University of
South Wales
Prifysgol
De Cymru

Final Report

Full-stack Web Application for Vehicle Plate Extraction

Adam Hughes

2020

Statement of Originality

This is to certify that, except where specific reference is made, the work described within this project is the result of the investigation carried out by myself, and that neither this project, nor any part of it, has been submitted in candidature for any other award other than this being presently studied. Any material taken from published texts or computerized sources have been fully referenced, and I fully realize the consequences of plagiarizing any of these sources.

Student Name (Printed)	Adam Hughes
Student Signature	A.Hughes
Registered Course of Study	Computer Science
Date of Signing 20.03.2020	

Abstract

This project covers the development and construction of a web-based full-stack application centred around the recovery of car details for public viewing. This is a small insight into the potential of computer vision. This kind of technology could be used to automate a lot of our visually based processes and could aid in improving the quality of life within our day-to-day activities. This application is simply an example of that technology and a glimpse into its potential.

In this project, the discussion will be based on how to create and implement this specific kind of technology into a web application. The application itself will be run using a virtual private server (VPS) that will include the ability to have multiple instances to allow for multiple people to use it all at once.

The application goes through the process of retrieving an image of a vehicle taken by a user and upon retrieval, an extraction process takes place to acquire the letters and digits of a number plate. These details are then sent to the DVLA website automatically. Once processed, the results for that specific vehicle are displayed to the user.

Table of Contents

1 Introduction **5**
1.1 Problem Statement & Objectives 6

2 Background Research **6**
2.1 Research on Technologies 7

3 Implementation **9**
3.2 Website Introduction 11
3.3 Retrieving Images Using Javascript & Python 12
3.4 Connecting Frontend & Backend 13

3.5 Unpredicted Image Issue 18
3.6 Extracting Image Characters 19
3.7 Getting Data From The DVLA 22
3.8 Displaying The Results 25
3.9 Starting up a Server 25
3.10 Implementing the Server 26
3.11 Setting up uWSGI 26
3.12 Selenium Roadblock 28
3.13 Implementing Websockets 30
3.14 Load Balancing Socketio 32
3.15 Multiple Instances & AJAX 34

4 Results 35
4.1 - Application Design 35
4.2 - Application Performance & Security 36

5 Evaluation 37

6 Conclusions and Future Work 38
6.1 - Conclusion 38
6.2 - Future Work 38

7 References 40

1 Introduction

This application is a view into vehicle number plate detection. As this is an automated approach that requires no behind the scenes human activity, it has a lot of potential. This can be used and tested either by someone that knows nothing about software or someone with the knowledge. A car we see on the road may pique our interest to the point where we want to know more on the current status of that specific car and that’s where this application comes into play.

With the idea in mind that there could be a way for people to simply take a picture of a vehicle's number plate and know the current status of MOT and tax, there could be more peace of mind when it comes to encounters on the road, good or bad. This sort of application would be aimed directly for public use which includes those who can drive as well as pedestrians.

Though this idea seems to be rather difficult at first glance, the main obstacles initially perceived at the beginning of this project are listed below. These were followed as an almost step-by-step goal process to start and complete in order:

1. Gather a picture of a vehicle's number plate and allow for this data to be received through the web app

2. Once the picture is received, the numbers and letters on the vehicle's number plate will need to be extracted into the program

3. After retrieving the appropriate characters, automation of retrieving the MOT and Tax status will need to be extracted from the Government website.

4. The final results will be displayed for the user.

5. Once previous steps are complete, set up a server to host this application.

1.1 Problem Statement & Objectives

There are far more vehicles on the road than there ever have been before. With that, more incidents can occur on the road. The ability for the public to take a quick picture of a vehicle and retrieve its MOT and Tax information would be useful. As a beneficial side point, people interested in a particular car type would be able to take that picture and discover the make/type of car it is too.

The ultimate aim of this project was to code and create a web application/progressive web app that will allow the user to know a vehicle's MOT and Tax status by way of taking a picture of the vehicle's number plate. The goal was for this app to be able to be used on practically any computer/device or downloaded as an app on a mobile phone. Although being able to use this app on a computer is beneficial enough, the ability to use this on the go on your mobile device is even more convenient. Being able to grant access to the web app through both a computer and mobile allows for more accessibility for different devices.

2 Background Research

Through research, it has been previously established that this kind of technology is already available to the police. However, there has been nothing available for the public to use. There is the ability to search a number plate online, but there is nothing out there which is photo-based and able to be used through a mobile device.

As police have been mentioned, it’s also been discovered that this project seems quite similar to the Automatic Number Plate Recognition (ANPR) cameras which are widely placed across the entirety of the United Kingdom. With more than 10,000 of them in place, it has become a massive resource for the police in being able to detect numerous types of crimes.

Although this application is not entirely on the same level as sophisticated ANPR cameras, there was the challenge of implementing something that was a refined piece of software. People will benefit from the finished product and will find it handy to use out in public spaces.

For example, a car parked on the street that has not moved for an extended period of time. Should it be there? Is the MOT and/or tax up to date? Another example is in minor collisions. Where a vehicle in the wrong bumps into you. A quick move of the phone to take a picture after a bump and you have the offending car’s MOT and tax status. The app will prove beneficial for the general public in many different situations.

When it came to the initial research process, there was a lot to look at when it came to what technologies were to be used and how to deliver it. After gathering information, these are the three ways that appeared most beneficial to provide this project:

1. Mobile Application - With this option, there was some wariness on what devices to target, either Apple iOS or Android. It was possible to do cross-platform, but this comes with a new set of problems. The decision was made to attempt to create an Android application with the use of Java, though Kotlin was another option. This didn’t entirely feel right for this kind of application.

2. Desktop / GUI - This was the most simplistic of all the options and seemed to be the easiest to come up with a decent solution for the project, but with the downfall of a Desktop application being quite restrictive and only really usable on a local machine. This was not ideal.

3. Web Application - This option appears to be the most complex, but offers a wide range of possibilities. Due to this, the decision was made to continue forth with this as the delivery point for the application. This option allowed for use on both Android and iOS devices as well as any other computer device with access to the internet.

2.1 Research on Technologies

With the final decision made to create a web application, research on the technologies required to piece together the project began. Below is a list of three components that were essential in making this application come to life. In Michael Wales' blog (2020), he states, "The front end of a website is the part that users interact with."

With the above in mind, the three components below were the main parts that aided in making this project interactable. All three were vital for the user to perform smooth interactions with the application. The interactions must be handled smoothly otherwise potential users would not like to use the website. It is always best to keep the user in mind when constructing such an application.

1. HTML (Hyper-Text Markup Language) - Every web application must begin with an HTML file. This was where the structure of the website comes from. Every website out there has an HTML file that will display what the website looks like to the user.

2. CSS (Cascading Style Sheets) - CSS is not exactly necessary to have a website, but if it is not used, the website will result in a rather glum appearance. This is because all of the style of your website will be down to this language. With this, you can change colours, make animations, loading screens and so on. This part has the potential to turn an unaesthetic website into a very aesthetic one.

3. JavaScript - This part is what makes a website come to life. With this, we can create events when users click buttons, gather data, collect data and display things to the user.

In Josh Long's frontend vs. backend based article (2012), it is said that, "The backend usually consists of three parts: a server, an application, and a database". It was discovered from this how possible construction of the application would take place. A backend was needed for this application to process the data that users input in from the frontend.

Choosing a backend language, Long states that, "Backend languages usually consist of languages like PHP, Ruby, Python". It was decided that Python would be best for the backend. Using Python as a backend language required a web framework to make things a lot smoother to set up. By reading Gareth Dwyer's article on frameworks (2017), it was discovered that Flask and Django are Python's two most popular frameworks. He goes on to say that Flask is a more watered down and simplistic approach than Django.

With the above and some analysis of some code examples, the decision was made that Flask would be the more natural framework to work with. It was also found that Django's additional functionality would not be necessary or could complicate things a lot more than it needed to be.

As stated above, a part of the backend should include a server as well as a database. There were already options out there like Heroku and PythonAnywhere which seemed very promising as they don't require any knowledge of setting up a server from scratch. However, Hansel (2013) states that when uploading user media, it is not possible to do so when using Heroku and is only possible with the addition of a separate data storage provider. This was somewhat tricky as the application needed to be able to upload and save images from the users.

Because of the above, Heroku didn't seem like a good option and PythonAnywhere was beginning to be more likely because of the ability to upload and save images. This was an essential part of the entire project so this piece of functionality would be a must-have.

3 Implementation

The main aim of this project was to extract a vehicle's number plate from an image and then gather the information on the vehicle from the DVLA Government Website.

1. Take and retrieve pictures.

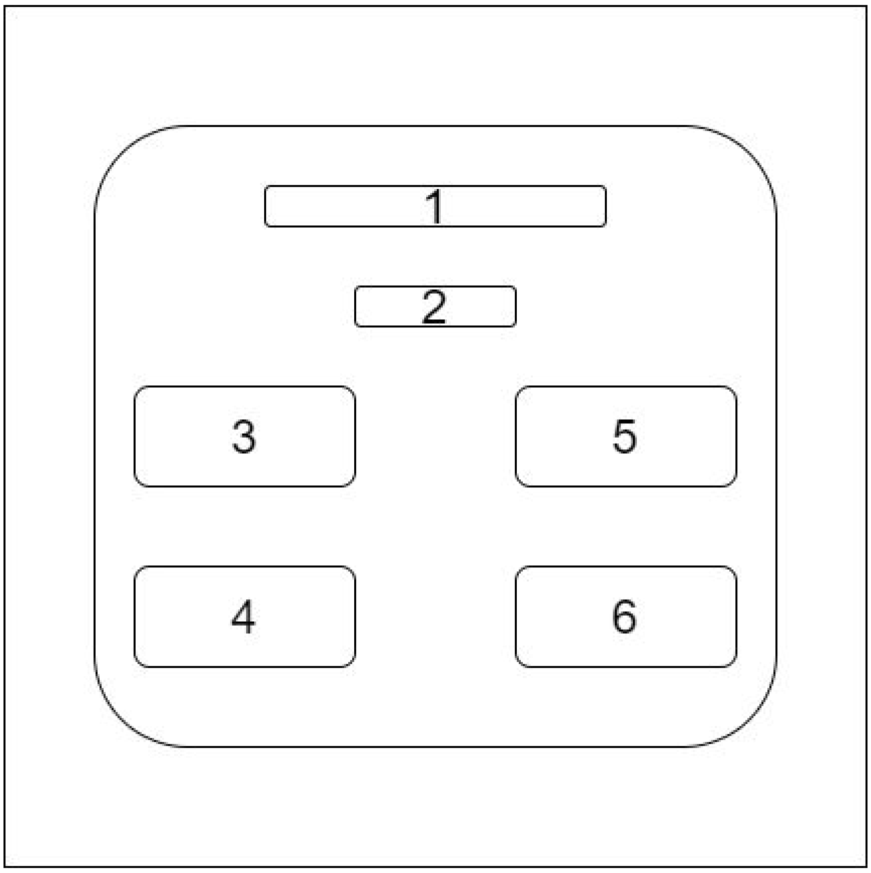
2. Extract the number plate from the image.

