**A**

# Project Report

**On**

# "Plant disease detection using ReactJS and React Native"


**Prepared by**

Jayshil Jatin Patel (18EC068)

**Under the guidance of**

Prof. Vishal Tank


**Submitted to**

Charotar University of Science & Technology
For partial fulfillment of the requirements for the
Degree of Bachelor of Technology
In Electronics & Communication
EC448 - Project
Of 8th Semester of B.Tech

**Submitted at**



Department of Electronics & Communication
Faculty of Technology & Engineering, CHARUSAT
Chandubhai S. Patel Institute of Technology
At: Change, Dist: Anand - 388421


April 2022

CHARUSAT
CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY

# CERTIFICATE

This is to certify that the report entitled **"Plant disease detection using ReactJS and React Native"** is a bonafide work carried out by **Jayshil Jatin Patel (18EC068)** under the guidance and supervision of **Prof. Vishal Tank** for the subject **Project (EC448)** of 8th Semester of Bachelor of Technology in Electronics & Communication at Faculty of Technology & Engineering (C.S.P.I.T) - CHARUSAT, Gujarat.

To the best of my knowledge and belief, this work embodies the work of the candidate himself, has duly been completed, and fulfills the requirement of the ordinance relating to the B.Tech. degree of the University and is up to the standard in respect of the content, presentation, and language for being referred to the examiner.

Under the supervision of,

Prof. Vishal Tank
Assistant Professor
Department of EC Engineering,
C.S.P.I.T., CHARUSAT-Changa.

Dr. Trushit K. Upadhyaya
Head of Department,
Department of Electronics & Communication
C.S.P.I.T., CHARUSAT-Changa. Gujarat.

---

**Chandubhai S Patel Institute of Technology (C.S.P.I.T.)**
**Faculty of Technology & Engineering, CHARUSAT**
At: Changa, Ta. Petlad, Dist. Anand, Gujarat - 388421

# ABSTRACT

The user interface of a simple machine learning model is inadequate and does not match up to the mark of what the machine learning model is capable of doing. The user must be very competent with machine learning to extract the information and must have a technical background for doing so. Hence, the models (plant disease detection, property price prediction) which are solely made to ease the job of people from non-technical backgrounds (farmers, economists) would always require a technician to troubleshoot the issues they encounter while they work.

A graphical user interface would ease the work and would provide a better user experience for the users. Hence this project exports a model and a website (GUI) would access the model using API (Application program interface) for a better user experience. In this case, the user would not have to go through a machine learning model to give input and have a hard time extracting the output.

# ACKNOWLEDGMENT

I would like to express my sincere thanks to my internal guide, Prof. Vishal Tank, for his valuable guidance, enthusiastic attitude, and encouragement throughout the period of the project and thesis work. His guidance, suggestion, and very constructive appreciation have contributed immensely to the evolution of the project.

I would also like to express my sincere gratitude and sincere thanks to **Dr. Trushit Upadhyaya** Hod, Department of Electronics and Communication Engineering, Charotar University of Science and Technology, Change for providing me the opportunity to undertake such a challenging and intellectual work.

**Jayshil Jatin Patel (18EC068)**
**Department of EC Engineering,**
**CSPIT, CHARUSAT.**

# MOTIVATION

Machine learning models are very complicated to look at, and their non-user friendliness is highly renowned. When I was doing a course on web development, the idea of combining the machine learning model with a frontend (website) to create a fully functional application that aids people from non-technical backgrounds. This improvement in user experience could help to gain consumers' confidence and provide the results/information that they are looking for.

This project's CNN model is targeted to a very specific group of individuals, mainly farmers, and it is very highly unlikely that an average farmer could access the machine learning model straight out of a jupyter notebook. Here, a great website to access the same makes sense. It would not only improve the user experience as expected but it would also make the job of farmers easier by giving them proper results with ease.

# List of Figures

# Code snippets

# Table of Contents

# Chapter 1: Introduction

With the evolution of technology, programmers can develop and train a CNN model which takes chest x-rays, brain x-rays, or even images of alphabets and classify them with an insignificant range of error. The person who develops a model does not need to have any knowledge regarding the classification the person is working on. For instance, a CNN developer can make a model to classify brain tumor x-rays from normal brain x-rays without any knowledge of radiology, this same concept applies to classify leaves.

## 1.1 Problem statement

Farmers face a huge economic loss due to fungal and viral diseases in potato plants. There are two common diseases, early blight (caused by a fungus), and late blight (caused by specific microorganisms). If farmers can detect these diseases early and implement appropriate treatment then it can prevent a lot of waste and save the economic loss. As a result, this CNN model is made to help farmers to shorten the extent of their economic loss and aid their production per hectare.

The images given below show the difference between healthy and unhealthy potato plant leaves. Upon a close inspection, a normal person can easily differentiate between a healthy and an unhealthy leaf. But while differentiating between two infected plants becomes a tedious task. Here, a pathologist or botanist would easily separate the infected plant and classify them, but a CNN model could also do so.

Figure 1.1 (Healthy plant leaf)

Figure 1.2 (Early Blight)

Figure 1.3 (Late Blight)

## 1.2 Technical Architecture

The project includes an array of tools and technologies to cope with the issues faced by the farmers. The front-end is responsible for a better user interface and user experience, while the backend is composed of python script, fast API, and saved CNN model used to serve information to the front-end.

The dataset is fed into the .ipnb file and the trained model using CNN is exported for TF serving. The TF serving serves the model and front end website made using ReactJS uses a fast API to access this model. On average, it takes 40 minutes to train the model and a couple of seconds to predict the result post-training.

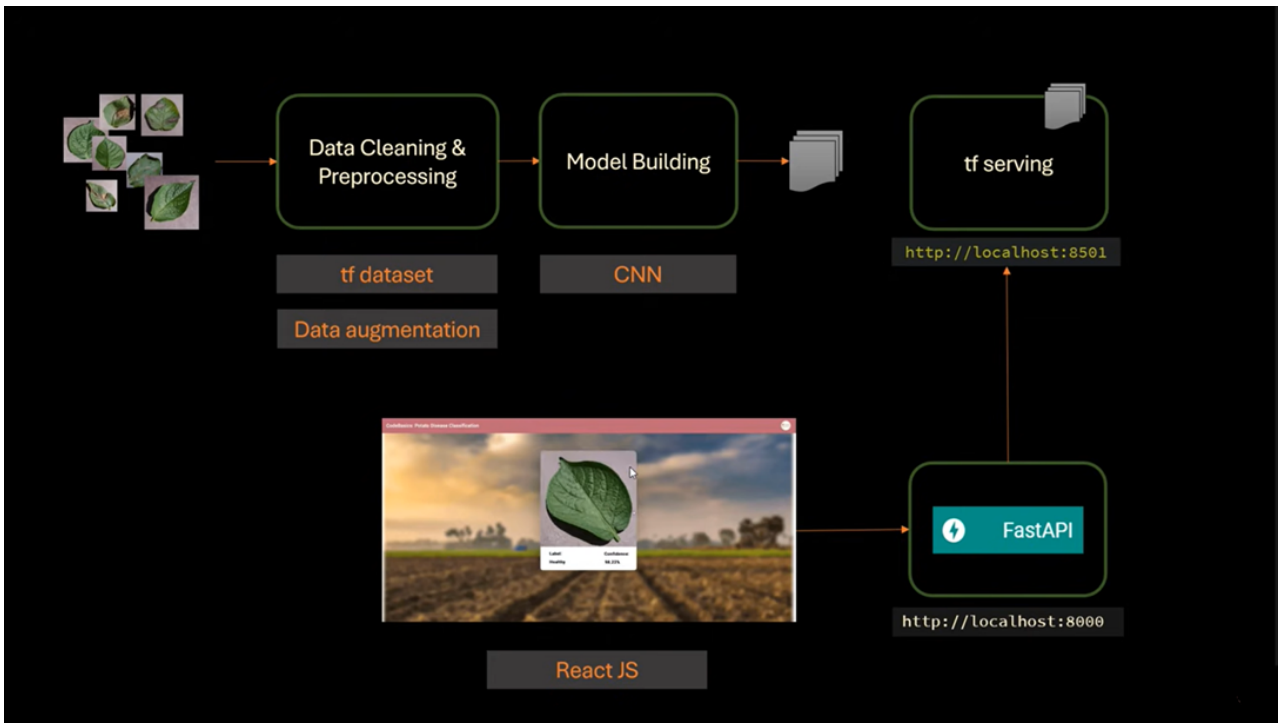


Figure 1.4 (Flowchart of the project)

### 1.2.1 Back-end Architecture

The model is made using a jupyter notebook (.ipynb file) and the TF serving is pulled as a container using docker. This TF serving is started using a python script which fetches the model and provides the result to the front-end.

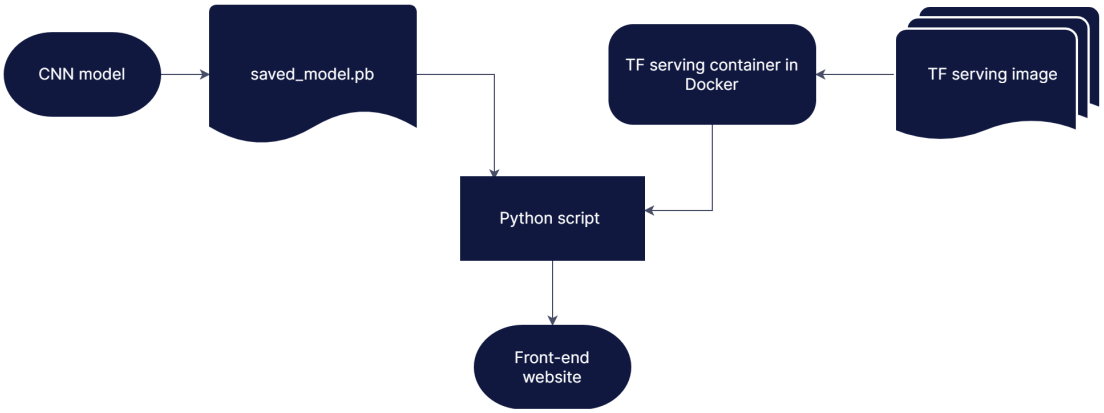


Figure 1.5 (Flowchart of the Back-end)

### 1.2.2 Front-end Architecture

The front end is composed of a website. This website is made up using HTML, CSS, and reactJS. These technologies are used to make a website that will provide a better user interface for the model to be served. FastAPI is used to connect our front-end server with our backend server.

