
GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

# WhereTo? - Desarrollo de una Aplicación Móvil para la Gestión de Estacionamiento en Tiempo Real

Autor: Velilla Arana, Marta

Director: Pisano, Alan

Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

WhereTo? - Desarrollo de una Aplicación Móvil para la Gestión de Estacionamiento en Tiempo Real

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2024/06 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: Marta Velilla Arana Fecha: …10/ 05/ 2024…

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Alan Pisano Fecha: …10/ 05/ 2024…


GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

# WhereTo? - Desarrollo de una Aplicación Móvil para la Gestión de Estacionamiento en Tiempo Real

Autor: Velilla Arana, Marta

Director: Pisano, Alan

Madrid

# Acknowledgements

Thank you to everyone in Boston University who accompanied me during this exchange year. Specially thank you to John Burke, Muhammad Ahmad Ghani, Erick Tomona and Haochen Sun.

# WHERETO

**Autor: Velilla Arana, Marta.**
Director: Pisano, Alan.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas.

# RESUMEN DEL PROYECTO

Este proyecto, titulado "WhereTo", surge como una respuesta innovadora y tecnológica a la complejidad de la búsqueda de estacionamiento en entornos urbanos. Se propone diseñar una aplicación móvil destinada a proporcionar información en tiempo real sobre localización de los parquímetros y señales de tráfico en Estados Unidos. La necesidad de una solución eficiente se intensifica con la continua expansión de las ciudades y la creciente dependencia de la sociedad en los vehículos.

**Palabras clave**: Aplicación, Dirección, Radio, Parquímetros, Señales, Mapa

## 1. Introducción

En las áreas urbanas, la disponibilidad de estacionamiento se ha convertido en un gran desafío para los conductores debido a la expansión de las áreas metropolitanas. Esta situación genera no solo una carga económica, sino también emocional. La confusión sobre dónde estacionar provoca multas frecuentes; en 2022, Nueva York emitió 8.4 millones de multas, lo que muestra el considerable desperdicio de tiempo y recursos [1].

La aplicación WhereTo busca mitigar estos problemas, reducir emisiones, ahorrar combustible y mejorar la movilidad urbana. Utilizando tecnologías avanzadas y la integración de APIs de Google, WhereTo ofrece una solución escalable que complementa las aplicaciones GPS y de pago de estacionamiento, simplificando la búsqueda de estacionamiento y promoviendo una movilidad urbana más eficiente y sostenible.

## 2. Definición del proyecto

Los objetivos de este proyecto son:

- Simplificar la búsqueda de aparcamiento mediante el desarrollo de una aplicación móvil que permita a los conductores encontrar de manera sencilla información sobre la disponibilidad y regulaciones de estacionamiento en áreas urbanas.
- Optimizar recursos y tiempo. Otro objetivo es minimizar el impacto económico y ambiental causado por la búsqueda ineficiente de estacionar. De esta manera, se ayuda a ahorrar tiempo, combustible, y, por lo tanto, reducir las emisiones.
- Implementar tecnologías innovadoras: emplear modelos avanzados de inteligencia artificial para interpretar de manera precisa las señales de tráfico y de estacionamiento.
- Ofrecer respuestas rápidas y precisas: crear un sistema ágil que responda a las consultas de los usuarios es un tiempo razonable.

Estos objetivos se logran a través de la aplicación creada. Esta es capaz de, con una dirección y radio, mostrar al usuario un mapa donde puede encontrar fácilmente los parquímetros, tanto de espacio simple como múltiple, y las señales de tráfico.

## 3. Descripción del modelo/sistema/herramienta

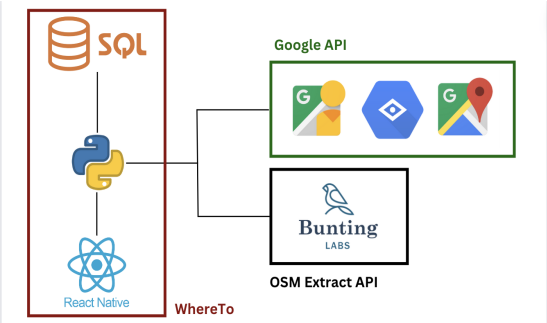



*Ilustración 1- WhereTo high level system*

Desde una perspectiva de alto nivel, el sistema WhereTo es relativamente sencillo. Los componentes que conforman el sistema son las APIs principales en Python alojadas en la nube, la aplicación móvil en React Native, la base de datos SQL alojada en la nube, y las APIs relevantes que WhereTo utiliza desde las APIs principales en Python.

Las APIs en Python son responsables de calcular y entregar resultados. Hay dos endpoints: el endpoint Park, responsable del cálculo de resultados, y el endpoint Detail, encargado de proporcionar información detallada sobre una detección específica. Al recibir una solicitud, la API Park utiliza la API de Bunting Labs para obtener información de calles del área especificada por la dirección decodificada por la API de Google Geocoding. El frontend luego consulta la API de Google Cloud Static Street View utilizando esta información. La API principal en Python utiliza los modelos de aprendizaje automático de WhereTo para evaluar cada imagen de Google y detectar información de estacionamiento relevante. Cada detección de información de estacionamiento o parquímetro se adjunta a una respuesta JSON que se envía al cliente que realiza la solicitud. El endpoint Detail es mucho más simple. Es un contenedor alrededor de la caché, que permite aceptar una solicitud para una detección específica y devolver toda la información sobre esa detección en formato JSON.

La aplicación móvil WhereTo es el punto de entrada del usuario al sistema. Interactúa con el sistema a través de dos endpoints de API en el backend. El frontend envía datos de dirección y radio como parámetros en formato JSON al endpoint de la API Park y recibe los resultados de la detección en aproximadamente un minuto en formato JSON.

Para cada detección, la API principal en Python envía información de la detección a la base de datos SQL, donde permanece hasta que debe ser actualizada debido a una actualización en los modelos de aprendizaje automático que utiliza WhereTo o una actualización en los endpoints de imágenes de Google, posiblemente proporcionando información más nueva y precisa.

## 4. Resultados

El resultado consiste en una aplicación llamada ‘WhereTo’ cuyo funcionamiento es muy sencillo, permitiendo a los usuarios aprovechar todas sus funciones. Con la aplicación instalada, el usuario debe:

1. Ingresar una dirección en el campo de entrada "Enter Address" o usar el botón "Current Location".
    - La barra de entrada tiene la función de autocompletar y el botón "Current Location" funciona si se han habilitado los servicios de ubicación.
2. Seleccionar un radio en metros desde el menú desplegable.
3. Presionar el botón "Find Parking".
4. Esperar unos minutos mientras la aplicación procesa la información.
5. Ver los resultados en la vista de mapa donde los pines representan parquímetros y señales de tráfico.
    - Al presionar una detección determinada, se abrirá un modal con detalles.
    - El usuario puede salir del modal presionando 'Close'.
6. Volver a la vista de entrada presionando el icono 'Home'.

Si se ingresa una dirección real y un radio válido, el usuario podrá ver parquímetros o señales detectadas por el modelo y obtener información detallada sobre ellos.

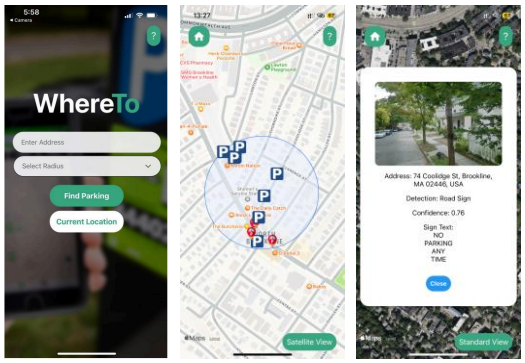



*Ilustración 2- Pantallas principales de WhereTo*

## Conclusiones

El proyecto logró avances significativos al integrar un modelo de machine learning para detectar señales de tráfico y parquímetros, mejorando la precisión y velocidad del procesamiento de datos. La interfaz de usuario, desarrollada con React Native, garantiza compatibilidad con dispositivos iOS y Android, ofreciendo una experiencia intuitiva y eficiente mediante validación de entradas. La integración fluida del frontend y backend, facilitada por el uso de APIs, permitió el procesamiento de datos en tiempo real, mientras que el despliegue en Google Cloud aseguró eficiencia operativa y escalabilidad.

Se han identificado varias áreas para futuras mejoras. Estas incluyen la expansión internacional, mejor visualización de mapas, integración con servicios de pago de estacionamiento y el uso de algoritmos avanzados de aprendizaje automático para ofrecer actualizaciones en tiempo real y análisis predictivos.

## 5. Referencias

[1] Nir, S. M. (2023, october 9). Parking in New York city really is worse than ever. The New York Times. https://www.nytimes.com/2023/10/09/nyregion/nycparking-cars.html

# WHERETO

**Author: Velilla Arana, Marta.**
Supervisor: Pisano, Alan.
Collaborating Entity: ICAI – Universidad Pontificia Comillas

# ABSTRACT

This project, titled "WhereTo", emerges as an innovative and technological response to the complexity of searching for parking in urban environments. It is proposed to design a mobile application aimed at providing real-time information on the location of parking meters and traffic signs in the United States. The need for an efficient solution is intensifying with the continued expansion of cities and society's increasing dependence on vehicles.

**Keywords**: Application, Address, Radius, Parking meters, Parking signs, Map

## 1. Introduction

In urban areas, parking availability has become a major challenge for drivers due to the expansion of metropolitan areas. This situation generates not only an economic burden, but also an emotional one. Confusion over where to park leads to frequent tickets; In 2022, New York issued 8.4 million tickets, showing the considerable waste of time and resources [1].

The WhereTo application seeks to mitigate these problems, reduce emissions, save fuel and improve urban mobility. Using advanced technologies and the integration of Google APIs, WhereTo offers a scalable solution that complements GPS and parking payment applications, simplifying the search for parking and promoting more efficient and sustainable urban mobility.

## 2. Project Definition

The objectives of this project are:

- Simplify the search for parking by developing a mobile application that allows drivers to easily find information about parking availability and regulations in urban areas.

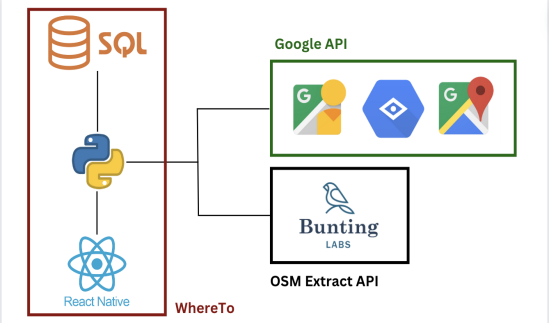
- Optimize resources and time. Another objective is to minimize the economic and environmental impact caused by inefficient parking searches. In this way, it helps save time, fuel, and, therefore, reduce emissions.

- Implement innovative technologies: use advanced artificial intelligence models to accurately interpret traffic and parking signs.

- Offer fast and accurate responses: creating an agile system that responds to user queries takes a reasonable amount of time.

These objectives are achieved through the created application. This is capable of, with an address and radius, showing the user a map where they can easily find parking meters, both single and multiple spaces, and traffic signs.

## 3. System/model description




*Ilustration 1- High level diagram of WhereTo*

From a high-level perspective, the WhereTo system is relatively straightforward. The components compromising the system are the cloud-hosted core Python APIs, the React Native mobile application, the cloud-hosted SQL database, and the relevant APIs that WhereTo calls from the core Python APIs.

The Python APIs are responsible for computing and delivering results. There are two endpoints: the Park endpoint, responsible for the computation of results, and the Detail endpoint, responsible for delivering detailed information about a specific detection. Upon receiving a request, the Park API uses the Bunting Labs API to get street information from the area specified by the address as decoded by the Google Geocoding API. The frontend then queries Google Cloud Static Street View API using this information. The core Python API then uses WhereTo's machine learning models to evaluate each image from Google and detect relevant parking information. Each detection of parking information or meter is then appended to a JSON response forwarded to the client making the request. The Detail endpoint is much simpler. It is a wrapper around the cache, allowing it to accept a request for a specific detection and return all the information about that detection in JSON format.

The WhereTo Mobile application is the user's entry point into the system. It interacts with the system through two API endpoints in the backend. The frontend sends address and radius data as JSON formatted parameters to the Park API endpoint and receives the detection results back within around a minute in JSON format.

For each detection, the core Python API sends detection information to the SQL database, where it lives until it has to be updated due to an update in the machine learning models that WhereTo uses or an update to Google's image endpoints, possibly giving newer, more accurate information.

## 4. Results

Under normal circumstances, the operation of the WhereTo application is exceptionally straightforward, allowing many users to utilize its functions. With the application already installed, all the user has to do to use the application is the following:

1. Enter an address into the first input field with the placeholder "Enter Address" or use the "Current Location" button for the address.
    a. This input field has an autofill address bar, so as the user types, suggestions populate the address bar.

    b. The “Current Location” button will work, provided the user has enabled location services for Expo Go.
2. Select a radius from the dropdown placeholder “Radius in Meters (50,100,150).”
3. Press the ‘Find Parking’ button.
4. Allow the application to compute the necessary information, which may take a couple of minutes depending on street density and radius of selection.
5. View the results once loaded onto the screen as a map view.
    a. Pressing on any pinned detection will open the detail modal, allowing the user to see the location they are interested in in more detail.
    b. The user can press the ‘Close’ button to leave the detailed modal view.
6. Once the user is done examining the results, they can leave the map view and return to the input view by pressing the ‘Home’ icon at the top of their screen.

Under these circumstances, provided that an actual address and valid radius are entered into the input fields, the user will be able to view any parking meters or road signs that the WhereTo model has detected and view detailed information about each one of these that appears within the specified radius.

## 7. Conclusions

The project made significant progress by integrating a machine learning model to detect traffic signs and parking meters, improving the accuracy and speed of data processing. The user interface, developed with React Native, guarantees compatibility with iOS and Android devices, offering an intuitive and efficient experience through input validation. Seamless frontend and backend integration, facilitated by the use of APIs, enabled real-time data processing, while deployment on Google Cloud ensured operational efficiency and scalability.

Several areas have been identified for future improvements. These include international expansion, better map display, integration with parking payment services, and the use of advanced machine learning algorithms to deliver real-time updates and predictive analytics.

## 8. References

[1] Nir, S. M. (2023, october 9). Parking in New York city really is worse than ever. The New York Times. https://www.nytimes.com/2023/10/09/nyregion/nycparking-cars.html

# *Index of the report*

# *Figures Index*

# *Table Index*

# CHAPTER 1: INTRODUCTION

In the dynamic contemporary urban landscape, parking availability has emerged as a significant challenge for all drivers around the world. The constant growing expansion of metropolitan areas has intensified this problem, generating a considerable economic and emotional burden for those seeking to park their vehicles.

As urban areas evolve, the reliance on accurate and reliable information is becoming increasingly more essential. The unpredictability of where to park affects negatively not only to visitors, but also to residents, impacting quality of life and potentially hindering economic growth and development.

For many drivers, the issue of urban parking availability can often result in confusion and bewilderment. For instance, research has shown that the average U.S. driver when attempting to locate a parking space spends $345 in wasted time, fuel and emissions annually [2]. This indicates that locating parking has become a systemic problem. As a problem it contributes to emissions, waste of fuel, and a waste of time for many people. This is not to mention the traffic it causes. On average 34% of cars on the road contributing towards traffic are doing so in search of parking [3]. Therefore, as these urban areas continue to expand, there is going to be a continued growing reliance on real-time parking information, in order to successfully navigate the complex web of city streets.

The dependance on consistent and reliable parking information grows stronger as cities continue to develop and grow, especially in a car-dominant society. Inside these cities, visitors and locals alike are negatively impacted by the unpredictability and difficulty of parking. These negative effects lower the standard of living, and if parking availability is vital to an area but not easily found, it can certainly inhibit economic growth and activity.

Confusion regarding parking availability can cause drivers to– purposefully or otherwise– park in locations that are not valid. As a partial consequence of this fact, in 2022,

law enforcement authorities in New York issued a total of 8.4 million tickets [4]. Not only is this issuance of tickets upsetting to the drivers receiving them, but the volume of these tickets shows that there is a considerable waste of time, money, and resources of law enforcement agencies in attempting to resolve the issue of parking.

This proposal not only aims to mitigate the negative effects of searching for parking, but also aims to contribute to the reduction of emissions, fuel savings and improvement of efficiency in urban mobility. Through the fusion of technologies and the integration of Google APIs, WhereTo is positioned as a scalable and generalizable solution that complements conventional GPS applications and parking payment applications, thus offering a comprehensive experience for users. In this context, the successful implementation of WhereTo promises not only to simplify the search for parking, but also to pave the way towards more efficient and sustainable urban mobility.

The proposed solution to the issue of parking is to incorporate artificial intelligence models into a software system used to deliver real-time information regarding parking regulations and guidelines to a user.

## *1.1 PROJECT MOTIVATION*

In light of the challenges posed by urban parking availability, it has become increasingly evident that traditional methods of navigating to search for parking spaces are no longer sufficient. As cities continue to grow and evolve, the need for accurate, reliable, and real-time parking information has become essential.

The reliance on such information is not only important for visitors, who navigate unfamiliar city streets, but also for residents, seeking to complete their daily routines efficiently. The economic and emotional toll of navigating parking challenges is substantial, with wasted time, fuel, and emissions representing a huge cost for drivers. Moreover, the nature of the problem is underscored by its contribution to traffic congestion, with a significant percentage of vehicles on the road dedicated to the search for parking.