3. Get the vehicles data from the DVLA website.

4. Display the data to the user.

As previously mentioned, the application was written using Python as the backend language and Flask was the web framework used. From the very start of this project, the aesthetic of the website was an essential part. The functionality is a lot more critical, but the look and feel of the website must be up to standards as well.

Implementing the design for a website is always a tricky process no matter the website as there are so many ways to construct it. For this website, it wasn't meant to be overcomplicated. Eventually, a simplistic but effective design was found.



This was the design idea that took hold. All the components are as follows:

1. This part is a button that users can click to either take a picture on their smartphone or upload an existing image from their computer.

2. Once the image has been uploaded, this is where the number plate of the vehicle will display.

3. This is where the vehicle's make will be displayed.

4. Here the tax status of the vehicle will be displayed.

5. This is where the year of manufacture for the vehicle will be displayed.

6. This is where the MOT status of the vehicle will be displayed.

With the design of the website complete and a clear vision taking hold, it was now time for the implementation of it. To begin, here’s a rebrief of the main components that had to be met for this to work as intended:

1. Design and construct website.

2. Start implementing a Backend.

3. Retrieve image from Frontend to Backend.

4. Extract the characters from the image.

5. Perform DVLA check on vehicle.

6. Display results to the user.

3.2 Website Introduction

With goals clearly set and in order, it was time to begin the first task. Firstly, the button needed to be configured to allow access to a smartphone's camera, as well as allowing the same button access on a computer. This seemed particularly awkward as one button was wanted for two separate devices.

Firstly, a basic HTML file setup with the necessary format was required to start the first task. It was already known that a HTML input tag would be the possible way to solve this issue, but this didn't actually seem to work as intended and only worked on a computer.

Brandon Beeks article 'HTML5 Mobile Device Camera Access' (2019) was able to give guidance on the next implementation steps of the code at hand.

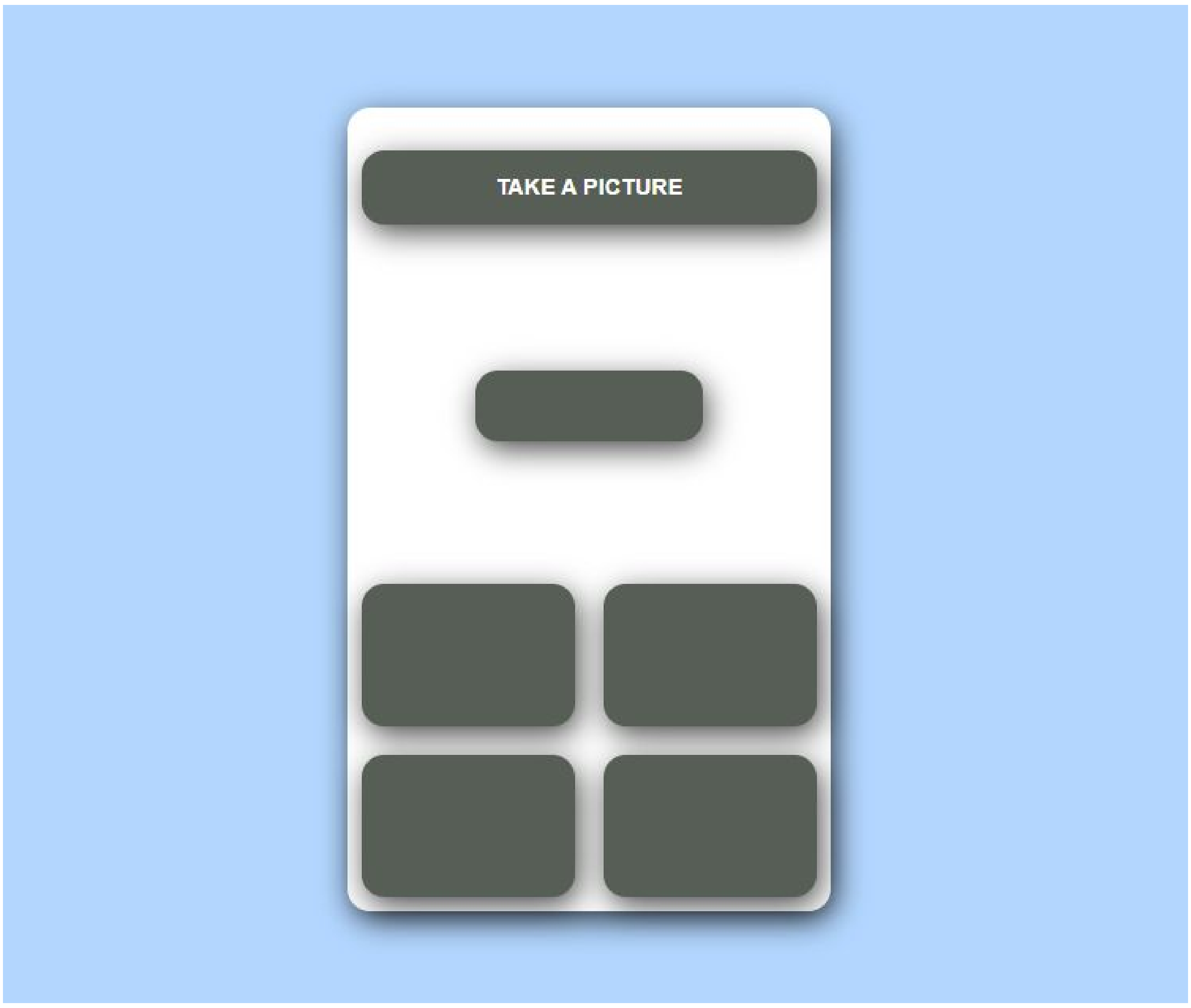
```
"<input type="file" accept="image/*" capture="camera" />"
```

This snippet was extremely helpful towards getting a full solution towards this task. It was known that an input tag would most likely be the solution, but this type of input tag was not expected.

The one problem that Beeks mentioned is, "It only works with iOS6+ or Android 3.0+", so this would have potentially been an issue with some older devices. From this point, it did not seem like it would be much of a problem.

Getting this part working was an excellent first step forward. The button was complete. As the design is quite minimalistic, the initial thought was that this would be reasonably simple to construct. Still it was actually quite challenging to make everything come together fluidly in web design. When it comes to the design, the user must be kept in mind. This application was always intended for public use. Anyone with simple smartphone literacy should be able to use this application with ease.

After some time, the bulk of the HTML was functional enough so that we could move on towards the next step of implementing some of the events on the website. The HTML has been structured into two div tags. These div tags are what separates or divides elements of the website apart. We then go down into a form tag which allows for capturing input from input tags, so this would be a way to capture the image a user uploads.



3.3 Retrieving Images Using Javascript & Python

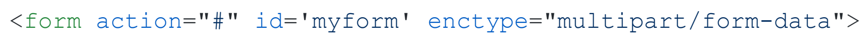
The main functionality of retrieving an image from the user was the next step into trying to get the project going forward. The issue with this was that once the user clicks the button and chooses the image, the image would need to be retrieved and then passed from the Frontend to the backend. As previously mentioned, Python was used for this task and the Python web framework Flask.

This was probably one of the trickier parts of the project as it was required to capture the image from the Frontend (Client) to the Backend (Server). The breakdown for this was to start by creating a Form HTML tag.

```
<form action="/action_page.php" method="get">
  First name: <input type="text" name="fname"><br>
  Last name: <input type="text" name="lname"><br>
  <input type="submit" value="Submit">
</form>
```
(HTML Form Attribute, 2020)

The above is an example piece. At first glance, this seemed to be for some sort of user input. Forms are an interactive part of the website. This is where a user might fill in a survey, but in our case, it was a little different as we accept a type of file which is an image.

Furthermore, a different take on the form tag above was used. This was because it must be explicit with what type of input it would be able to capture.



```
<form action="#" id='myform' enctype="multipart/form-data">
```

In the above image, this form tag is not much different at all, but the ‘enctype’ part is different from the previously mentioned form tag. This is because what type of input we want to retrieve must be precise. The multipart/form-data is used when looking to retrieve an <input> tag which is of “type=file”.

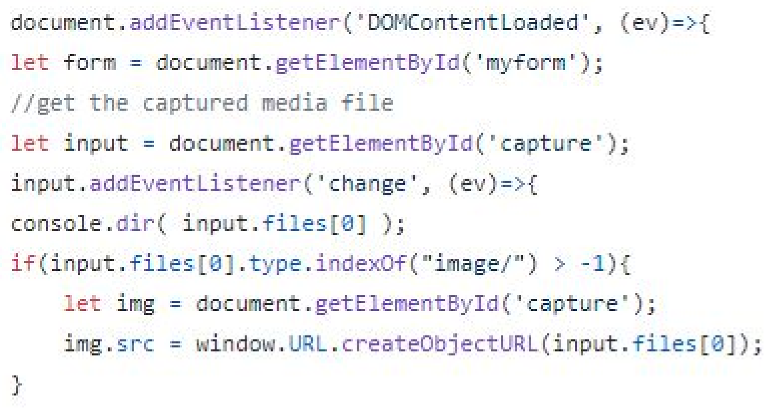
3.4 Connecting Frontend & Backend

This allowed for the ability to retrieve an image from the Frontend and be able to save the image to the server. This will then give the ability to perform processes on the Backend, such as extracting the characters from the image, gathering the data from the DVLA and displaying the results to the user.

Time was taken to research on how to pass an image from the Frontend using JavaScript and then retrieving it using Python. It was found out that the most common way to do this was to use a JavasScript library called JQuery which makes the use of JavasScript much simpler to write.

Alongside this, a JavaScript methodology was used called AJAX (Asynchronous JavaScript and XML). This is the part that will allow the ability to pass the image data from the Frontend to the Backend.