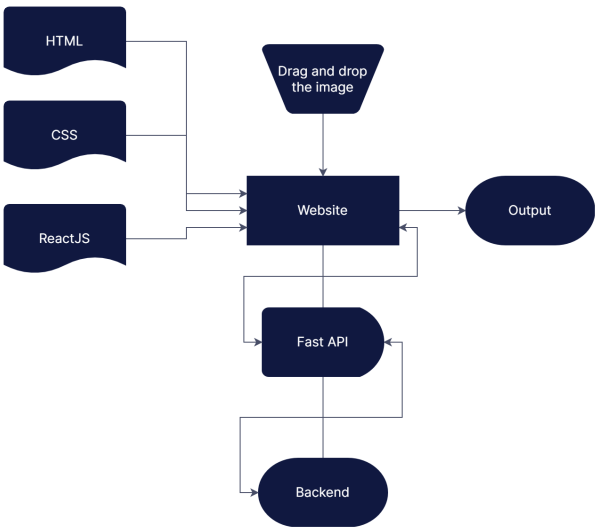


Figure 1.6 (Flowchart of the front end)

# Chapter 2: Convolution Neural Network

Convolutional neural network (CNN), a type of artificial neural network that has been dominant in numerous computer vision tasks, is gaining popularity in a range of fields, including radiology. CNN is aimed to learn spatial hierarchies of features automatically and adaptively by backpropagation by utilizing several building blocks such as convolution layers, pooling layers, and fully connected layers.

CNN is a deep learning model for processing data with a grid pattern, such as photographs, that is inspired by the architecture of the animal visual cortex and intended to learn spatial hierarchies of characteristics automatically and adaptively, from low- to high-level patterns. CNN is a mathematical construct made up of three kinds of layers (or building blocks): convolutional, pooling, and fully connected layers. The first two layers, convolution, and pooling extract feature, while the third, a fully connected layer, transfers the extracted features into the final output, such as categorization. A convolution layer is essential in CNN, which is made up of a stack of mathematical operations such as convolution, a sort of linear operation. Pixel values in digital images are stored in a two-dimensional (2D) grid, i.e., an array of numbers and a small grid of parameters called the kernel, and an optimizable feature extractor is applied at each image position, making CNN extremely efficient for image processing because a feature can occur anywhere in the image. Extracted features can evolve hierarchically and progressively more complicated as one layer feeds its output into the next layer.

## 2.1 Python for building CNN

Convolution neural networks can be implemented using various programming languages such as MATLAB (Matrix Laboratory), Python, JavaScript, etc. But, Python is most preferred because of its widest community on stack overflow (website to troubleshoot the issues by the community). Furthermore, using a python notebook (Jupyter notebook) is fairly simpler than learning a whole new programming language for the same purpose.

Furthermore, there are an array of libraries and modules available to aid the programming and making earlier for us to make a CNN model. For instance, Google’s TensorFlow library is renowned for making machine learning models. If a computer is having CUDA activated GPU, then the training time of CNN also decreases significantly because TensorFlow divides the task among CPU and GPU.

## 2.2 Epochs

The number of epochs is a hyperparameter that defines the number of times that the learning algorithm will work through the entire training dataset. One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters. An epoch is comprised of one or more batches.

```
history = model.fit(
    train_ds,
    batch_size=BATCH_SIZE,
    validation_data=val_ds,
    verbose=1,
    epochs=50,
)
```

Code snippet 2.1 (Number of Epochs)

In our model, there is a total of 50 epochs for training the model with 54 steps in each model. Considering 998ms/step as an average and each epoch having 54 of these steps, we predict the estimated time to train the model using the following formula.

Estimated time = Number of Epochs * steps per epoch * time per step.
In our case, we get 50*54*998ms = 2,694,600ms ≈ 45 minutes.

```
Epoch 48/50
54/54 [==============================] - 54s 995ms/step - loss: 0.0325 - accuracy: 0.9850 - val_loss: 0.0366 - val_accuracy: 0.9844
Epoch 49/50
54/54 [==============================] - 54s 995ms/step - loss: 0.0198 - accuracy: 0.9936 - val_loss: 0.0520 - val_accuracy: 0.9844
Epoch 50/50
54/54 [==============================] - 54s 994ms/step - loss: 0.0255 - accuracy: 0.9890 - val_loss: 0.2266 - val_accuracy: 0.9375
```

Figure 2.1 (training epochs)

# 2.3 Libraries Required

## 2.3.1 Back-end libraries

For building the CNN model, the TensorFlow library is required which included Keras and its different models. Matplotlib is required to plot accuracy and validation loss for reference purposes. IPython.display is used to imbibe HTML in the ipynb file. OS module is used to access the directory structure using python and read, write and modify files in the operating system, in this case, we have used os to save the model in a specific sequential directory structure named according to the modified date to the saved model. Finally, NumPy is used to convert arrays of binary data of images to appropriate forms and dimensions for proper functionality.

```
import tensorflow as tf
from tensorflow.keras import models, layers
import matplotlib.pyplot as plt
from IPython.display import HTML
import numpy as np
import os
```

Code snippet 2.2 (libraries for CNN)

For building the backend server, we will require FastAPI as discussed earlier to connect our backend with the front-end. Uvicorn is used to start the server on localhost (similar to a live server). Here numpy is used to resolve the dimensions of the received image from the web or our API server. IO module is used to convert the values of a required image to bytes for proper functionality. PIL is used to rectify the dimensions of the input image to feed into the model. CORSMiddleware is a library to resolve the errors raised due to “cross-origin resource sharing”. This topic will be discussed in detail in the further part of the report.

```
import numpy as np
from fastapi import FastAPI, UploadFile, File
import uvicorn
import numpy as np
from io import BytesIO
from PIL import Image
import tensorflow as tf
from fastapi.middleware.cors import CORSMiddleware
import requests
```

Code snippet 2.3 (libraries for backend python script)

### 2.3.2 Front-end Libraries

Backend is composed of an array of libraries that are used to add beauty to the website. These imports are imported using NPM (node package manager)

```
import { useState, useEffect } from "react";
import { makeStyles, withStyles } from "@material-ui/core/styles";
import AppBar from "@material-ui/core/AppBar";
import Toolbar from "@material-ui/core/Toolbar";
import Typography from "@material-ui/core/Typography";
import Avatar from "@material-ui/core/Avatar";
import Container from "@material-ui/core/Container";
import React from "react";
import Card from "@material-ui/core/Card";
import CardContent from "@material-ui/core/CardContent";
import { Paper, CardActionArea, CardMedia, Grid, TableContainer, Table, TableBody, TableHead, TableRow,
TableCell, Button, CircularProgress } from "@material-ui/core";
import cblogo from "./cblogo.PNG";
import image from "./bg.png";
import { DropzoneArea } from 'material-ui-dropzone';
import { common } from '@material-ui/core/colors';
import Clear from '@material-ui/icons/Clear';
```

Code snippet 2.4 (imports for home.js)

There are versions of the libraries which are changed and updated on a regular basis. So if a code works perfectly, but after the update of the libraries, the code might not work in the same way. Hence, specific versions of the code are required to initialize the imports.

```
{
 "name": "photo",
 "version": "0.1.0",
 "private": true,
 "dependencies": {
  "@material-ui/core": "^4.11.4",
  "@material-ui/icons": "^4.11.2",
  "axios": "^0.21.4",
  "eslint-plugin-react-hooks": "^4.2.1-rc.2-next-2bf7c02f0-20220314",
  "material-ui-dropzone": "^3.5.0",
  "react": "^17.0.2",
  "react-dom": "^17.0.2",
```

```
    "react-scripts": "^4.0.3",
    "web-vitals": "^1.1.2"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Code snippet 2.5 (Versions of imported modules)

## 2.4 Dataset

The dataset is imported from Kaggle (Kaggle, a subsidiary of Google LLC, is an online community of data scientists and machine learning practitioners). PlantVillage directory consists of an array of categories to choose from as shown in the given figure.

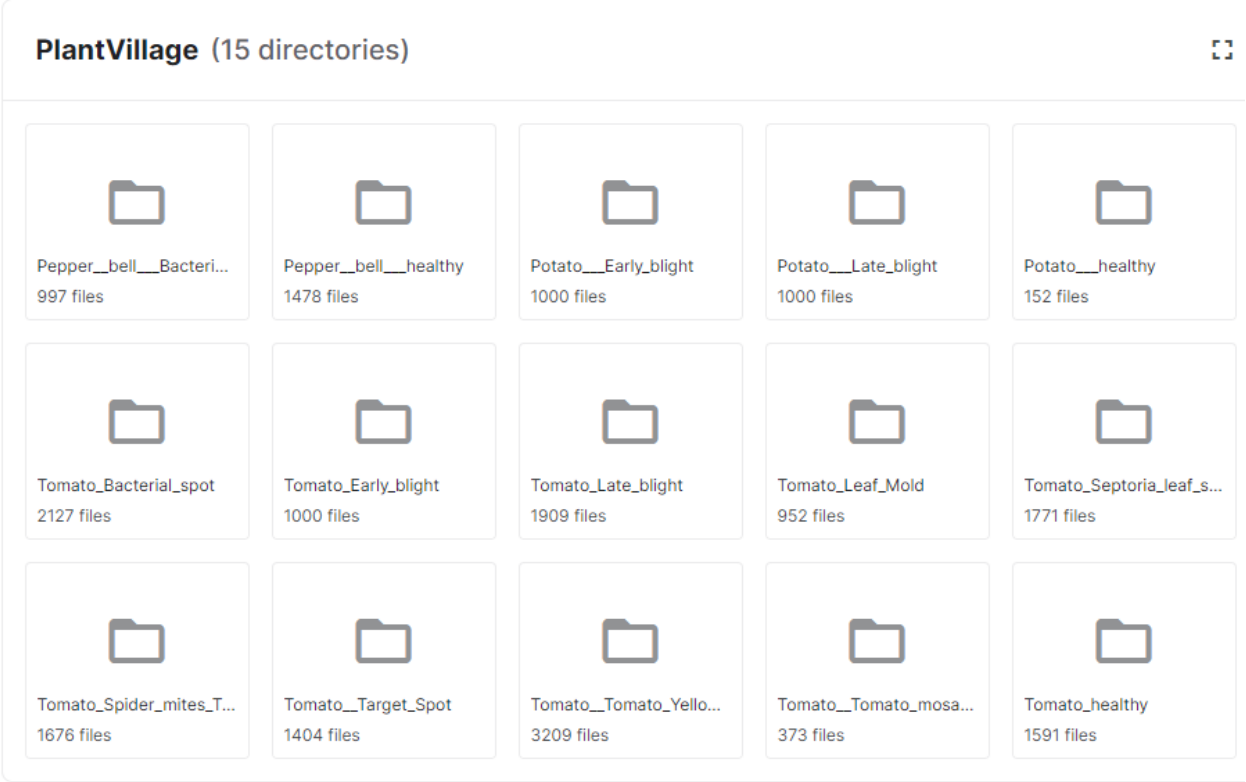


Figure 2.2 (Plant Village directories)

For potato disease classification, we would only use three directories labeled “potato healthy”, “potato late blight”, and “potato early blight”.

The dataset plays a crucial role in building the CNN model. The dataset is used to generate a model and adjust weights, if the wrong dataset is provided then the weights would not be adjusted as they should be, and hence the accuracy of the model would be hindered. The range of the dataset, along with the partition of training, validation and test also play a crucial role in determining the accuracy and time elapsed on training.

### 2.4.1 Dataset directory structure.

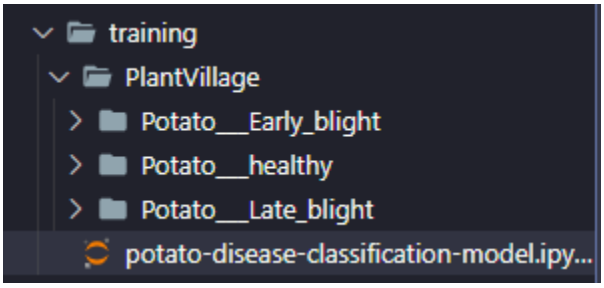


Figure 2.3 (Directory structure of the dataset)

The dataset ("PlanVillage") is placed into the "training" folder which contains the .ipynb file which is the CNN model. The class names should be carefully written such that they are identical to the directory names.

In total, there are 2152 images

- 1000 Early blight
- 1000 Late Blight
- 152 healthy images