In this context, the need for innovative solutions that leverage technology to address parking challenges is clear. By harnessing the power of artificial intelligence and integrating real-time parking data, the project aims to revolutionize the way drivers navigate urban parking in United States. The application WhereTo seeks to provide users with comprehensive information on parking regulations and availability, empowering them to make informed decisions and streamline their parking experience.

Furthermore, by incorporating new technologies and leveraging partnerships with platforms like Google APIs, WhereTo aims to offer a scalable and adaptable solution that complements existing GPS applications and parking payment systems. By doing so, it does not only seek to simplify the search for parking but also to contribute to more efficient and sustainable urban mobility.

# CHAPTER 2: TECHNOLOGIES DESCRIPTION

## *2.1 BACKEND*

### 2.1.1 PYTHON FLASK RESTFUL APIS

Python flask is a popular framework used for building web applications. Not only it is lightweight and requires minimal setup is a great choice for building small to medium-sized APIs, but developers find a quick and efficient way to create a RESTful API.

On the other hand, REST (Representational State Transfer) is a set of architectural constraints which defines the way an API (Application Programming Interface) should work. It allows an easy, efficient, and secure communication between the frontend and the backend. They use HTTP requests to perform operations on data and typically support standard CRUD (Create, Read, Update, Delete) operations.

Therefore, "Python Flask RESTful APIs," they are likely referring to building web APIs using Python and Flask that follow the principles of REST. This involves defining endpoints (URLs) for different resources, handling HTTP methods (GET, POST, PUT, DELETE) for each endpoint, and returning data in a format like JSON or XML. Flask provides tools and extensions to make building and serving RESTful APIs straightforward and efficient.

### 2.1.2 YOLOV8 MODEL

YOLO (You Only Look Once) is an object detection architecture. It´s main feature is that it is a single-stage detector, which implies it does not require intermediate processes such as RoI (regions of interest) extraction. Instead, YOLO is able to predict the the bounding boxes and class labels for the objects in the image that was used as input using a single neural network and deep learning.

It incorporates several methods such as:

- Classification: It is widely used to identify the general content of an image without knowing any specific object location by determining the class it belongs to.
- Object detection: The way it works is based on dividing the input into a grid and assigning bounding boxes to each cell of that grid. After, the neural network determines whether if each box contains an object or not. This way it can locate multiple objects within an image making it a powerful tool for tasks like robotics, video surveillance or autonomous driving.
- Image segmentation: Once it knows the location of an object, it identifies the exact shape and boundaries of it. It provides a detailed analysis and understanding of the content up to a pixel level information in an efficient and accurate manner.

  A visual example of these three methods is shown in the Figure 1.


*Figure 1- Visual example of YOLOv8 methods [4]*

It emerged in 2015, when a Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi released their research. YOLO marked a breakthrough in object-detection history by introducing a framework that transformed the landscape of computer vision. As it can be seen in Table 1**¡Error! No se encuentra el origen de la referencia.**, the first version of YOLO introduced real-time detection object, since then, it has been continuously improving and adding new features such as more accuracy and speed.

| YOLO Version | Year | Main Advancements |
|---|---|---|
| YOLOv1 | 2015 | Introduction of real-time object detection using a grid-based approach |
| YOLOv2 | 2016 | Incorporation of anchor boxes, feature pyramid networks, and multi-scale prediction |
| YOLOv3 | 2018 | Improvements in accuracy and speed with the introduction of Darknet-53 and multiple detection scales |
| YOLOv8 | 2021 | State-of-the-art advancements in real-time object detection with improved accuracy and speed |

*Table 1: comparison between YOLO versions [6].*

### 2.1.3 GOOGLE APIS

Google APIs are a set of programming interfaces created by Google. They enable the interaction with other Google Services, such as Search, Gmail, Translate or Google Maps, facilitating their integration into other applications. The features this services offer can be used by external applications that utilize these APIs.

These interfaces offer a huge variety of functionalities like analytics, machine learning and access to useful data. For instance, embedding a Google map onto a webpage can be done through these Google APIs.

### 2.1.4 BUNTING LABS OSM EXTRACT API

The Bunting Labs OMS (OpenStreetMaps) Extract API is a tool provided by the company called Bunting Labs which allows users to extract geographic data from the OpenStreetMap database.

OpenStreetMap is a public collaborative project that creates a free geographic data, including maps or coordinates, to anyone who requires them. This database contains ample information about landscapes, roads, buildings, and other features geographically related. It is maintained by volunteers who update the database on a regular basis.

The Bunting Labs OSM Extract API can be used for several reasons:

- Geospatial analysis: to extract data for analysis and decision-making purposes researchers and businesses may be interested in using this API.
- Mapping Applications: software developers may use the database to access up-to-day information about the streets to integrate in their program.
- Location-based services: companies that use location-based services like deliveries or navigation apps, could use the detailed information the database provides to make the business more effective and appealing to customers.
- Customized Mapping Solutions: Individuals may want to use the database for their own benefit.

### 2.1.5 SQL DATABASE CACHE

It is used to compliment the primary database by storing the frequently accessed data temporarily in order to improve the performance of the application. The benefits of using a cache SQL are:

- High performance and scalability,
- Integration with cache objects technology,
- Low maintenance,
- Support for standard SQL queries.

The architecture of the core of the Cache SQL consists of:

- The Unified Data Dictionary: it is the central repository and contains meta-information stored in the form of class definitions. It automatically generates relational access in tables for each class stored.
- The SQL Processor and Optimizer: suite of programs designed to interpret and assess SQL queries. The purpose of this system is to determine which is the most efficient search strategy for each query.
- Cache SQL server: processes responsible for all communication with the Cache ODBC (Open Database Connectivity) and JDBC (Java Database Connectivity) drivers. It contains the frequently accessed queries, therefore eliminating all the redundant queries.

## *2.2 FRONTEND*

### 2.2.1 AXIOS

It is a library that allows software developers to make requests to a server, either their own or a third-party, to fetch data. It offers different options for making an HTTP request such as GET, POST, PUT/PATCH and DELETE.

It uses NodeJS and XMLHttpRequests to work by making an HTTP request. There are three possible outcomes: Success, it will receive a response with the data required; failed, an error will be received; and it could have been intercepted in case a modification might be needed. The way it works is being explained in the Figure 2.

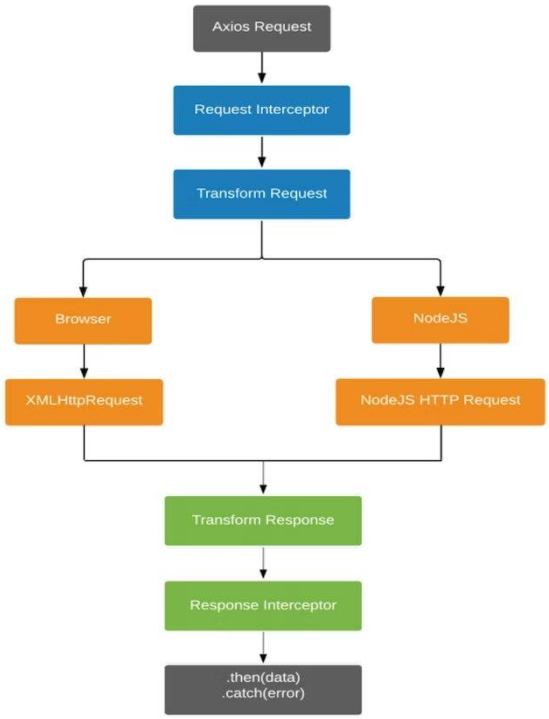



*Figure 2 - diagram of how Axios works [7].*

### 2.2.2 NATIVE-BASE

It is an open-source library that eases the work to developers by providing them with UI (User Interface) components like buttons, images, alerts, modals or cards. Native base was built for React Native and it is supported by Expo, Web and React Native CLI.

It is widely used due to the fact that it creates consistent and aesthetic components for both Android and iOS platforms across all different devices. They are also highly customizable and easily integrated in the React Native projects saving time and effort in designing user interfaces.

# CHAPTER 3: QUESTION STATUS

In the realm of parking assistance, various applications have been tailored to aid users in locating parking spaces or facilitating payment for them. These applications can generally be classified into two primary categories based on their distinct functionalities: General GPS Applications and Parking Payment Applications. Additionally, certain municipalities offer comprehensive information regarding parking regulations within their respective areas.

However, WhereTo is not a direct competitor of either of these types of applications, instead, it would serve as a complementary tool to enhance the user experience and utility of all types of applications.

## *3.1 GENERAL GPS APPLICATIONS*

The most widely used applications in this category are Apple Maps, Google Maps or Waze. They exist for both operating systems, iOS and Android. These applications allow users to search for any location they desire and will provide directions to get to it using the different transportation vehicles available. Most of them use their machine learning algorithms in order to search for the fastest car routes, which takes into consideration the amount of traffic that might be found in the area to calculate alternative routes though smaller roads.

Most of these applications allow the users to search for parking in a general area, but the results for the queries only include parking lots and public garages. Therefore, they are more useful when the user knows in advance exactly where they want to go and are willing to pay the parking lot fees. They are not capable of providing parking information in the way WhereTo intends by leveraging deep learning algorithms. It is positioned to be a more useful and user-friendly application than these existing GPS applications for finding parking, as it will also be able to provide information about street parking to a user.

## *3.2 PARKING PAYMENT APPLICATIONS*

Other potentially competitive technologies are the applications designed to help people with the parking payment throughout their mobile devices. This group includes applications like SpotHero, Passport or Telpark. They allow the user to pay for parking at a specific location without the need to search for parking meters on street parking or paying machines on garages. Additionally, in certain areas, these applications can be used to reserve spaces in advance, typically parking spaces within lots and garages. They are very helpful if the user has already parked and is looking for paying methods in either streets or garages. However, they are not able to inform the user with the parking regulations or parking signs from the location that the client might be interested in.

## *3.3 OPEN DATA PLATFORMS*

Some cities and municipalities provide its citizens access to open data related to parking regulations and traffic in their areas. These datasets may include information about the location of parking meters, signs and regulations on the streets and roads of the area.

An example of an open data platform could be the portal of the city of New York [8]. This portal provides free access to a wide range of datasets related to the city, including data on street parking, parking meter location, parking regulations and restrictions, and more.

The main screen has some information about the open data platform and a search bar for the user to look up their required information as it can be seen on Figure 3. Useful information can be found such as the location of the parking meters in the city center when searching for “Parking meters”. The website features an interactive map interface that allows users to visualize parking meter locations with precision as shown in Figure 4. Users have the flexibility to zoom in on specific areas of interest to gain a closer view of the parking meter distribution across the region. Additionally, complementary to the map display, a

detailed table (Figure 5) is presented below, providing comprehensive information such as coordinates, street names, and the current status of each parking meter.

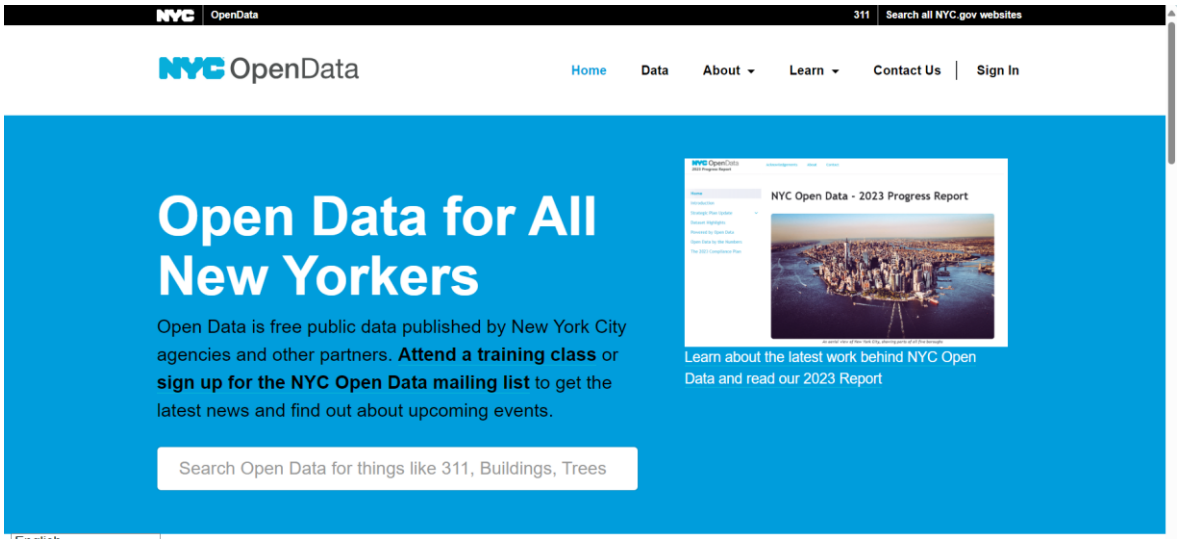



*Figure 3- Main screen of New York open data web [8].*

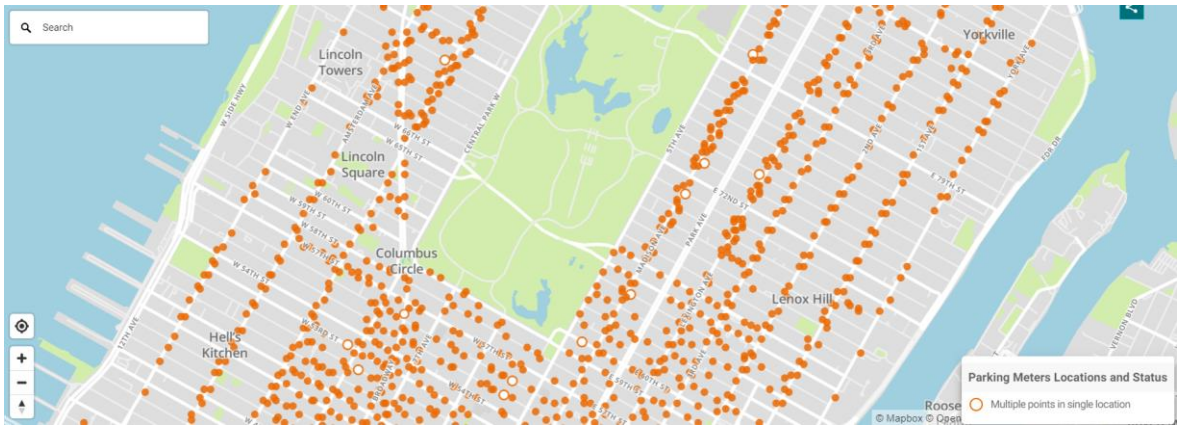



*Figure 4- Map displayed with New York parking meters [9].*

| Object... | Meter ... | Status | Pay B... | Meter... | Parkin... | Facility | Borou... | On_Str... | Side_o... | From_... | To_Str... | Latitud | Longit... | X | Y | Locati... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15382 | 1133111 | Active | 104236 | 2HR Com ... | | On Street | Manhattan | 1 AVENUE | W | EAST 13 S... | EAST 14 S... | 40.730933... | -73.98302... | 988,955.2... | 205,575.6... | POINT (-7... |
| 6722 | 2313022 | Active | 210224 | 2 HR Pas ... | | On Street | Bronx | WESTCHE... | N | BUHRE AV... | HOBART A... | 40.847410... | -73.83196... | 1,030,740.... | 248,056.5... | POINT (-7... |
| 14032 | 3193088 | Active | 325501 | 2 HR Pas ... | | On Street | Brooklyn | BROADWAY | E | MADISON ... | PUTNAM ... | 40.688062... | -73.91940... | 1,006,601.... | 189,966.4... | POINT (-7... |
| 1976 | 2162816 | Active | 214324 | 2 HR Pas ... | | On Street | Bronx | WILLIAMS... | E | EASTCHE... | POPLAR S... | 40.844712... | -73.84632... | 1,026,769.... | 247,066.3... | POINT (-7... |
| 3768 | 4443052 | Active | 456736 | 2 HR Pas ... | | On Street | Queens | 104 STRE... | W | 103 AVEN... | LIBERTY A... | 40.682375... | -73.83722... | 1,029,396.... | 187,926.0... | POINT (-7... |
| 3509 | 4293069 | Active | 419794 | 2 HR Pas ... | | On Street | Queens | LIBERTY A... | N | 120 STRE... | 121 STRE... | 40.686805... | -73.82288... | 1,033,370.... | 189,547.9... | POINT (-7... |
| 12405 | 3053018 | Active | 315008 | 2 HR Pas ... | | On Street | Brooklyn | EAST 16 S... | W | AVENUE P | KINGS HI... | 40.610000... | -73.95763... | 996,013.4... | 161,518.8... | POINT (-7... |
| 6930 | 2073001 | Active | 207067 | 2 HR Pas ... | | On Street | Bronx | WEST GU... | S | EAST GUN... | KNOX PLA... | 40.882895... | -73.88174... | 1,016,950.... | 260,962.5... | POINT (-7... |
| 12340 | 1323001 | Active | 101872 | 2HR Pas ... | | On Street | Manhattan | AVENUE O... | W | BROOME ... | SPRING S... | 40.724653... | -74.00449... | 983,005.1... | 203,287.2... | POINT (-7... |
| 512 | 1058387 | Active | 113769 | 2 HR Pas ... | | On Street | Manhattan | WEST 181... | S | MAGAW P... | FT WASHI... | 40.850689... | -73.93774... | 1,001,473.... | 249,212.7... | POINT (-7... |
| 499 | 1264951 | Active | 114518 | 3HR Com ... | | On Street | Manhattan | 2 AVENUE | E | EAST 79 S... | EAST 80 S... | 40.773804... | -73.95458... | 996,827.7... | 221,197.7... | POINT (-7... |
| 3054 | 4283050 | Active | 459332 | 2 HR Pas ... | | On Street | Queens | NORTHER... | S | 157 STRE... | 158 STRE... | 40.763509... | -73.80768... | 1,037,523.... | 217,502.4... | POINT (-7... |
| 2324 | 5053335 | Active | 521125 | 2 HR Pas ... | | On Street | Staten Isla... | GUYON AV... | W | AMBOY R... | NORTH RA... | 40.564742... | -74.12786... | 948,724.3... | 145,053.1... | POINT (-7... |

< Previous  Next >  Showing Parking Munimeter in NYCs 1 to 13 out of 13,336

*Figure 5- Table displayed with information about New York parking meters [9].*

# CHAPTER 4: PROJECT DEFINITION

## *4.1 JUSTIFICATION*

### 4.1.1 PROVIDES INFORMATION ABOUT TRAFFIC SIGNS

It is the first ever mobile application that offers comprehensive insights into parking signage information. It uses cutting-edge technology to detect and interpret a wide range of traffic signs, presenting users with an indispensable tool for navigating complex cities. It can accurately interpret the content and context of each sign, extracting relevant information pertaining to parking regulations, restrictions, and conditions.