The implementation took some time to get right as there was an issue with connecting both languages and the actual retrieval of the image on the Backend. After some time, a solution was near complete.



```
document.addEventListener('DOMContentLoaded', (ev)=>{
let form = document.getElementById('myform');
//get the captured media file
let input = document.getElementById('capture');
input.addEventListener('change', (ev)=>{
console.dir( input.files[0] );
if(input.files[0].type.indexOf("image/") > -1){
    let img = document.getElementById('capture');
    img.src = window.URL.createObjectURL(input.files[0]);
}
```

This appears as though there is a lot of information to take in. A debrief of the above is as follows:

1. With the first line, there is an event listener that is for the DOMContentLoaded. This means an event will run when the HTML document has been completely loaded.

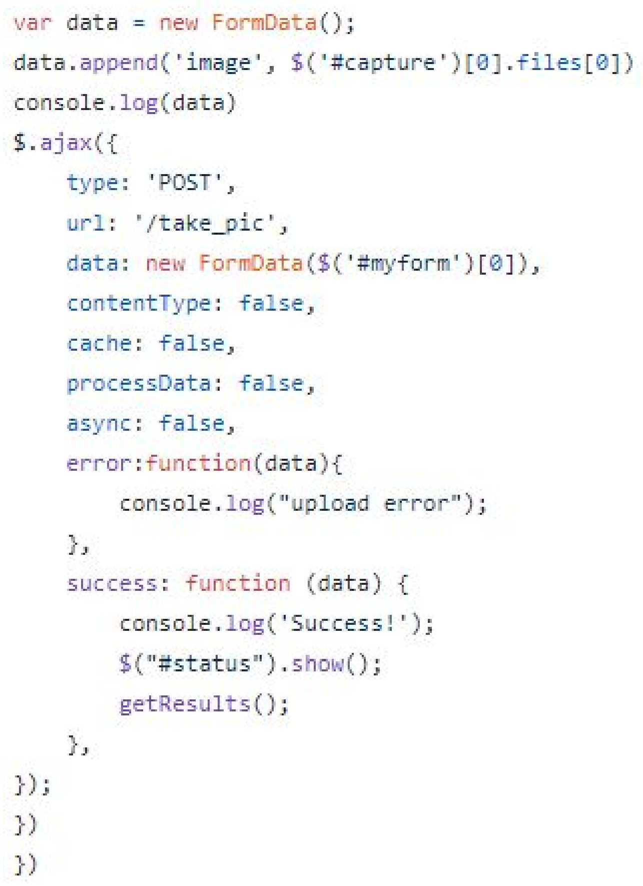
2. The next two lines of the variables form and input returns the elements ID. This allows for quicker changes to elements.

3. It then moves into an event if the element input catches any changes towards it. A change of its state would mean now that a user has selected an image.

4. Next is the file part of capturing the actual image. This is done by enclosing the image inside an ObjectURL and capturing the image. At this stage, it is now close to being ready to be able to pass to the Backend.

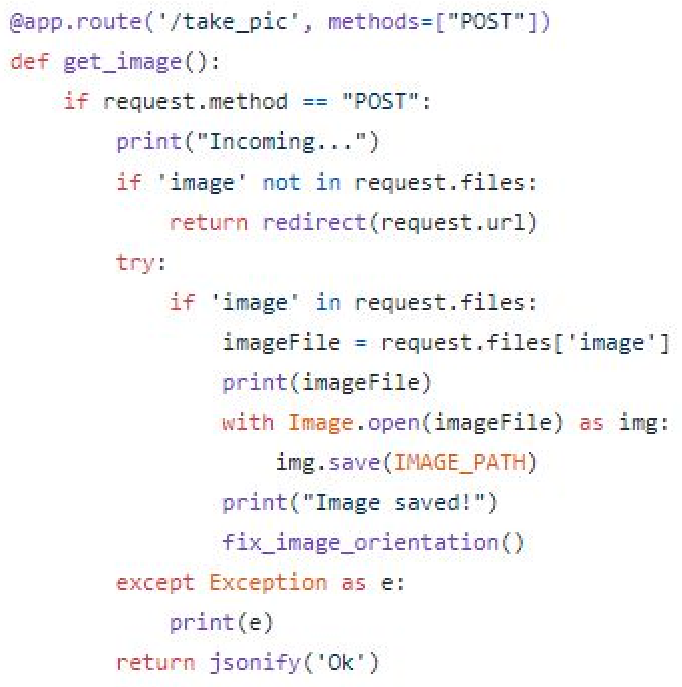
As previously mentioned above, adding in the ability to capture the image is now complete at this point. With this, the process of passing the image from the Frontend to the Backend using JQuery and AJAX begins.

Once our event has been fired, it will capture the image and then proceed to start passing the image over. This is done by communicating to the Backend using an XMLHttpRequest object. This opens a request with our Backend (Server), then our server can proceed to catch this request. If this is then successful, the data will be transferred and then the ability to save the image that was just received is achieved.


```
var data = new FormData();
data.append('image', $('#capture')[0].files[0])
console.log(data)
$.ajax({
    type: 'POST',
    url: '/take_pic',
    data: new FormData($('#myform')[0]),
    contentType: false,
    cache: false,
    processData: false,
    async: false,
    error:function(data){
        console.log("upload error");
    },
    success: function (data) {
        console.log('Success!');
        $("#status").show();
        getResults();
    },
});
})
})
```

The above is the AJAX query. This is still part of the previous function, but this is the part where we will pass the image over to the Backend. Explaining this part is quite difficult as there are two parts. The JavasScript and AJAX side, which is referred to above and the second part is the Backend Python code.

To receive the image on the Backend, the application must be able to process the image by capturing the AJAX request. The following Python code will allow for this to be achieved.



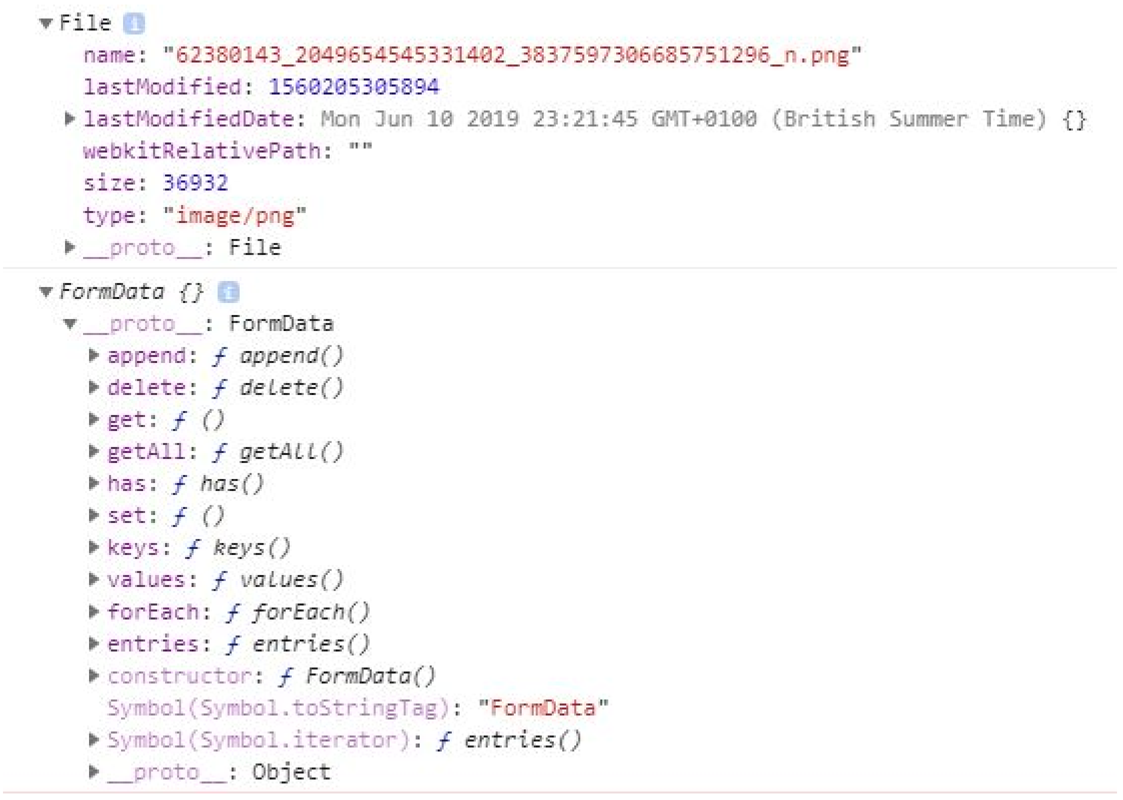
```
@app.route('/take_pic', methods=["POST"])
def get_image():
    if request.method == "POST":
        print("Incoming...")
        if 'image' not in request.files:
            return redirect(request.url)
        try:
            if 'image' in request.files:
                imageFile = request.files['image']
                print(imageFile)
                with Image.open(imageFile) as img:
                    img.save(IMAGE_PATH)
                print("Image saved!")
                fix_image_orientation()
        except Exception as e:
            print(e)
        return jsonify('Ok')
```

In the above code, both functions have the same '/take_pic'. This is used to determine when this specific function will run. In the AJAX query above under URL, the route '/take_pic' is being passed which will look for this route. If the request is successful, the AJAX function will output in the terminal that the request has gone through with no errors. At this point, it'll be the Backend processing left to complete.

Here in the function 'get_image', 'methods=[POST]' is used. This is also used in the above AJAX function. Out of the methods, there are two different ways of going about this by using either POST or GET. The only thing known about these HTTP methods was that there were two options.

W3School's 'HTTP Request Methods' (2020) mention that the POST HTTP request can pass more data with no restrictions and that this method is a little more secure than the other GET method. Due to the data length, POST was the only option that was possible. The added security is also a nice benefit.

After this, a request takes place to look for the 'image'. If this request is successful, then the process continues down through the code coming into the try and except clause. This will then take the received image file and save it to the server. At this point, the image has successfully been retrieved from the Frontend to the Backend and is now ready to go through the other processes of the application.



```
▼File
   name: "62380143_2049654545331402_3837597306685751296_n.png"
   lastModified: 1560205305894
 ▶ lastModifiedDate: Mon Jun 10 2019 23:21:45 GMT+0100 (British Summer Time) {}
   webkitRelativePath: ""
   size: 36932
   type: "image/png"
 ▶ __proto__: File
▼FormData {}
 ▼__proto__: FormData
   ▶ append: ƒ append()
   ▶ delete: ƒ delete()
   ▶ get: ƒ ()
   ▶ getAll: ƒ getAll()
   ▶ has: ƒ has()
   ▶ set: ƒ ()
   ▶ keys: ƒ keys()
   ▶ values: ƒ values()
   ▶ forEach: ƒ forEach()
   ▶ entries: ƒ entries()
   ▶ constructor: ƒ FormData()
     Symbol(Symbol.toStringTag): "FormData"
   ▶ Symbol(Symbol.iterator): ƒ entries()
   ▶ __proto__: Object
```

On the website, by pressing Crtl + Shift + I, inspection mode is applied and then access onto the Console is granted. Here, outputs can be seen that were written to the console. This is an example of a successful image transfer to the server. It can be seen here that this is an image of type png and it was a part of the FormData.