```
[34]  class_names = dataset.class_names
      class_names

...   ['Potato___Early_blight', 'Potato___Late_blight', 'Potato___healthy']
```

Figure 2.4 (Class names)

### 2.4.2 Dataset categories

As mentioned the dataset is categorized into three parts:-

- Training: Dataset to be used while training.
- Validation: Dataset to be tested against while training.
- Test: Dataset to be tested against after we trained a model.

To make partitions in the dataset, a function is made to ease and automate the task.

```
def get_dataset_partitions_tf(ds, train_split=0.8, val_split=0.1, test_split=0.1, shuffle=True,
shuffle_size=10000):
    assert (train_split + test_split + val_split) == 1

    ds_size = len(ds)

    if shuffle:
        ds = ds.shuffle(shuffle_size, seed=12)

    train_size = int(train_split * ds_size)
    val_size = int(val_split * ds_size)

    train_ds = ds.take(train_size)
    val_ds = ds.skip(train_size).take(val_size)
    test_ds = ds.skip(train_size).skip(val_size)

    return train_ds, val_ds, test_ds
```

Code snippet 2.6 (function to split dataset)

This function will return the training dataset, test dataset, and validation dataset. There is a total of 32 batches and each batch would have 68 images in total, this results in a total number of images when multiplied. Hence, according to the above function, we would have 54 training images, 6 validation, and 8 testing images which sums up to 68 images as stipulated.

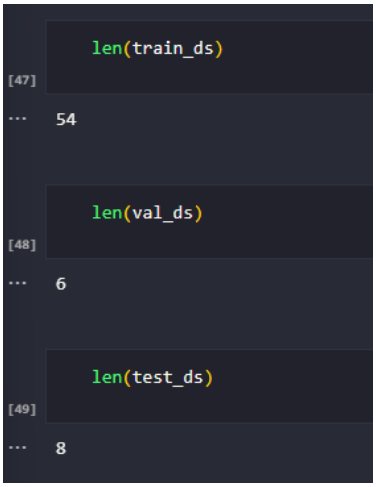


Figure 2.5 (Train, Test, Validation length)

## 2.5 Model Architecture

At first, all the dependencies are imported into the .ipynb file. Followed by setting all the constants like batch size, image size, channels, and epochs. Then the data is imported into the TensorFlow dataset object and class names are obtained. The dataset is split into three categories as mentioned earlier. After cache, shuffling, and prefetching the dataset, we would start to build the model.

The model starts after creating a layer for resizing and normalization for data augmentation. After adding all the layers to our sequential model we review the summary of it. After compiling the model for 45 minutes, we plot the accuracy and loss curves to review our trained model's performance. Now I ran a few inferences on a few sample images to test the performance and functionality of the model. Finally, using to OS module, we will save the model for future purposes to build the website.

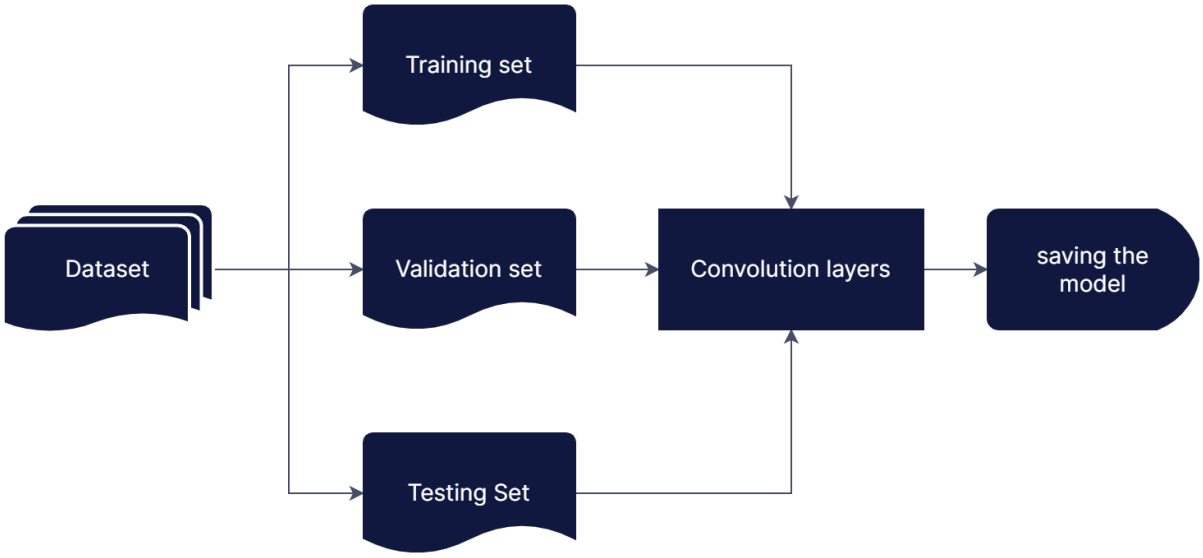


Figure 2.6 (Model Architecture)

## 2.6 Running prediction on sample images

After compiling the model, we try to view the result and see our model in action for the first time using the following code.

```
import numpy as np
for images_batch, labels_batch in test_ds.take(1):

  first_image = images_batch[0].numpy().astype('uint8')
  first_label = labels_batch[0].numpy()

  print("first image to predict")
  plt.imshow(first_image)
  print("actual label:",class_names[first_label])

