWS Management Console using https://us-west-2.console.aws.amazon.com/console .
- On the AWS Management Console, you can select “Launch a Virtual Machine.” Here we are trying to set up the machine where we will deploy our FastAPI API.
- In the first step, you need to choose the AMI template for the machine. I am selecting the 18.04 Ubuntu Server since Ubuntu.

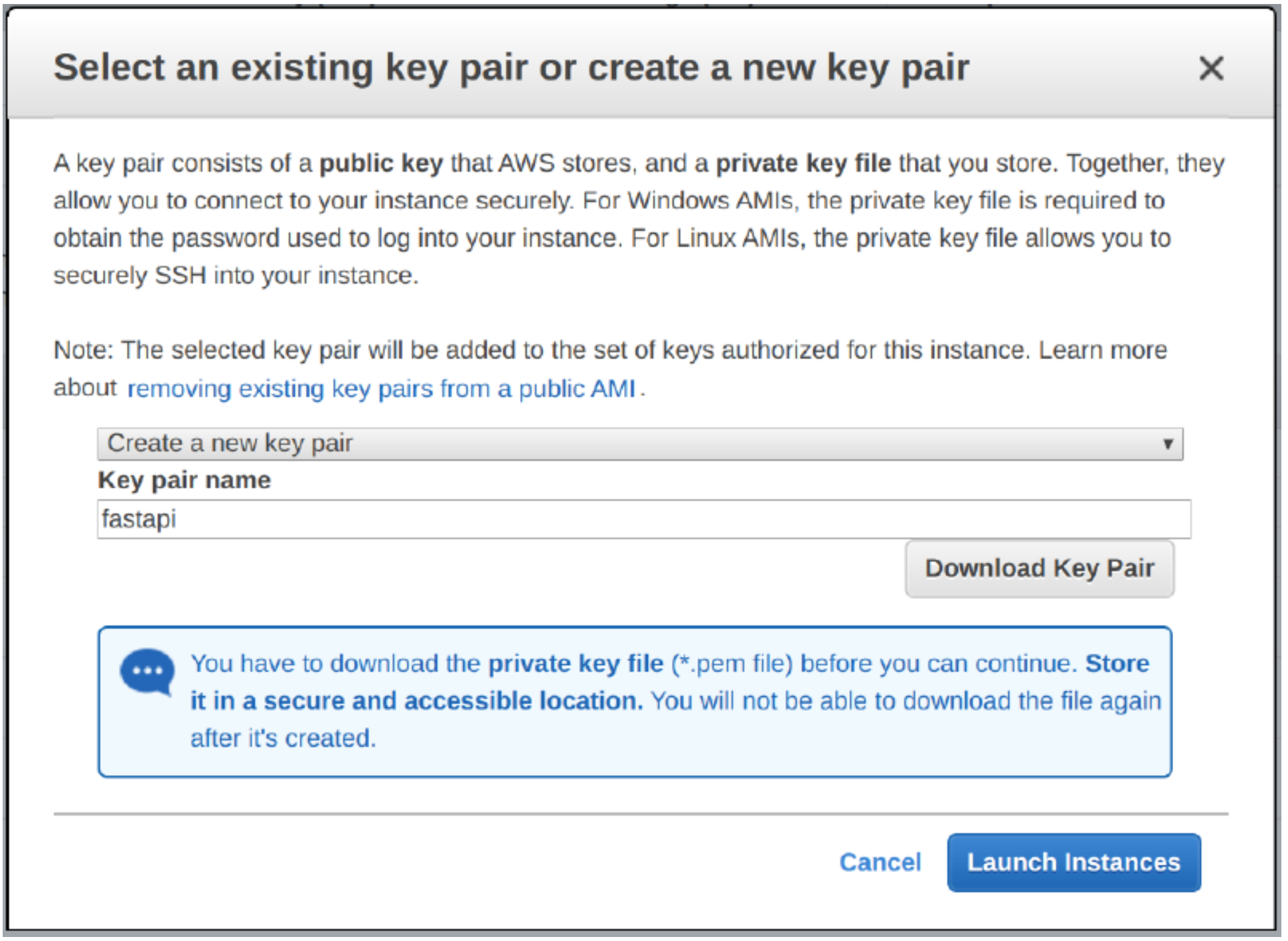


- In the second step, I select the t2.xlarge machine, which has 4 CPUs and 16GB RAM rather than the free tier since I want to use an Object Detection model and will need some resources.

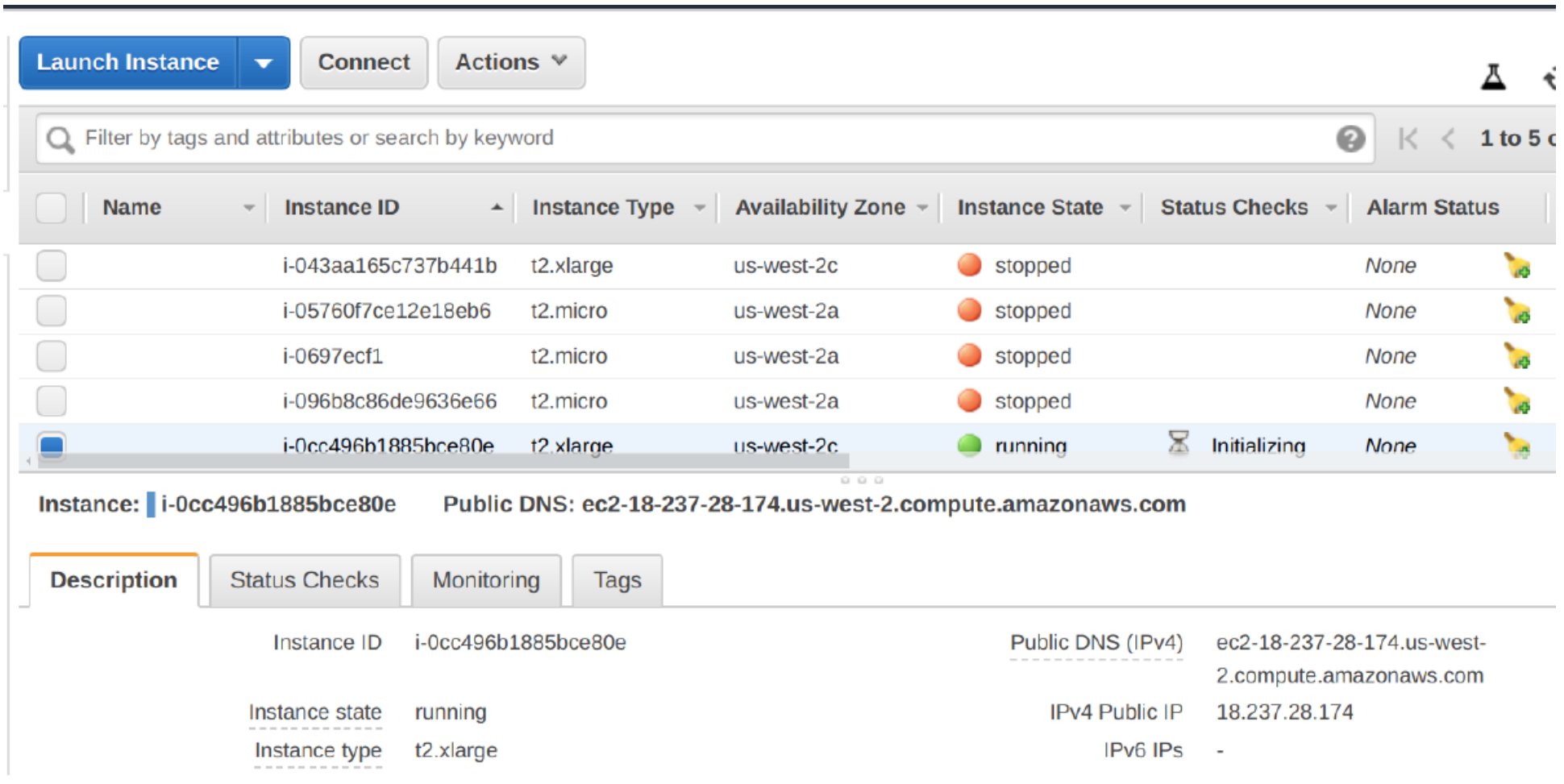


- Keep pressing Next until you reach the “6. Configure Security Group” tab. This is the most crucial step here. You will need to add a rule with Type: “HTTP” and Port Range:80.

- You can click on “Review and Launch” and finally on the “Launch” button to launch the instance. Once you click on Launch, you might need to create a new key pair. Here I am creating a new key pair named fastapi and downloading that using the “Download Key Pair” button. Keep this key safe as it would be required every time you need to login to this particular machine. Click on “Launch Instance” after downloading the key pair



- You can now go to your instances to see if your instance has started. Hint: See the Instance state; it should be showing “Running.”