3.5 Unpredicted Image Issue

Below, discussion takes place on how the implementation of how the extraction process will work. One thing that was not at all predicted was that images from smartphones cause a strange bug. With the previous process about sending the image from the Frontend to the Backend, everything ran smoothly as long as the user was on a computer or laptop.

The only issue that comes along is if the user is on a smartphone. Once the image has been transferred, the image gets altered and rotates by 90 degrees. This was a significant issue, as to be able to extract the characters correctly, we must be able to see the characters on the number plate horizontally.

This bug could not be overlooked, especially with this app being aimed more towards mobile users. After some sifting through the web, the culprit was eventually found out. This took some time as searching for a solution to the actual problem was awkward. This was because the only thing to go on was that the image was rotating.

It was such a strange problem to come across. It was not a bug that was at all expected. As mentioned above the issue was found. The reason this bug exists is to do with the HTML input tag discussed previously.

“`<input type="file" accept="image/*" capture="camera" />`”

This does not seem to play well on smartphones for some reason and causes the data to be altered. To go even more specific, it is to do with the EXIF data. EXIF (Exchangeable Image File Format). With some research, an article was found that was able to provide the required information to fix the issue at hand.

Olayinka Omole’s ‘Fixing Rotated Mobile Image Uploads In PHP’ (2020) explains that the camera causes the issue of the mobile phone changing the orientation tag. This then changes the EXIF data so that the orientation of the image is rotated by 90 degrees. This was not at all predicted and was particularly confusing when the issue first came to light.

3.6 Extracting Image Characters

This is where things became more complicated. The initial ideas and planning on how to come up with a way to extract image characters are as follows:

1. The initial idea was to use a Machine / Deep learning route and use Tensorflow, but this route appeared to be much more complicated.

2. The second idea was to go along the route of using graphics programming to try and extract the characters. The idea was to use a library called OpenCV.

3. The final idea was to find some sort of API (Application Programmable Interface) that has already been created to allow for a more efficient and accurate way of extracting the characters. This would allow for more time on evolving other areas of the project.

All of the ideas above were very captivating and interesting. The idea jumping out was the OpenCV idea as this appeared more suitable than the Deep learning route.

The initial route was to go with the graphic library OpenCV and try to do Optical Character Recognition (OCR). After a couple of weeks or close to a month of implementing and getting stuck on the extracting of the characters, using OpenCV did not seem like the most viable option. It was not working as intended.

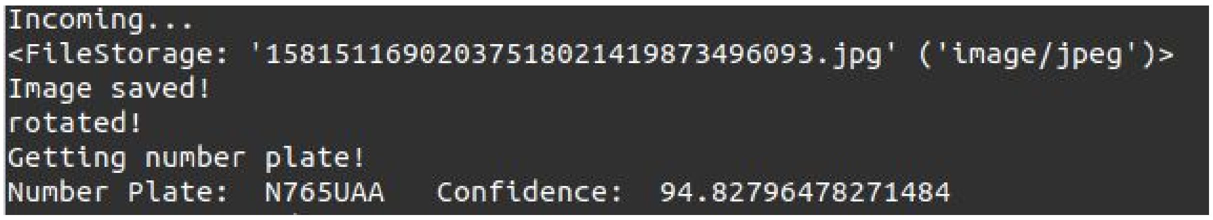
After some time, progress was made and it was able to extract the characters from a number plate, but this came at a price of being right 20% of the time. With this specific project, there was no room for any error. It was required to be correct 100% of the time rather than just the majority of the time. Overall, any errors in use would leave users feeling confused and they would leave using the app altogether which is not ideal at all.

Due to the inefficiency of this route and the time-consuming efforts to get OpenCV working as intended, it was decided that an API would be more appropriate as the central part of the application is the character extraction of the image. This part is essential and needs to work completely.

This was one of the main parts of the entire application and if an API meant that the extraction would have a 95% accuracy, this would be much better as the application would be much more user friendly and would work with as little error as possible. The main priority here is user experience. Without good user experience on this app, there will be little point to it existing as users will avoid the app if it is too dysfunctional.

After some time searching for an API along the lines of character extraction, a company was found called OpenAlpr (2019). This company has an API available for detecting car number plates and extracting the characters.

With some details noted by looking through the documentation on the API and trying different ways of extracting the characters, a solution was found that was successful and the first number plate had been tested achieving a 94% accuracy.



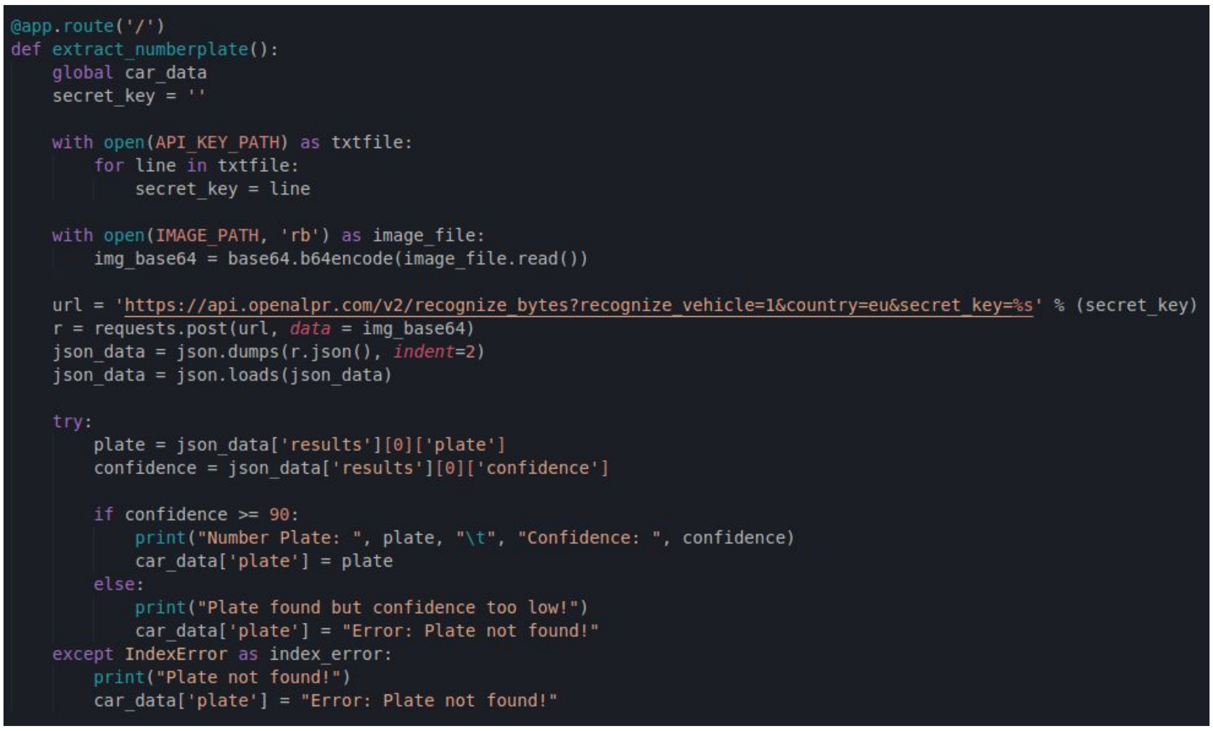
```
Incoming...
<FileStorage: '15815116902037518021419873496093.jpg' ('image/jpeg')>
Image saved!
rotated!
Getting number plate!
Number Plate:  N765UAA   Confidence:  94.82796478271484
```

As previously said above, the current API OpenAlpr that is being used was getting quite a high accuracy for detecting the right characters in a number plate. This was considered useful. With this part being highly efficient, that meant that other parts of the application could be focused on more. Although it would be nice to have a 100% accuracy throughout, that would have been asking for too much of the current tools available.

The implementation of using the actual API was not exactly simple though. To get the above results took some time. A large amount of time went into working out how to connect the API up properly. The API is connected to a server. To get this fully up and running, it was necessary to write some Python code on the Backend that would now also send the image that was saved to the server of this API.

After the above, it would then be passed to their server and go through some checks. One of the checks was that a secret key would be needed. For this, it would require signing up for an account with this service. As a result of doing this, a secret key was received and the ability to use their service free of charge.

After some time looking through the above API documentation, an implementation was successful, as seen from above. The implementation is as follows:



```python
@app.route('/')
def extract_numberplate():
    global car_data
    secret_key = ''

    with open(API_KEY_PATH) as txtfile:
        for line in txtfile:
            secret_key = line

    with open(IMAGE_PATH, 'rb') as image_file:
        img_base64 = base64.b64encode(image_file.read())

    url = 'https://api.openalpr.com/v2/recognize_bytes?recognize_vehicle=1&country=eu&secret_key=%s' % (secret_key)
    r = requests.post(url, data = img_base64)
    json_data = json.dumps(r.json(), indent=2)
    json_data = json.loads(json_data)

    try:
        plate = json_data['results'][0]['plate']
        confidence = json_data['results'][0]['confidence']

        if confidence >= 90:
            print("Number Plate: ", plate, "\t", "Confidence: ", confidence)
            car_data['plate'] = plate
        else:
            print("Plate found but confidence too low!")
            car_data['plate'] = "Error: Plate not found!"
    except IndexError as index_error:
        print("Plate not found!")
        car_data['plate'] = "Error: Plate not found!"
```

1. Firstly, the text file is opened which has the secret key inside and this is read into a variable. This way, the secret key is not a part of the source code and makes it more secure.

2. The image is then opened, saved and encoded into a base64 string and then a request can start to the URL with the encoded image.

3. The data is received from the request as a JSON string and parse it using json.loads() method.

4. Complete access is granted to the JSON data and can begin to find the characters for the number plate. This was done using a try and except clause. The route that was decided was only accepting 90% accuracy and above, otherwise we could let number plates through that are wrong. Anything below that accuracy threshold would not be acceptable.

3.7 Getting Data From The DVLA

With the use of the API that is being used, the ability to accurately capture the characters of a number plate from an image is now available. Since this part was now accomplished, information on the vehicles must now be gathered.