  batch_prediction = model.predict(images_batch)
  print("predicted label:",class_names[np.argmax(batch_prediction[0])])
```

Code snippet 2.7 (Running prediction on a sample image)

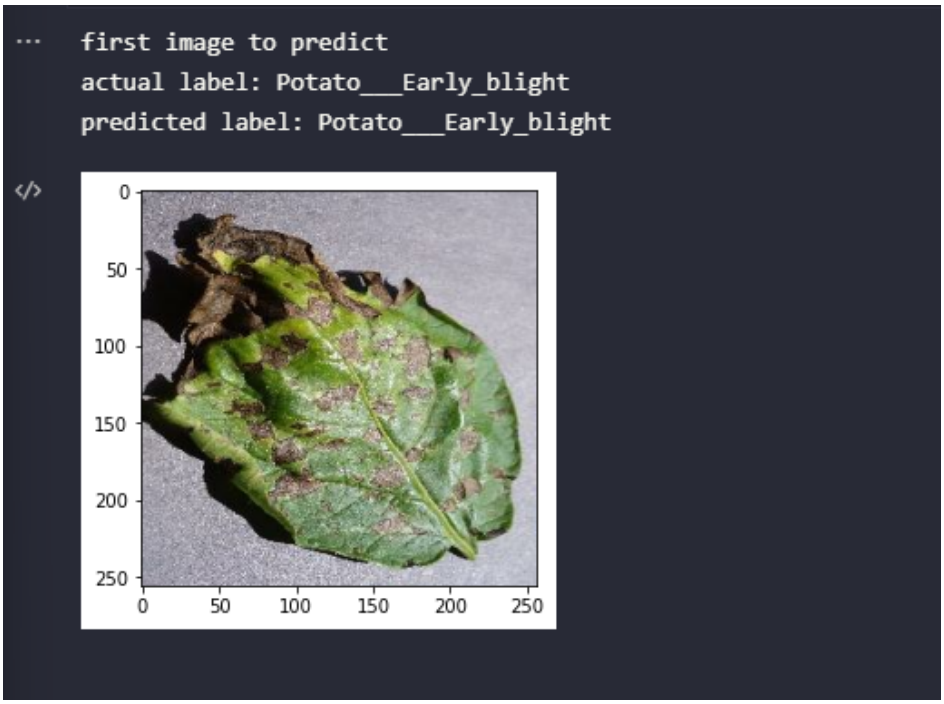


Figure 2.7 (Result of predicting a sample image)

Here we can observe that the predicted label matches the actual label. Hence we can conclude that our model works perfectly as predicted.

## 2.7 Accuracy of the model

The accuracy of the model plays a crucial role since the decision to be made depends upon the accuracy of the model.

The images below show the change in the accuracy of the model with a change in epochs. And by the end of the training, the accuracy of the model reaches around 99%

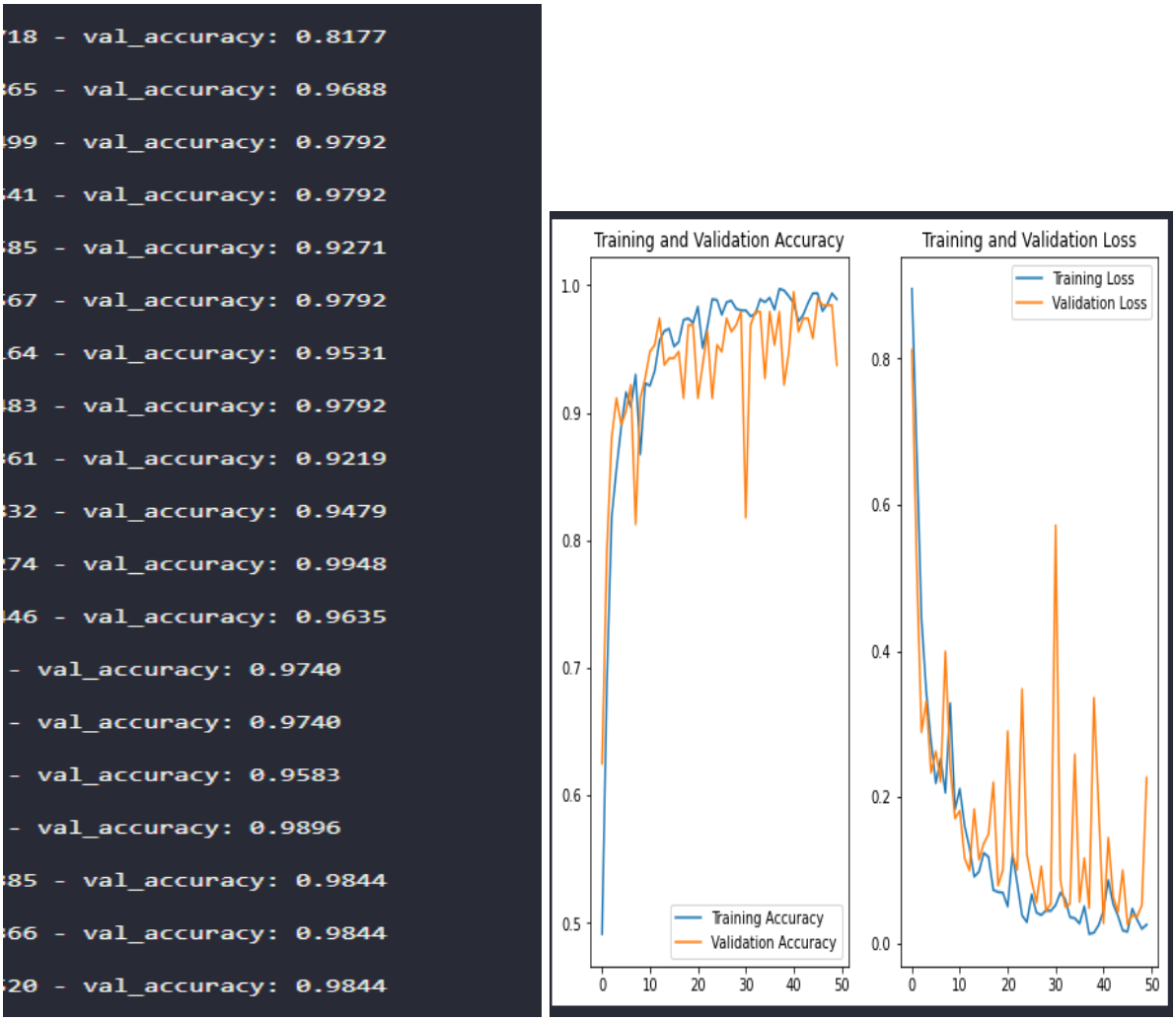


Figure 2.8 (Accuracy of the model)

## 2.8 Saving the model

Finally after verifying that the model works perfectly fine, as it should be, now it is time to export the model and export it to use TF serving on it for serving it to the front end.

```
import os
model_version=max([int(i) for i in os.listdir("../saved_models/") + [0]])+1
model.save(f"../models/{model_version}")
```

Code snippet 2.8 (Saving the model)

Using the OS module, we would use functions like listdir and save to access our file in the operating system. This code would save the model dynamically into the directory as specified and would name the saved model on the basis of its date of modification (The latest model would be ranked the highest). Using this concept we can access different versions of the model with different accuracy performing differently for a different set of images. Here, the concept of beta users comes into the picture.

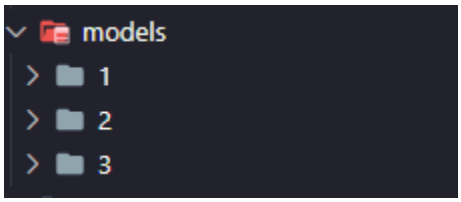


Figure 2.9 (Dynamic naming of the models)

# Chapter 3: Application Programming Interface

APIs or Application Programming Interface plays a crucial role to connect the frontend and the backend. The following image depicts the role of API in the most creative manner. For instance, consider you are in a restaurant and you order a dish. You cannot directly speak to the chef, instead, you would talk to the waiter to note your order. The waiter sends this request to the chef in FIFO order (First In First Out). Once the dish is ready, again, the chef will not be serving you the dish even though he deserves the credit for making it. It is the duty of the waiter to garnish the dish and make sure it reaches the correct table on time.

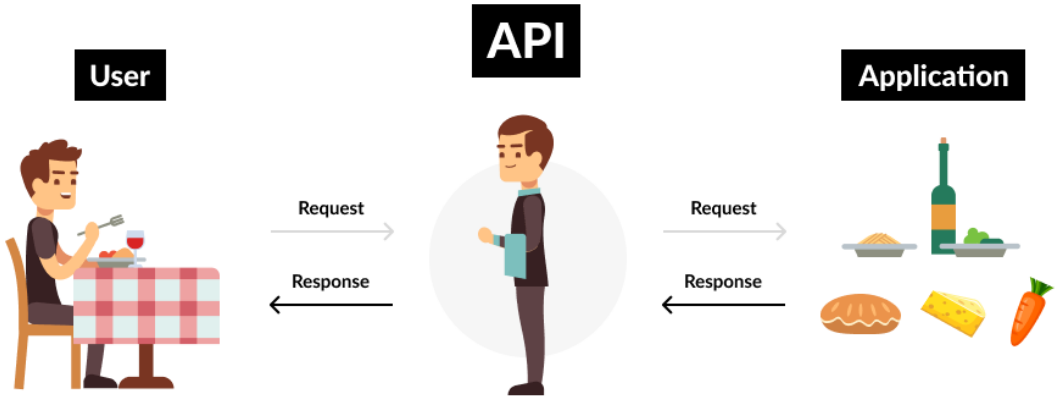


Figure 3.1 (API)

Using the same analogy, when you use an application on your mobile phone, the application connects to the Internet and sends data to a server. The server then retrieves that data, interprets it, performs the necessary actions, and sends it back to your phone. The application then interprets that data and presents you with the information you wanted in a readable way. This is what an API is - all of this happens via API.

# 3.1 Fast API

In this project, API is used to connect the front end and backend. We used FastAPI as our primary API and postman for intermediate research testing of the python script. FastAPI is a modern, high-performance web framework for building APIs with Python-based on standard type hints. It is fast to run and code, it has reduced the number of bugs and it is intuitive.

The following piece of code describes how the Fast API is initialized.