The user Interface displays this data in an intuitive and visually attractive map screen, where the dynamic pins point the location of, not only traffic signs, but also parking meters. When interacting with a pin, users obtain access to detailed information about the desired traffic sign. This includes insights into the confidence level of the model´s interpretation, textual transcriptions of the sign´s context and supplementary visual evidence captured by the application.

### 4.1.2 COMPLEMENTARY TO OTHER APPLICATIONS

Unlike conventional applications for paying parking like SpotHero or Telpark, which serve the function of simplifying the parking payments in different areas, WhereTo offers a different approach to urban mobility solutions. While these platforms exist to speed up the payment process for parking spaces within predetermined lots or garages, the application goes beyond mere transactional facilitation.

WhereTo does not intend to replace or these applications, since it does not feature a way of paying for a parking space. Instead, it leverages advanced technology to interpret and display real-time information regarding parking-related traffic signs and the precise

locations of parking meters. By integrating this data into its interface, WhereTo empowers users with valuable insights into parking regulations and availability, allowing them to make informed decisions about where and how to park.

### 4.1.3 USEFUL AMONG ALL UNITED STATES

The capability of WhereTo to detect traffic signs and parking meters across the entire United States presents a significant and competitive advantage over the open data platforms, such as those provided by certain cities or municipalities. This is due to the fact that said datasets offer valuable information about local parking regulations among their respective areas, so their scope is limited to specific jurisdictions.

On the contrary, WhereTo transcends geographical boundaries by providing coverage among all United States. This range assures that users can access parking-related information regardless of their location within the country. Whether users are driving on urban streets like New York or exploring suburban roads, WhereTo remains a reliable and comprehensive resource for parking information.

The ability to offer coverage on a national level improves the utility and practicality of the application for all users who are traveling or residing in different states of the country. Instead of relying on fragmented or localized data sources, users can access a centralized platform that encompasses information related to parking from diverse regions into a simple and single interface.

The main limitation of the app is it depends on Google Street View data since WhereTo leverages it for visual and locational information. Consequently, if Google Street View lacks data from a particular area, the application will also be unable to provide information for that location. This dependency means that the application´s functionality is directly related to the accuracy of Google Street View´s data. Therefore, if users experience gaps or inconsistencies in the application´s coverage is because Google has not yet captured or updated its imagery.

### 4.1.4 ACCESSIBLE PARKING INFORMATION

WhereTo goes beyond offering general information about parking since it is able to provide details about specialized parking spaces, including those designated for handicap parking. The application is designed taking into account the inclusivity of everyone, recognizing the importance of accessible parking for individuals with disabilities. By using the latest technology of image recognition, WhereTo is capable of identifying these crucial spaces on its interactive map. Users who require accessible parking options can locate easily these parking spaces due to its simple user interface.

## *4.2 OBJECTIVES*

The objectives of the project are:

1. Create and design a user-friendly mobile application. The user should be able to write a desired address and a radius and, using these inputs, the app will display a visual map of the location with all the parking information available. The goal is to make a simple user interface for all users with different technical skills, ensuring that all of them can understand how the application works and can find all the detections found by the model.
2. The integration of Google APIs is capable for enhancing the application´s functionality. By using accurate and up-to-date information about the different areas in United States, the user is presented with reliable and enriched with contextual information about the location of the parking meters and parking signs available.
3. Develop and implement efficient image processing algorithms. The application requires a model to process large amounts of data and images, it must handle tasks such as breaking down street segments into more manageable areas, calculating distances or detecting objects like parking meters.

4. Must be able to use AI in order to read text from the images where the model detected a parking sign. Is essential to ensure that WhereTo can provide the most accurate information about parking regulations, which includes being able to understand the signs.
5. Use a cache to store temporal data to avoid redundant calculations. This mean that when the user makes a request, the detections found will be saved in a SQL cache so if another user request the same information, the application can quicky retrieve it instead of reprocessing the same coordinates.
6. The user must be able to find the application simple and easy to use. Some functionalities are the use of different icons to show each of the detection types, add an autocomplete form on the address input, or put help buttons to show the new users how to use WhereTo.
7. It is necessary to maintain a high confidence level on all the detections. This ensures that the information provided to the users is both accurate and relevant. In order to do this, the application will use a combination of algorithms and APIs to guarantee a high confidence.
8. The main objective of WhereTo is to provide users with a straight-forward mobile application that has available parking information. By offering a real-time data and easy access to all the parking meters and signs. Consequently, the stress related to parking will be reduced as well as reducing the time of it.

## *4.3 METHODOLOGY AND TASK BREAKDOWN*

The methodology for completing the WhereTo system involves a series of specific tasks for each component. Below are the main planned tasks for each section:

First of all, the introduction and motivation of the project was necessary. The problem of limited and inconvenient parking information to drivers in urban areas was identified. The

motivation for the project was made and research to determine if the idea was unique or not too.

The goal was to create a user-friendly application that visually represents real-time information about parking for a desired area.

### 4.3.1 REACT NATIVE UI

**Task 1: Design the Initial Application Screen**

Develop a basic and simple user interface, including the name of the application, the input for the user (address and radius), and the search button.

Once the user has input the address and clicks the 'Find parking' button, the frontend should verify that both inputs are in an accepted format and that the address is valid.

**Task 2: Implement the Map Screen**

When the request is made, after a few minutes, the frontend should be able to develop a screen that will show the detection information received for that address within a certain radius.

The functionalities implemented must allow the user to click on the icons of the map to open a modal with detailed information.

**Task 3: Implement the Detailed Information Modal Screen**

Develop a modal screen that will show the user the detailed information about a specific detection where the user has clicked the pin. It shows information such as the exact location, a picture of it, the type of detections and the confidence level of the model.

**Task 4: General frontend enhancements**

Other functionalities that are implemented are:

- the autocomplete feature when the user is typing the address,
- the current location button in the first screen,
- being able to change from standard view to satellite view,
- The 'help' buttons that explain the functionalities of the app to the new users.

### 4.3.2 PYTHON API

**Task 1: Create Street Traversal Algorithm**

Develop an algorithm that takes a string address and a radius as inputs from the user and returns a dictionary indexed by the street name with latitude-longitude coordinate pairs.

Said coordinates, should be ordered alphabetically to travel in both directions along the streets.

**Task 2: Develop Object Detection Model: Parking signs**

Create an object detection model that generates a bounding box around anything the model believes it to be a parking sign related to traffic.

**Task 3: Develop Object Detection Model: Parking meters**

Train the same model to detect, not only parking signs, but also parking meters. The bounding box is also generated for these images.

**Task 4: Create Algorithm to Read Text**

Develop a function capable of processing Google Street View images and read the texts that appear on them. It is necessary to understand the text in order to provide the user with that information.

**Task 5: Develop Algorithm to Display Information on the Map**

Once everything is done, it is essential to create a function for the final stage of the pipeline, which, with the list of streets and the parking detections, is capable of making a query to Google Static Maps API and indicate where all the detections are located.

### 4.3.3 MACHINE LEARNING MODEL

**Task 1: Comprehensive Model Improvement with Diverse Data**

Expand the machine learning model´s capabilities by training it with a more diverse and extensive dataset that includes a wider variety. It is important to include parking sings and meters from different countries across all United Stated to improve the model´s accuracy and precision.

**Task 2: Optimize Training and Model Deployment Efficiency**

In order to improve the application, only the detections with a high confidence level will be shown to the user, therefore, it is extremely important to optimize the training by exploring advanced techniques.

### 4.3.4 SQL CACHING

**Task 1: Define the Data to Store**

Create an SQL database for the data to be stored in. It must be ordered and coherent so that the data is index by the unique pair of coordinates for each segment or detection.

**Task 2: Implement a Database Interface in the Python API**

Create functions to read and write information in the SQL database from the Python API. Before that, it is necessary to verify again that the address exists, and it is in a valid format for the backend to use.

## *4.4 ECONOMIC PLANNING AND ESTIMATION*

### 4.4.1 COST BREAKDOWN

WhereTo has no initial cost to it. The only cost of WhereTo exhibits is maintenance and hosting. Additionally, WhereTo must pay to access new Google Street View image data past a certain point each month.

In terms of hosting, WhereTo can be comfortably hosted on a C2 Standard instance from Google Cloud. This instance has 16 vCPU, 8 cores, and 64 GB of memory. The database for WhereTo can be comfortably hosted on a Standard 2 vCPU machine with 8GB of memory from Google Cloud SQL. These incur no initial cost but rather an expected monthly cost. Google provides estimates for the expected cost. The exact breakdown can be seen in Table 2.

As for accessing the image data, Google Street View charges $7 per 1,000 images. On average, it was researched that WhereTo will analyze anywhere from 250 to 350 images from Google in a single large pass with no value in the cache. It has been therefore estimated that a given run past the initial $300 per month Google Street View credit will cost around $2.10 in Google Street View Images. The good news is that this batch of images is only paid for once, as the data is then stored in the cache.

| Name | Description | Expected Cost |
|---|---|---|
| Google Compute - C2 Standard 16 vCPU, 8 core, 64 GB memory | Cloud server to host backend API | $488.78 per month |

| | | |
|---|---|---|
| Google Cloud SQL - Standard 2 vCPU, 8 GB memory | Cloud SQL storage instance | $201.60 per month |
| Google Street View Static API Images | Cost of image data from Google | ~$2.10 per API call (past 150 calls) |

*Table 2- Cost of Production and Maintenance of Beta Version.*

As seen from the table, there is a definitive expected monthly cost of $690.38 from combining the cost of the server and SQL storage instances. In addition to this monthly cost, there is the expected cost of around $2.10 for each full call to the Park API after $300 in monthly Google API use credits is exhausted. It can not be determined at this point how much that will amount to, but it should be kept in mind, and the actual operating cost should be reevaluated once more data is available.

Overall, the WhereTo beta version should initially expect a monthly operating cost of around $700 to provide the service to users.

### 4.4.2 GANTT CHART

The Gantt chart presented shows the timeline planning of a development project divided into different phases and tasks. Each row represents a specific task, while the columns represent the calendar weeks from November to March. The colored blocks indicate the duration of each task and when it is scheduled to be performed.

| | Duration | November | | December | | | | January | | | | February | | | | March | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 16-23 | 24-30 | 1-7 | 8-14 | 15-21 | 22-31 | 1-7 | 8-14 | 15-21 | 22-31 | 1-7 | 8-14 | 15-21 | 22-29 | 1-7 | 8-14 | 15-21 | 22-31 |
| **React Native UI** | 58 | | | | | | | | | | | | | | | | | | |
| Design lnitial Apl cation Screen | 7 | | | | | | | | | | | | | | | | | | |
| lmplement the Map | 21 | | | | | | | | | | | | | | | | | | |

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Return Screen | | | | | | | | | | | | | | | | | | |
| lmplement the Detailed lnformatio n Modal Screen | 30 | | | | | | | | | | | | | | | | | |
| **Python API** | 70 | | | | | | | | | | | | | | | | | |
| Create Street Traversal Algorithm | 14 | | | | | | | | | | | | | | | | | |
| lmplement algorithm for performing object detection | 14 | | | | | | | | | | | | | | | | | |
| Add text recognition to road sign detection | 14 | | | | | | | | | | | | | | | | | |
| Add detail API for image and text data | 14 | | | | | | | | | | | | | | | | | |
| **Add detection location prediction** | 14 | | | | | | | | | | | | | | | | | |
| Machine Learning Model | 42 | | | | | | | | | | | | | | | | | |
| Developed a model for detecting single space parking meters | 14 | | | | | | | | | | | | | | | | | |
| Developed a model for detecting multispace d parking meters | 14 | | | | | | | | | | | | | | | | | |
| Developed a model for detecting road signs Pertaining to parking | 14 | | | | | | | | | | | | | | | | | |

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **SQL Caching** | 35 | | | | | | | | | | | | | | | | | | |
| Develop how the data is going to be stored | 7 | | | | | | | | | | | | | | | | | | |
| Created schema file for storing coordinate s and detections | 14 | | | | | | | | | | | | | | | | | | |
| Added an interface in the python api for connecting to the database | 14 | | | | | | | | | | | | | | | | | | |
| **Deployme nt** | 56 | | | | | | | | | | | | | | | | | | |
| Deploy the Python API | 14 | | | | | | | | | | | | | | | | | | |
| lntegrate the Python API with the U1 | 14 | | | | | | | | | | | | | | | | | | |
| Deploy SQI database to cloud | 14 | | | | | | | | | | | | | | | | | | |

*Table 3- Gantt Chart for WhereTo.*

As it can be seen in Table 3, the project has distinct phases for the User Interface design, backend development, model training, SQL caching and deployment.

At first, the efforts are significantly more focused on designing and implementing the React Native UI, with tasks starting on November and finishing in January. This is done because it is essential to have a functional frontend before integrating it with the backend functionalities.

The Python API development and machine learning model training run from early December through January. The SQL cache tasks are scheduled during December, focusing on setting up the data storage mechanisms early in the project to reduce the calls to the Google API.

The deployment tasks are at the end, in March, indicating that the final effort is put towards ensuring that all components are integrated effectively.

| | Duration | April | | | | May | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Writting the paper** | 70 | 1-7 | 8-14 | 15-21 | 22-30 | 1-7 | 8-14 | 15-21 | 22-31 |
| Technologies Description and Question Status | 14 | | | | | | | | |
| System Description | 14 | | | | | | | | |
| Results and conclusions | 14 | | | | | | | | |
| Summary | 14 | | | | | | | | |

*Table 4- Gantt Chart for writting paper.*

Once the project was finished, it was essential to write the paper. It was divided into four categories: Technologies Description and Question Status, System Description, Results and conclusions and Summary. It has a structured procedure with a clear progression as shown in Table 4.

# CHAPTER 5: SYSTEM/MODEL DEVELOPMENT

## *5.1 SYSTEM ANALYSIS AND REQUIREMENTS*

### 5.1.1 REQUIRED MATERIALS

**Hardware:**

- Personal Computer: to host the React Native Application as well as interact with the Google Cloud Compute instance via SSH (Secure Shell), because it provides a secure encrypted transmission of the data.
- Personal Smartphone (iOS or Android): to experience with the user interface of WhereTo via Expo Go.

**Software:**

- Python Park API: API that makes use of ML (Machine Learning) models as well as algorithmic design and API calls in order to analyze parking information for the user.
- Python Detail API: API that makes use of the cache as well as Google Services to forward more detailed information about detections to the user interface.
- React Native UI: React code to display the user interface on mobile devices.
- Expo Go: a mobile application designed for testing React Native applications on iOS.
- SQL Database: the cache, used to reduce excessive calls to the Google Services APIs.

### 5.1.2 FUNCTIONAL REQUIREMENTS

The functional requirements for the WhereTo application can be broken down into multiple key aspects, each corresponding to different areas of user interaction.

**Address Entry:**

- Input field: The main screen provides an input field where the user must write the desired address. It has also an autofill feature that suggest possible addresses as the user types.
- Current location detection: It includes a "Current Location" button that retrieves the user's current location, assuming that location services have being enabled for the Expo Go application.

**Radius Selection:**

- The radius selection is done through a dropdown menu with predefined options (50, 100 or 150 meters) for users to select the desired search area.

**Search functionality:**

- There is a "Find Parking" button that starts the search. It only makes a query once the user has input a location and a radius. Once it is pressed, the data processing can take several minutes to cover the area and send back the results.

**Display of results:**

- Map screen: Once the results are received, they are shown in an interactive map, showing the location of the detected parking meters (both single-space and multi-space) and parking signs. Different pins are used to differentiate between both.
- Details modal: the user is able to select a specific pin from the map and a modal will appear providing more detailed information about the parking meter or sign. It shows the location, model confidence and visual evidence of the desired pin. In order to exit said modal, there is a close button where the user returns to the map view.
- Home icon: at the top of the screen there is a "home" icon that allows the user to return to the initial input screen.