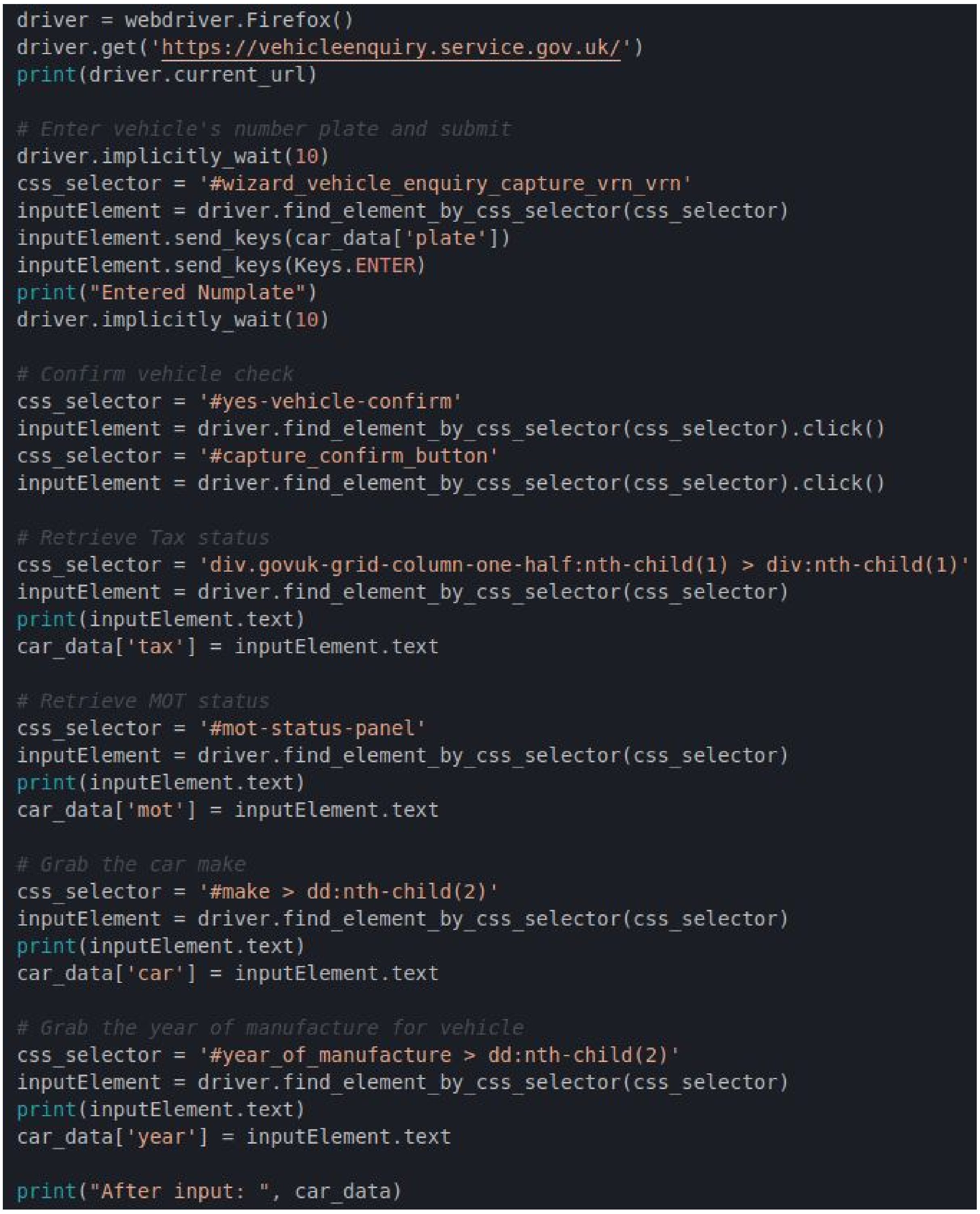
This process first appeared to be somewhat daunting and tricky. Extracting this type of information seemed to be complicated. If this were not accomplished, the web app would not be able to work for its intended purpose. The way this needed to work was to use the DVLA's Vehicle Check website through the use of code. This seemed like a difficult thing to accomplish at first glance.

After initially checking out the website, it was found that there are two checks before you can get through to the actual data on the car. To begin, the number plate needs to be typed into a text box and then either click a button or enter to continue. The next page gives a short description of the vehicle and asks if said vehicle provided from the typed number plate is correct. After this, 'yes' or 'no' must be clicked and then the response must be submitted.

Upon submitting the response, a page will be presented that has all sorts of data of the vehicle at hand. With all the data on the page, it seemed near impossible at first glance to be able to provide and pass all this information through using Python code on the backend. With this being a necessary process for the web app, research was required to be able to tackle this problem.

After researching on how this would be possible, a Python library kept appearing within the search called Selenium. This library is for automating web processes, which was needed. After reading through the documentation thoroughly as to not miss any essential details, a plan was starting to formulate that would work.

The way this worked out is by first specifying the browser you want Selenium to use while making sure the drivers are acquired for said browser and the correct version of the drivers as well. This then allows navigation through web pages by looking for specific parts of HTML, CSS, JavaScript and XPath. This was quite a strange part of the project and something that wasn't first considered at the very start when this project was a simple idea. After a lot of trial and error of issues caused by wrong FireFox drivers and making sure the correct ones were installed, the code that was needed to automate this part was quite different than usual. However, after further reading of the documentation provided, things were becoming far more clearer.



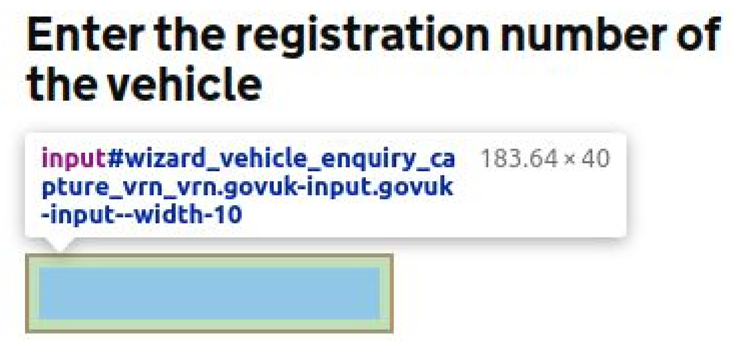
```python
driver = webdriver.Firefox()
driver.get('https://vehicleenquiry.service.gov.uk/')
print(driver.current_url)

# Enter vehicle's number plate and submit
driver.implicitly_wait(10)
css_selector = '#wizard_vehicle_enquiry_capture_vrn_vrn'
inputElement = driver.find_element_by_css_selector(css_selector)
inputElement.send_keys(car_data['plate'])
inputElement.send_keys(Keys.ENTER)
print("Entered Numplate")
driver.implicitly_wait(10)

# Confirm vehicle check
css_selector = '#yes-vehicle-confirm'
inputElement = driver.find_element_by_css_selector(css_selector).click()
css_selector = '#capture_confirm_button'
inputElement = driver.find_element_by_css_selector(css_selector).click()

# Retrieve Tax status
css_selector = 'div.govuk-grid-column-one-half:nth-child(1) > div:nth-child(1)'
inputElement = driver.find_element_by_css_selector(css_selector)
print(inputElement.text)
car_data['tax'] = inputElement.text

# Retrieve MOT status
css_selector = '#mot-status-panel'
inputElement = driver.find_element_by_css_selector(css_selector)
print(inputElement.text)
car_data['mot'] = inputElement.text

# Grab the car make
css_selector = '#make > dd:nth-child(2)'
inputElement = driver.find_element_by_css_selector(css_selector)
print(inputElement.text)
car_data['car'] = inputElement.text

# Grab the year of manufacture for vehicle
css_selector = '#year_of_manufacture > dd:nth-child(2)'
inputElement = driver.find_element_by_css_selector(css_selector)
print(inputElement.text)
car_data['year'] = inputElement.text

print("After input: ", car_data)
```

After thorough analysis, the above piece of code was finally completed. The main issue with this part was that the drivers were awkward, which was due to minor experience in this field. Substantial research was required.

The above piece of code is explained as follows.

1. Firstly, a driver is created that uses FireFox and then it is pointed towards the DVLA website that we want to visit.

2. The selector of the specific input must then be gathered that we want to automate. The way this is achieved is by pressing Ctrl + Shift + I on the DVLA website



Enter the registration number of the vehicle

input#wizard_vehicle_enquiry_capture_vrn_vrn.govuk-input.govuk-input--width-10 183.64 × 40

In the above, the '#wizard_vehicle_enquiry_capture_vrn_vrn' is the actual selector path of the box. With this, this part can be edited by passing in the number plate and this process is then repeated over until we get through to the page where all the car data is held. Once on this page, the output is converted to text and it is passed into the 'car_data' dictionary.

3.8 Displaying The Results

The application is now very close to being nearly completed. When it came to the process of constructing how the results were to be displayed, things seemed to be less stressful as all the data collection construction had been completed. Passing the data back to the user to get displayed would now be straightforward due to having used AJAX already to pass the image from the Frontend to the Backend.

However, things were not going entirely to plan. The solution constructed appeared foolproof, but this solution was not working in a significant way. It was causing the webpage to completely freeze and be unusable until the results finally came through. This was a massive issue as there is nothing else to do while waiting for the results to be displayed, but the behaviour of this issue causes other issues and, on occasions, it would make the CSS fail. The webpage would also flash white for around 10 seconds.

This was not ideal at all. With the user in mind, a delay on results becoming provided would mean that the user may think that it is not working. However, this was later found out to be an issue with AJAX and a possible solution would be the use of WebSockets.

3.9 Starting up a Server

It was decided that a full production server setup would be the most ideal for this project. This would be great, as there would be no restrictions to allowing only one user at a time like on a local machine. The website would have its domain and it should have the ability to be accessed from either a smartphone or a computer as long as an internet connection is available.

This would also allow the ability for the use of WebSockets. As previously mentioned in Background Research, there are services out there like PythonAnywhere and Heroku. These services do not allow the use of WebSockets and this is a particular technology that would be useful to get the smooth behaviour required and without the strange behaviour of AJAX, but would require building an entire server.

After looking around on reviews and how to get started with acquiring a server, a company by the name of DigitalOcean seemed to be highly recommended and were quite inexpensive to run at only $5 a month. After some thought and browsing through their documentation, it was decided that it was perfect for what the project required.

Looking at the documentation and following along it was not as easy or straightforward at all. The actual construction of the server soon became a big part of this project. This was now an essential thing for the application actually to be launched with complete availability and success.

3.10 Implementing the Server

Looking through the documentation that DigitalOcean published was a massive help in tackling this hugely daunting task. The first thing that was needed was to acquire an actual domain so that people would be able to access the website.