- Also, to note here are the Public DNS(IPv4) address and the IPv4 public IP. We will need it to connect to this machine. For me, they are:

```
Public DNS (IPv4): ec2-18-237-28-174.us-west-2.compute.amazonaws.com

IPv4 Public IP: 18.237.28.174
```

- Once you have that run the following commands in the folder, you saved the fastapi.pem file. If the file is named fastapi.txt you might need to rename it to fastapi.pem.

```
# run fist command if fastapi.txt gets downloaded.
# mv fastapi.txt fastapi.pem

chmod 400 fastapi.pem
ssh -i "fastapi.pem" ubuntu@<Your Public DNS(IPv4) Address>
```

```
(base) rahul@rahul-MS-7A93:~$ ssh -i "fastapi.pem" ubuntu@ec2-18-237-28-174.us-west-2.compute.amazonaws.com
The authenticity of host 'ec2-18-237-28-174.us-west-2.compute.amazonaws.com (18.237.28.174)' can't be established.
ECDSA key fingerprint is SHA256:o5W1tLXwN0nLcqa2LhtDDDmdbGkAL5Mvg3sPjTLrJrM.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-18-237-28-174.us-west-2.compute.amazonaws.com,18.237.28.174' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-1065-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Sat Jun 27 21:56:00 UTC 2020

  System load:  0.14              Processes:           129
  Usage of /:   13.8% of 7.69GB   Users logged in:     0
  Memory usage: 1%                IP address for eth0: 172.31.12.30
  Swap usage:   0%

0 packages can be updated.
0 updates are security updates.



The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-12-30:~$
```

Now we have got our Amazon instance up and running. We can move on here to the real part of the post.

2. Creating a FastAPI API for Object Detection

Before we deploy an API, we need to have an API with us, right? In one of my last posts, I had written a simple tutorial to understand FastAPI and API basics. Do read the post if you want to understand FastAPI basics.

So, here I will try to create an Image detection API. As for how to pass the Image data to the API? The idea is — ***What is an image but a string?*** An image is just made up of bytes, and we can encode these bytes as a string. We will use the base64 string representation, which is a popular way to get binary data to ASCII characters. And, we will pass this string representation to give an image to our API.

A. Some Image Basics: What is Image, But a String?

So, let us first see how we can convert an Image to a String. We read the binary data from an image file using the ‘rb’ flag and turn it into a base64 encoded data representation using the base64.b64encode function. We then use the decode to utf-8 function to get the base encoded data into human-readable characters. Don’t worry if it doesn’t make a lot of sense right now. ***Just understand that any data is binary, and we can convert binary data to its string representation using a series of steps.***

As a simple example, if I have a simple image like below, we can convert it to a string using:

```
import base64

with open("sample_images/dog_with_ball.jpg", "rb") as image_file:
    base64str = base64.b64encode(image_file.read()).decode("utf-8")
```

```
import base64

with open("sample_images/dog_with_ball.jpg", "rb") as image_file:
    base64str = base64.b64encode(image_file.read()).decode("utf-8")
print(base64str)
```

/9j/4AAQSkZJRgABAgAAAQABAAD/2wBDAAYEBQYFBAYGBQYHBwYIChAKCgkJChQODwwQFxQYGBcUFhYaHSUfGhsjHBYWICwgIyYnKSopGR8tMC0oMCU
oKSj/2wBDAQcHBwoIChMKChMoGhYaKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCj/wAARCAGbAmcDASIAAh
EBAxEB/8QAHwAAAQUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQoL/8QAtRAAAgEDAwIEAwUFBAQAAAF9AQIDAAQRBRIhMUEGE1FhByJxFDKBkaEII0Kxw
RVS0fAkM2JyggkKFhcYGRolJicoKSo0NTY3ODk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqDhIWGh4iJipKTlJWWl5iZmqKjpKWmp6ip
qrKztLW2t7i5usLDxMXGx8jJytLT1NXW19jZ2uHi4+Tl5ufo6erx8vP09fb3+Pn6/8QAHwEAAwEBAQEBAQEBAQAAAAAAAAECAwQFBgcICQoL/8QAtRE
AAgECBAQDBAcFBAQAAQJ3AAECAxEEBSExBhJBUQdhcRMiMoEIFEKRobHBCSMzUvAVYnLRChYkNOEl8RcYGRomJygpKjU2Nzg5OkNERUZHSElKU1RVVl
dYWVpjZGVmZ2hpanN0dXZ3eHl6goOEhYaHiImKkpOUlZaXmJmaoqOkpaanqKmqsrO0tba3uLm6wsPExcbHyMnK0tPU1dbX2Nna4uPk5ebn6Onq8vP09
fb3+Pn6/9oADAMBAAIRAxEAPwDx9osS5qyIwv41Vu5dknJqzZuJV5NfNyTSuec9SC6wgJzzVGK9e0Q/NVrU007IORWJKG35FbUoKUdQSZ08OpF4yAcG

Here I have got a string representation of a file named dog_with_ball.png on my laptop.

Great, we now have a string representation of an image. And, we can send this string representation to our FastAPI. But we also need to have a way to read an image back from its string representation. After all, our image detection API using PyTorch and any other package needs to have an image object that they can predict, and those methods don’t work on a string.

So here is a way to create a PIL image back from an image’s base64 string. Mostly we just do the reverse steps in the same order. We encode in ‘utf-8’ using .encode. We then use base64.b64decode to decode to bytes. We use these bytes to create a bytes object using io.BytesIO and use Image.open to open this bytes IO object as a PIL image, which can easily be used as an input to my PyTorch prediction code.*** Again simply, it is just a way to convert base64 image string to an actual image.***

```
import base64
import io
from PIL import Image

def base64str_to_PILImage(base64str):
   base64_img_bytes = base64str.encode('utf-8')
   base64bytes = base64.b64decode(base64_img_bytes)
   bytesObj = io.BytesIO(base64bytes)
   img = Image.open(bytesObj)
   return img
```

So does this function work? Let’s see for ourselves. We can use just the string to get back the image.

```
def base64str_to_PILImage(base64str):
    base64_img_bytes = base64str.encode('utf-8')
    base64bytes = base64.b64decode(base64_img_bytes)
    bytesObj = io.BytesIO(base64bytes)
    img = Image.open(bytesObj)
    return img

base64str_to_PILImage(base64str)
```

And we have our happy dog back again. Looks better than the string.

B. Writing the Actual FastAPI code

So, as now we understand that our API can get an image as a string from our user, let's create an object detection API that makes use of this image as a string and outputs the bounding boxes for the object with the object classes as well.

Here, I will be using a Pytorch pre-trained fasterrcnn_resnet50_fpn detection model from the torchvision.models for object detection, which is trained on the COCO dataset to keep the code simple, but one can use any model. You can look at these posts if you want to train your custom image classification or image detection model using Pytorch.

Below is the full code for the FastAPI. Although it may look long, we already know all the parts. In this code, we essentially do the following steps:

- Create our fast API app using the FastAPI() constructor.
- Load our model and the classes it was trained on. I got the list of classes from the PyTorch docs .
- We also defined a new class Input , which uses a library called pydantic to validate the input data types that we will get from the API end-user. Here the end-user gives the base64str and some score threshold for object detection prediction.
- We add a function called base64str_to_PILImage which does just what it is named.
- And we write a predict function called get_predictionbase64 which returns a dict of bounding boxes and classes using a base64 string representation of an image and a threshold as an input. We also add @app .put("/predict") on top of this function to define our endpoint. If you need to understand put and endpoint refer to my previous post on FastAPI.

```
from fastapi import FastAPI
from pydantic import BaseModel
import torchvision
from torchvision import transforms
import torch
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from PIL import Image
import numpy as np
import cv2
import io, json
import base64


app = FastAPI()

# load a pre-trained Model and convert it to eval mode.
# This model loads just once when we start the API.
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
COCO_INSTANCE_CATEGORY_NAMES = [
    '__background__', 'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
    'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A', 'stop sign',
    'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
    'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack', 'umbrella', 'N/A', 'N/A',
    'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
    'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket',
    'bottle', 'N/A', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
    'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
    'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table',
    'N/A', 'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard', 'cell phone',
    'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A', 'book',
    'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush'
]
model.eval()

# define the Input class
class Input(BaseModel):
    base64str : str
    threshold : float

def base64str_to_PILImage(base64str):
    base64_img_bytes = base64str.encode('utf-8')
    base64bytes = base64.b64decode(base64_img_bytes)
    bytesObj = io.BytesIO(base64bytes)
    img = Image.open(bytesObj)
    return img

@app.put("/predict")
def get_predictionbase64(d:Input):
    '''
    FastAPI API will take a base 64 image as input and return a json object
    '''
    # Load the image
    img = base64str_to_PILImage(d.base64str)
    # Convert image to tensor
    transform = transforms.Compose([transforms.ToTensor()])
    img = transform(img)
    # get prediction on image
    pred = model([img])
    pred_class = [COCO_INSTANCE_CATEGORY_NAMES[i] for i in list(pred[0]['labels'].numpy())]
    pred_boxes = [[(float(i[0]), float(i[1])), (float(i[2]), float(i[3]))] for i in list(pred[0]
['boxes'].detach().numpy())]
    pred_score = list(pred[0]['scores'].detach().numpy())
    pred_t = [pred_score.index(x) for x in pred_score if x > d.threshold][-1]
    pred_boxes = pred_boxes[:pred_t+1]
    pred_class = pred_class[:pred_t+1]
    return {'boxes': pred_boxes,
        'classes' : pred_class}
```

C. Local Before Global: Test the FastAPI code locally

Before we move on to AWS, let us check if the code works on our local machine. We can start the API on our laptop using:

```
uvicorn fastapiapp:app --reload
```

The above means that your API is now running on your local server, and the –reload flag indicates that the API gets updated automatically when you change the fastapiapp.py file. This is very helpful while developing and testing, but you should remove this –reload flag when you put the API in production.