**Error handling:**

- Input validation: Once the user clicks on the search button, to eliminate useless calls to the backend, the address and the radius are verified.
- Error messages: In case the address is not recognized, an error message appear on the screen explaining the issue to the user.

## *5.2 SYSTEM DESIGN AND IMPLEMENTATION*

### 5.2.1 OVERALL ARCHITECTURE

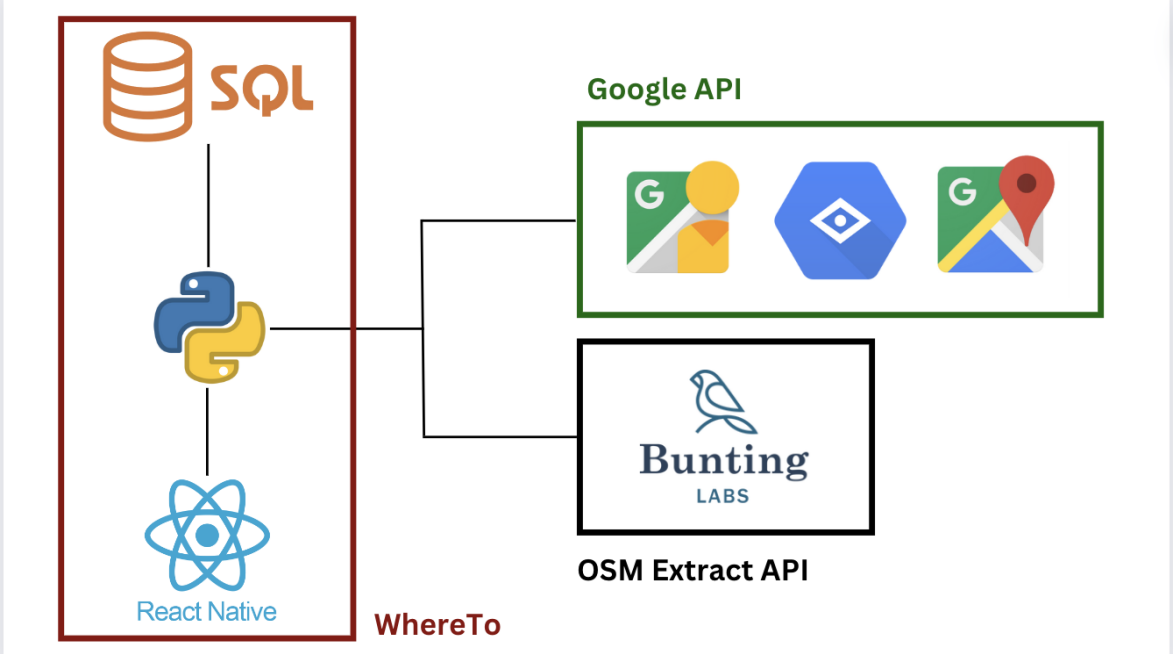



*Figure 6- System block diagram for entire WhereTo application.*

From a high-level perspective, the WhereTo system can be considered relatively straightforward. The components making up the system are the cloud-hosted core Python APIs, a cloud-hosted SQL database, a React Native mobile application and the essential APIs that WhereTo calls from the core Python APIs as it can be seen on Figure 6.

The Python APIs are an integral part of the WhereTo application since they handle both the computation of the parking information and the deliver to the detailed data to the user. The architecture includes two primary endpoints: endpoint Park and endpoint Detail. The first one is responsible for the computation of the parking information. When a request is received on the backend, it goes through several steps to ensure the data is accurate and reliable. Said requests is a JSON with the location and the radius that the user has input before. The Park API uses the Google Geocoding API to decode the provided address into geographical coordinates, which then are sent to the Bunting Labs to gather street information for the specified area. With the information retrieved, the frontend then queries the Google Cloud Street View API to obtain the images of the area and display a map to the user.

The core Python API employs machine learning models to analyze the images gotten previously from Google. These models are trained to detect relevant parking information such as parking meters, traffic signs and other pertinent details. Each piece of pertinent information is appended to a JSON response, which contains the type of detection, geographical coordinates, the confidence of the model and the URL where the image is stored. This JSON is then forwarded to the frontend, providing the user with comprehensive parking information for the requested area.

The Detail endpoint is designed to provide specific information about a certain detection. It functions like a wrapper around the cache, accepting requests for these determined detections. When receiving a request, it retrieves all the relevant information from the cache, like confidence level or an image among others. The endpoint then returns the detailed information in JSON format, making sure the user can access in-depth data about any specific detection.

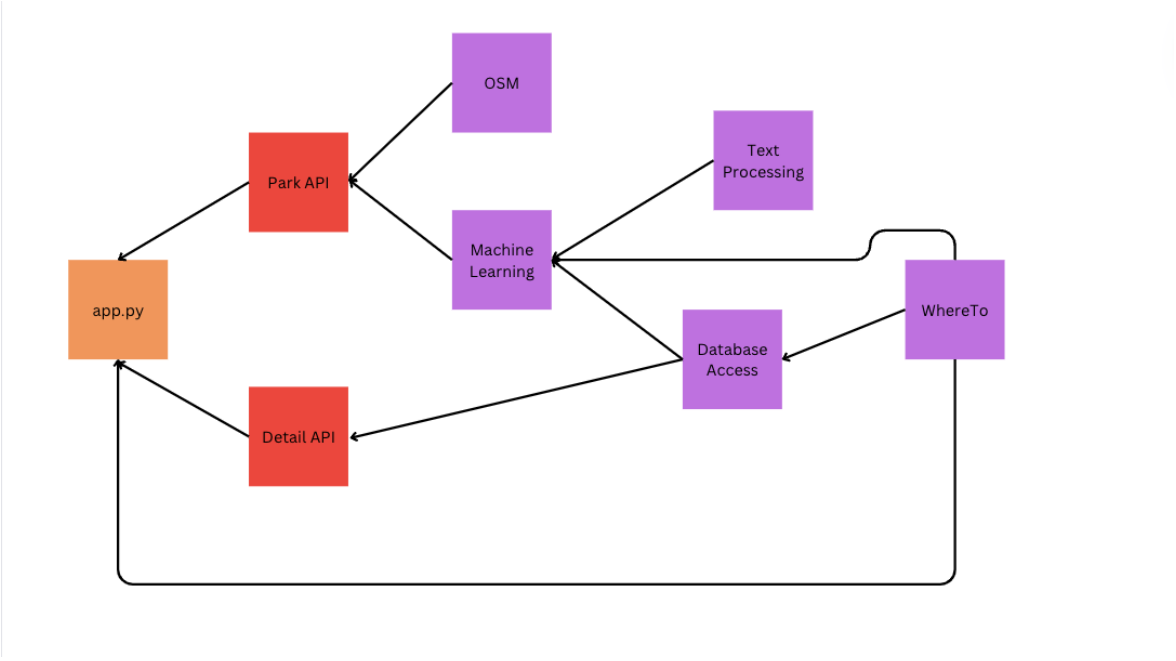
The WhereTo mobile app serves as the user´s interface to the system, interacting with the backend though the endpoints. The process starts with the user typing a desired address and selecting a radius, which the frontend later sends to the Park API endpoint as a JSON request. The frontend receives the results from the same API, typically within a minute if it

is not already saved information on the cache and displays the results on a map. When the user selects a specific detection pin on the map, the frontend sends a request to the Detail API endpoint to get more specific information about that detection. The information received is shown in a modal view.

Additionally, the core Python API sends detection information to an SQL Server, where it is stored temporarily. The stored data is updated whenever there is an improvement in WhereTo´s machine learning models or updates to Google´s image endpoints, ensuring the information is always accurate and up to date.

### 5.2.2 BACKEND DESIGN

*Figure 7- WhereTo backend software system diagram.*




The backend of the WhereTo application is made by the combination of several modules, each with different responsibilities and task that contribute to the overall functionality and performance of the system. Below are detailed descriptions of each module shown in Figure 7.

#### 5.2.2.1 Database

There are two tables present within the SQL Cache, coordinate, and detection.

The coordinate table simply stores the longitude and latitude of every coordinate, as well as a unique 'cid' standing for coordinate id. This id is used to link the coordinate table (Table 5) to the detection table (Table 6).

The detection table stores the 'did' or detection id of a detection in addition to the coordinate id representing the coordinate from where the detection was found. Additionally, the detection table stores the 'lat' (latitude) and 'lng' (longitude) that the detection was predicted to be located at. These values are not equal to the coordinate table's 'lat' and 'lng', but rather to the guess made in the API based on the heading and x location of the detection. The detection table also stores the type of detection, the confidence score of the detection, the image url for the detection, and text read from the sign if the detection was a road sign.

| Coordinate Table | | |
|---|---|---|
| Name | Type | Description |
| CID | Int (PK) | ID value to reference the coordinate being stored |
| LAT | Float not null | Latitude of the coordinate being stored |
| LNG | Float not null | Longitude of the coordinate being stored |

*Table 5- schema of the coordinate table.*

| Detection table | | |
|---|---|---|
| Name | Type | Description |

| | | |
|---|---|---|
| DID | Int (PK) | ID value to reference the detection being stored |
| CID | Int not null (FK) | ID value to the reference coordinate from where the detection was seen |
| LAT | Float not null | Latitud (predicted) of the detection being stored |
| LNG | Float not null | Longitud (predicted) of the detection being stored |
| CLASS_NAME | Text not null | Detection classifier, road sign or meter |
| CONF | Float not null | Confidence score output from model, between 0 and 1 |
| TEXT_READ | Text | Text read from detected road signs |
| IMAGE_URL | Text not null | URL to see this detection via Google Street View |

*Table 6- schema of the detection table.*

In these tables, “PK” refers to Primary Key, while “FK” refers to Foreign Key. It is a one-to-many relationship, which indicates that one coordinate can be associated with multiple detections.

#### 5.2.2.2 Database Access Module:

This module is essential for defining the models for the data being stored in the SQL cache. Additionally, this module implements the functions necessary for interacting with the SQL cache, which are used by other modules.

It has five functions:

- First of all, there is a function called 'locationExists', that checks if a coordinate with the specific latitude and longitude exists in the database. It returns the coordinate Id ('Cid') if it exists, otherwise 'None'.

```
def locationExists(point): # function that will return if a point exists in our
database yet
    try:
        result = coordinate.query.filter_by(lat=point[0], lng=point[1]).first()
        if (result):
            return result.cid # the coordinate has been analyzed and is in DB
        else:
            return None
    except:
        return None
```

- Another function called 'getDetection' is responsible for returning a list of detections with details that are associated with a given coordinate Id ('Cid').

```
def getDetections(cid): # function that will get every detection for a given CID
in the database
    try:
        detections = []
        results = detection.query.filter_by(cid=cid)
        for result in results:
            new_result = {
                "did": result.did,
                "class_name": result.class_name,
                "lat": result.lat,
                "lng": result.lng,
                "conf": result.conf,
                "text_read": result.text_read,
                "image_url": None
            }
            detections.append(new_result)
        return detections
    except Exception as e:
        logger.debug(e)
```

- The function called 'WriteCoordinate' generates a random 'Cid', which must be unique, to add a new coordinate in the coordinate database.

```
def writeCoordinate(point): # function taht will write a new coordinate to the
database
    min = 1
```

```
    max = 100000000
    rand = randint(min, max)
    while detection.query.filter_by(did=rand).limit(1).first() is not None:
        rand = randint(min, max)
    # now rand is our id
    new_coord = coordinate(rand, point[0], point[1])
    db.session.add(new_coord)
    db.session.commit()
    return rand
```

- A similar one ('writeDetection') uses the 'Cid' from before and new data to generate a new random 'Did', also unique, and creates a new detection to the detection table.

```
def writeDetection(data, cid): # function that will write a new detection to the
database
    min = 1
    max = 100000000
    rand = randint(min, max)
    while detection.query.filter_by(did=rand).limit(1).first() is not None:
        rand = randint(min, max)
    # now rand is our id
    new_detect = detection(rand, cid, data['lat'], data['lng'],
data['class_name'], data['conf'], data['text_read'], data['image_url'])
    db.session.add(new_detect)
    db.session.commit()
    data['did'] = rand
    data['image_url'] = None
    return data
```

- 'readDetection' fetches detailed information for a given 'did'. Before, it retrieves the image from 'image_url' and encodes it in base64.

```
def readDetection(did): # function that returns detailed information for a DID in
the database
    try:
        results = detection.query.filter_by(did=did)
        for result in results:
            to_query = result.image_url
            image_data = requests.get(to_query).content
            base64_string = base64.b64encode(image_data)
            im = base64_string.decode('utf-8')
            new_result = {
                "did": result.did,
                "class_name": result.class_name,
                "lat": result.lat,
```

```
                "lng": result.lng,
                "conf": result.conf,
                "text_read": result.text_read,
                "image_data": im
            }
            return new_result
    except Exception as e:
        logger.debug(e)
```

### 5.2.2.3 Detail API Module:

The Detail API is the module necessary to allow the requester to input a detection id and receive a response containing detailed information about that detection. It uses Flask-RESTful to create API resources and to parse request arguments. The way this module works is:

1. It receives a detection Id (‘Did’), latitude (‘lat’), and a longitude (‘lng’) as parameters.
2. Uses ‘readDetection’ from the module before, to get detailed information about the detection from the database.
3. Uses the Google Maps Geocoding API to get the address associated with those coordinates.
4. The obtained addresses are returned along with the detailed detection.
5. In case there is any error, it returns a message with status code 500, which is related with Internal Server Error.

```
class DetailAPI(Resource):
    def post(self): # endpoint delivers detailed information to the caller
        try:
            new_parser = reqparse.RequestParser()
            new_parser.add_argument('did', required=True, help="DID may not be
blank...").add_argument('lat', required=True, help="lat may not be
blank...").add_argument('lng', required=True, help="lng may not be blank...")
            args = new_parser.parse_args()
            did = int(args['did'])
            lat = args['lat']
            lng = args['lng']
            result = readDetection(did)
```

```
            response =
requests.get(f"https://maps.googleapis.com/maps/api/geocode/json?latlng={lat},{ln
g}&key={config.map_api_key}")
            result['address'] =
response.json()['results'][0]['formatted_address']
            return result
        except Exception as e:
            logger.debug(e)
            return "Error with your request...", 500
```

Below, there is a table called Table 7, showing the backend system from the outside perspective, where the Detailed endpoint can be seen more visually.

| Method | Description | Request Body | Response |
|---|---|---|---|
| POST | Get information about the detection ID | Detection ID | Detailed detection information |

*Table 7- Detailed endpoint information.*

#### 5.2.2.4 OSM Module:

This module is responsible for loading extract data from Open Street Maps and formatting it to the specifications. It utilizes an external API to get the street information and format the response into a list of streets.

The first function (‘fits’) is responsible for checking is a given point (latitude and longitude) is within the specified bounding box. In case it is, it returns ‘True’, otherwise ‘False’.

```
def fits(point, b_box): # helper function for staying within radial bounds
    if isinstance(point[0], list) or isinstance(point[1], list):
        return False
    if point[0] < b_box[0]:
        return False
    if point[0] > b_box[2]:
        return False
    if point[1] > b_box[1]:
        return False
    if point[1] < b_box[3]:
```

```
            return False
    return True
```

The ‘map_geo_data’ is a function that processes raw geographical data and orders them by street names, filtering out points outside of the bounding box thanks to the function previously created. It also filters specific highway types such as bus stops, motorways, or pedestrian paths. With the remaining coordinates, it calculates pairwise distances between them to find which points are at the end of the street. For each coordinate, it finds the maximum distance to any other point, and it focuses on the highest maximum distance in order to find the ‘source point’.

Then it orders the remaining streets by their coordinates using the distances calculated before.

Example:

There are three coordinates that are being studied.

- Point A at (1,1)
- Point B at (1,4)
- Point C at (4,1)

The calculated distances are:

- A to B = $\sqrt{(1-1)^2 + (4-1)^2} = 3$
- A to C = $\sqrt{(4-1)^2 + (1-1)^2} = 3$
- B to C = $\sqrt{(4-1)^2 + (4-1)^2} \approx 4{,}2426$

From these results, it can be seen that the maximum distances are:

- For A is 3
- For B is 4,2426

- For C is 4,2426

Since B and C have the largest maximum distances, one of them will be chosen as the source point.