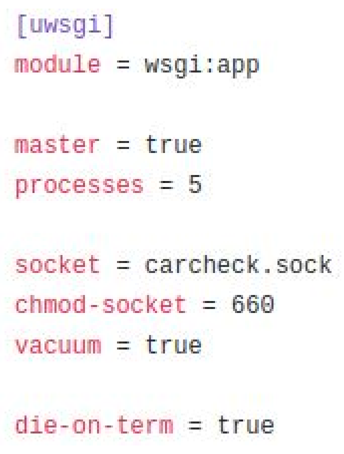
As recommended in their documentation, a domain via Namecheap was acquired for a very minimal price. The name that was decided upon for the website and application is *visualmot.app*.

The first thing to do is to log into the server (Virtual Private Server) which was logged into via SSH (Secure Shell). This was reasonably straightforward to do on the DigitalOcean website. A SSH key was then created and a password was used for logging in to the terminal on Linux.

Initially, the most significant and most complicated task was that all the code would have to be transferred over and it was not definite that the same code would work on this machine fully. The first step was to begin by installing all the necessary libraries and setting up a virtual environment for Python.

3.11 Setting up uWSGI

To start this, a new file was created called uwsgi_config.ini. This INI extension is a configuration file and in this case is specifically for uWSGI.



```
[uwsgi]
module = wsgi:app

master = true
processes = 5

socket = carcheck.sock
chmod-socket = 660
vacuum = true

die-on-term = true
```

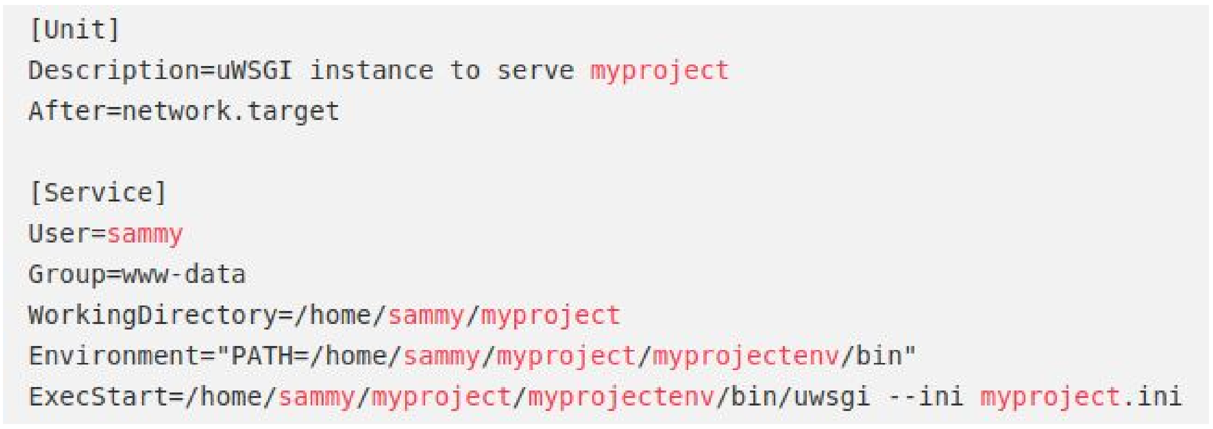
This part works as follows.

1. Module is the actual application that uWSGI will call to execute the backend code.

2. Processors is the maximum number of workers allowed.

3. Socket allows the ability to use a Unix socket which gives a better speed, more security and chmod-socket changes in the permissions on the socket.

Next, another file is created that is a systemd file. This will give the ability to link uWSGI up to Ubuntu’s init system and fire up the application whenever the server boots up. An example of what this looks like is as follows. (Ellingwood, J. Juell, K, 2018)



```
[Unit]
Description=uWSGI instance to serve myproject
After=network.target

[Service]
User=sammy
Group=www-data
WorkingDirectory=/home/sammy/myproject
Environment="PATH=/home/sammy/myproject/myprojectenv/bin"
ExecStart=/home/sammy/myproject/myprojectenv/bin/uwsgi --ini myproject.ini
```

This is a little more simple than the previous image we saw. It starts with a basic description and then we go into the services part. This is where we specify the directory of the application, the Python virtual environment and the uWSGI configuration INI file. After this, the service can be started and enabled.

After a lot of reading on DigitalOcean's documentation, the server was finally up and running using uWSGI and NGINX. The final thing to be done was to make the website secure, changing it from HTTP to HTTPS. This was achieved by using Certbot which is a plugin that reconfigures the configuration automatically. This was extremely useful for getting a quick SSL certificate and having the website be secure. The website must be as secure as possible.

3.12 Selenium Roadblock

The website was all up and running. All the code seemed to be working somewhat fine until the local solution that was working on the original machine was not working as intended on the server. The local version of Selenium is using FireFox as its browser which was fine, but for some reason, it wasn’t working on the server.

Installing FireFox on the server seemed to go in the right direction, but produced errors. After hours of searching through the web, the problem was not exactly clear, but instead it was decided that trying Google Chrome would be another option.

Installing Google Chrome and the Chrome drivers on the server was successful. Then came the task of changing the code over to work on the Chrome browser, but the same errors occurred again. It was finally discovered that servers do not have a screen and Selenium was trying to open the standard GUI version that humans would use. Instead, it was found that Chrome would need to run in headless mode as servers don’t need to have physical screens. The result of this change now looks as follows.

```python
chrome_options = Options()
chrome_options.add_argument('--no-sandbox')
chrome_options.add_argument('--no-extensions')
chrome_options.add_argument('--headless')

try:

    driver = webdriver.Chrome(chrome_options=chrome_options)
    driver.get('https://vehicleenquiry.service.gov.uk/')
    print(driver.current_url)

    driver.implicitly_wait(10)
    inputElement = driver.find_element_by_css_selector(selector[0])
    inputElement.send_keys(car_data['plate'])
    inputElement.send_keys(Keys.ENTER)
    print("Entered Numplate")

    driver.implicitly_wait(10)
    inputElement = driver.find_element_by_css_selector(selector[1]).click()
    inputElement = driver.find_element_by_css_selector(selector[2]).click()

    inputElement = driver.find_element_by_css_selector(selector[3])
    print(inputElement.text)
    car_data['tax'] = inputElement.text

    inputElement = driver.find_element_by_css_selector(selector[4])
    print(inputElement.text)
    car_data['mot'] = inputElement.text

    inputElement = driver.find_element_by_css_selector(selector[5])
    print(inputElement.text)
    car_data['car'] = inputElement.text

    inputElement = driver.find_element_by_css_selector(selector)
    print(inputElement.text)
    car_data['year'] = inputElement.text

finally:
   driver.close()
   driver.quit()
```

This change completely altered the behaviour of the application and it was working exactly as the local version was. Still, due to it using a headless Chrome instance which does not run graphically, it performed quicker than the local version. This is not a bad thing at all as the application would always be previewed on the server.

3.13 Implementing Websockets

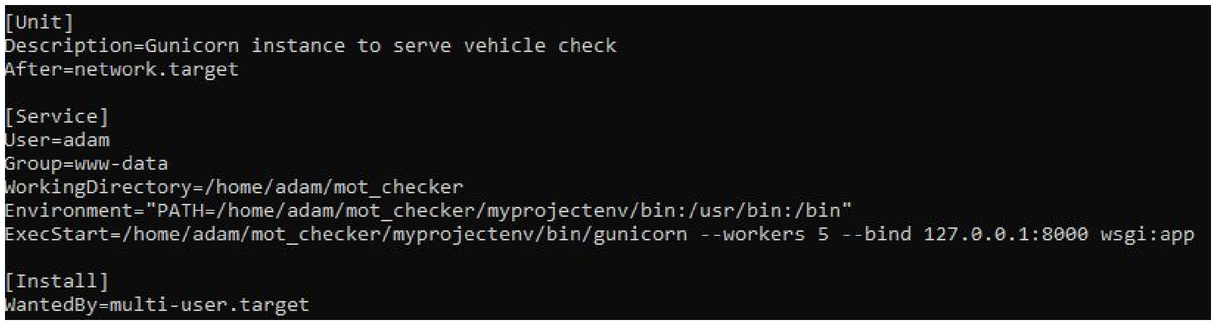
Implementing WebSockets was a very tricky part of this project. It was initially thought that it would not be too difficult as it looks much easier to implement than AJAX does, but this was not the case at all.

Upon trying to implement WebSockets using Flask Socket IO library, it was a little confusing at first to get all of it to work together. Eventually, it was all working and things seemed to be quite fine.

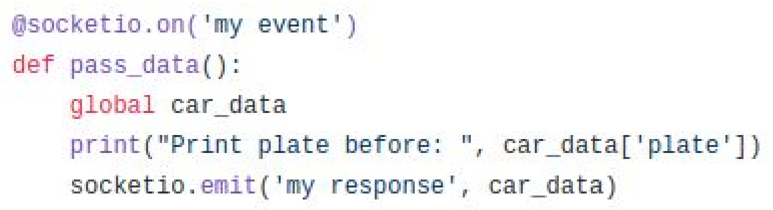
The only issue with switching to this library for WebSockets was that uWSGI was not as looked on as the best approach to implementing this. Guniorn was the widely recommended approach, but this did mean that we would need to switch from using uWSGI. Due to not having a lot of options and development taking a lot of time to perfect, it was decided to switch over to Gunicorn to implement WebSockets

This resulted in a change and “uwsgi_config.ini” was no longer required as stated in ‘Implementing uWSGI 3.11’. From looking at the Gunicorn documentation, it was more straightforward than initially thought and much easier to set up than uWSGI.

The resulting setup now only resides with the use of the systemd file and the NGINX configuration file. The systemd is now as follows and does not use a Linux socket.