```
app = FastAPI()
@app.get("/ping")
async def ping():
    return "Hello, from Jayshil!"
```

Code snippet 3.1 (Initialization of FastAPI)

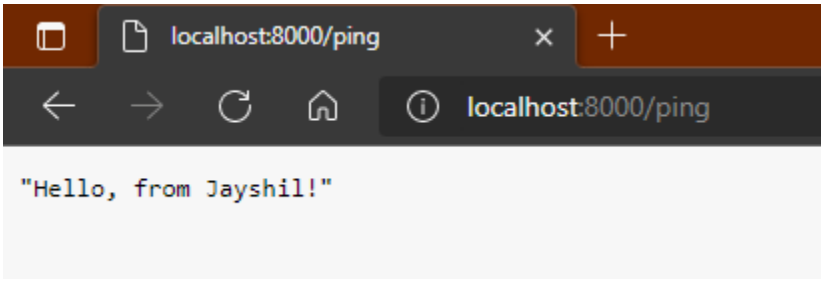


Figure 3.2 (Fast API ping)

There are various methods for FastAPI to choose from like POST, GET, PUT, DELETE, OPTIONS, HEAD, PATCH, and TRACE. These methods have different functions as the name suggests. For this project, we would be using the GET method to test the server and POST to get the predicted response.

# 3.2 Fast API documentation

Fast API provides an array of documentation options to explore the project and improve the API performance.

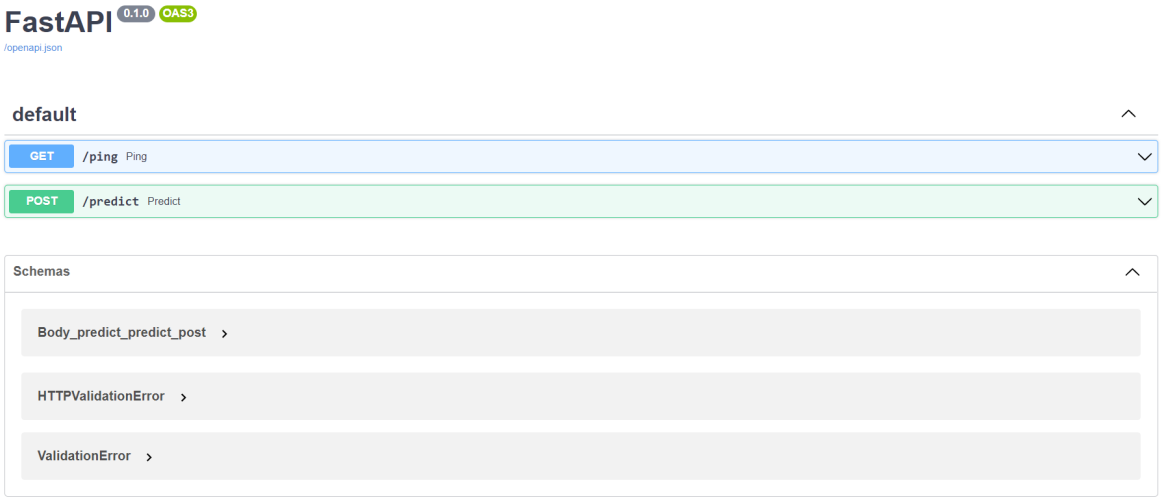


Figure 3.3 (Fast API docs)

We can try out individual methods for debugging purposes as portrayed in the following figure.

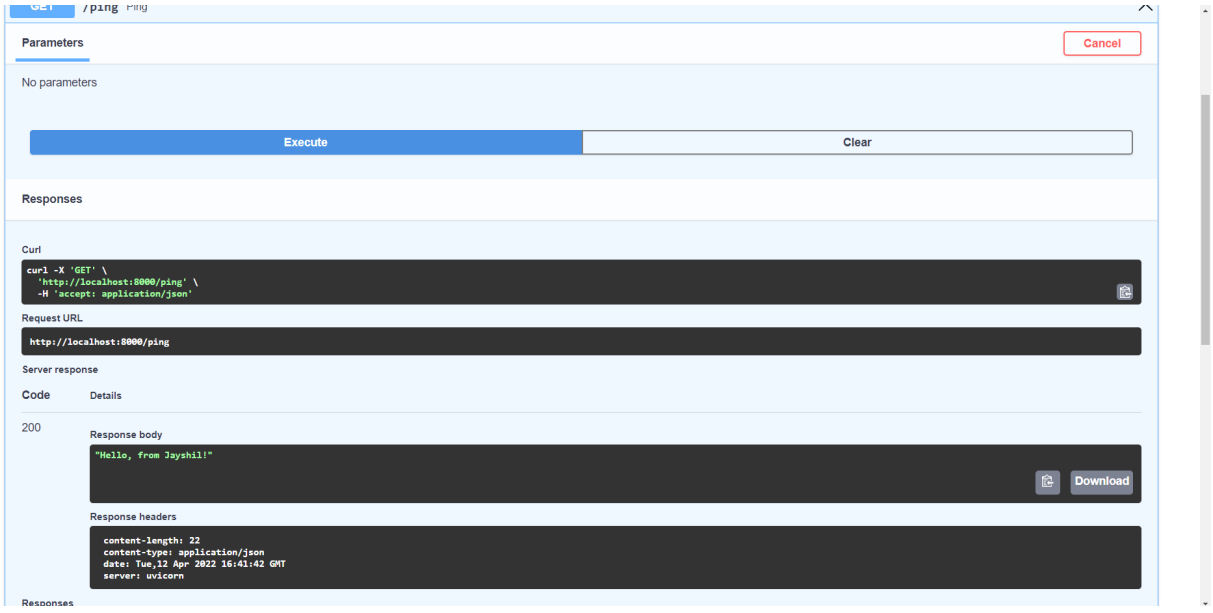


Figure 3.4 (Fast API debugging)

\

# Chapter 4: Docker

Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide your applications and services. Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

Consider the following example scenario:

- Your developers write code locally and share their work with their colleagues using Docker containers
- They use Docker to push their applications into a test environment and execute automated and manual tests.
- When developers find bugs, they can fix them in the development environment and redeploy them to the environment for testing and validation
- When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.

The container-based Docker technology enables extremely portable workloads. Docker containers can operate on a developer's laptop, physical or virtual computers in a data center, cloud providers, or a combination of these settings. Docker's mobility and lightweight characteristics also make it simple to dynamically manage workloads, scaling up or down apps and services in near real-time as business demands dictate. Docker is a lightweight and quick application. It offers a realistic, cost-effective alternative to hypervisor-based virtual machines, allowing you to make better use of your computing power to fulfill your business objectives. Docker is ideal for high-density situations as well as small and medium deployments where more can be done with fewer resources.

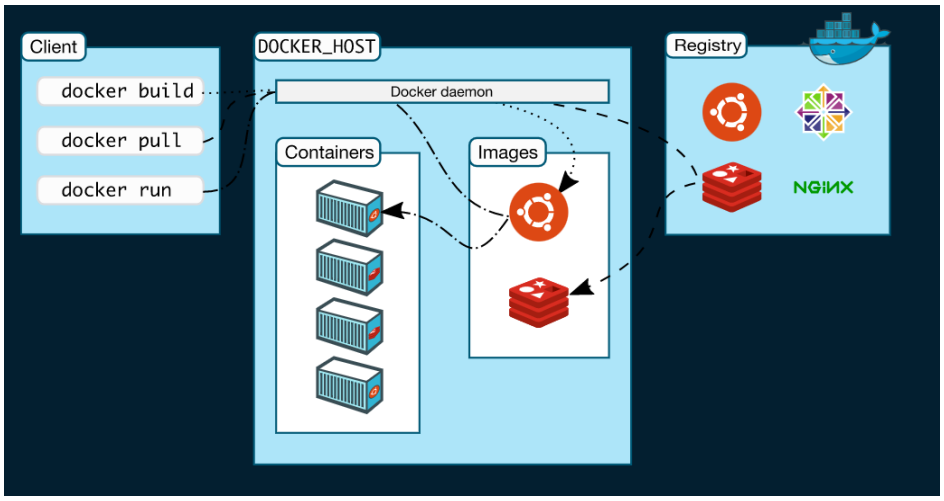


Figure 4.1 (Docker demonstration)

## 4.1 Docker and TensorFlow serving

TensorFlow serving depends on a lot of dependencies and to use TensorFlow serving, we need to install and maintain the specific versions to compel the code to work as it is supposed to be. There are various updates and performance enhancements of the libraries dropping out on daily basis, so to lock a specific version, we used the docker image which is available on the docker hub (similar to GitHub), and pull the image using the docker pull command. The TF serving image can be seen on the Docker desktop app.

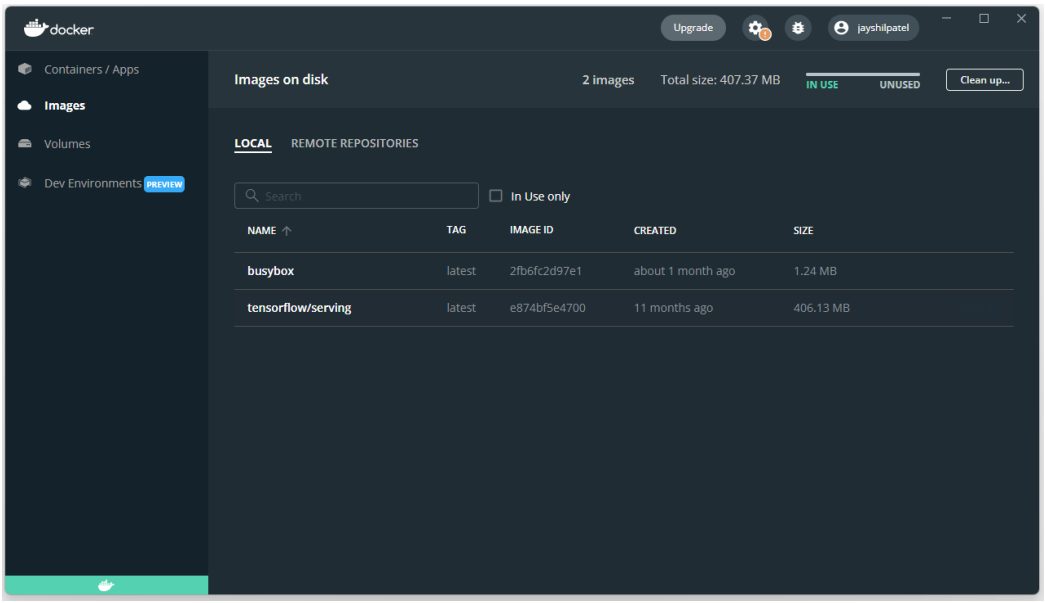


Figure 4.2 (TFserving image)

## 4.2 TensorFlow Serving

TensorFlow Serving is a production-ready, flexible, high-performance serving solution for machine learning models. TensorFlow Serving simplifies the deployment of new algorithms and experiments while maintaining the same server architecture and APIs. TensorFlow Serving integrates with TensorFlow models out of the box, but it can be readily expanded to serve other types of models and data. Can serve multiple models, or multiple versions of the same model simultaneously. Exposes both gRPC as well as HTTP inference endpoints. Allows deployment of new model versions without changing any client code. Supports canarying new versions and A/B testing experimental models.
Adds minimal latency to inference time due to efficient, low-overhead implementation.

## 4.3 Trigger command for Docker container

```
PS C:\Users\jaysh> docker run -t --rm -p 8502:8502 -v C:/Code/potato-disease:/potato-disease tensorflow/serving --rest_a
pi_port=8502 --model_config_file=/potato-disease/models.config
```

Figure 4.3 (Container trigger command)

The command shown in figure 21 extracts the pulled docker image of TensorFlow serving and runs it making it an active container. This container is mapped on port 8502 and the directory in which we would find the host file is mentioned in the next part of the trigger command (C:/Code/potato-disease:/potato-disease) which is mapped to the docker image's directory (tensorflow/serving). Rest API's (used by fast API) port is configured to run on the same port number as our docker container is running using a configuration file which will be discussed further.

## 4.4 Models.config file

models.config