After choosing this source point, the function orders the points by connecting each point to the nearest neighbor, starting from the previously chosen point.

```
def map_geo_data(old_data, b_box):
    new_data = {} # we are going to index by street
    for point in old_data:
        street_name = point.get("properties").get("name")
        if point.get("properties").get("highway") == "bus_stop" or
point.get("properties").get("highway") == "motorway" or
point.get("properties").get("highway") == "pedestrian":
            continue
        if street_name is None:
            continue
        if new_data.get(street_name) is None: # first time seeing street
            if point.get("geometry").get("coordinates")[0] is not None:
                new_data[street_name] = []
                for coordinate in point.get("geometry").get("coordinates")[0]:
                    if fits(coordinate, b_box):
                        new_data[street_name].append(coordinate)
        else: # here we have to append to existing street info
            temp_list = new_data.get(street_name)
            if point.get("geometry").get("coordinates")[0] is not None:
                for coordinate in point.get("geometry").get("coordinates")[0]:
                    if fits(coordinate, b_box):
                        temp_list.append(coordinate)
            new_data[street_name] = temp_list
    # we have street level data, place coordinates in the correct order:
    street_source = {}
    street_coord_list = {}
    for street in new_data:
        coordinate_list = new_data[street]
        if len(coordinate_list) == 0:
            continue
        # EVALUATE EVERY DISTANCE
        dist_mat = []
        for coordinate_pair in coordinate_list:
            dist_vec = []
            for inner_pair in coordinate_list:
                dist_vec.append(math.sqrt(math.pow(coordinate_pair[0]-
inner_pair[0],2) + math.pow(coordinate_pair[1]-inner_pair[1],2)))
            dist_mat.append(dist_vec)
        # USE DIST_MAT MAX INDICES AS SOURCES
        max_list = []
```

```
        for idx, _ in enumerate(coordinate_list):
            dist_row = dist_mat[idx]
            max_list.append(max(dist_row))
        source_idx = max_list.index(max(max_list))
        street_source[street] = (source_idx, coordinate_list[source_idx])
        street_coord_in_order = []
        source = street_source[street][0]
        count = 0
        while True:
            if count > 200: # should never be here
                break
            count += 1
            street_coord_in_order.append(coordinate_list[source])
            coord = coordinate_list[source]
            coordinate_list.pop(source)
            d = []
            if(len(coordinate_list) == 0):
                break
            for pair in coordinate_list:
                d.append(math.sqrt(math.pow(pair[0]-coord[0],2) +
math.pow(pair[1]-coord[1],2)))
            source = d.index(min(d))
        street_coord_list[street] = street_coord_in_order

    return street_coord_list
```

The last function on this module is 'query_osm'. It sends a query to the Open Street Map API to get the geographical data within a specified radius around a given location. It calculates the bounding box based on the given radius and sends a request to OSM to get information about the streets within that box. Then it utilizes the function 'map_geo_data' to clean and organize the data that got returned. This way it eliminates unnecessary points and orders the remaining coordinates and sends them in a usable format.

```
def query_osm(lat, lng, rad):
    degpermile_lat = 1 / 69.172 # conventional conversion rate of lat to miles
    degpermile_lng = 1 / (69.172 * math.cos(math.radians(lat))) # conventional
conversion rate of lng to miles given lat
    off_lng = degpermile_lng * rad
    off_lat = degpermile_lat * rad
    bottom = lat - off_lat
    top = lat + off_lat
    left = lng - off_lng
    right = lng + off_lng
    b_box = [left, top, right, bottom]
    bbox = str(left) + ',' + str(bottom) + ',' + str(right) + ',' + str(top)
```

```
    geo_data_params = {
        "api_key": config.osm_extract_key,
        "bbox": bbox,
        "tags": "highway=*"
    }
    # this returned function will map the response to something more usable by
our ParkAPI
    return map_geo_data(requests.get(config.osm_extract_http,
params=geo_data_params, verify=False).json().get("features"), b_box)
```

#### 5.2.2.5 Machine Learning Module:

This module is the main section of the application since it holds the ability to detect parking meters and traffic signs. It uses a pre-trained PyTorch model to scan images within a specific area, identifying relevant parking regulations and meters. It is capable of loading the model, processing the images and detecting the objects wanted.

The first function is called 'checkLatLng' it is used to check if the latitude and longitude provided are inside a specific radius. In order to do this, it calculates the distance between the latitude and the longitude with the center. Then it converts that distance into miles and calculates the total distance using the Pythagorean theorem. It compares the last result with the given radius.

Example:

- Center Point: (40, -74)
- Point to check: (40.1, -73.9)
- Radius: 10 miles

First it calculates the difference between both points:

- Difference in latitude: lat – center_lat = 40,1 – 40 = 0,1
- Difference in longitude: lng – center_lng = -73,9 – (- 74) = 0,1

Later it converts the latitude and longitude differences to miles for easier calculations. The rates used are 1 degree of latitude is approximately 69 miles, while 1 degree of longitude is approximately 69 miles * cos (latitude in radians).

- Latitude difference in miles: 0,1 * 69 = 6,9 miles
- Longitude difference in miles: 0,1 * 69 * cos(latitude) = 6,9 miles (For simplicity it is assumed that cos (latitude) is 1)

Calculate the total distance: $\sqrt{6{,}9^2 + 6{,}9^2} \approx 9{,}758$ miles.

Since that distance is less than the radius, it has been proven that the desired coordinated are inside the bounded box.

```
def checkLatLng(lat, lng, center_lat, center_lng, radius):
    degpermile_lat = 1 / 69.172 # conventional conversion rate of lat to miles
    degpermile_lng = 1 / (69.172 * math.cos(math.radians(center_lat))) #
conventional conversion rate of lng to miles given lat
    # convert lat, lng to normalized lat, lng
    lat = lat - center_lat
    lng = lng - center_lng
    # now convert to miles
    lat_miles = lat * (1/degpermile_lat)
    lng_miles = lng * (1/degpermile_lng)
    # now check the miles against radius
    if math.sqrt(lat_miles * lat_miles + lng_miles * lng_miles) < radius:
        return True
    return False
```

Another function called ‘generate_base_heading’ is responsible for calculating the degrees (direction) based on the difference between dy and dx coordinates.

```
def generate_base_heading(dy, dx):
    base_heading = math.atan2(dy, dx) * 180 / math.pi
    if base_heading < 0:
        base_heading = 360 + base_heading
    return base_heading
```

In order to make a request to the Google Street View API a function called 'run_query' is used. It sends a request with some coordinates and the heading. The response is an image, the URL used to get said image and the original input data.

```
def run_query(head_x_y):
    heading = head_x_y['head']
    x = head_x_y['x']
    y = head_x_y['y']
    size = "?size=640x640"
    pitch = "&pitch=0"
    fov = "&fov=80"
    api = "&key=" + config.map_api_key
    location = "&location=" + str(x) + "," + str(y)
    try:
        return
[Image.open(io.BytesIO(requests.get("https://maps.googleapis.com/maps/api/streetv
iew" + size + location + pitch + fov + "&heading=" + str(heading) +
api).content)), "https://maps.googleapis.com/maps/api/streetview" + size +
location + pitch + fov + "&heading=" + str(heading) + api, head_x_y]
    except:
        return None
```

The main function however is called 'run_model' which processes street coordinates to detect and classify desired objects, traffic signs and parking meters, within a specified radius around a center location. In order to do this, it follows some steps:

1) Initializes a dictionary to store the coordinates and detections for each street.

```
def run_model(street_coord_list, center_lat, center_lng, radius):
    locations = {}
    for street in street_coord_list:
        locations[street] = {
            "coordinates": [],
            "detections": []
        }
```

2) It utilizes the 'street_coord_list' that was given in the input to calculate the direction (North, East, West or South) and divide the segments into smaller steps for a more accurate result.

```
        for idx in range(len(street_coord_list[street])): # iterate through each
coordinate in the street segment
            if idx != len(street_coord_list[street]) - 1:
                xi = street_coord_list[street][idx][0]
                xf = street_coord_list[street][idx + 1][0]
                yi = street_coord_list[street][idx][1]
                yf = street_coord_list[street][idx + 1][1]
                dy = yf - yi
                dx = xf - xi
                base_heading = generate_base_heading(dy, dx)
                headings = [base_heading + 45]
                for _ in range(3):
                    base_heading = (base_heading + 90) % 360
                    headings.append(base_heading)
                d = math.sqrt((xf - xi) ** 2 + (yf - yi) ** 2)
                steps = int(d * 250)
                steps = steps + 1
                dx = (xf - xi) / steps
                dy = (yf - yi) / steps
                for count in range(steps + 1):
                    y = xi + count * dx
                    x = yi + count * dy
                    logger.debug([x, y])
                    locations[street]["coordinates"].append([x, y])
```

3) With the function previously created called 'generate_base_heading', it calculates the heading for each small segment.
4) Before analyzing each coordinate, it checks if the information related to that location is already stored in the database. If it does, it retrieves the existing detections, otherwise, it writes the new coordinates in the database.

```
                    cid = locationExists([x, y])
                    if cid is not None:
                        try:
                            detections = getDetections(cid)
                            for detection in detections:
                                temp_list = locations[street]["detections"]
                                already_detected = False
                                for item in temp_list:
                                    if item == detection:
                                        already_detected = True
                                if not already_detected:
                                    if detection['conf'] > .75 and
checkLatLng(detection['lat'], detection['lng'], center_lat, center_lng, radius):
                                        locations[street]["detections"].append(de
tection)
```

```
                        except Exception as e:
                            logger.debug(e)
                    else:
                        new_cid = writeCoordinate([x, y])
                        count = count + 1
```

5) For each new coordinate, it fetches images from Google Street View at different headings using 'ThreadPool'.

```
                    heading_x_ys = []
                    for heading in headings:
                        heading_x_y = {
                            "head": heading,
                            "x": x,
                            "y": y
                        }
                        heading_x_ys.append(heading_x_y)
                    img = []
                    with ThreadPool() as pool:
                        for result in pool.map(run_query, heading_x_ys):
                            img.append(result)
```

6) Those images are then processed and run by the prediction model which detect the objects wanted. When an object is detected, it calculates its position and stores the detection information along with their coordinates.

```
for im in img:
                    if im is None:
                        continue
                    results = model.predict(im[0])
                    result = results[0]
                    if len(result.boxes):
                        logger.debug("Image Analyzed - Meter Found")
                        classifier = ""
                        conf = 0
                        box_info = None
                        for box in result.boxes: # iterate through
detections
                            if box.conf[0].item() > conf:
                                conf = box.conf[0].item()
                                classifier =
result.names[box.cls[0].item()]
                                box_info = box.xyxy.data[0]
                                norm_info = box.xyxyn.data[0]
                                size_info = box.xywhn.data[0]
```

```
                                # we use x (lat) and y (lng) + heading
(im[2]['head']) to guess real placement of these objects
                                w = .00020 # some coordinate offset
                                norm_info = norm_info.numpy()
                                size_info = size_info.numpy()
                                x_norm_disp = ((norm_info[0] + norm_info[2]) /
2.0) - .5
                                norm_size = size_info[3] * size_info[2]
                                guessed_lat = x + w *
math.cos(math.radians(im[2]['head']) + x_norm_disp * 1.5)
                                guessed_lng = y + w *
math.sin(math.radians(im[2]['head']) + x_norm_disp * 1.5)
                                temp = {
                                    "class_name": classifier,
                                    "lat": guessed_lat,
                                    "lng": guessed_lng,
                                    "conf": conf,
                                    "image_url": im[1],
                                    "text_read": None
                                }
```

7) When the object detected is a traffic sign, the image is processed with OCR (Optical Character Recognition) to read any text on the sign.

```
                                if classifier == "Road Sign" and conf > .75:
                                    buffered = io.BytesIO()
                                    left = box_info.data[0].item() - 30
                                    if left < 0:
                                        left = 0
                                    right = box_info.data[2].item() + 30
                                    if right > 640:
                                        right = 640
                                    top = box_info.data[1].item() + 30
                                    if top > 640:
                                        top = 640
                                    bottom = box_info.data[3].item() - 30
                                    if bottom < 0:
                                        bottom = 0
                                    cropped_im = im[0].crop((left, 140, right,
500)) # anything below 500 will read google and block anyways..
                                    cropped_im.save(buffered, format="JPEG")
                                    img_str = buffered.getvalue()
                                    text_read = detect_text(img_str)
                                    temp['text_read'] = text_read
```

8) Among all the detections, the ones with a high confidence and within the radius specified by the user are stored.

```
                                if conf > .75 and checkLatLng(temp['lat'],
temp['lng'], center_lat, center_lng, radius): # only write if we're confident
                                    locations[street]["detections"].append(writeD
etection(temp, new_cid))
                            else:
                                logger.debug("Image Analyzed - Meter Not Found")
                                continue
```

9) Finally, it returns a final dictionary containing all the coordinates and detections.

```
    return locations
```

### 5.2.2.6 Park API Module

This module is responsible for running the main algorithm pipeline for analyzing parking information in a given area. This module calls functions from the Open Street Map (OMS) and Machine Learning Modules in order to complete its workflow.

The module´s first action is related to logging configuration. It configures it to show messages of debug level as well as creating a logger with the name of the current module.

```
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)
```

It initializes a parser for processing incoming JSON requests and adds the address and radius as required arguments since it would be inefficient to send a query when one of the main arguments is missing.

```
parser = reqparse.RequestParser() # to parse JSON request
parser.add_argument('address', required=True, help="Address may not be blank...")
parser.add_argument('radius', required=True, help="Radius cannot be blank...")
```

The first function is called 'distance', and it is used in order to calculate the separation between two geographical points. To explain how this distance is computed, an example can be very useful.

```
def distance(lat1, lng1, lat2, lng2):
    """
    Calculate the distance between two points on the Earth's surface
    using the Haversine formula.
    """
    # Radius of the Earth in kilometers
    R = 6371.0

    # Convert latitude and longitude from degrees to radians
    lat1 = radians(lat1)
    lng1 = radians(lng1)
    lat2 = radians(lat2)
    lng2 = radians(lng2)

    # Calculate the change in coordinates
    dlng = lng2 - lng1
    dlat = lat2 - lat1

    # Calculate the distance using the Haversine formula
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlng / 2)**2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    distance = R * c

    print(distance)
    return distance
```

Example:

- Point 1: Lat = 0º; Lng = 0º
- Point 2: Lat = 1º; Lng = 1º

It starts by converting the degrees to radians.

- Latitude 1: 0º = 0 radians
- Longitude 1: 0º = 0 radians
- Latitude 2: 1º = 0,0174533 radians (1º * π /180)
- Longitude 2: 1º = 0,0174533 radians (1º * π /180)

The difference between both coordinates is needed.

- Difference_longitude = longitude 2 – longitude 1 = 0,0174533 radians
- Difference_latitude = latitude 2 – latitude 1 = 0,0174533 radians

With these results, the Haversine formula is applied [10]. It is known that R is the radius of the Earth in kilometers.

$$a = \sin^2\left(\frac{\Delta lat}{2}\right) + \cos(lat1) * \cos(lat2) * \sin^2\left(\frac{\Delta lng}{2}\right)$$

$$a = \sin^2\left(\frac{0{,}0174533}{2}\right) + \cos(0) * \cos(0{,}0174533) * \sin^2\left(\frac{0{,}0174533}{2}\right)$$

$$a = 0{,}0001523$$

$$c = 2 * atan2\left(\sqrt{a}, \sqrt{1-a}\right)$$

$$c = 2 * atan2\left(\sqrt{0{,}0001523}, \sqrt{1-0{,}0001523}\right)$$

$$c = 0{,}024682$$

$$distance = R * c$$

$$distance = 6371 * 0{,}024682$$

$$distance = 257{,}249\ km$$

This way it is calculated that the distance between both points is 257,249 kilometers. This function returns this distance.

Another function in this module is called 'cleanupLocations' and it is responsible for cleaning the detected locations by removing the duplicates. It starts by creating a list to keep track of all the detections that would be removed, then it iterates through the streets in the examined locations and ensures that there are no duplicates, in case there are, they are added to the list before created. It also removes those that are too close to each other. For reference, in order to be removed because of proximity, both points need to be closer than 3,5 meters. The response of this function is a list with all the cleaned-up locations.

```
def cleanupLocations(examined_locations):
    for street in examined_locations:
        # print(examined_locations[street])
        if street == 'center_lat' or street == 'center_lng' or street ==
'radius':
            continue
        # print(street)
        to_remove = []
        for i, detection1 in enumerate(examined_locations[street]['detections']):
            for other_street in examined_locations:
                if other_street == 'center_lat' or other_street == 'center_lng'
or other_street == 'radius':
                    continue
                # print(examined_locations[other_street])
                to_remove = []
                time_to_leave = False
                for j, detection2 in
enumerate(examined_locations[other_street]['detections']):
                    if j != i:
                        if distance(detection1['lat'], detection1['lng'],
detection2['lat'], detection2['lng']) < .0035:
                            print("REMOVED DIST")
                            to_remove.append(i)
                            time_to_leave = True
                            break
                if time_to_leave:
                    break # get a new iter
            for ind in to_remove:
                examined_locations[street]['detections'].pop(ind)
                for iter, val in enumerate(to_remove):
                    to_remove[iter] = val - 1
    return examined_locations
```

Lastly, in this module there is class created called 'ParkAPI'. It is a RESTful resource that handles HTTP POST requests. The method defined requires some arguments like radius and address, which are converted into de desired type of data, like float for the radius. After, it validates that the radius is between 0,01 miles and 0,25. The parameters are prepared to be sent and a request is made to the Google Geocoding API. From the response, it extracts the latitude and longitude and queries OpenStreetMap data around the location. On this queried data, the machine learning model is run. The results are cleaned up thanks to the function previously created. The processed locations are retuned. There are also exceptions catch in this function to ensure that nothing goes wrong.

```
class ParkAPI(Resource):
    def post(self):
        try:
            ######################################
            # BEGIN ERROR CHECKING OF PARAMETERS #
            ######################################
            args = parser.parse_args()
            radius = float(args['radius'])
            address = args['address']
            logging.info("Received request -- Address: " + address + "; Radius: "
+ str(radius))
            if radius is None:
                return {"Error": "Parameter Error: No radius supplied"}, 400
            if address is None:
                return {"Error": "Parameter Error: No address supplied"}, 400
            if radius < .01 or radius > .25:
                return "Parameter Error: radius should be between .01 and .25
miles", 500
            geocode_params = {
                "key": config.map_api_key,
                "address": address
            }
            response =
requests.get("https://maps.googleapis.com/maps/api/geocode/json",
params=geocode_params)
            lat =
response.json().get("results")[0].get("geometry").get("location").get("lat")
            long =
response.json().get("results")[0].get("geometry").get("location").get("lng")
            if long is None or lat is None:
                return "Parameter Error: Issue with locating address", 500
            ################## MAP DATA QUERY ##################
            street_coord_list = query_osm(lat, long, radius)
            ################## ML ######################
            examined_locations = run_model(street_coord_list, lat, long, radius)
```

```
            examined_locations['radius'] = radius
            examined_locations['center_lat'] = lat
            examined_locations['center_lng'] = long
            examined_locations = cleanupLocations(examined_locations)
            return examined_locations
        except Exception as e:
            logger.debug(e)
            return "Error with your request...", 500
```

Below, there is a table called Table 8, showing the backend system from the outside perspective, where the Park endpoint can be seen more visually.

| Method | Description | Request Body | Response |
|---|---|---|---|
| POST | Evaluate parking regulations in the area | Address and radius | Dictionary mapping streets to detections |

*Table 8- Park endpoint information.*

#### 5.2.2.7 Text Processing Module

This is a completely independent module responsible for implementing a single function in order to detect text in an image. It imports Google Cloud Vision API from the library Google Cloud SKD, and creates an instance of 'ImageAnnotatorClient', which is used to interact with this API.