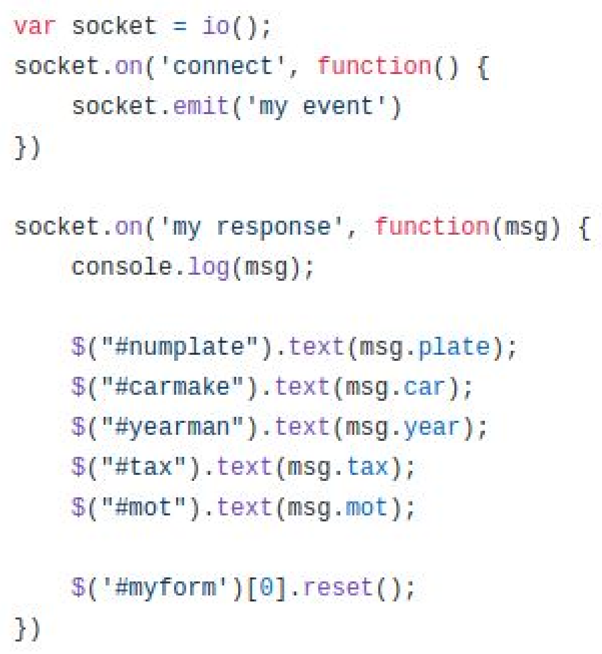


```
[Unit]
Description=Gunicorn instance to serve vehicle check
After=network.target

[Service]
User=adam
Group=www-data
WorkingDirectory=/home/adam/mot_checker
Environment="PATH=/home/adam/mot_checker/myprojectenv/bin:/usr/bin:/bin"
ExecStart=/home/adam/mot_checker/myprojectenv/bin/gunicorn --workers 5 --bind 127.0.0.1:8000 wsgi:app

[Install]
WantedBy=multi-user.target
```



```python
@socketio.on('my event')
def pass_data():
    global car_data
    print("Print plate before: ", car_data['plate'])
    socketio.emit('my response', car_data)
```

This is the implementation on the Backend side. This works by a route ‘my event’.



```javascript
var socket = io();
socket.on('connect', function() {
    socket.emit('my event')
})

socket.on('my response', function(msg) {
    console.log(msg);

    $("#numplate").text(msg.plate);
    $("#carmake").text(msg.car);
    $("#yearman").text(msg.year);
    $("#tax").text(msg.tax);
    $("#mot").text(msg.mot);

    $('#myform')[0].reset();
})
```

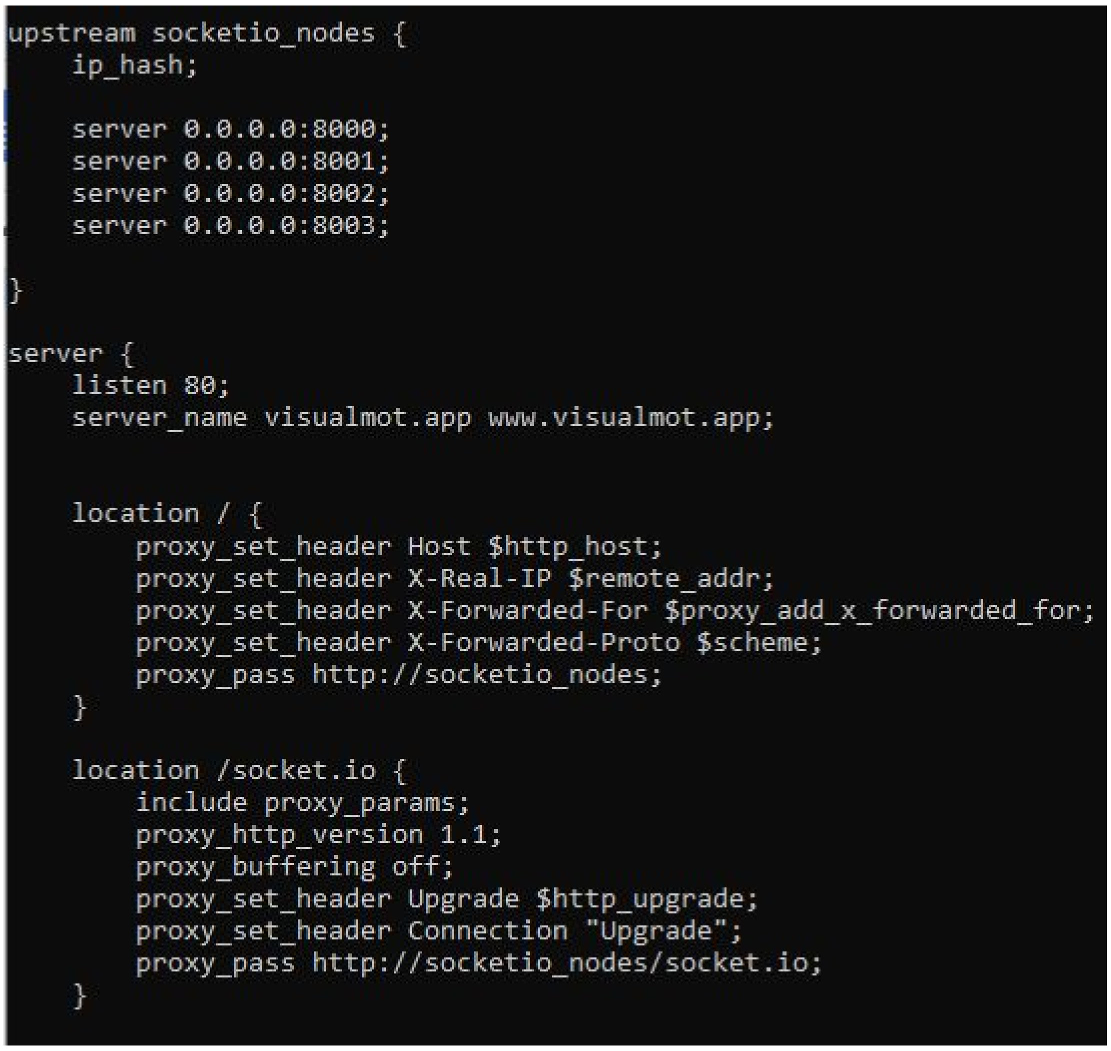
This part is the javascript side. Here, connection to the backend side occurs as well as preparation of the data for displaying it on the HTML page.

3.14 Load Balancing Socketio

After getting socketio to function well and imitate the behaviour of the local version, there was one big issue. The issue was that it only allowed Gunicorn to be able to run on one worker. This caused problems as there was only one instance of the website available. With this kind of application, more than one instance would be required.

With there being only one, this meant that no matter how many users were using the website, they would be in the same state. They would be receiving the same information which is not ideal at all. Because of this issue, the website was practically unusable and went against the reasoning of building a website to begin with.

While looking over the documentation for socketio (Flask-SocketIO, 2020), it was found that the only possible way to make this work was to implement a sticky sessions/load balancer.



```
upstream socketio_nodes {
    ip_hash;

    server 0.0.0.0:8000;
    server 0.0.0.0:8001;
    server 0.0.0.0:8002;
    server 0.0.0.0:8003;

}

server {
    listen 80;
    server_name visualmot.app www.visualmot.app;


    location / {
        proxy_set_header Host $http_host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_pass http://socketio_nodes;
    }

    location /socket.io {
        include proxy_params;
        proxy_http_version 1.1;
        proxy_buffering off;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
        proxy_pass http://socketio_nodes/socket.io;
    }
```

The above is the NGINX configuration setup. This is modified beyond the socketio documentation to try and allow for multiple instances. After some time with the issue, it was decided to go backwards and attempt to try another solution that would allow for multiple instances with each user having their specific state.

The above did not seem to work out and solve the issue. The load balancers intention was to spin up a new instance of the application for each user that requested the website. This was most likely an issue to do with HTTPS, but ultimately it is unknown why this idea did not fully work out as intended

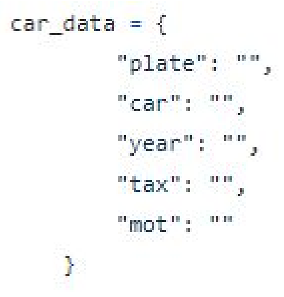
The decision was to go back and try to implement this using AJAX. With this, the ability to use more than one web worker using Gunicorn would be allowed which should allow things to run more smoothly.

3.15 Multiple Instances & AJAX

This was quite a quick process to implement as AJAX has already been achieved for passing images to the server. This seemed to have better behaviour than the previous attempt, but it still had issues with allowing for multiple instances and did not exclude data to different clients.

It was realised that this could potentially be because of a global variable that is being used to pass the vehicle data around. This global was used as a prototype to be able to pass the data around easier quickly. The issue was with the state of the data. Upon uploading an image of a number plate, the application would run as intended and would assume that nothing foreign was occurring. Upon another test of a different image, it would display the previous data. This was causing the data to always be behind the true state of what was happening on the backend of the server.

After refactoring the code and not using the global anymore, the application was successfully functional and allowed for more than one instance. Below is an image of the global dictionary that was being used. This global was used from the start of the application. Globals are a terrible practice in the software industry, but the idea that a global could cause complete failure of an application was not thought of when it was initially placed within the application.



```
car_data = {
        "plate": "",
        "car": "",
        "year": "",
        "tax": "",
        "mot": ""
    }
```

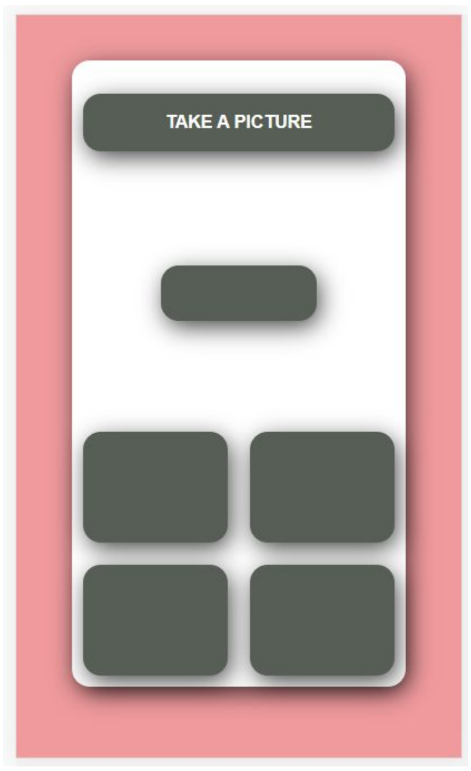
4 Results

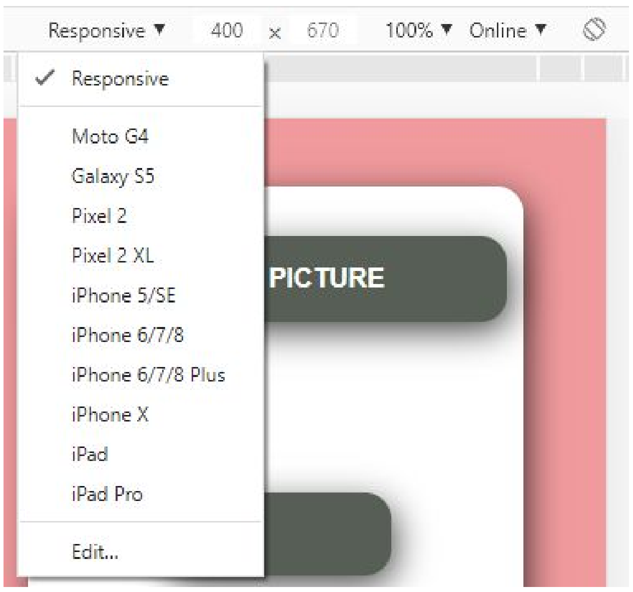
4.1 - Application Design

The application design process has not changed from the initial idea that was originally planned out. This image below is what will be presented with upon arriving on the website. The background colour may be different for different devices. The reason for this was to differentiate which device the application was being used on during development. It was not something that needed to be altered.