You should see something like:

```
(pyt) rahul@rahul-MS-7A93:~/projects/deployFastApi/fastapi$ uvicorn fastapiapp:app --reload
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [18965] using statreload
INFO:     Started server process [19001]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

You can now try to access this API and see if it works using the requests module:

```
import requests,json

payload = json.dumps({
  "base64str": base64str,
  "threshold": 0.5
})

response = requests.put("[http://127.0.0.1:8000/predict](http://127.0.0.1:8000/predict)",data = payload)
data_dict = response.json()
```

```
import requests,json

payload = json.dumps({
  "base64str": base64str,
  "threshold": 0.5
})

response = requests.put("http://127.0.0.1:8000/predict",data = payload)
data_dict = response.json()
print(data_dict)
```

```
{'boxes': [[[272.754638671875, 48.12982177734375], [433.81219482421875, 376.4888916015625]], [[312.01104736328125,
149.79376220703125], [367.9121398925781, 207.1046142578125]]], 'classes': ['dog', 'sports ball']}
```

And so we get our results using the API. This image contains a dog and a sports ball. We also have corner 1 (x1,y1) and corner 2 (x2,y2) coordinates of our bounding boxes.

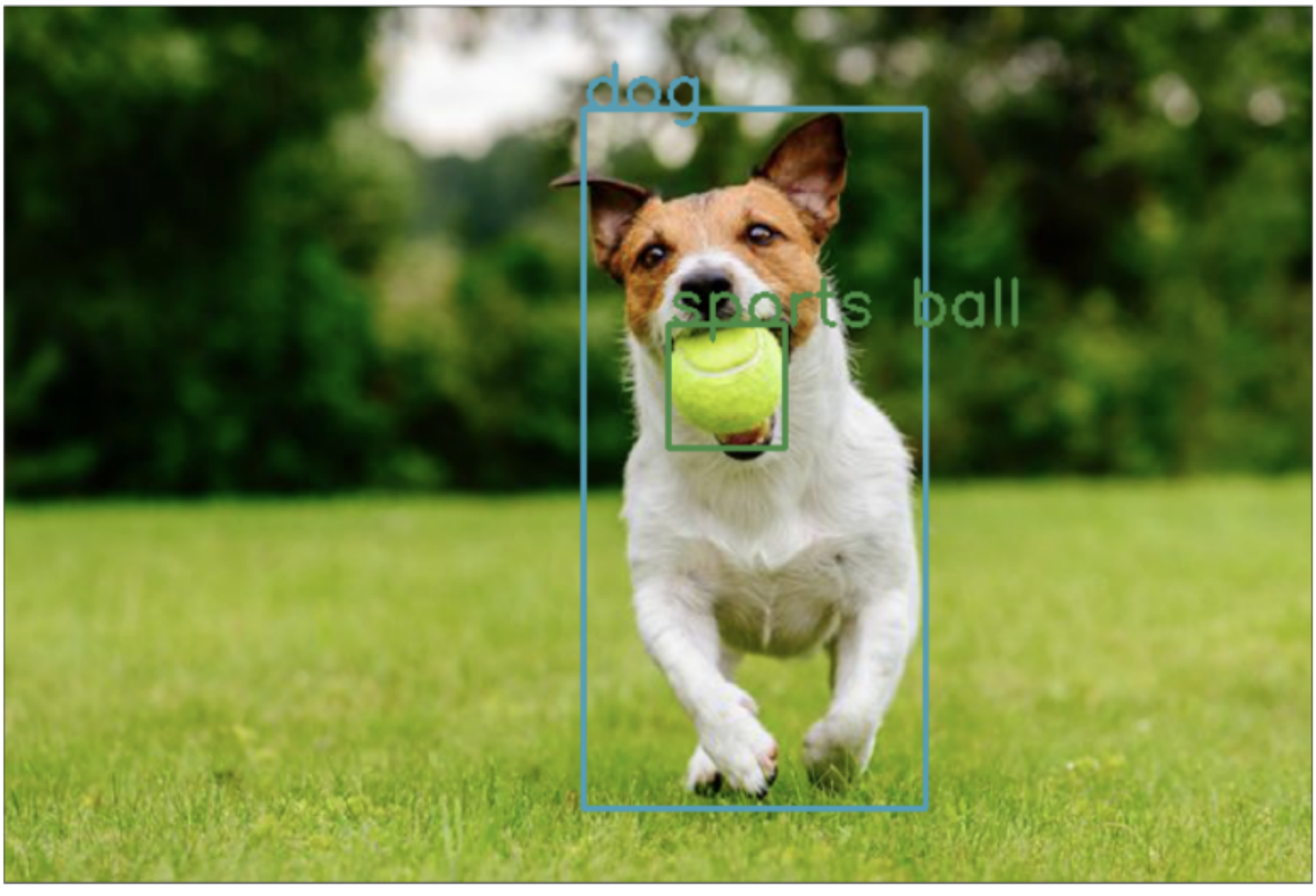
D. Lets Visualize

Although not strictly necessary, we can visualize how the results look in our Jupyter notebook:

```
from PIL import Image
import numpy as np
import cv2
import matplotlib.pyplot as plt

def PILImage_to_cv2(img):
    return np.asarray(img)

def drawboundingbox(img, boxes,pred_cls, rect_th=2, text_size=1, text_th=2):
    img = PILImage_to_cv2(img)
    class_color_dict = {}

    #initialize some random colors for each class for better looking bounding boxes
    for cat in pred_cls:
        class_color_dict[cat] = [random.randint(0, 255) for _ in range(3)]

    for i in range(len(boxes)):
        cv2.rectangle(img, (int(boxes[i][0][0]), int(boxes[i][0][1])),
                      (int(boxes[i][1][0]),int(boxes[i][1][1])),
                      color=class_color_dict[pred_cls[i]], thickness=rect_th)
        cv2.putText(img,pred_cls[i], (int(boxes[i][0][0]), int(boxes[i][0][1])),  cv2.FONT_HERSHEY_SIMPLEX,
text_size, class_color_dict[pred_cls[i]],thickness=text_th) # Write the prediction class
    plt.figure(figsize=(20,30))
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
    plt.show()

img = Image.open("sample_images/dog_with_ball.jpg")
drawboundingbox(img, data_dict['boxes'], data_dict['classes'])
```

Here is the output:

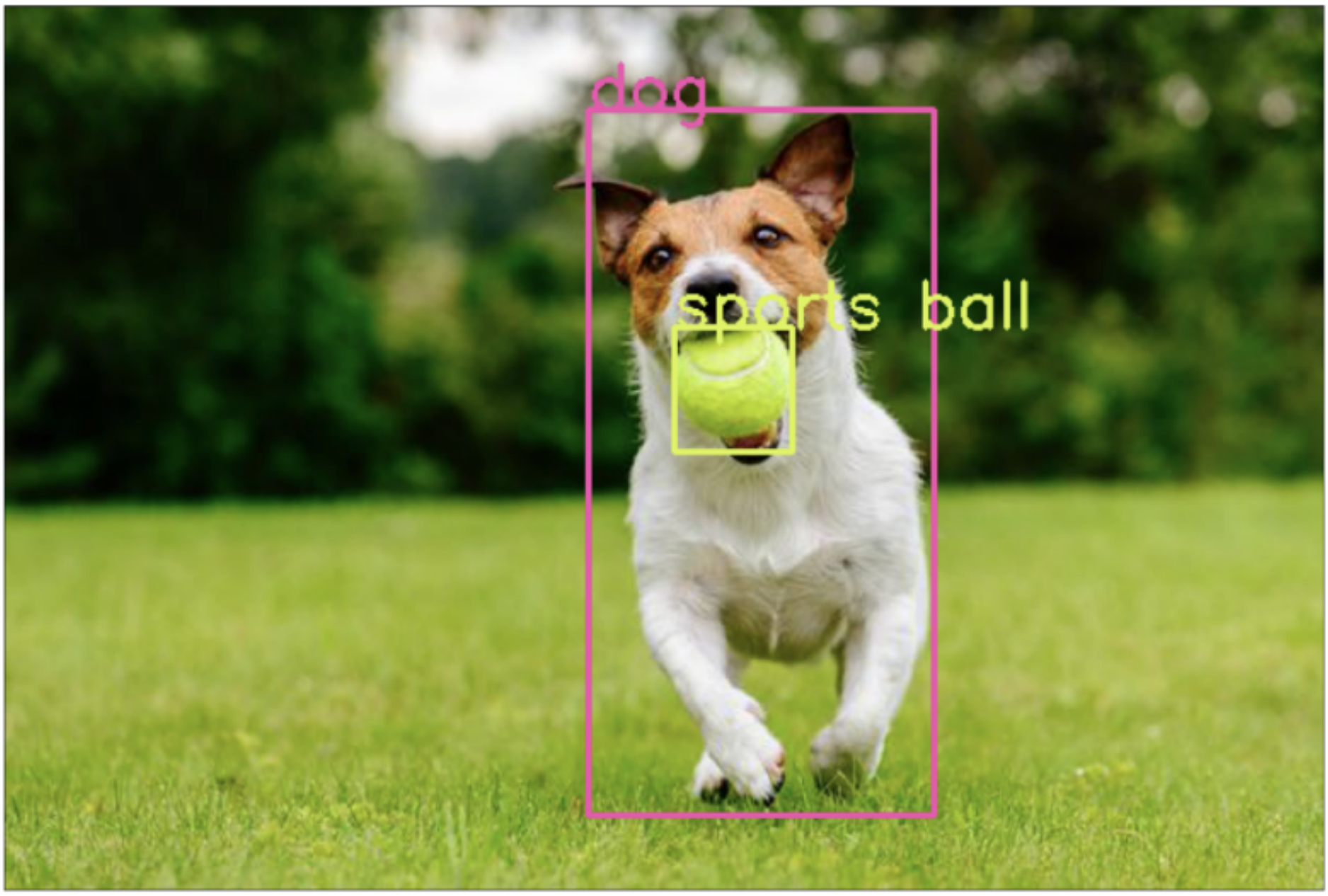
```
img = Image.open("sample_images/dog_with_ball.jpg")
drawboundingbox(img, data_dict['boxes'], data_dict['classes'])
```



Here you will note that I got the image from the local file system, and that sort of can be considered as cheating as we don’t want to save every file that the user sends to us through a web UI. We should have been able to use the same base64string object that we also had to create this image. Right?

Not to worry, we could do that too. Remember our base64str_to_PILImage function? We could have used that also.