```
model_config_list {
 config {
    name: 'potatoes_model'
    base_path: '/potato-disease/saved_models'
    model_platform: 'tensorflow'
    model_version_policy: {all: {}}
        }
}
```

Figure 4.4 (models.config file)

This file serves the correct and updated version of the models to TF serving which will ultimately serve the model to the website using API to generate results. The configuration list includes the name of the model and its base path. Instead of an absolute path, the relative path is provided concerning the docker container. Model version policy serves the latest version to the TF serving. Here, we can change it and set a specific model version as the default version to be served.

# Chapter 5: Backend

The server-side of a website is referred to as the backend. It saves and organizes data while also ensuring that everything on the client-side of the website functions properly. It is the section of the website that you are unable to view or interact with. It is the part of the software that has no direct touch with the users. Users have indirect access to the pieces and attributes created by backend designers via a front-end application. The backend also includes activities such as building APIs, generating libraries, and interacting with system components without user interfaces or even scientific programming systems.

The backend server in our program comprises various technologies, API script is written in python, and TF serving container in the Docker are the elements of the backend used in this project as shown in the images below.

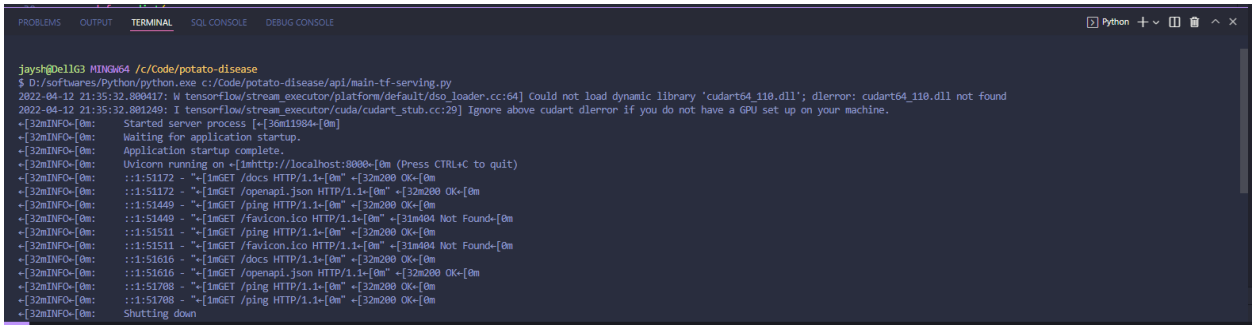


Figure 5.1 (API server)

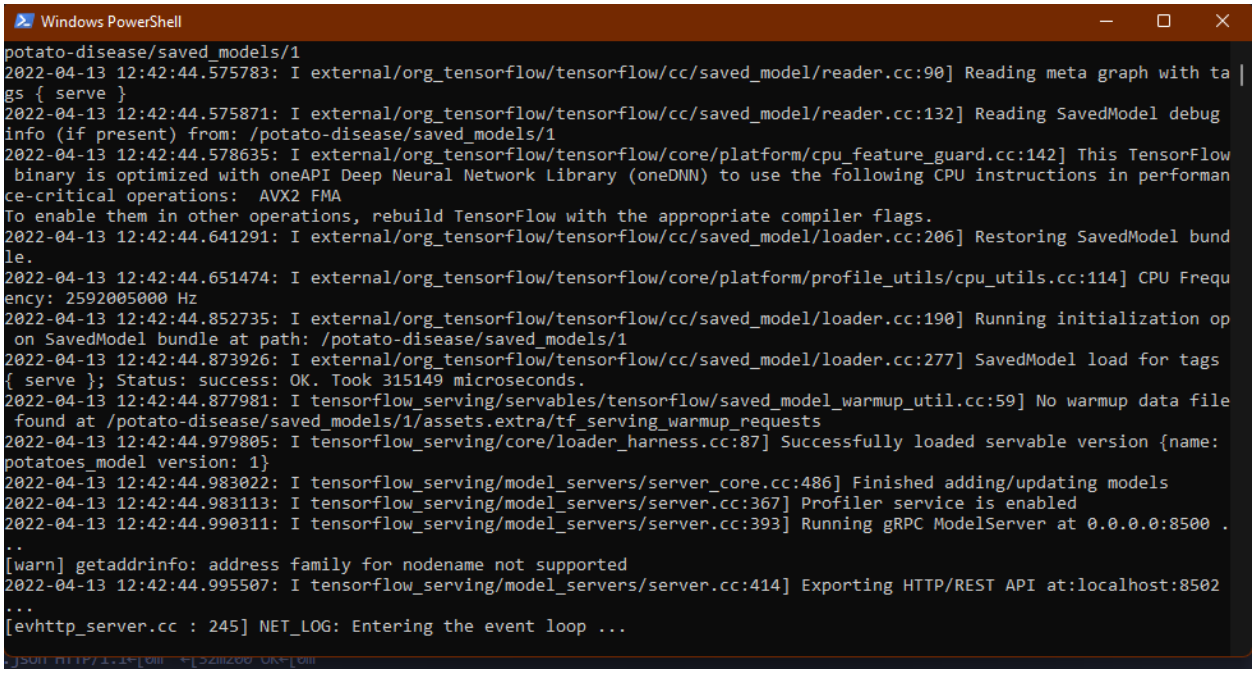


Figure 5.2 (Docker TF serving server)

## 5.2 Python script

The Python script is made to serve a Fast API-based server (placed in the “api” folder in the directory). The requirements.txt contains all the dependencies for making the server. The requirements can be initialized using “pip install -r requirements.txt”.

```
tensorflow==2.5.0
fastapi
uvicorn
python-multipart
pillow
tensorflow-serving-api==2.5.0
matplotlib
numpy
```

Code snippet 5.1 (requirements.txt)

The POST routine is used to create an endpoint name “predict” for the actual prediction of the received images. The argument of this function would be a file (image) sent by the website [Upload file to be specific].

```
@app.post("/predict")
async def predict(
  file: UploadFile = File(...)
```

Code snippet 5.2 (predict method)

After reading the file we get bytes as a return type, hence we must convert these bytes into a numpy array. This array is fed to the served model which runs prediction and returns class type and confidence.

```
  def read_file_as_image(data) -> np.ndarray:
        image = np.array(Image.open(BytesIO(data)))
        return image
  response=requests.post(endpoint,json=json_data)
  prediction=np.array(response.json()["predictions"][0])

  predicted_class=CLASS_NAMES[np.argmax(prediction)]
  confidence=np.max(prediction)

  return {
    "class":predicted_class,
    "confidence":float(confidence)
  }
```

Code snippet 5.3 (feeding image to the model)

## 5.3 Ports and endpoints

A web service endpoint is simply a web address (URL) through which clients of a specific service can gain access to it. Clients can access operations provided by that service by mentioning that URL. A port is a distinct endpoint that has its own address. A protocol is a set of rules that govern how people interact with one another. A message is a piece of jargon that facilitates communication. What operations this port may execute is defined by its port type. An operation is an activity that the service can do on your behalf. A service is a collection of endpoints that all have something in common.

In our case, the endpoint is the predict method for posting the image and expecting the results. The endpoint maps to the docker's TensorFlow serving image on the 8502 port as discussed earlier.

```
endpoint = "http://localhost:8502/v1/models/potatoes_model:predict"
```

Code snippet 5.4 (endpoint)

### 5.3.1 Cross-Origin Resource Sharing

Cross-Origin Resource Sharing (CORS) is an HTTP-header-based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources. CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

An example of a cross-origin request: the front-end JavaScript code served from https://domain-a.com uses XMLHttpRequest to make a request for https://domain-b.com/data.json.

For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts. For example, XMLHttpRequest and the Fetch API follow the same-origin policy. This means that a web application using those APIs can only request resources from the same origin the application was loaded from unless the response from other origins includes the right CORS headers.

### 5.3.2 Resolving CORS error

The resolution of CORS can be done by importing the CORSMiddleware library provided by FastAPI.

```
from fastapi.middleware.cors import CORSMiddleware
```

Code snippet 5.5 (CORS library)

By mentioning all the possible origins in form of a list as mentioned in the code snippet 15, we can set the CORS policy according to our requirements. Here we have allowed all the origins (origins from different ports) and gave a green flag to all the methods and the headers.

```
origins = [
    "http://localhost",
    "http://localhost:3001",
    "http://localhost:3000",
    "http://localhost:8000"
]
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Code snippet 5.6 (code to resolve CORS error)

## 5.4 Postman

Postman is an API testing program. It is an HTTP client that tests HTTP requests by using a graphical user interface to acquire various sorts of answers that must then be verified. Its better user interface and request tracking capabilities come as a great aid when we are prototyping the project. During the initial stages of development, we would need a tool to check our responses since our website is not ready yet.

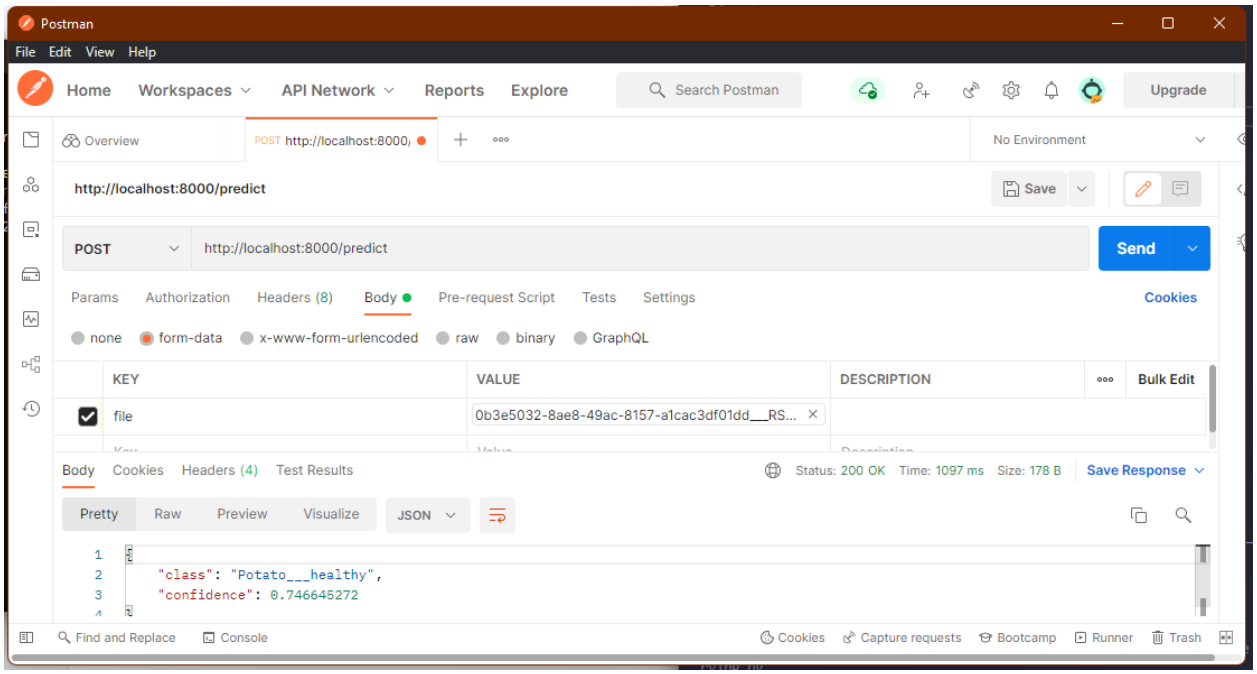


Figure 5.3 (Postman API)

The postman's user-friendliness can be used to obtain fast results and prototype our model. The POST method should be specified and the endpoint link should be pasted in the URL box before adding value into the body and triggering the send command. Here the "KEY" is a file since we have initialized the "UploadFile" function as a file. The response can be observed below which is a set of class and confidence.