This image was previously used in the introduction. As stated, this is the original design that was thought out. The design is simplistic and modern. There are no fancy animations, but the aesthetic was thought out. Throughout the development of this, accessibility and user-ability has been a key goal that was predominantly at the forefront when thinking about this project. The website had to be responsive to allow it to be used on any device.

As seen below using Google Chromes tools, Ctrl + Shift + M can be pressed. This opens a device toggle toolbar and allows us to simulate how the website will look on a particular device.



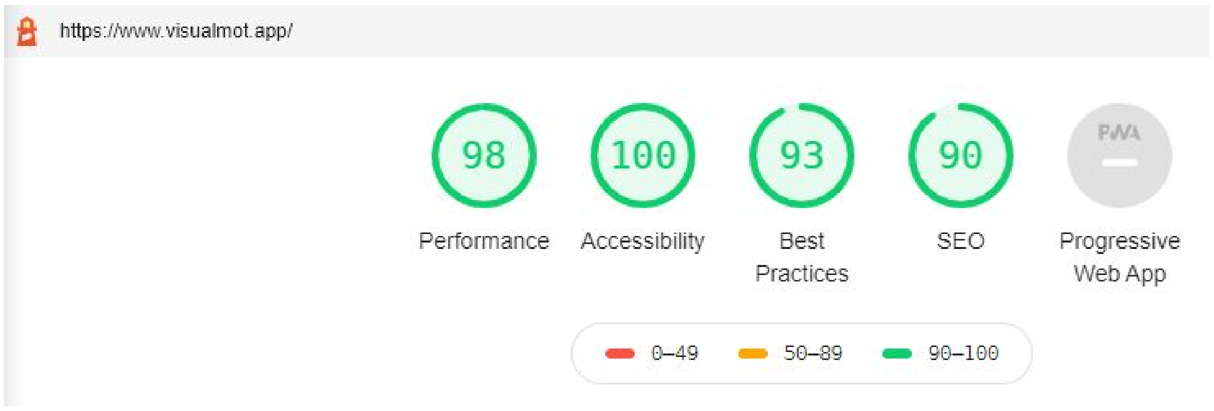


4.2 - Application Performance & Security

The application speed was not particularly as enormous of a contributor as accessibility was, but it also was not left out entirely. Some thought process went into trying to make the application run as smoothly and as quickly as possible.

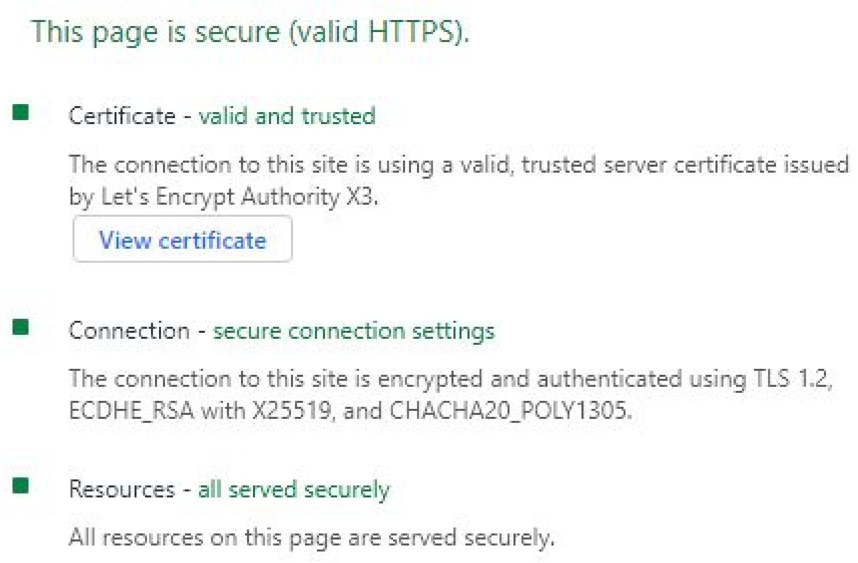
The current speed of the application is around 10 - 15 seconds. The reason for this wait is due to the image being uploaded, character extraction from the number plate, a web scrape on the DVLA's web pages and the actual results being displayed. The time may differ depending on the device's internet connection at the time when it comes to uploading an image.

Another useful tool from Google is the ability to do an overall check of the website to see the main statistics of the website. This is incredibly useful. With this, things were able to be checked on during the development stages to see just how well the application was performing within certain areas.



This runs a simulation, testing each one of these topics. The application scores extremely high marks. With a result like this, and performance not particularly being a big goal with this application, this is an incredible outcome. Another part of this successful result has a lot to do with the fact that an SSL certification was acquired to allow for encrypted traffic.

If this were not the case, the website would not be considered safe by Google. They now notify users if a website does not use HTTPS. As this is a standard within the industry, this is essential.


This page is secure (valid HTTPS).

Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by Let's Encrypt Authority X3.

Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.2, ECDHE_RSA with X25519, and CHACHA20_POLY1305.

Resources - all served securely

All resources on this page are served securely.

5 Evaluation

From the goals that were created at the start of this application, there were lots of difficult parts within the development. One of the main challenges was with setting up the DigitalOcean server to allow for this application to be accessed and used by anyone for testing or just general curiosity. Getting that to be fully operational was a true task and one that was not foreseen at the beginning of this project.

Due to the goals of this project, accessibility was the most crucial factor and because of this, the project has gone through multiple different fields. It has gone from Frontend, Backend and System Administration. This opened up many different types of challenges as the application relied upon not only one technology, but many.

The project eventually took a large step back due to the uses of Websockets and the issues that were causing the application to fail on one of the crucial goals of accessibility. This became obvious that the Websocket technology was not fully capable of the specifications that were set out. True accessibility of the website and multiple instances were the prime components of delivering a fully accessible application.

Due to some parts of the application, it is difficult to be successful in capturing the data in certain instances. The application prefers pictures of number plates that are within the range of around 6 feet or closer. The closer the picture, the better chance that it’ll be successful.

The application does not offer a generalised text extraction and only works with number plates. This is the prime key feature of the application, but a more generalised approach would have had more use cases and could have opened up the application to more possibilities. This is something that could be left towards the future work of the application.

6 Conclusions and Future Work

6.1 - Conclusion

The aims and goals for the application were successful. The main part of the project's functionality explores text extraction and applies this successfully with the use of a deep learning API. The purpose of the application was to be able to offer it for use to anyone with a computer device, whether that be a smartphone, laptop or desktop. This difficult problem was accomplished with the aid of using a DigitalOcean server and the application now resides online via www.visualmot.app/

Overall, the application is how it was initially thought of with most goals being hit. The only goals that were not hit was the use of (OCR) and WebSockets. These two solutions would have been more ambitious. Still, through the project's development and timeline, these options were removed due to taking too much allotted time to complete and were not as successful as other approaches. There was too much uncertainty in the behaviour of the application and a more robust and accessible application was much more captivating.

6.2 - Future Work

One of the processes of the application is using an API to extract the letters and digits from an image. A decision to use this API was due to the unpredictable results from using OCR (Optical Character Recognition). Another way to improve upon this project would have been an artificial intelligence centred approach, more specifically deep learning. This was in mind due to the possibility of a more generalised approach to extracting text from images, instead of solely number plates.

There are other ways that this application could be implemented in the way of helping to automate trivial tasks. This includes the potential to automate your home garage door upon leaving or returning home. There would need to be some alterations made, but it could scan for the number plate when motion is detected and if the plate matches to one on the list of approved entries, the door will open for the car to enter.

This idea would not necessarily need a website to work either. This could potentially be done by using a camera above the garage door and this could then take pictures every couple of seconds. It could then run through the API until a number plate is detected.

The above is a simple thought experiment on how this would potentially work. The only major flaw with this is if there were to be any security flaws that cause the garage door opening to any car. This would be a significant security issue that should not go unchecked. This could be remedied by only having the script run during times that coincide with your schedule on the times you regularly return home.

Another idea for this application would be an automated car parking system. This would be a much more significant undertaking of the application in its current state, but with more modification, it is a possible approach that could be made.

7 References

Beeks, B. (2019) HTML5 Mobile Device Camera Access. Available at:
https://coderwall.com/p/epwmoa/html5-mobile-device-camera-access
(Accessed: 15 November)

Dwyer, G. (2017) Flask vs. Django: Why Flask Might Be Better. Available at:
https://www.codementor.io/@garethdwyer/flask-vs-django-why-flask-might-be-better-4xs7md f8v
(Accessed: 11 November 2019)

Ellingwood, J. Juell, K. (2018) How To Serve Flask Applications with uWSGI and Nginx on Ubuntu 18.04. Available at:
https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-uswgi -and-nginx-on-ubuntu-18-04
(Accessed: 10 December 2019)

Flask-SocketIO (2020) Available at:
https://flask-socketio.readthedocs.io/en/latest/
(Accessed: 5 January 2020)

Hansel. (2013) PythonAnywhere vs Heroku. Available at:
https://blog.pythonanywhere.com/65/
(Accessed: 15 November 2019)

Long, J. (2012) I Don't Speak Your Language: Frontend vs. Backend. Available at:
https://blog.teamtreehouse.com/i-dont-speak-your-language-frontend-vs-backend
(Accessed: 15 November 2019)

Omole, O. (2018) Fixing Rotated Mobile Image Uploads In PHP. Available at:
https://medium.com/thetiltblog/fixing-rotated-mobile-image-uploads-in-php-803bb96a852c
(Accessed: 1 December)

OpenALPR. (2017) OpenALPR API. Available at:
http://doc.openalpr.com/api.html
(Accessed: 10 February 2020)

Wales, M. (2020) *3 Web Dev Careers Decoded: Front-End vs Back-End vs Full Stack. Available at:*
https://blog.udacity.com/2014/12/front-end-vs-back-end-vs-full-stack-web-developers.html
(Accessed: 10 November 2019)

W3Schools (2020) HTTP Request Methods. Available at:
https://www.w3schools.com/tags/ref_httpmethods.asp

(Accessed: 20 November 2019)

W3Schools (2020) HTML Form Attribute. Available at:
https://www.w3schools.com/html/html_forms.asp
(Accessed: 22 November 2019)