```
img = base64str_to_PILImage(base64str)
drawboundingbox(img, data_dict['boxes'], data_dict['classes'])
```



That looks great. We have our working FastAPI, and we also have our amazon instance. We can now move on to Deployment.

3. Deployment on Amazon ec2

Till now, we have created an AWS instance and, we have also created a FastAPI that takes as input a base64 string representation of an image and returns bounding boxes and the associated class. But all the FastAPI code still resides in our local machine. ***How do we put it on the ec2 server? And run predictions on the cloud.***

A. Install Docker

We will deploy our app using docker, as is suggested by the fastAPI creator himself. I will try to explain how docker works as we go. The below part may look daunting but it just is a series of commands and steps. So stay with me.

We can start by installing docker using:

```
sudo apt-get update
sudo apt install docker.io
```

We then start the docker service using:

```
sudo service docker start
```

B. Creating the folder structure for docker

```
└── dockerfastapi
    ├── Dockerfile
    ├── app
    │   └── main.py
    └── requirements.txt
```

Here dockerfastapi is our project's main folder. And here are the different files in this folder:

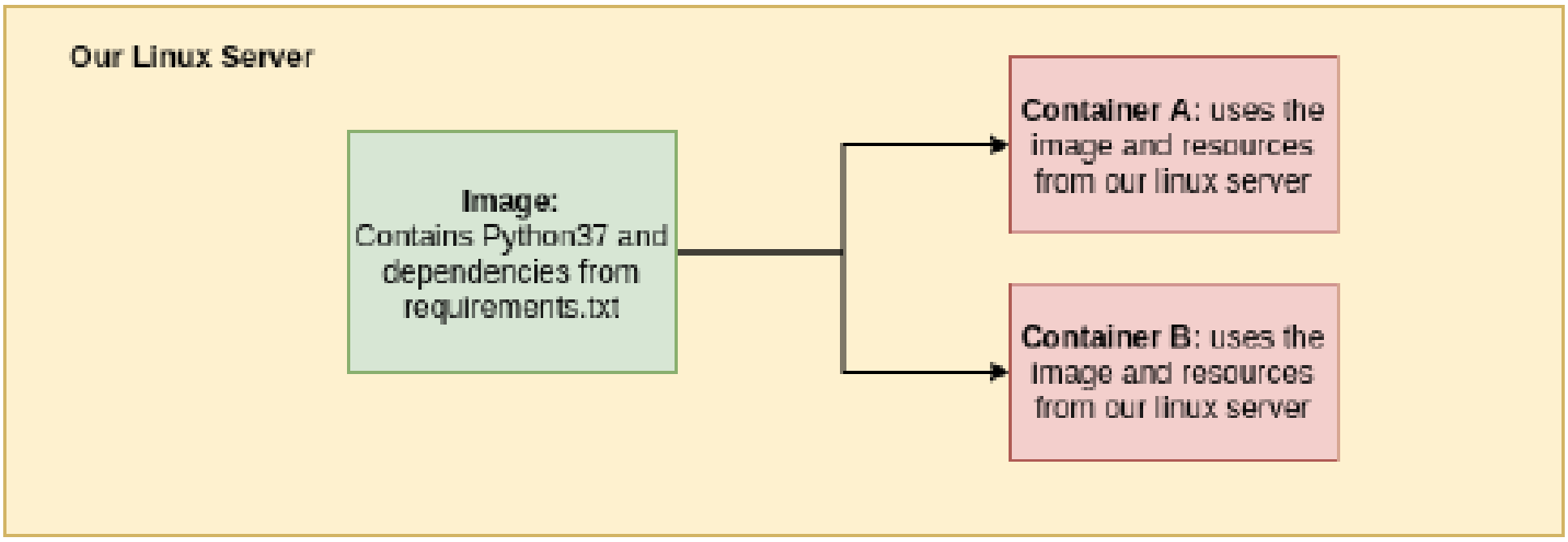
i. requirements.txt: Docker needs a file, which tells it which all libraries are required for our app to run. Here I have listed all the libraries I used in my Fastapi API.

```
numpy
opencv-python
matplotlib
torchvision
torch
fastapi
pydantic
```

ii. Dockerfile: The second file is Dockerfile.

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7

COPY ./app /app
COPY requirements.txt .
RUN pip --no-cache-dir install -r requirements.txt
```

How Docker works?: You can skip this section, but it will help to get some understanding of how docker works.



The dockerfile can be thought of something like a sh file,which contains commands to create a docker image that can be run in a container. One can think of a docker image as an environment where everything like Python and Python libraries is installed. A container is a unit which is just an isolated box in our system that uses a dockerimage. The advantage of using docker is that we can create multiple docker images and use them in multiple containers. For example, one image might contain python36, and another can contain python37. And we can spawn multiple containers in a single Linux server.

Our Dockerfile contains a few things:

- FROM command: Here the first line FROM specifies that we start with tiangolo's (FastAPI creator) Docker image. As per his site: "*This image has an "auto-tuning" mechanism included so that you can just add your code and get that same high performance automatically. And without making sacrifices*". What we are doing is just starting from an image that installs python3.7 for us along with some added configurations for uvicorn and gunicorn ASGI servers and a start.sh file for ASGI servers automatically. For adventurous souls, particularly commandset 1 and commandset2 get executed through a sort of a daisy-chaining of commands.
- COPY command: We can think of a docker image also as a folder that contains files and such. Here we copy our app folder and the requirements.txt file, which we created earlier to our docker image.
- RUN Command: We run pip install command to install all our python dependencies using the requirements.txt file that is now on the docker image.

iii. main.py: This file contains the fastapiapp.py code we created earlier. Remember to keep the name of the file main.py only.

C. Docker Build

We have got all our files in the required structure, but we haven't yet used any docker command. We will first need to build an image containing all dependencies using Dockerfile.