The only function in this module is called 'detect_text', and it needs a binary argument, which is the image. It sends a request to the Vision API, if the length of the response is more than 0, then the text is stored, otherwise, the stored text will be 'Unable to read text from sign'.

```
from google.cloud import vision
client = vision.ImageAnnotatorClient()

def detect_text(content):
    """Detects text in the file."""
    image = vision.Image(content=content)
    response = client.text_detection(image=image)
    texts = response.text_annotations
```

```
    if len(texts) > 0:
        response = texts[0].description
    else:
        response = "Unable to read text from sign."
    return response
```

#### 5.2.2.8 WhereTo Module

In this module is where the constants for the application can be found. For instance, it imports Flask, which is used to create the web application; os, a module for interacting with the operating system; YOLO, to use the model created for object detection; flask_restful to create a RESTful APIs with Flask; and flask_sqlalchemy.

Then, it proceeds to initiate all the necessary instances of these previously mentioned imports.

```
from flask import Flask
import os
from ultralytics import YOLO
from flask_restful import Api
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
api = Api(app)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' +
os.path.join(os.path.abspath(os.path.dirname(__file__)),
'../database/whereto.db')
db = SQLAlchemy(app)
model = YOLO("model4.pt")
```

#### 5.2.2.9 Testing Module

This module is design to hold the functionality unit tests for each module. It is divided into two files: test_detail and test_park, which verify that certain HTTP methods are not allowed in the endpoint.

Both of them follow the same structure:

First of all, a unique client is created and will be used to simulate the requests without the need to run the server.

The methods that are tried are:

- GET request but with wrong arguments,
- PATCH request but with wrong arguments,
- DELETE request but with wrong arguments,
- PUT request but with wrong arguments.

All of them are meant to fail, and their response should be the error 405 (Method not allowed), since all the arguments are wrong.

```
import pytest
from flask.testing import FlaskClient
from app import app

@pytest.fixture
def client():
    return app.test_client()

# TESTING PROJECT API (ParkAPI)

def test_park_api_sad_paths(client: FlaskClient):
    resp = client.get('/park', json={'username': 'username'}) # this should fail
    assert resp.status_code == 405 # we cannot perform a get
    resp = client.patch('/park', json={'username': 'username'}) # this should
fail
    assert resp.status_code == 405 # we cannot perform a patch
    resp = client.delete('/park', json={'username': 'username'}) # this should
fail
    assert resp.status_code == 405 # we cannot perform a delete
    resp = client.put('/park', json={'username': 'username'}) # this should fail
    assert resp.status_code == 405 # we cannot perform a put
```

```
import pytest
from flask.testing import FlaskClient
from app import app

@pytest.fixture
def client():
    return app.test_client()
```

```
# TESTING PROJECT API (DetailAPI)

def test_detail_api_sad_paths(client: FlaskClient):
    resp = client.get('/detail', json={'username': 'username'}) # this should
fail
    assert resp.status_code == 405 # we cannot perform a get
    resp = client.patch('/detail', json={'username': 'username'}) # this should
fail
    assert resp.status_code == 405 # we cannot perform a patch
    resp = client.delete('/detail', json={'username': 'username'}) # this should
fail
    assert resp.status_code == 405 # we cannot perform a delete
    resp = client.put('/detail', json={'username': 'username'}) # this should
fail
    assert resp.status_code == 405 # we cannot perform a put
```

#### 5.2.2.10 App.py Module

This is the main driver module for the backend since it adds the resource APIs to the application and begins running it.

The application is run by using the app.py driver code. This main module depends directly on the Park API and Detail API, as well as the WhereTo core module constants. App.py adds both the Park and Detail APIs as resources, giving them endpoints on the server. App.py then begins to run the server if it is run as a script, rather than a module. It does this through importing both the api and app constants from the WhereTo core module.

It creates a login in configuration with the name of the current module that can be used to log messages throughout the code. Then, it launches the backend only if this module is called directly and not when it is imported form another one.

```
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

api.add_resource(ParkAPI, '/park')
api.add_resource(DetailAPI, '/detail')

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=7001, debug=True)
```

The acceptable methods for the endpoints can be seen in Table 7, for the Detail API, and Table 8 for the Park one.

### 5.2.3 FRONTEND DESIGN

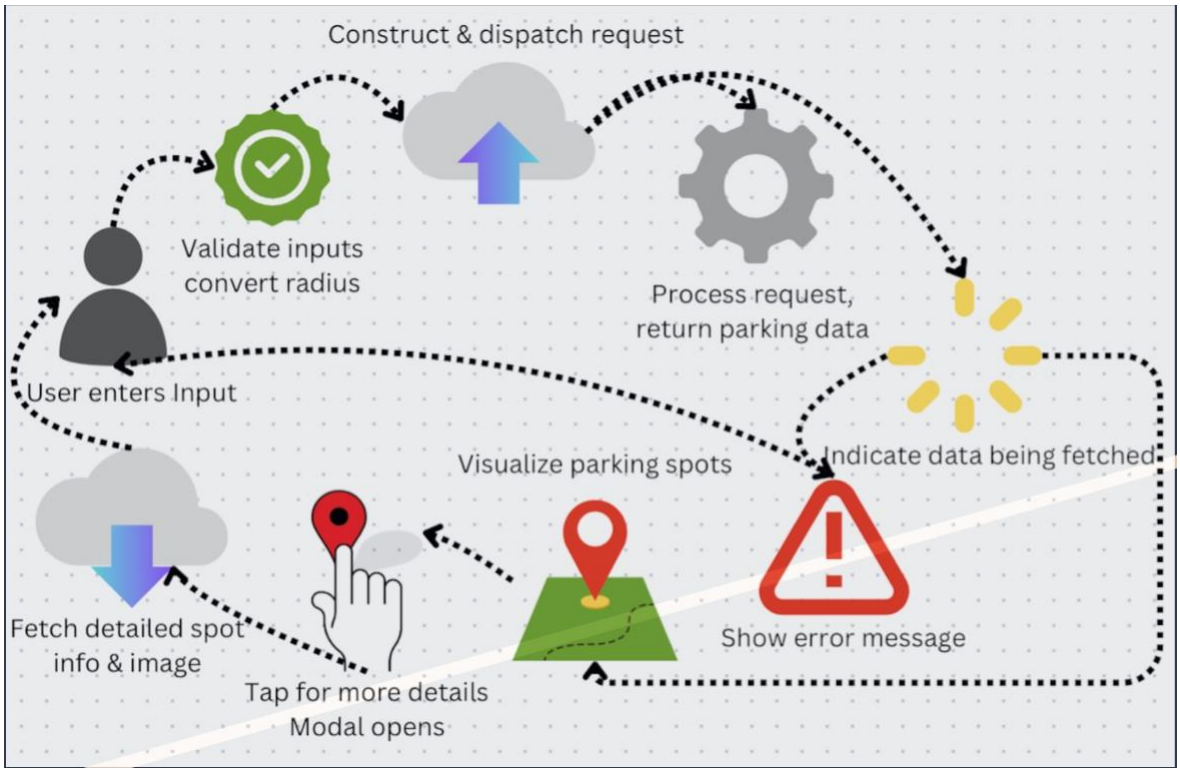



*Figure 8- Frontend Software overview.*

This diagram (Figure 8) shows the workflow of the frontend application designed to help users find and visualize parking information. Initially, the user inputs data such as address and search radius, which the application verifies is real and converts the data into the necessary format. Then a request is made and sent to the backend, which processes the query and sends back the requested information. The frontend indicates that data is being fetched and then it shows a map with the available parking information. Users can also tap on the different pins to open a modal window with detailed information about specific detection. If any issue occurs during the process, such as a request failure, an error message

is displayed to the user. This workflow ensures a smooth and informative user experience to find important parking related information.

The WhereTo application is structured into distinct modules and components. Here is a detailed breakdown of the frontend architecture.

### *5.2.3.1 Input Display Module*

In order to use the Google API, it is necessary to create a profile and get an API key. This is a constant that is going to be used in every request made.

This module is the first screen that the user can see, so it is important to create a button with all the information on how to use the app. The component that is responsible for this is called 'HelpModal' and provides instructions on how to use the app. It explains how to enter an address and radius to find parking information in that area. The modal can be opened and closed by pressing a button.

```
function HelpModal({ isVisible, onClose }) {
  return (
    <Modal
      animationType="slide"
      transparent={true}
      visible={isVisible}
      onRequestClose={onClose}>
      <View style={styles.centeredModalView}>
        <View style={styles.modalContent}>
          <Text style={styles.modalTitle}>How to Use This App</Text>
          <Text style={styles.modalText}>
            Enter an address and a radius to find parking spots near you.
          </Text>
          <Text style={styles.modalExample}>Example: 700 Commonwealth Ave,
Boston, MA 02215</Text>
          <Text style={styles.modalText}>
            Press the "Find Parking" button to view spots on the map.
          </Text>
          <Pressable
            style={[styles.button, styles.buttonClose]}
            onPress={onClose}>
            <Text style={styles.textStyle}>Close</Text>
          </Pressable>
        </View>
      </View>
    </Modal>
```

```
  );
}
```

The main function of these module is called ‘InputDisplay’. There are many states managed here since the ‘useState’ hook is being used for radius selection, loading status, error status, visibility of the modals, address input or autocomplete feature.

Inside this function, the ‘getCurrentLocation’ function can be found. This function’s purpose is to fetch the users address and set it as the address in the input. In order to do this, it requests location permissions, then fetches the current geographic coordinates, it reverse geocodes the coordinates into a human comprehensive address and sets the results in the first input.

```
  const getCurrentLocation = async () => {
    let { status } = await Location.requestForegroundPermissionsAsync();
    if (status !== 'granted') {
      console.error('Permission to access location was denied');
      return;
    }

    let location = await Location.getCurrentPositionAsync({});
    let reverseGeocode = await Location.reverseGeocodeAsync({ latitude:
location.coords.latitude, longitude: location.coords.longitude });
    if (reverseGeocode.length > 0) {
      const { street, city, region, postalCode } = reverseGeocode[0];
      const formattedAddress = `${street}, ${city}, ${region} ${postalCode}`;
      googlePlacesAutocompleteRef.current?.setAddressText(formattedAddress);
    }
  };
```

The function ‘findParking’ is responsible for sending the request to the backend server to find parking spots based on the input address and radius. Therefore, it needs to convert into the desired format these two inputs because, for example, the radius needs to be changed from meters to miles. The frontend architecture utilizes Axios for API interactions, constructing requests that are then dispatched to the backend's /park endpoint. It handles these interactions by managing the states of the requests and systematically presenting the user with UI feedback. During the request lifecycle, Axios plays a pivotal role in managing

state transitions that indicate data fetching progress and relaying any errors back to the user interface. This ensures that users are well-informed of the app's current state, enhancing the user experience. As expressed before, it handles most of the states like setting the 'isLoading' to true and, once the results are back, it is responsible for changing the screen the user is seen to the map one. It is an asynchronized function since it depends on how long the data takes to come back from the backend.

```
  async function findParking(address, radius) {
    setIsLoading(true);
    try {
      const params = {
        "address": address,
        "radius": parseInt(radius) * 0.000621371 // meter to mile convert
      };
      const result = await axios.post('http://192.168.4.97:8000/park', params);
      setResponseData(result.data);
      setIsLoading(false);
      setIsOnMap(true);
    } catch (error) {
      console.error(error);
      setIsLoading(false);
      setIsError(true);
      setIsOnMap(false);
    }
  }
```

The styles are also defined in this component, see Appendix II.

### *5.2.3.2 MapDisplay Module*

The main purpose of this module is to display a map for the user to see the map with all the detections and allow them to interact with the different pins. The core function is determined by two parameters: 'responseData', which is the response from the backend; and 'setIsOnMap', which is a function to switch back to the initial screen. It controls not only multiple states such as 'modalVisible', 'selectedInfo', 'isHelpModalVisible' or 'mapType', but also fetches for the details of a specific detection when the user is interested in them.

```
function MapDisplay({ responseData, setIsOnMap }) {
    const [modalVisible, setModalVisible] = useState(false);
    const [selectedInfo, setSelectedInfo] = useState(detailedInfo);
    const [isHelpModalVisible, setIsHelpModalVisible] = React.useState(false); //
State for help modal visibility
    const [mapType, setMapType] = useState('standard');
```

```
    const handleSelectMarker = async (did, lat, lng) => {
      // use the did to call the detail endpoint
      const params = {
        "did": did,
        "lat": lat,
        "lng": lng
      }
      result = await axios.post('http://192.168.4.97:8000/detail', params)
      let base64Image = ''
      base64Image = base64Image + result.data.image_data
      result.data.image_data = base64Image
      setSelectedInfo(result.data);
      setModalVisible(true);
    };
    return (
      <ImageBackground source={require('../assets/sample.jpeg')}
style={styles.backgroundImage} blurRadius={3}>
        <View style={styles.container}>
        <MapView
  style={styles.map}
  mapType={mapType}
  initialRegion={{
    latitude: responseData['center_lat'],
    longitude: responseData['center_lng'],
    latitudeDelta: 0.0122,
    longitudeDelta: 0.0042,
            }}>
            <Circle
              center={{ latitude: responseData['center_lat'], longitude:
responseData['center_lng'] }}
              radius={responseData['radius'] * 1609.34}
              strokeWidth={1}
              strokeColor={'#1a66ff'}
              fillColor={'rgba(230,238,255,0.5)'}
            />
            <Markers responseData={responseData} onSelect={handleSelectMarker} />
          </MapView>
          {/*<TouchableOpacity style={styles.backButton} onPress={() =>
setIsOnMap(false)}>
            <Text style={styles.backButtonText}>Back to Home</Text>
          </TouchableOpacity>*/}
          <MarkerInfoModal visible={modalVisible} onClose={() =>
setModalVisible(false)} info={selectedInfo} />
          {/* Help button */}
        <Pressable
          onPress={() => setIsHelpModalVisible(true)}
          style={styles.helpButton}>
          <Text style={styles.helpButtonText}>?</Text>
        </Pressable>
        <Pressable
  style={styles.mapTypeButton}
```

```
  onPress={() => setMapType(mapType === 'standard' ? 'satellite' : 'standard')}>
  <Text style={styles.mapTypeButtonText}>
    {mapType === 'standard' ? 'Satellite View' : 'Standard View'}
  </Text>
</Pressable>
        {/* Help modal */}
        <HelpModal
          isVisible={isHelpModalVisible}
          onClose={() => setIsHelpModalVisible(false)}
        />
        <Pressable onPress={() => setIsOnMap(false)} style={styles.homeButton}>
          <Icon name="home" size={30} color="#FFFFFF" />
        </Pressable>
        </View>
        </ImageBackground>
      );
    }
```

In order to do this, it uses other functions created previously such as 'MarketInfoModal'. In the whole application, interactivity is key; users can engage with the map markers to reveal comprehensive details of each parking location. When a marker is selected, a modal is triggered, displaying information such as the parking spot's address, the confidence level of the detection, and the associated image. The image handling mechanism within this module is particularly adept, converting base64-encoded strings back to binary format for image rendering, thereby streamlining the data transfer process. This efficient encoding strategy allows the embedding of image data directly within API responses, reducing the need for additional HTTP requests.

In terms of visual options, the MapDisplay module provides a feature to toggle between standard and satellite views of the map, enhancing user experience by offering diverse visual representations of the terrain. This function exemplifies the module's versatile presentation capabilities, accommodating user preferences for map displays. Another function is called 'HelpModal' and provides users with guidance on how to navigate the map and utilize the application effectively. Aims to enhance user understanding and ensure a smoother user experience by clarifying application functionalities. The function called 'CustomMarker' is used to differentiate between types of detections (e.g., parking meters vs. road signs) by using distinct icons, making it visually easy for users to identify the type of

information provided. For example, a traffic sign will be show with a pin like the one in Figure 10, while a parking meter will be show with Figure 9.