# Chapter 6: Frontend

The part of a website that the user interacts with directly is termed the front end. It is also referred to as the ‘client side’ of the application. It includes everything that users experience directly: text colors and styles, images, graphs and tables, buttons, colors, and a navigation menu. HTML, CSS, and JavaScript are the languages used for Front End development. The structure, design, behavior, and content of everything seen on browser screens when websites, web applications, or mobile apps are opened up, is implemented by front End developers. Responsiveness and performance are two main objectives of the Front End. The developer must ensure that the site is responsive i.e. it appears correctly on devices of all sizes no part of the website should behave abnormally irrespective of the size of the screen.

HTML: HTML stands for Hypertext Markup Language. It is used to design the front-end portion of web pages using a markup language. HTML is the combination of Hypertext and Markup language. Hypertext defines the link between the web pages. The markup language is used to define the text documentation within the tag which defines the structure of web pages.

CSS: Cascading Style Sheets fondly referred to as CSS is a simply designed language intended to simplify the process of making web pages presentable. CSS allows you to apply styles to web pages. More importantly, CSS enables you to do this independent of the HTML that makes up each web page.

JavaScript: JavaScript is a famous scripting language used to create magic on the sites to make the site interactive for the user. It is used to enhance the functionality of a website to run cool games and web-based software.

There are various frontend frameworks available to ease the development stages and reduce the redundant effort to make the same webpage over and over again. For HTML, emmet abbreviations are used (Emmet is not a framework, but it is used to ease the development process). For CSS, frameworks like Tailwind and Bootstrap are used, and for javascript, “nodeJS”, “reactJS”, and “angularJS” are used.

## 6.1 ReactJS

React (also known as React.js or ReactJS) is a free and open-source front-end JavaScript library for building user interfaces based on UI components. It is maintained by Meta (formerly Facebook) and a community of individual developers and companies. React can be used as a base in the development of single-page, mobile, or server-rendered applications with frameworks like Next.js. However, React is only concerned with state management and rendering that state to the DOM, so creating React applications usually requires the use of additional libraries for routing, as well as certain client-side functionality. To initialize a reactJS project we must use an NPX (node package execution) command using the following command.

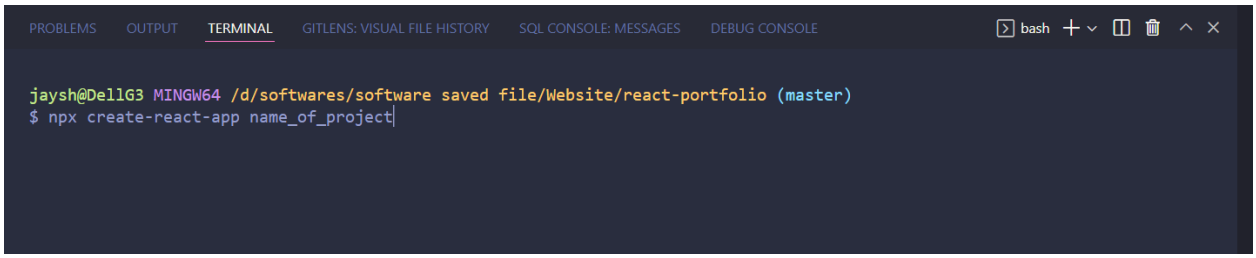


Figure 6.1 (NPX trigger command to create the app)

This command will initialize the react project and create a set of skeleton code which will reduce the redundant efforts to initialize files. The following image shows the directory structure made dynamically by the react automated script.

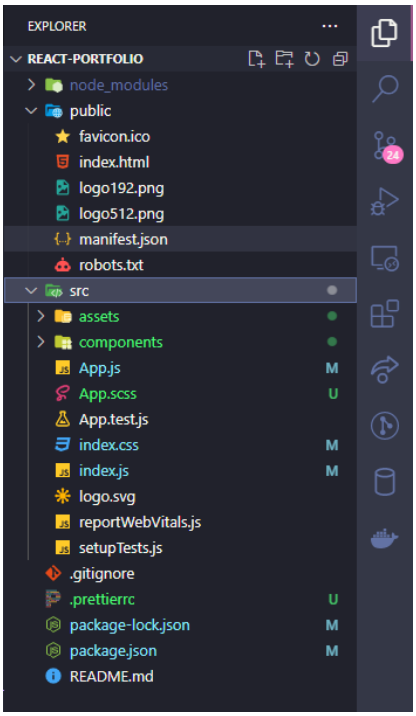


Figure 6.2 (initialized directories and file by React app)

## 6.2 Accessing the model using the website

Using the readymade modules of the ReactJS application we can use them to access the backend hosted site. sendFile is an asynchronous function that accepts the data (image file) and sets the data for output.

```
const sendFile = async () => {
  if (image) {
    let formData = new FormData();
    formData.append("file", selectedFile);
    let res = await axios({
      method: "post",
      url: process.env.REACT_APP_API_URL,
      data: formData,
    });
    if (res.status === 200) {
      setData(res.data);
    }
    setIsloading(false);
  }
}
```

Code snippet 6.1 (sendFile in home.js)

Here, the .env file is mentioned which corresponds to our backend hosted localhost port 8000 where our uvicorn server is running. This further points to the predict method.

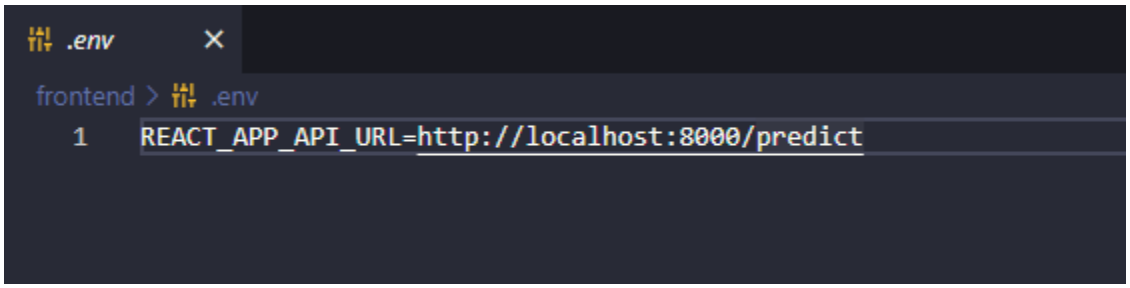


Figure 6.3 (.env file)

The website can be initialized using “npm run start” which will automatically load our website on the default browser. This command must be run on the directory which contains the package.json file.

## 6.3 Obtaining the output

The results can be obtained simply by dragging and dropping the image to the website or by navigating through the file explorer.

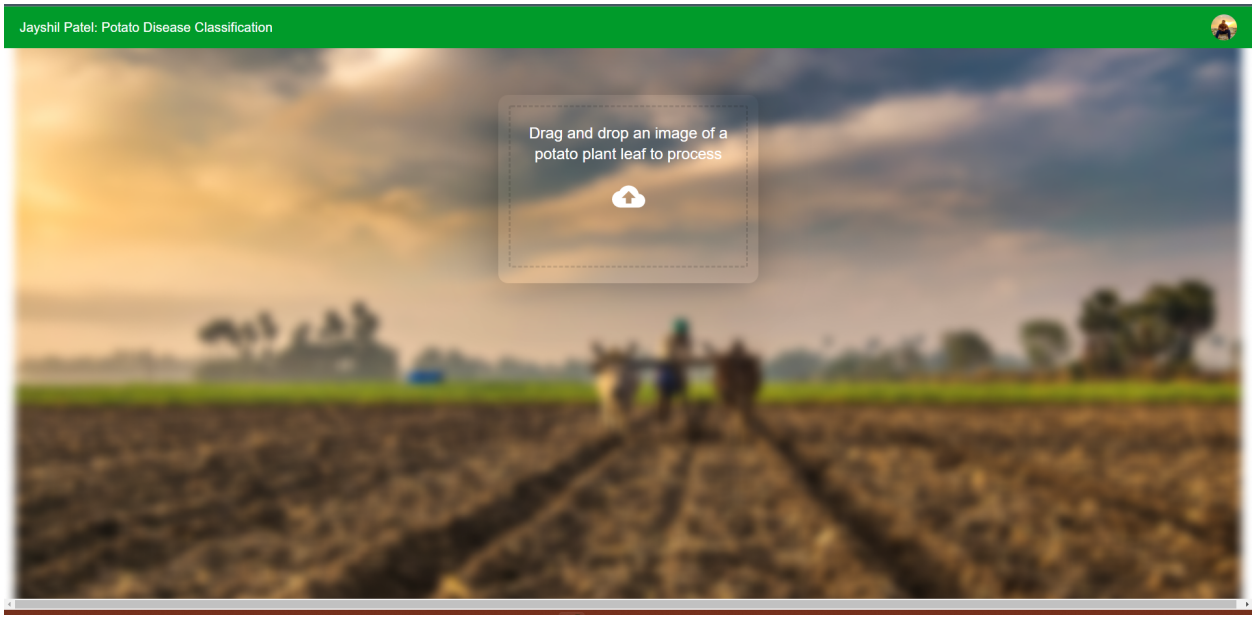


Figure 6.4 (Website on startup)