We can do this simply by:

```
sudo docker build -t myimage .
```

This downloads, copies and installs some files and libraries from tiangolo's image and creates an image called myimage. This myimage has python37 and some python packages as specified by requirements.txt file.

```
ubuntu@ip-172-31-12-30:~$ cd dockerfastapi/
ubuntu@ip-172-31-12-30:~/dockerfastapi$ sudo docker build -t myimage .
Sending build context to Docker daemon  7.168kB
Step 1/4 : FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7
python3.7: Pulling from tiangolo/uvicorn-gunicorn-fastapi
90fe46dd8199: Pull complete
35a4f1977689: Pull complete
bbc37f14aded: Pull complete
74e27dc593d4: Pull complete
4352dcff7819: Pull complete
deb569b08de6: Pull complete
1aa33568be24: Pull complete
a8034acd5893: Pull complete
7dfa9b763153: Pull complete
a035e7b052a3: Pull complete
d4f2cd280496: Pull complete
e227f25dc86f: Pull complete
97a4044094b0: Pull complete
5d40afec4e92: Pull complete
f6b5e3636b46: Pull complete
85a8adaae1ef: Pull complete
84e6014bde90: Pull complete
a326685691d9: Pull complete
Digest: sha256:cae1a92ba5b15c9e92b5b00875fef83fe16d8c7332f30195cc529be879dd3aed
Status: Downloaded newer image for tiangolo/uvicorn-gunicorn-fastapi:python3.7
 ---> 57c75d7c41d8
Step 2/4 : COPY ./app /app
 ---> 07345e73e2e1
Step 3/4 : COPY requirements.txt .
 ---> 3d69e9e9b443
Step 4/4 : RUN pip --no-cache-dir install -r requirements.txt
 ---> Running in 0200428817bd
Collecting numpy
  Downloading numpy-1.19.0-cp37-cp37m-manylinux2010_x86_64.whl (14.6 MB)
Collecting opencv-python
  Downloading opencv_python-4.2.0.34-cp37-cp37m-manylinux1_x86_64.whl (28.2 MB)
Collecting matplotlib
  Downloading matplotlib-3.2.2-cp37-cp37m-manylinux1_x86_64.whl (12.4 MB)
Collecting torchvision
  Downloading torchvision-0.6.1-cp37-cp37m-manylinux1_x86_64.whl (6.6 MB)
Collecting torch
  Downloading torch-1.5.1-cp37-cp37m-manylinux1_x86_64.whl (753.2 MB)
Requirement already satisfied: fastapi in /usr/local/lib/python3.7/site-packages (from -r requirements.txt (line 6)) (0.55.1)
Requirement already satisfied: pydantic in /usr/local/lib/python3.7/site-packages (from -r requirements.txt (line 7)) (1.5.1)
Collecting cycler>=0.10
  Downloading cycler-0.10.0-py2.py3-none-any.whl (6.5 kB)
Collecting pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1
  Downloading pyparsing-2.4.7-py2.py3-none-any.whl (67 kB)
Collecting kiwisolver>=1.0.1
  Downloading kiwisolver-1.2.0-cp37-cp37m-manylinux1_x86_64.whl (88 kB)
Collecting python-dateutil>=2.1
  Downloading python_dateutil-2.8.1-py2.py3-none-any.whl (227 kB)
Collecting pillow>=4.1.1
  Downloading Pillow-7.1.2-cp37-cp37m-manylinux1_x86_64.whl (2.1 MB)
Collecting future
  Downloading future-0.18.2.tar.gz (829 kB)
Requirement already satisfied: starlette==0.13.2 in /usr/local/lib/python3.7/site-packages (from fastapi->-r requirements.txt (line 6)) (0.13.2)
Collecting six
  Downloading six-1.15.0-py2.py3-none-any.whl (10 kB)
Building wheels for collected packages: future
  Building wheel for future (setup.py): started
  Building wheel for future (setup.py): finished with status 'done'
  Created wheel for future: filename=future-0.18.2-py3-none-any.whl size=491058 sha256=d4749a3f1b32702d360de998aac28be89f9ba34adb8a8593ac92bf7814b92820
  Stored in directory: /tmp/pip-ephem-wheel-cache-fa3tsv82/wheels/56/b0/fe/4410d17b32f1f0c3cf54cdfb2bc04d7b4b8f4ae377e2229ba0
Successfully built future
Installing collected packages: numpy, opencv-python, six, cycler, pyparsing, kiwisolver, python-dateutil, matplotlib, future, torch, pillow, torchvision
Successfully installed cycler-0.10.0 future-0.18.2 kiwisolver-1.2.0 matplotlib-3.2.2 numpy-1.19.0 opencv-python-4.2.0.34 pillow-7.1.2 pyparsing-2.4.7 python-dateutil-2.8.1 six-1.15.0 torch-1
.5.1 torchvision-0.6.1
WARNING: You are using pip version 20.0.2; however, version 20.1.1 is available.
You should consider upgrading via the '/usr/local/bin/python -m pip install --upgrade pip' command.
Removing intermediate container 0200428817bd
 ---> 53f7dc8f9053
Successfully built 53f7dc8f9053
Successfully tagged myimage:latest
```

We will then just need to start a container that runs this image. We can do this using:

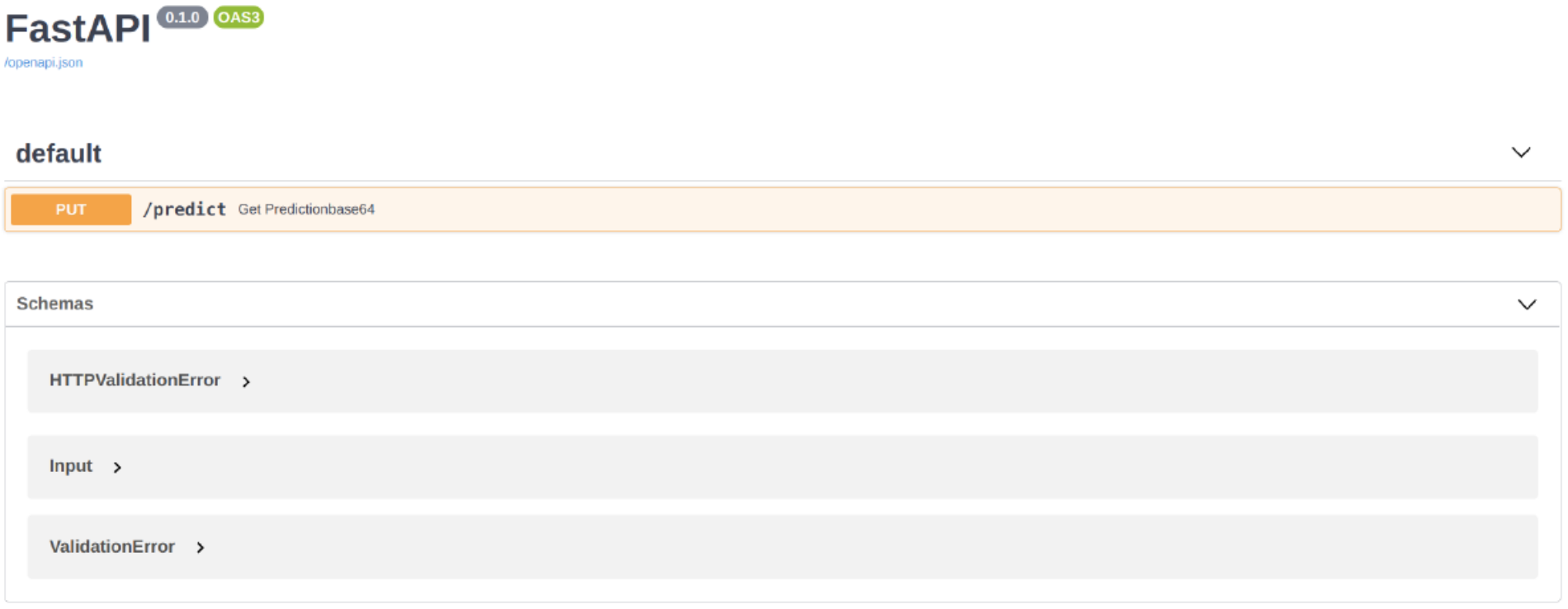
```
sudo docker run -d --name mycontainer -p 80:80 myimage
```

This will create a container named mycontainer which runs our docker image myimage. The part 80:80 connects our docker container port 80 to our Linux machine port 80.

```
ubuntu@ip-172-31-12-30:~/dockerfastapi$ sudo docker run -d --name mycontainer -p 80:80 myimage
97eb062326a9eb04306aba66142ff04229d9ac790bd9028750f19ed3bbc0501a
ubuntu@ip-172-31-12-30:~/dockerfastapi$
```

And actually that’s it. At this point, you should be able to open the below URL in your browser.

```
# <IPV4 public IP>/docs
URL: 18.237.28.174/docs
```



And we can check our app programmatically using:

```
payload = json.dumps({
  "base64str": base64str,
  "threshold": 0.5
})

response = requests.put("[http://18.237.28.174/predict](http://18.237.28.174/predict)",data = payload)
data_dict = response.json()
print(data_dict)
```

```
payload = json.dumps({
  "base64str": base64str,
  "threshold": 0.5
})

response = requests.put("http://18.237.28.174/predict",data = payload)
data_dict = response.json()
print(data_dict)
```

```
{'boxes': [[[272.754638671875, 48.12983703613281], [433.81219482421875, 376.4888916015625]], [[312.01104736328125,
149.79376220703125], [367.9121398925781, 207.1046142578125]]], 'classes': ['dog', 'sports ball']}
```

Yup, finally our API is deployed.

D. Troubleshooting as the real world is not perfect

All the above was good and will just work out of the box if you follow the exact instructions, but the real world doesn't work like that. You will surely get some errors along the way and would need to debug your code. So to help you with that, some docker commands may come handy:

- **Logs:** When we ran our container using sudo docker run we don't get a lot of info, and that is a big problem when you are debugging. You can see the real-time logs using the below command. If you see an error here, you will need to change your code and build the image again.

```
sudo docker logs -f mycontainer
```

```
ubuntu@ip-172-31-12-30:~/dockerfastapi$ sudo docker logs -f mycontainer
Checking for script in /app/prestart.sh
Running script /app/prestart.sh
Running inside /app/prestart.sh, you could add migrations to this file, e.g.:

#! /usr/bin/env bash

# Let the DB start
sleep 10;
# Run migrations
alembic upgrade head

[2020-06-27 23:18:25 +0000] [1] [INFO] Starting gunicorn 20.0.4
[2020-06-27 23:18:25 +0000] [1] [INFO] Listening at: http://0.0.0.0:80 (1)
[2020-06-27 23:18:25 +0000] [1] [INFO] Using worker: uvicorn.workers.UvicornWorker
[2020-06-27 23:18:25 +0000] [12] [INFO] Booting worker with pid: 12
[2020-06-27 23:18:25 +0000] [13] [INFO] Booting worker with pid: 13
[2020-06-27 23:18:25 +0000] [14] [INFO] Booting worker with pid: 14
[2020-06-27 23:18:25 +0000] [15] [INFO] Booting worker with pid: 15
Downloading: "https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth" to /root/.cache/torch/checkpoints/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth
0.0%Downloading: "https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth" to /root/.cache/torch/checkpoints/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth
2.6%Downloading: "https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth" to /root/.cache/torch/checkpoints/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth
29.9%Downloading: "https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth" to /root/.cache/torch/checkpoints/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth
67.5%%
[2020-06-27 23:18:30 +0000] [15] [INFO] Started server process [15]
67.5%[2020-06-27 23:18:30 +0000] [15] [INFO] Waiting for application startup.
[2020-06-27 23:18:30 +0000] [15] [INFO] Application startup complete.
97.9%%
[2020-06-27 23:18:32 +0000] [12] [INFO] Started server process [12]
[2020-06-27 23:18:32 +0000] [12] [INFO] Waiting for application startup.
[2020-06-27 23:18:32 +0000] [12] [INFO] Application startup complete.
87.0%%
[2020-06-27 23:18:33 +0000] [13] [INFO] Started server process [13]
[2020-06-27 23:18:33 +0000] [13] [INFO] Waiting for application startup.
[2020-06-27 23:18:33 +0000] [13] [INFO] Application startup complete.
100.0%
[2020-06-27 23:18:34 +0000] [14] [INFO] Started server process [14]
[2020-06-27 23:18:34 +0000] [14] [INFO] Waiting for application startup.
[2020-06-27 23:18:34 +0000] [14] [INFO] Application startup complete.
{"loglevel": "info", "workers": 4, "bind": "0.0.0.0:80", "graceful_timeout": 120, "timeout": 120, "keepalive": 5, "errorlog": "-", "accesslog": "-", "workers_per_core": 1.0, "use_max_workers
": null, "host": "0.0.0.0", "port": "80"}
49.207.57.66:5307 - "PUT /predict HTTP/1.1" 200
```

- **Starting and Stopping Docker:** Sometimes, it might help just to restart your docker. In that case, you can use:

```
sudo service docker stop
sudo service docker start
```

- **Listing images and containers:** Working with docker, you will end up creating images and containers, but you won't be able to see them in the working directory. You can list your images and containers using:

```
sudo docker container ls
sudo docker image ls
```

```
ubuntu@ip-172-31-12-30:~/dockerfastapi$ sudo docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                   NAMES
97eb062326a9        myimage             "/start.sh"         5 minutes ago       Up 5 minutes        0.0.0.0:80->80/tcp      mycontainer
ubuntu@ip-172-31-12-30:~/dockerfastapi$ sudo docker image ls
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
myimage                             latest              53f7dc8f9053        27 minutes ago      2.58GB
tiangolo/uvicorn-gunicorn-fastapi   python3.7           57c75d7c41d8        3 weeks ago         970MB
```

- **Deleting unused docker images or containers:** You might need to remove some images or containers as these take up a lot of space on the system. Here is how you do that.

```
# the prune command removes the unused containers and images
sudo docker system prune

# delete a particular container
sudo docker rm mycontainer

# remove myimage
sudo docker image rm myimage

# remove all images
sudo docker image prune — all
```

- **Checking localhost:**The Linux server doesn't have a browser, but we can still see the browser output though it's a little ugly:

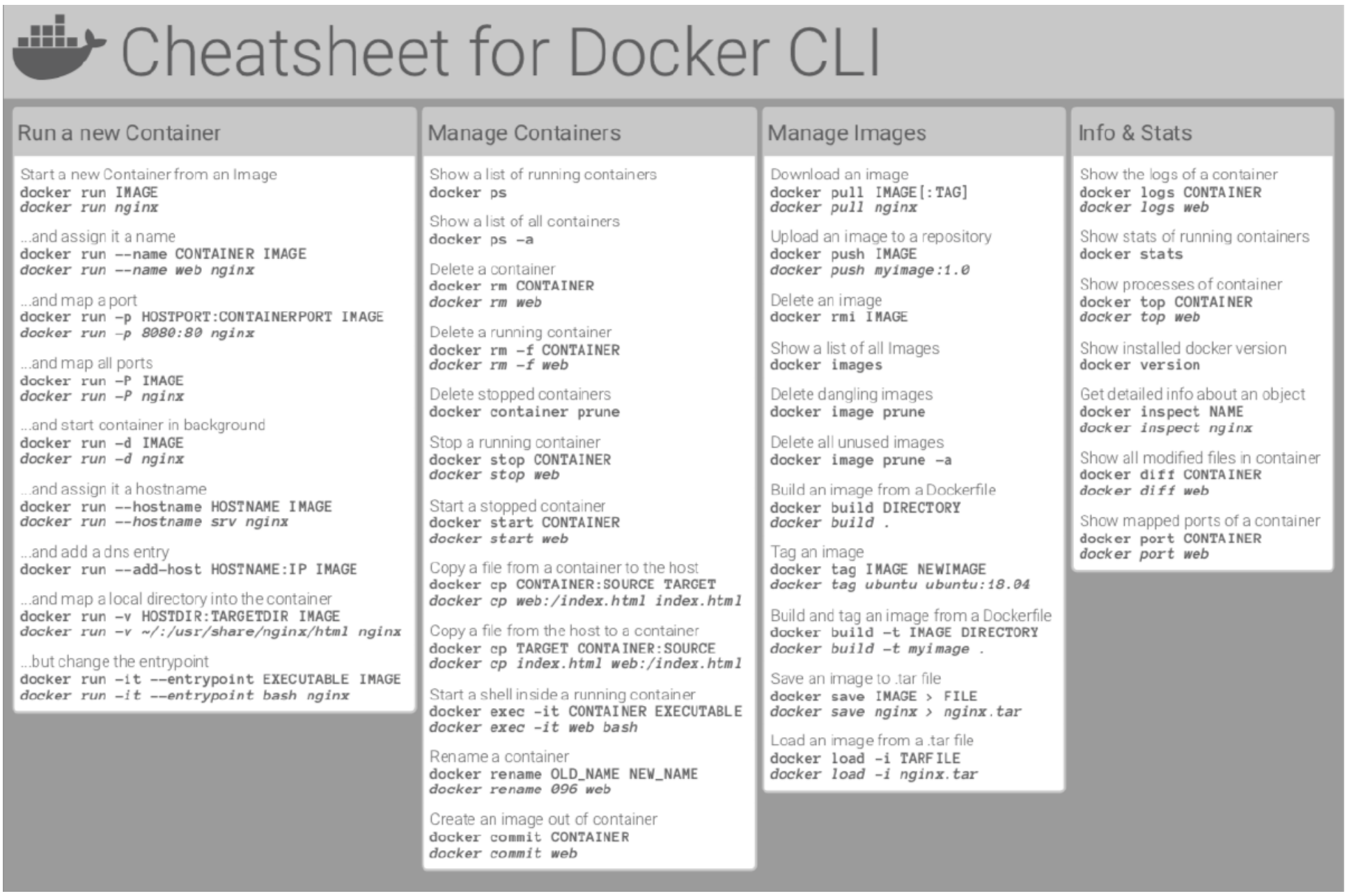
```
curl localhost
```

```
ubuntu@ip-172-31-12-30:~/dockerfastapi$ curl localhost/docs

    <!DOCTYPE html>
    <html>
    <head>
    <link type="text/css" rel="stylesheet" href="https://cdn.jsdelivr.net/npm/swagger-ui-dist@3/swagger-ui.css">
    <link rel="shortcut icon" href="https://fastapi.tiangolo.com/img/favicon.png">
    <title>FastAPI - Swagger UI</title>
    </head>
    <body>
    <div id="swagger-ui">
    </div>
    <script src="https://cdn.jsdelivr.net/npm/swagger-ui-dist@3/swagger-ui-bundle.js"></script>
    <!-- `SwaggerUIBundle` is now available on the page -->
    <script>
    const ui = SwaggerUIBundle({
        url: '/openapi.json',
    oauth2RedirectUrl: window.location.origin + '/docs/oauth2-redirect',
        dom_id: '#swagger-ui',
        presets: [
        SwaggerUIBundle.presets.apis,
        SwaggerUIBundle.SwaggerUIStandalonePreset
        ],
        layout: "BaseLayout",
        deepLinking: true
    })
    </script>
    </body>
    </html>
```

- **Develop without reloading image again and again:** For development, it's useful to be able just to change the contents of the code on our machine and test it live, without having to build the image every time. In that case, it's also useful to run the server with live auto-reload automatically at every code change. Here, we use our app directory on our Linux machine, and we replace the default (/start.sh) with the development alternative /start-reload.sh during development. After everything looks fine, we can build our image again run it inside the container.

```
sudo docker run -d -p 80:80 -v $(pwd):/app myimage /start-reload.sh
```

If this doesn't seem sufficient, adding here a docker cheat sheet containing useful docker commands:

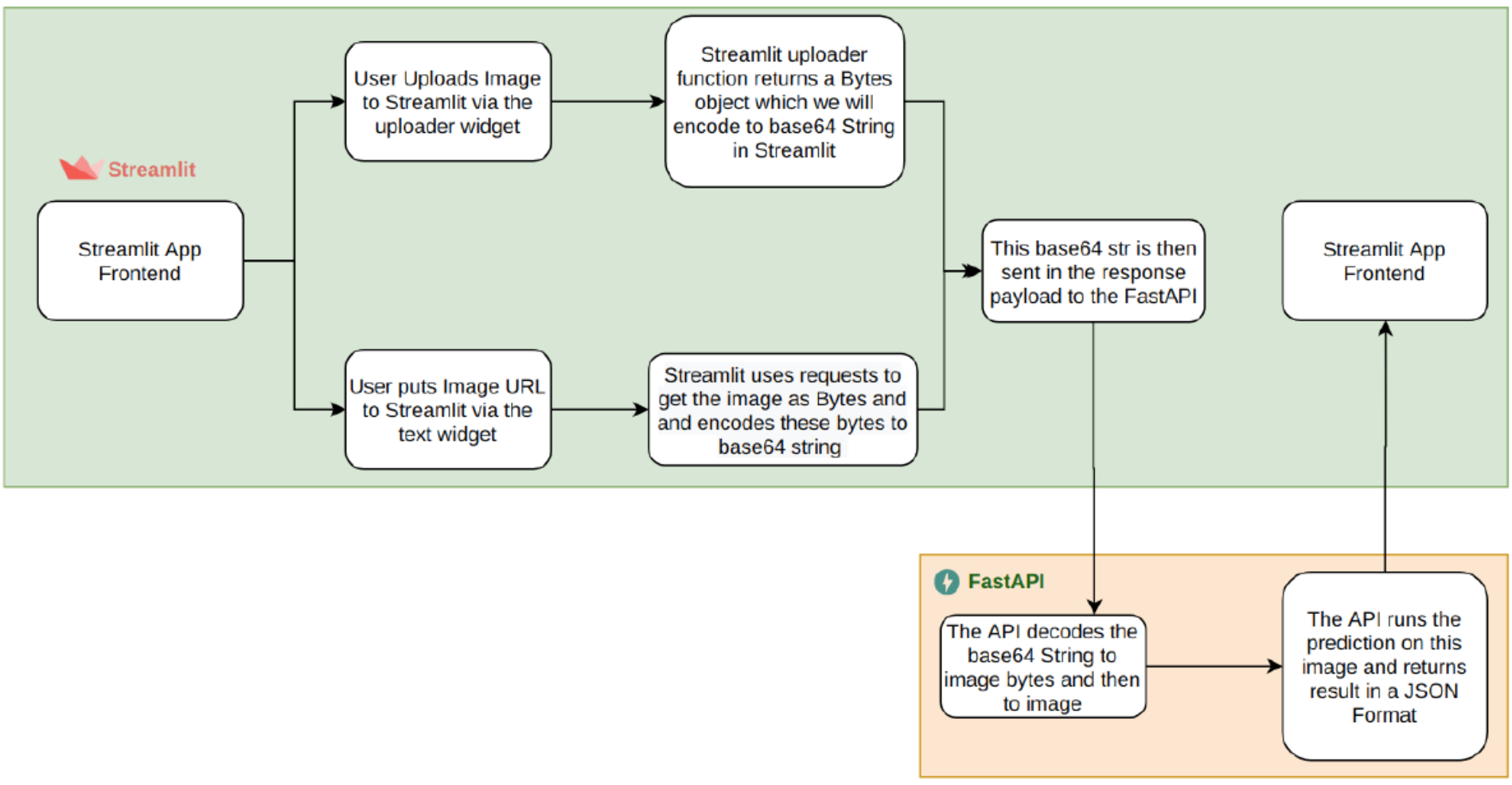


4. An End to End App with UI

We are done here with our API creation, but we can also create a UI based app using Streamlit using our FastAPI API. This is not how you will do it in a production setting (where you might have developers making apps using react, node.js or javascript)but is mostly here to check the end-to-end flow of how to use an image API. I will host this barebones Streamlit app on local rather than the ec2 server, and it will get the bounding box info and classes from the FastAPI API hosted on ec2.

If you need to learn more about how streamlit works, you can check out this post . Also, if you would want to deploy this streamlit app also to ec2, here is a tutorial again.

Here is the flow of the whole app with UI and FastAPI API on ec2:



Project Architecture

The most important problems we need to solve in our streamlit app are:

How to get an image file from the user using Streamlit?

A. Using File uploader: We can use the file uploader using:

```
bytesObj = st.file_uploader("Choose an image file")
```

The next problem is, what is this bytesObj we get from the streamlit file uploader? In streamlit, we will get a bytesIO object from the file_uploader and we will need to convert it to base64str for our FastAPI app input. This can be done using:

```
def bytesioObj_to_base64str(bytesObj):
   return base64.b64encode(bytesObj.read()).decode("utf-8")

base64str = bytesioObj_to_base64str(bytesObj)
```

B. Using URL: We can also get an image URL from the user using text_input.

```
url = st.text_input('Enter URL')
```

We can then get image from URL in base64 string format using the requests module and base64 encode and utf-8 decode:

```
def ImgURL_to_base64str(url):
    return base64.b64encode(requests.get(url).content).decode("utf-8")

base64str = ImgURL_to_base64str(url)
```

And here is the complete code of our Streamlit app. You have seen most of the code in this post already.

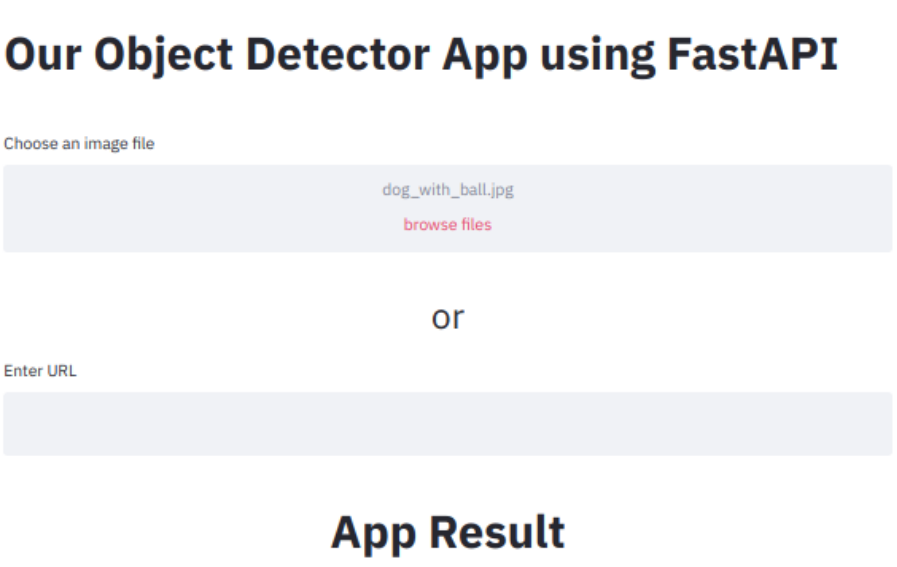
```
import streamlit as st
import base64
import io
import requests,json
from PIL import Image
import cv2
import numpy as np
import matplotlib.pyplot as plt
import requests
import random

# use file uploader object to recieve image
# Remember that this bytes object can be used only once
def bytesioObj_to_base64str(bytesObj):
    return base64.b64encode(bytesObj.read()).decode("utf-8")

# Image conversion functions

def base64str_to_PILImage(base64str):
    base64_img_bytes = base64str.encode('utf-8')
    base64bytes = base64.b64decode(base64_img_bytes)
    bytesObj = io.BytesIO(base64bytes)
    img = Image.open(bytesObj)
    return img

def PILImage_to_cv2(img):
    return np.asarray(img)

def ImgURL_to_base64str(url):
    return base64.b64encode(requests.get(url).content).decode("utf-8")

def drawboundingbox(img, boxes,pred_cls, rect_th=2, text_size=1, text_th=2):
    img = PILImage_to_cv2(img)
    class_color_dict = {}

    #initialize some random colors for each class for better looking bounding boxes
    for cat in pred_cls:
        class_color_dict[cat] = [random.randint(0, 255) for _ in range(3)]

    for i in range(len(boxes)):
        cv2.rectangle(img, (int(boxes[i][0][0]), int(boxes[i][0][1])),
                      (int(boxes[i][1][0]),int(boxes[i][1][1])),
                      color=class_color_dict[pred_cls[i]], thickness=rect_th)
        cv2.putText(img,pred_cls[i], (int(boxes[i][0][0]), int(boxes[i][0][1])),  cv2.FONT_HERSHEY_SIMPLEX,
text_size, class_color_dict[pred_cls[i]],thickness=text_th)
    plt.figure(figsize=(20,30))
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
    plt.show()

st.markdown("<h1>Our Object Detector App using FastAPI</h1><br>", unsafe_allow_html=True)

bytesObj = st.file_uploader("Choose an image file")

st.markdown("<center><h2>or</h2></center>", unsafe_allow_html=True)

url = st.text_input('Enter URL')

if bytesObj or url:
    # In streamlit we will get a bytesIO object from the file_uploader
    # and we convert it to base64str for our FastAPI
    if bytesObj:
        base64str = bytesioObj_to_base64str(bytesObj)

    elif url:
        base64str = ImgURL_to_base64str(url)

    # We will also create the image in PIL Image format using this base64 str
    # Will use this image to show in matplotlib in streamlit
    img = base64str_to_PILImage(base64str)
```