*Figure 10- Parking Sign pin [11].*

*Figure 9- Parking Meter pin [12].*

### 5.2.3 DETECTION MODEL

The purpose of the model is to identify relevant objects on images taken from the Google Street View. In order to do this, it is used the YOLOv8 framework by Ultralytics.

The YOLOv8 model is utilized for its ability to detect multiple objects efficiently within a single model stream. This approach enhances efficiency compared to running separate models for each class. The focus is primarily on detecting two classes:

1. Parking Meters: distinguishing between single and multiple parking meters,
2. Road Signs.

For training the model it was necessary to examine an extensive dataset of approximately 800 images containing parking meters and road signs. These images where taken from Google Street View to ensure the relevance for the real-world. In order to generalize the whole model across all United States. The main 4 cities where the images were taken where Boston, Charlotte, Los Angeles and New York. This diversity is essential

due to the fact that the different parking signs and meters vary from state to states as it can be seen in Figure 11.


*Figure 11- Different signs of 'No Stopping Anytime'.*

Each one of these images was analyzed before classifying it. This consisted of meticulously annotate them with bounding boxes and labels for the different types of detection. This is done in order to get a higher accuracy on the objects of interests since it is only considered the area of interest.

The advantages for this type of approach are:

- Efficiency: Using a single model able to detect different objects reduces computational overhead and simplifies the detection model.
- Relevance: The use of real-world data from Google Street View which encompasses all the United States. This ensures that the model is well-suited to the practical applications and prepared for WhereTo.
- Accuracy: The model was trained with nearly 800 images and the response contains only the images that have a higher confidence score. This means that if the confidence is less than a 0,75 the map will not display that detection.
- Optimized Mapping: The integration of the street transversal algorithm and OpenStreetMap data ensures efficiency and accurate mapping.

## *5.3 TESTING*

### 5.3.1 UNIT TESTING

This section outlines the unit test designed to validate the functionality of each endpoint. It is organized in two files: 'test_destail' and 'test_park', which ensure that certain HTTP methods are not permitted. As it was stated on 5.2.2.9, the methods tested where GET, PATCH, DELETE and PUT. All of them were expected to fail and return an error 405 (Method Not Allowed), given that the arguments provided are incorrect.

### 5.3.2 SYSTEM TESTING

#### *5.3.2.1 First testing*

**Equipment:**

Hardware:

- A personal computer

Software:

- Python3 Flask API
    - Image gathering
        - Google Street View Static API
    - Image text processing
        - Google Cloud Vision API
- Web Browser
    - Accessing locally hosted API via URL

**Testing Procedure:**

The test was done entirely with software. The only hardware that was required was a personal computer on which the code was run. The Python API script was launched locally

in a virtual environment in order to access the endpoint using a web browser on the same machine. This allowed to run the Python API without having to host it on the cloud or some other means. The API was designed to take as inputs two coordinates: an initial and a final coordinate. It progresses between these two coordinates and takes a set of pictures by querying Google's Street View Static API every few meters. The API is designed to take 8 images at each location, at intervals of 45 degrees. After collecting the images between the two supplied coordinates, the images are fed into Google's Cloud Vision API in order to have any text present within the images be read and output locally to a JSON formatted file.

**Measurements:**

The following is what is considered to be the measurable criteria for this prototype test:

1. Whether the Python API proves accessible locally, verified via the health endpoint.
2. Whether the Python API captures and stores into a specified directory sets of images between the specified/supplied coordinates. These images should be in sets of 8, as they are taken every 45 degrees for an entire rotation.
3. Whether the Python API outputs a JSON formatted file containing the results of processing the images gathered between the two coordinates through Google Cloud Vision Text analysis.
    a. For every image processed between the provided coordinates, the JSON file will have a distinct section detailing the text extracted from that specific image.

Here are the **results** from the testing in regard to these criteria:

1. The Python API did prove accessible using the health endpoint.
2. The Python API successfully captured the images expected between the two supplied coordinates. It was also successful in that for each location, a set of 8 pictures taken at each different angle was provided by querying the Google Street View Static API.
3. The Python API successfully outputs the expected JSON file. This file contained the text information that the Google Cloud Vision text analysis tool API was able to see.

While this step was successful, the text was read from all parts of the image, not just parts relevant to parking signage.

#### *5.3.2.2 Second Testing*

**Required Equipment:**

Hardware:

- Personal Computer
  - To host both the backend of the application, as well as the database
- Personal Smartphone
  - To experience the frontend user interface of the application

Software:

- Python Park API
  - This is hosted on the machine, makes use of ML models as well as algorithmic design and API calls in order to analyze parking information for the user.
- React Native UI
  - This is hosted on the machine, React code to display the user interface on mobile. This interacts directly with the backend via an API call.
- Expo Go
  - A mobile application which is designed for testing react native applications on iOS or Android devices.
- SQLite Database
  - This is a cache, to reduce excessive calls to the Google Services APIs

**Pre-Testing Setup Procedure:**

In order to set up the test correctly, every component of the software must be correctly configured as well as actively running. Because of this, each component has its own setup that must be performed prior to attempting the test:

ParkAPI Component:

1. Ensure the backend repository is in the expected location on the personal computer being used for testing.
2. Ensure that all dependencies for the backend are installed via pip.
3. Ensure that the correct configuration variables are set in the config.py file. This is vital to allow for both use of the OSM Extract API as well as the Google Maps API.
4. Run the application by navigating to the root directory of the API in terminal and executing: 'python3 app.py'
    - This will run the application on every open host possible, so it is accessible through the React Native Application

React Native UI Component:

1. Ensure that the smartphone being used for testing has the Expo Go application installed, this is necessary to run the application prototype.
2. Once the ParkAPI is hosted (in the above steps for ParkAPI Component) it will output its URL to the terminal. One must ensure this URL is the URL that the React application attempts to access via axios.
3. Run the application by executing 'npx expo start'.
    a. This will output a QR code to the terminal, scan this and open Expo Go to download the application and begin testing.

Database Component:

1. An empty database is needed for the start of the test, so any existing db file should be deleted and a new empty database should be created using sqlite commands.
    - This database should have the schema present in the .sql file in the backend repository. This schema must be able to store two tables: coordinates being looked at and detections made.

**Testing Procedure:**

The testing procedure for the application was as follows:

1. Load the application by scanning the QR code outputted from the 'npx expo start' command.
2. For each address and radius combination for which it is wanted to examine:
    a. On the InputDisplay screen for the UI, enter a valid address string and radius value.
    b. Press the "Find Parking" button on the UI.
    c. Wait for the backend to finish completing the request.
    d. Examine the UI/map output present after the request is done processing.
3. Additionally, it should be performing each combination of inputs twice, to ensure that proper caching is being performed; that is, it is not performing duplicate calls to the Google API services if it has analyzed them previously/recently.

During the test, the testing plan was followed precisely. Example inputs were processed twice each to demonstrate the capability to locate relevant parking information in various cities and to cache significant results. Additionally, an example address and radius pair for a CVS in Newton, MA, was tested as a Professor Osama's request. This example successfully detected the parking meters at the specified location with reasonable accuracy.

**Measurements Taken:**

The measurable criteria for the Python API was as follows:

1. The Python API should be capable of analyzing the streets in a given area, this is validated by a local file in the map/ directory which displays markers on every analyzed coordinate.
2. The Python API should be capable of using the pretrained model to make and deliver predictions with a confidence level of possible parking meters in the area as a JSON response. This is also verified partially in the map/ directory, where markers displaying detections are placed.
3. The Python API should return an error in the case of an invalid parameter and should be capable of receiving multiple requests in a row and processing them accurately to the requirements of the application.

The measurable criteria for the React Native UI was as follows:

1. The React Native UI should allow for the user to input a radius and address that they desire.
2. The React Native UI should be capable of graphically displaying the JSON response in the case of a successful API procedure.
3. The React Native UI should be capable of handling and informing the user of an error in the case of an unsuccessful API procedure.

The measurable criteria for the SQLite Database was as follows:

1. The SQLite Database should be used instead of a duplicate call to the Google API services and machine learning model if we have examined a coordinate before, this is going to be displayed/shown through API logging.

**Results:**

Overall, this second test was a very successful one for the design. It performed as it was hoped and expected that it would during the test.

The Python Park API was capable of analyzing streets within the given radius and was able to forward the detections it saw to the frontend as well as store the detections inside of the database. When it was fed incorrect parameters to the API, it was able to return an error to our frontend. It could be determined that the caching was working with the SQLite database as the second time it was ran any given input, it completed the request much faster than the original time. This would be due to less calls to the Google API and AI components, as it was using the database for the information.

The React Native UI was also successful in all of the criteria we demanded of it during the test. It allowed for user input and was able to correctly handle both an error and success response from the backend API. The UI was also able to visually graph the locations of detected parking meters on a map for the user.

#### *5.3.2.3 Final Testing*

**Required Materials**

Hardware:

- Personal Computer
    - To host the React Native Application as well as interact with our Google Cloud Compute instance via SSH.
- Personal Smartphone (iOS or Android)
    - To experience with the user interface of WhereTo via Expo Go.

Software:

- Python ParkAPI

  - API that makes use of ML models as well as algorithmic design and API calls in order to analyze parking information for the user.
- Python DetailAPI
  - API that makes use of our cache as well as Google Services to forward more detailed information about detections to the user interface.
- React Native UI
  - React code to display our user interface on mobile.
- Expo Go
  - A mobile application designed for testing React Native applications on iOS.
- SQL Database
  - The cache, used to reduce excessive calls to the Google Services APIs.

**Set-Up**

This test requires a computer and a smartphone with Expo Go installed. The Python API as well as SQL server are hosted on Google Cloud, and they can be started through accessing Google Cloud Services through a personal computer via SSH. The React Native application is to be hosted on the personal computer. The smartphone is then used to download the React Native application and begin testing the backend. Once configuration is complete, the backend should be accessible from the frontend seamlessly to execute testing.

**Pre-Testing Setup Procedure**

ParkAPI and DetailAPI:

1. Ensure that the repository is initialized and downloaded to the VM on Google Cloud Compute Engine.
2. Ensure that relevant dependencies are installed on the file system where we are running the API.
3. Ensure that the necessary API keys and URL configurations are present in the config.py file in the root of the application on Google Cloud.

4. Run the application on port 8080 by navigating to the root directory of the API in terminal and executing: 'python3 app.py'
    a. This will run the application on the port 8080, which can be accessed by the external IP address listed in Google Cloud for the application.

React Native UI:

1. Ensure the smartphone has the Expo Go application installed.
2. Ensure that the React Native code base url is the external IP address from Google Cloud through which we can access the Python API.
3. Ensure that relevant dependencies are installed to the machine through the use of the 'npm i' command.
4. Run the application by executing: 'npx expo start'.
    a. This will output a QR code to the terminal, scan this and open Expo Go to download and run the application.

Database:

1. Nothing must be done for the database; it is connected directly to the ParkAPI. If there is no whereto.db file present within the API/database folder, use the queries in API/database/whereto.sql to generate the database tables for population.

**Testing Procedure**

1. For each address and radius combination for which we want to examine:
    a. On the Input display screen for the UI, enter a valid address string and radius value.
    b. Press the "Find Parking" button on the UI.
    c. Wait for the backend to finish completing the request.
    d. Examine the Map display output present after the request is done processing.
        i. Ensure that both parking meters as well as road signs are being detected by the system and placed correctly.

ii. Ensure that the marker pins create a modal with image and detailed information upon tapping, with reasonable information presented.

2. Additional testing actions
    a. Ensure that the help modal is functioning appropriately on both the Map and Input displays.
    b. Ensure that error messages can be handled appropriately on the frontend.
    c. Ensure that the find by location feature is able to autofill the general location of the user.
    d. Ensure that the autocomplete for addresses is working correctly.

**Measurable Criteria**

The measurable criteria for the Python API is as follows:

1. The ParkAPI should be capable of analyzing the streets in a given area, this is validated by DEBUG logs within the API, which will print out every examined coordinate. This is also validated by the JSON response from the server to the React Native application.
2. The Python API should be capable of using the pretrained model to make and deliver predictions with an associated confidence level of possible parking meters and road signs in the area as a JSON response. This is verified through API DEBUG logs as well.
    a. For a given road sign detection, the ParkAPI must attempt to read the text off of the sign. This is verified by the setting of the text_read parameter in the JSON response to any value besides null. If no text is able to be read from the sign, that information should be forwarded in the text_read field.
    b. For any given detection, the ParkAPI must attempt to place the detection in an appropriate coordinate location, a prediction generated from the location and heading of the photo, as well as the relative size of placement of the detection within the photo.

3. The Python API should return an error in the case of an invalid parameter and should be capable of receiving multiple requests in a row and processing them without issue.

The measurable criteria for the React Native UI is as follows:

1. The React Native UI should allow for the user to input a radius and address that they desire.
2. The React Native UI should be capable of graphically displaying the JSON response in the case of a successful API procedure.
3. The React Native UI should be capable of displaying a detailed information modal for each individual detection upon clicking on the marker.
    a. For a parking meter, this should include a photograph, a confidence score, detection type, and the address of the meter.
    b. For a road sign, this should include a photograph, a confidence score, detection type, and the text read from the road sign, or an indication the text could not be read.
4. The React Native UI should be capable of handling and informing the user of an error in the case of an unsuccessful API procedure.
5. The React Native UI should have a "help" functionality on both the input display and map display screens. This can be verified through attempting to use the help button on the top right of the screen during testing, ensuring the modal is displayed correctly.
6. The React Native UI should be able to autofill a general location to the address input field. This will be verified by tapping the button to use current location and verifying that the correct location is entered into the address input.

The measurable criteria for the SQL Database is as follows:

1. The SQL Database should be used instead of a duplicate call to the Google API services and machine learning model if we have examined a coordinate before, this is going to be displayed/shown through API logging.

2. The SQL Database should be successfully integrated with the DetailAPI. This will be verified by the end-to-end functionality of the DetailAPI, as it relies entirely on the cache.

**Results**

This final test was very successful. The design performed exactly as it was hoped and expected it would during the testing procedure. Each component of the design passed the measurable criteria that it was expected of them.

The Python Park API was capable of grabbing geographic information regarding streets within the given radius. Using this information, it was able to progress down the street, gathering Google Street View image data and analyzing it to produce detections of both road signs and parking meters. If these locations were new to the system, it would save these results in the cache for later retrieval. It was possible to say that this caching system was working, as the second time it was run any input during the testing it was sent to the frontend very fast, much faster than possible if it were to run the machine learning algorithm a second time. The Python Park API was also capable of handling errors in input, this includes radii that differed from anticipated values as well as incorrect street addresses.

The Python Detail API was capable of returning detailed information from the cache about any detection tapped on in the React Native user interface. This verified that the endpoint was successfully integrated into the design and with the SQL database, and that end-to-end functionality was working as intended.

The React Native UI was also successful in all of the criteria it was expected of it during the test. The revamped main page asked the user for address input, also allowing them to enter their current location via a “Find Location” button. In addition, the page features a radius selection mechanic. Upon clicking the “Find Parking” button, the UI was able to make a request to the Python Park API. This request was handled appropriately, and any errors were caught by the frontend and displayed correctly to the user as an error response message.

The UI also displayed the new “Help” modal, accessible by tapping on the top right corner of the application at any point to get detailed information on how to use the application.

The React Native UI was successful in how the map was displayed to the user, and how the detailed modal functioned. Each detection marker on the map was pressable, allowing the user to make a request to the Python Detail API. This allowed the user to see relevant image and address information, as well as a confidence score, about any detections the model made. As well, any detections that were road signs had the text read off of them and presented to the frontend for the user to read. These modals were easy to understand and navigate while exploring the results map. Often, the text processing was unable to interpret the text from road signs. Sometimes, the text was read perfectly off of road signs. This all depended on the angle, lighting, and size of the road sign within the image from the Google Street View endpoint.

# CHAPTER 6: RESULTS ANALYSIS

## *6.1 INSTALLATION, SET UP AND SUPPORT*

### 6.1.1 INSTALLATION

The WhereTo application is designed to ensure that minimal setup is required for use. Users need only to download and launch the application to activate the service. Considering that WhereTo is a digital application, one should ensure their smartphone has a stable and reliable internet connection to allow for real-time functionality.

When installing WhereTo, users should ensure their device has sufficient storage space and their operating system is up to date. Additionally, users should ensure their smartphone's GPS is enabled for accurate location services, as it is crucial to utilize the app's 'Current Location' feature.

### 6.1.2 SET UP

1. Install Expo Go from your smartphone's App Store or Google Play Store.
2. Initialize the React Native application's frontend code on the personal computer.
    a. Ensure that the base URL for the backend API is consistent with the currently cloud hosted WhereTo backend.
3. Launch the frontend service, generating a QR code with 'npx expo start.'
4. Open Expo Go on the smartphone and scan the QR code from the computer screen, downloading the application to the smartphone.
5. Utilize the application.

### 6.1.3 SUPPORT

To ensure a seamless user experience, WhereTo offers an interactive iOS and Android mobile application. The application allows users to press a help button in the top

right corner of the screen. This button will prompt the user with a guide on using the application services. There are no advanced configuration settings for using WhereTo, which helps modal educate users on the application's features.