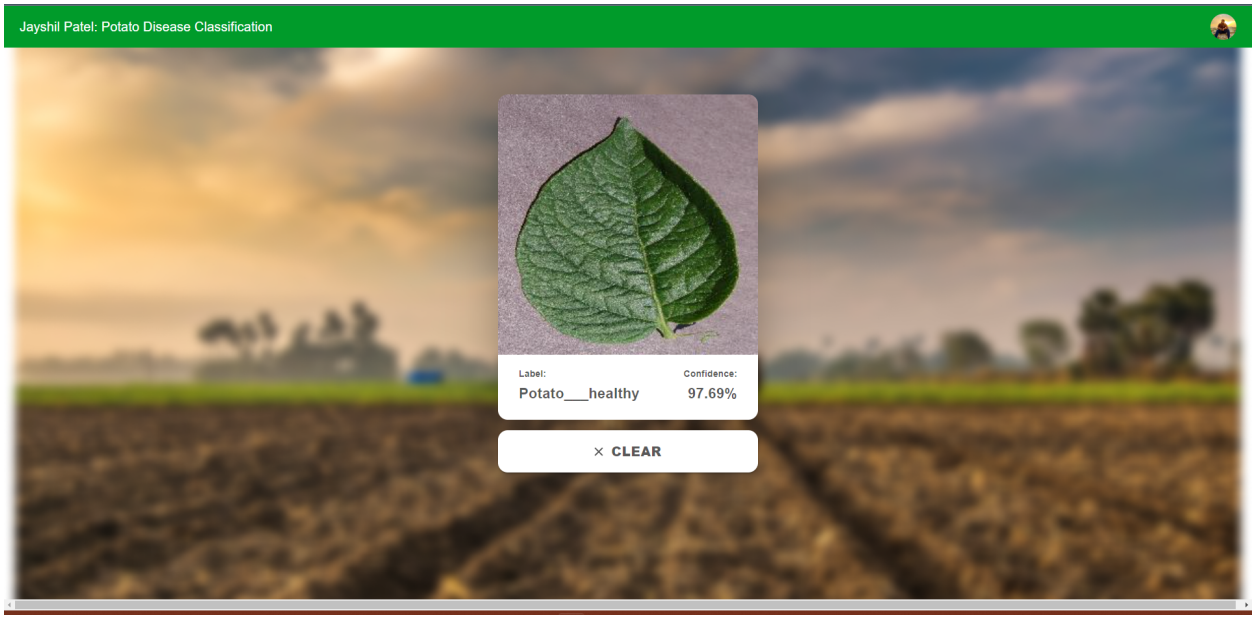


Figure 6.5 (Website output:- post prediction)

# Chapter 7: Version control

While working with a huge set of files, controlling the versions of files such that after a minor error in saved code, the previous version can be restored. Using GitHub also provides the portability of the code. We can access the cloud repository using any device. Furthermore, logs can be accessed to check the previous commits and changes made to a code.


Figure 7.1 (Git log)

I have also made this project open source for future development by other contributors. The complete source code is available on GitHub Repository.

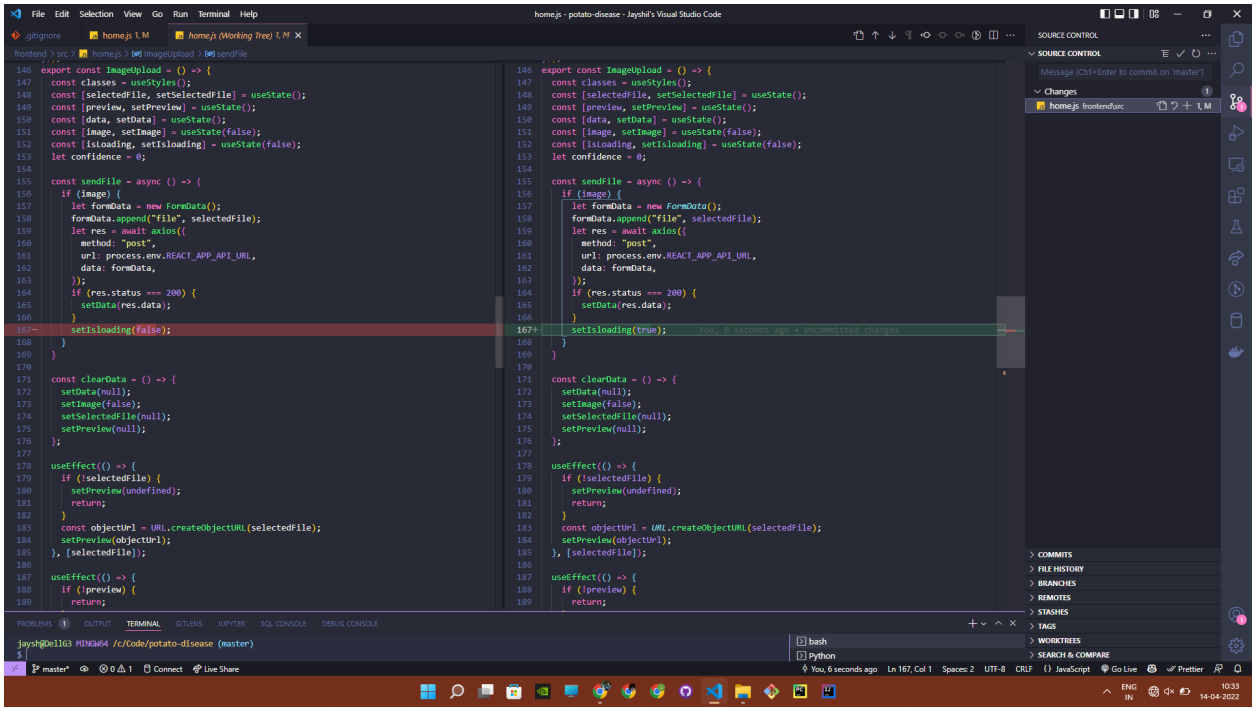

Figure 7.2 (Version control showing changes made to a file)

# Chapter 8: Possible future developments

This project holds a lot of potential for future development. I was constrained by limited knowledge of web development and mobile application development but with a team specialized in every individual field, this can be achieved without a doubt. Furthermore, we can expand the classification categories by upgrading the CNN model so that it targets a bigger group of people (farmers) and more people can benefit from it.

## 8.1 Uploading the project to a cloud server

The current project is limited to localhost only. All the devices connected to the same wifi can access the website but outside the wifi limit, users would not be able to access the website. Using a cloud server like Amazon Web Services, Azure, or Google cloud platform, we can cloud the project such that it would be accessible to everyone. Buying a server (or renting) is a costly process and requires in-depth knowledge of web development which is beyond the scope of this project.

## 8.2 Mobile application

Once the project is deployed on a cloud platform, queries from the mobile application can be used to get the output. I have made a mobile application using react native but with no cloud platform to respond to the query, it fails to generate output.

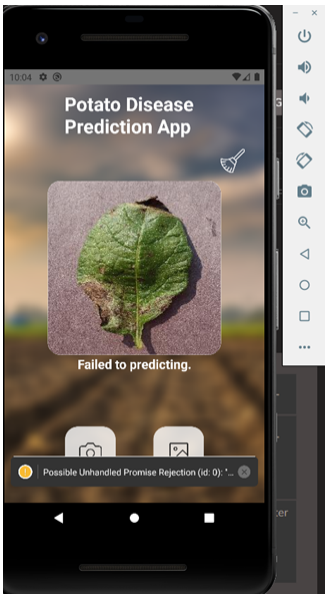


Figure 8.1 (mobile application)

# Chapter 7: Conclusion

With the advent of technology, an array of opportunities disclose themselves to improve the standard of classification. Machine learning and developments in artificial intelligence have played an important role in improving the lifestyle of people. This project focuses on the issues encountered by the farmers and helps them by letting them know about the diseases in their initial stages so that the economic loss can be reduced. The user interface of a simple machine learning model is inadequate and does not match up to the mark of what the machine learning model is capable of doing. The user must be very competent with machine learning to extract the information and must have a technical background for doing so. A graphical user interface would ease the work and would provide a better user experience for the users. Machine learning models are very complicated to look at, and their non-user friendliness is highly renowned. This improvement in user experience could help to gain consumers' confidence and provide the results/information that they are looking for. This project's CNN model is targeted to a very specific group of individuals, mainly farmers, and it is very highly unlikely that an average farmer could access the machine learning model straight out of a jupyter notebook. It would not only improve the user experience as expected but it would also make the job of farmers easier by giving them proper results with ease.

# References

[1]Dataset:-
https://www.kaggle.com/muhammadardiputra/potato-leaf-disease-dataset

[2]CNN:-
https://www.youtube.com/watch?v=Mubj_fqiAv8&list=PLeo1K3hjS3uu7CxAacxVndI4bE_o3BDtO

[3]HTML & CSS:-
https://www.freecodecamp.org/learn/responsive-web-design/

[4]JavaScript:-
https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/

[5]ReactJS:-
https://www.freecodecamp.org/learn/front-end-development-libraries/#react-and-redux

[6]Docker:-
https://docs.docker.com/get-started

[7]Query Solving:-
https://stackoverflow.com/users/15609296/jayshil-patel

[8]Postman:-
https://www.javatpoint.com/postman

[9]FastAPI:-
https://fastapi.tiangolo.com/tutorial/first-steps/

[10]CORS policy:-
https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[11]TensorFlow serving:-
https://www.tensorflow.org/tfx/guide/serving

[12]Project link:-
https://github.com/Jayshil-Patel/potato-disease

# Works Cited

[1] "Convolution Neural Networks." https://www.youtube.com/watch?v=Mubj_fqiAv8&list=PLeo1K3hjS3uu7CxAacxVndI4bE_o3BDtO.

[2] "Cross-Origin Resource Sharing (CORS) - HTTP | MDN." *MDN Web Docs*, 10 April 2022, https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. Accessed 14 April 2022.

[3] "Cross-Origin Resource Sharing (CORS) - HTTP | MDN." *MDN Web Docs*, 10 April 2022, https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. Accessed 14 April 2022.

[4] "First Steps - FastAPI." *FastAPI*, https://fastapi.tiangolo.com/tutorial/first-steps/. Accessed 14 April 2022.

[5] "Front End Development Libraries Certification." *freeCodeCamp*, https://www.freecodecamp.org/learn/front-end-development-libraries/#react-and-redux. Accessed 14 April 2022.

[6] "JavaScript Algorithms and Data Structures Certification." *freeCodeCamp*, https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/. Accessed 14 April 2022.

[7] "Orientation and setup." *Docker Documentation*, https://docs.docker.com/get-started. Accessed 14 April 2022.

[8] Patel, Jayshil Jatin. “Potato-disease.” *Potato-disease-classification*, Jayshil Jatin Patel, 14 April 2022, https://github.com/Jayshil-Patel/potato-disease.

[9] Patel, Jayshil Jatin. “StackOverflow Query solving.” *StackOverflow*, Jayshil Patel, https://stackoverflow.com/users/15609296/jayshil-patel.

[10] “Plant Village dataset.” https://www.kaggle.com/muhammadardiputra/potato-leaf-disease-dataset.

[11] “Postman Tutorial.” *Javatpoint*, https://www.javatpoint.com/postman. Accessed 14 April 2022.

[12] “Responsive Web Design Certification.” *freeCodeCamp*, https://www.freecodecamp.org/learn/responsive-web-design/. Accessed 14 April 2022.

[13] “Serving Models | TFX.” *TensorFlow*, 28 January 2021, https://www.tensorflow.org/tfx/guide/serving. Accessed 14 April 2022.