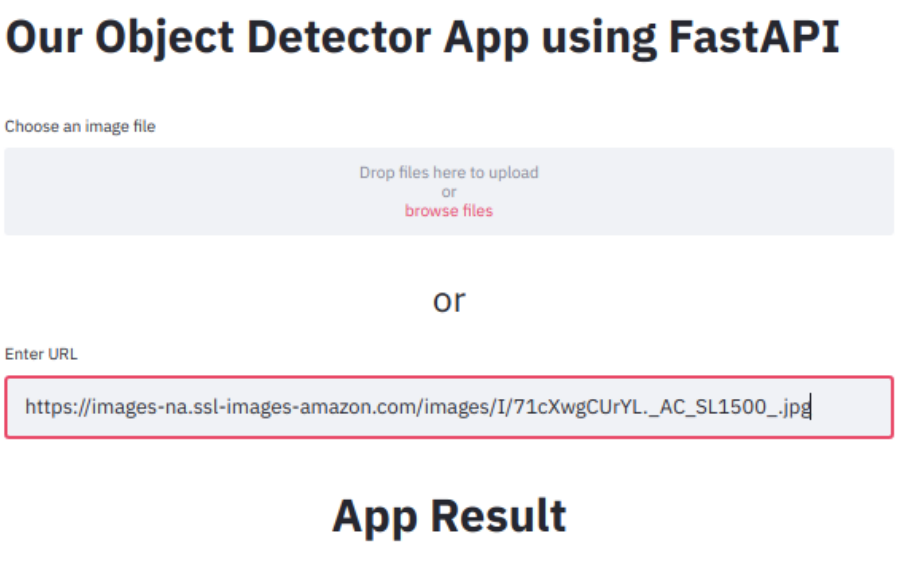
```
# Run FastAPI
payload = json.dumps({
  "base64str": base64str,
  "threshold": 0.5
})

response = requests.put("http://18.237.28.174/predict",data = payload)
data_dict = response.json()


st.markdown("<center><h1>App Result</h1></center>", unsafe_allow_html=True)
drawboundingbox(img, data_dict['boxes'], data_dict['classes'])
st.pyplot()
st.markdown("<center><h1>FastAPI Response</h1></center><br>", unsafe_allow_html=True)
st.write(data_dict)
```

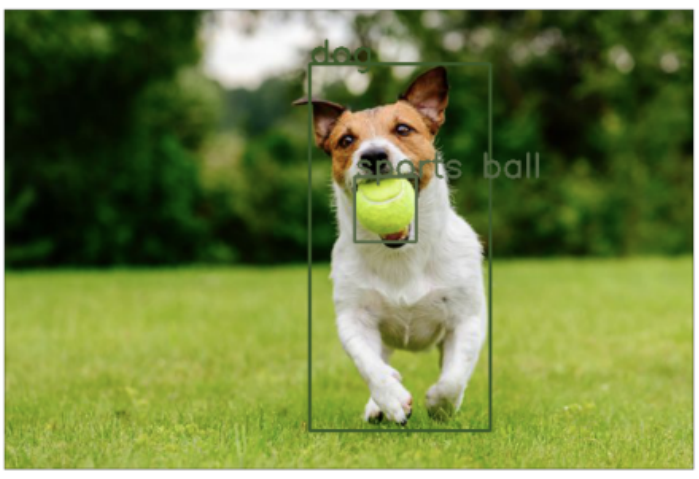
We can run this streamlit app in local using:

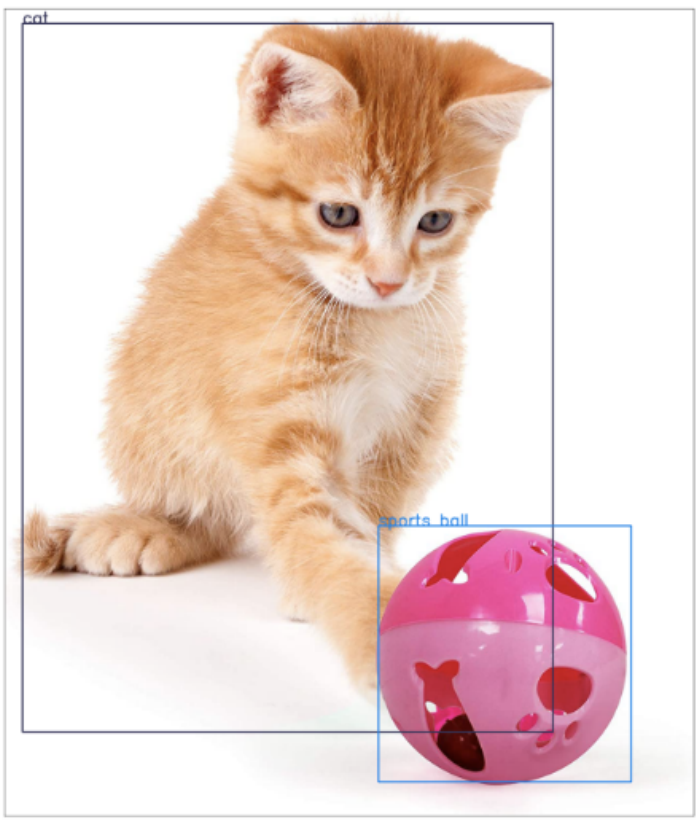
```
streamlit run streamlitapp.py
```

And we can see our app running on our localhost:8501. Works well with user-uploaded images as well as URL based images. Here is a cat image for some of you cat enthusiasts as well.









FastAPI Response


```
{
  "boxes" : [
    0 : [
      0 : [
        0 : 272.754638671875
        1 : 48.12982177734375
      ]
      1 : [
        0 : 433.81219482421875
        1 : 376.4888916015625
      ]
    ]
    1 : [
      0 : [
        0 : 312.01104736328125
        1 : 149.79376220703125
      ]
      1 : [
        0 : 367.9121398925781
        1 : 207.1046142578125
      ]
    ]
  ]
  "classes" : [
    0 : "dog"
    1 : "sports ball"
  ]
}
```


FastAPI Response


```
{
  "boxes" : [
    0 : [
      0 : [
        0 : 33.397422790527344
        1 : 27.73476791381836
      ]
      1 : [
        0 : 1017.8648071289062
        1 : 1343.2783203125
      ]
    ]
    1 : [
      0 : [
        0 : 693.4192504882812
        1 : 960.0377807617188
      ]
      1 : [
        0 : 1162.64599609375
        1 : 1436.177734375
      ]
    ]
  ]
  "classes" : [
    0 : "cat"
    1 : "sports ball"
  ]
}
```

So that's it. We have created a whole workflow here to deploy image detection models through FastAPI on ec2 and utilizing those results in Streamlit. I hope this helps your woes around deploying models in production. You can find the code for this post as well as all my posts at my GitHub repository.

Let me know if you like this post and if you would like to include Docker or FastAPI or Streamlit in your day to day deployment needs. I am also looking to create a much detailed post on Docker so follow me up to stay tuned with my writing as well. Details below.

Continue Learning

If you want to learn more about building and putting a Machine Learning model in production, this course on AWS for implementing Machine Learning applications promises just that.

Thanks for the read. I am going to be writing more beginner-friendly posts in the future too. Follow me up at Medium or Subscribe to my blog

Also, a small disclaimer — There might be some affiliate links in this post to relevant resources, as sharing knowledge is never a bad idea.

Your first name

Your email address

Get FREE "Advanced Python Tricks" Book

ALSO ON MLWHIZ.COM

Adding Interpretability to Multiclass Text …
a year ago • 1 comment
This post is about interpreting complex text classification models.

A Newspaper for COVID-19 — The …
7 months ago • 2 comments
This post is about how I created the Corona news dashboard

3 Programming concepts for Data …
a year ago • 3 comments
This post is about fast-tracking that study and panning some essential …

0 Comments mlwhiz.com Disqus' Privacy Policy 1 Login

Recommend Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

Subscribe Add Disqus to your siteAdd DisqusAdd Do Not Sell My Data

Support Me on Ko-fi

About Me

I’m a data scientist consultant and big data engineer based in Bangalore, where I am currently working with WalmartLabs .

Know More

Topics

Awesome Guides
Big Data
Computer Vision
Data Science
Deep Learning
Learning Resources
Natural Language Processing
Programming

Tags

Algorithms Artificial Intelligence Dask Deployment Ec2 Generative Adversarial Networks Graphs Image Classification Instance Segmentation Interpretability Jobs Kaggle Language Modeling Machine Learning Math Multiprocessing Object Detection Opinion Pandas Production Productivity Python Pytorch Spark Sql Statistics Streamlit Text Classification Timeseries Tools Transformers Translation Visualization Xgboost

Connect With Me

A SIMPLE BOOK ON ADVANCED PYTHON CONCEPTS
BY RAHUL AGARWAL

Join our newsletter

Receive a FREE PDF on Advanced Python Tricks. Also, get my Hand Picked Tutorials and manually curated courses to help you master ML and DS.

Your email address

Send it my way!

PYTORCH
VISION AND NLP
BY RAHUL AGARWAL

Is Deep Learning too Hard?

Lets make it a little easier. Subscribe here to get all the updates for my Upcoming Book

Your email address

Let Me Know!

MLWhiz

Contact Me

India, Bangalore

rahul@mlwhiz.com

Social Contacts

Linkedin

Medium

Twitter

Facebook

Github

Categories

Awesome Guides

Big Data

Computer Vision

Data Science

Deep Learning

Learning Resources

Natural Language Processing

Programming

Quick Links

About

Post

Copyright © 2020 MLWhiz All Rights Reserved