## *6.2 USER INTERFACE*

Upon launching the WhereTo application, users are greeted with a minimalist and intuitive interface. The focus is immediately on the two input fields: one for the address and the other for selecting the search radius as it can be seen in Figure 12. A prominent 'Find Parking' button prompts users to initiate their search. Additionally, the 'Current Location' button simplifies the search process using the device's GPS to fill in the user's current location automatically, making the process even more efficient and user-friendly.

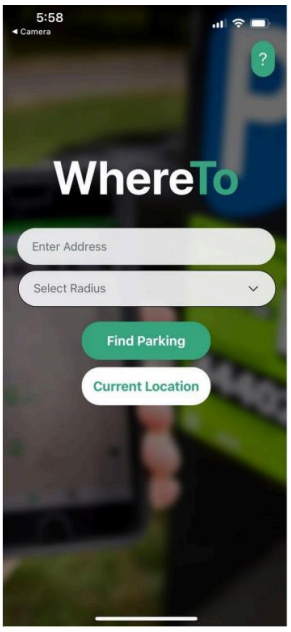



*Figure 12- WhereTo main screen.*

The application offers a user-friendly experience when entering an address. For instance, it has an autocomplete feature that suggest locations as the user types, as shown in Figure 13, reducing orthographic errors and speeding up the whole process. This feature

maximizes the usability and ensures that the user is able to find the parking information required.

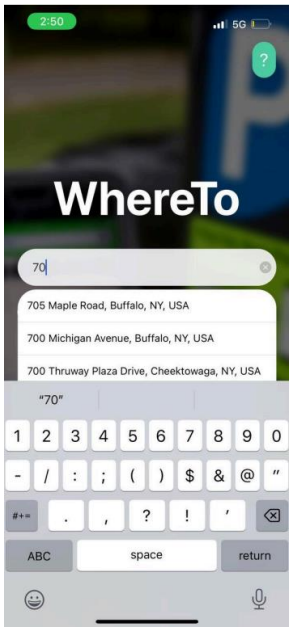



*Figure 13- WhereTo autocomplete feature.*

The selection of the radius started being an input, however, in order to make everything more straightforward, it was changed into a dropdown with 3 options: 50, 100 and 150 meters. This dropdown ensures that users can customize their search area to their immediate needs. It is very easy and logical to use as it is shown in Figure 14.

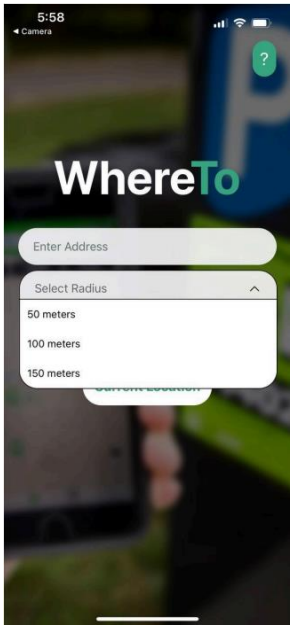


*Figure 14- WhereTo radius dropdown.*

In case the user required additional help to use the application, a button with a '?' has being implemented on the upper right side of the main screen (Figure 12). The purpose of this guide is to help first time users navigate the app without confusion. The help modal provides step by step information on how to use WhereTo in order to get the parking information, as shown in Figure 15.

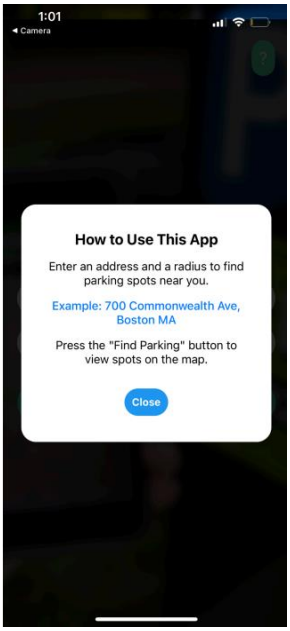



*Figure 15- WhereTo Help modal.*

After initiating a search, a map is displayed in the application (Figure 16) with pins identifying all the detections within the selected radius. The interactive map allows the user to zoom in and out for a closer examination of the area. In the map there is also a button in the right bottom corner where users can change the map view from standard (Figure 16) to satellite view (Figure 17), offering a real-world perspective of the area and helping with a more precise navigation. This feature is particularly useful for users who are unfamiliar with the area since it can provide a visual representation of the terrain, which makes it easier to identify each street.

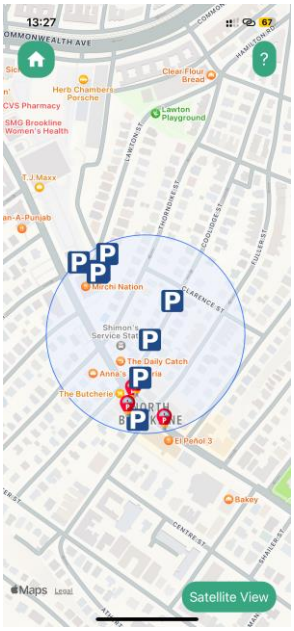



*Figure 16- WhereTo map standard view.*

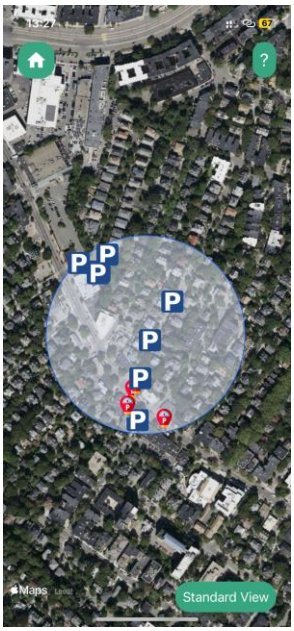



*Figure 17- WhereTo map satellite view.*

On Figure 16 there are two types of pins: The blue one is for parking signs while the red one is for parking meters. The user can interact with these pins to get more detailed information about that detection. In both cases it appears a modal with the precise address,

the detection type, the confidence of the model and an image from where the model found the detection, as it can be seen in Figure 18. If it is a parking sign, the text that was read appears under the other information, unless the Google Cloud Vision could not understand what it was written, as it can be seen in Figure 19.

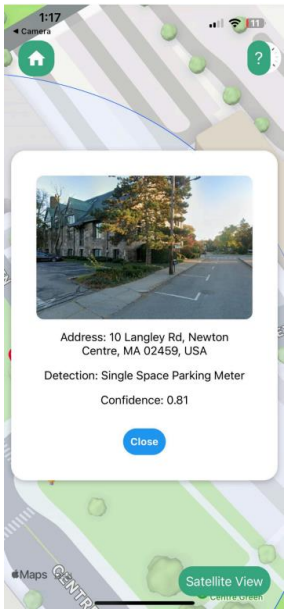



*Figure 18- WhereTo Parking meter detail modal.*

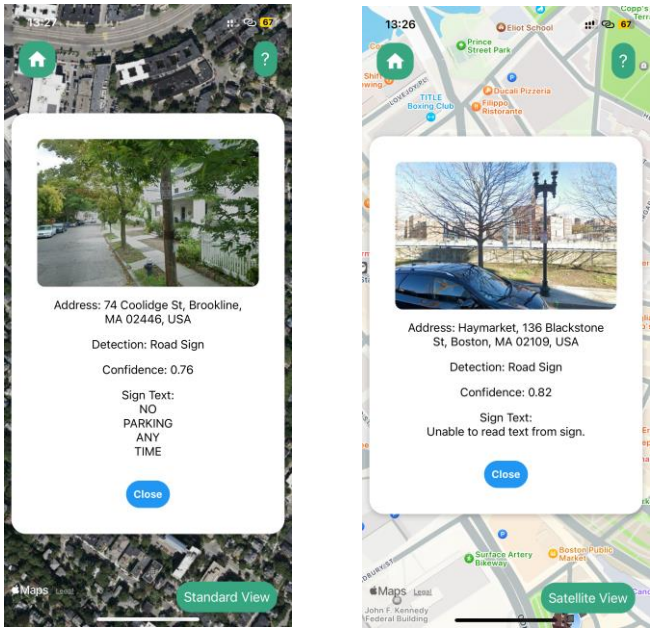



*Figure 19- WhereTo Parking sign detail modal.*

## *6.3 OPERATION OF THE PROJECT*

### 6.3.1 NORMAL OPERATIONS

Under normal circumstances, the operation of the WhereTo application is exceptionally straightforward, allowing many users to utilize its functions. With the application already installed, all the user has to do to use the application is the following:

1. Enter an address into the first input field with the placeholder “Enter Address” or use the “Current Location” button for the address.
    a. This input field has an autofill address bar, so as the user types, suggestions populate the address bar.
    b. The “Current Location” button will work, provided the user has enabled location services for Expo Go.
2. Select a radius from the dropdown placeholder “Radius in Meters (50,100,150).”
3. Press the ‘Find Parking’ button.
4. Allow the application to compute the necessary information, which may take a couple of minutes depending on street density and radius of selection.
5. View the results once loaded onto the screen as a map view.
    a. Pressing on any pinned detection will open the detail modal, allowing the user to see the location they are interested in in more detail.
    b. The user can press the ‘Close’ button to leave the detailed modal view.
6. Once the user is done examining the results, they can leave the map view and return to the input view by pressing the ‘Home’ icon at the top of their screen.

Under these circumstances, provided that an actual address and valid radius are entered into the input fields, the user will be able to view any parking meters or road signs that the WhereTo model has detected and view detailed information about each one of these that appears within the specified radius.

### 6.3.2 ABNORMAL OPERATIONS

There is only one condition due to user error that can lead to an error response from the backend when utilizing WhereTo. This user error would be inputting an invalid address, or no address, into the "Enter Address" input field. Another possible error case could be a failure in the cloud servers that WhereTo runs on, which is a server error. In either case, server or user, the user receives the error message that appears in Figure 20.

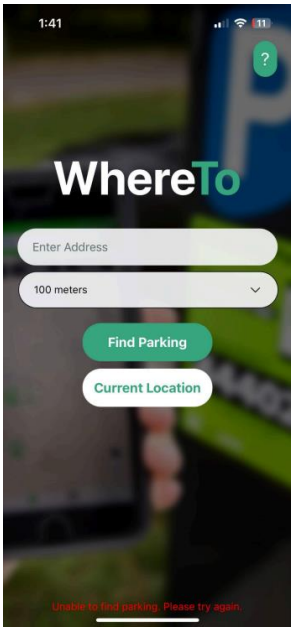



*Figure 20- WhereTo main screen error.*

The user gets an error message stating 'Unable to find parking. Please try again.' The core Python API and the React Native UI of WhereTo can handle these errors. Under these operating conditions, users must modify their inputs to be valid values to return to normal operating conditions and locate parking.

### 6.3.3 SAFETY ISSUES

Using WhereTo is generally safe as it only requires an address and a search radius, ensuring that no sensitive personal information is transmitted to the backend. However, users should adhere to safety guidelines, notably distracted driving: users must avoid interacting with WhereTo while driving.

# CHAPTER 7: CONCLUSIONS AND FUTURE WORK

## *7.1 CONCLUSIONS*

### 7.1.1 BACKEND ENHANCEMENTS

#### Machine Learning Model Integration

A major step forward in the backend development of the WhereTo application was the incorporation of a machine learning model, notably using Ultralytics' YOLOv8. This model was created to recognize various items, with a primary emphasis on road signs and parking meters. The selection of YOLOv8 was motivated by its ability to effectively manage several object detections in a single model stream, therefore improving processing speed and accuracy in general. The project required gathering a large dataset of about 800 photos from Google Street View that included a variety of urban settings from places including Boston, Charlotte, Los Angeles, and New York. The model's relevance and application across various regional designs was ensured by the inclusion of both single-space and multi-space parking meters in the dataset. The successful training and deployment of this model not only improved the detection accuracy but also set a foundation for future enhancements in recognizing and processing parking-related objects.

#### Detection Bounding Box System

Detection bounding box system implementation was a crucial part of the backend development. The goal of this system was to improve detection accuracy, especially at intricate urban crossroads where several items may overlap. Bounding boxes allowed the system to define the exact position and bounds of items it recognized, which decreased false positives and increased the accuracy of the detection findings. This strategy was crucial to ensuring that the app could give users accurate and useful information about parking spaces

that were available as well as pertinent road signs.

**API Endpoints and Error Handling**

The creation of effective API endpoints boosted the backend architecture even more. By enabling smooth connection between the backend server and the frontend user interface, these endpoints made sure that user requests were handled quickly and correctly. Strong error-handling procedures were also implemented to handle possible problems such erroneous user input, server outages, and unanticipated failures during data processing. These improvements were essential to preserving the stability of the program and offering a seamless user experience.

### 7.1.2 FRONTEND IMPROVEMENTS

**Cross-Platform Development with React Native**

React Native, a framework well-known for enabling the development of cross-platform applications, was used to create the WhereTo application's frontend. This decision expanded the application's possible user base by guaranteeing that it worked with both iOS and Android smartphones. During the development process, key functionality including address input fields, search radius selection, and map displays were prioritized in favor of a simple and user-friendly interface. The application's user-friendly interface makes it easy for users to explore and obtain the necessary information with the least amount of difficulty.

**Enhanced User Interface Features**

To improve usability and functionality, the user interface was enhanced with the integration of several essential elements. To guarantee that users provided legitimate addresses and search parameters, input validation methods were put in place. Additionally, a function known as autofill was added, which reduced input errors and expedited the search process by proposing potential addresses as users typed. To help users in the event of improper inputs or other problems, user-friendly error messages were also included. Together, these innovations made the user experience more effective and pleasurable.

**Interactive Map and Information Display**

One of the best aspects of the frontend development was the interactive map functionality. Within their designated search radius, users might see a map highlighted with detected parking meters and road signs. Users could examine comprehensive modal views with vital details including the exact address, photos of the site, and the AI model's degree of confidence in the detection by clicking on these markers. This interactive feature helped customers make judgments by giving them a thorough understanding of the parking situation.

### 7.1.3 SYSTEM INTEGRATION AND PERFORMANCE

**Seamless Integration via APIs**

The smooth integration of the frontend and backend components was a significant project milestone. This was made possible by thoughtfully created APIs that made it easier for the client application and server to exchange data effectively. Data encoding methods were used to make sure the transferred data was little and could be processed fast, such as base64 for picture data. This integration was essential to improving the application's

responsiveness and user happiness by enabling real-time modifications to the user interface depending on user inputs and backend responses.

**Deployment and Cost Efficiency**

Another notable accomplishment was the WhereTo system's implementation on Google Cloud. This cloud platform was selected due to its affordability, scalability, and dependability. Configuring the server environment, establishing the required API endpoints, and guaranteeing safe access to the database and other resources were all part of the deployment process. This deployment gave important insights into the economic dynamics of running the application in a cloud environment in addition to demonstrating the system's operational efficiency. Planning for future scalability and cost optimization of operations would greatly benefit from these insights.

**Real-Time Data Processing**

Real-time data processing was one of the WhereTo application's most important requirements. In order to guarantee that customers received the most recent information on parking availability and pertinent road signs, the backend system was designed to handle real-time data processing. Effective algorithms and data processing pipelines that reduced latency and increased throughput were used to achieve this. The system's capacity to deliver precise and timely information was essential to fulfilling user demands and improving the application's overall usefulness.

## *7.2 FUTURE WORK*

The WhereTo project has established a solid framework for a creative and intuitive parking management program. A number of areas have been selected for future improvement in order to build upon this achievement. With these improvements, the application should be able to reach a wider audience, work better, and provide users more benefits. The intended future work is described in depth in the following sections.

### 7.2.1 INTERNATIONAL EXPANSION

The WhereTo app is currently only intended for usage in the United States because the machine learning model was trained only on pictures taken in American cities. It is imperative to expand the dataset to encompass photographs from other worldwide areas in order to render the application feasible in other nations. This expansion would involve:

- Dataset Acquisitions: assembling and annotating pictures of road signage and parking meters from different nations. This might be accomplished by using global street view services, crowdsourcing data, and forming alliances with nearby municipalities. To guarantee that the model is applicable in many locations, special attention will be paid to capturing a broad range of parking meter and signage designs.
- Model Training and Validation: Retraining the YOLOv8 model with the enlarged dataset will guarantee that it recognizes and understands parking-related objects correctly across national borders. Strict validation would be needed for this process in order to take into consideration regional differences in meter and signpost designs. It will need repeated testing and revisions to improve the model's accuracy and dependability in various urban settings.
- Localization and Language Support: modifying the program interface to accommodate many languages and regional customs. This would entail localized help and support features as well as translation of the user interface elements. To

improve the user experience across different locations, cultural differences in preferences for user interface design will also be taken into account.

By expanding the geographic coverage, WhereTo can become a truly global application, catering to users in diverse urban environments worldwide. This would significantly broaden the user base and increase the application's utility for international travelers.

### 7.2.2 ENHANCED MAP VISUALIZATION

The ability to dynamically view parking zones and the prices associated with them on a map is one of the main improvements that have been suggested. Countries like Spain who have street parking with color-coded zones would find this feature especially helpful. The following would be involved in implementation:

- Data integration: is the process of combining data sources that offer parking zone and cost information. Municipal open data portals, commercial parking companies, and information gathered from the community could all fall under this category. Creating APIs to retrieve real-time data and guarantee that it is current would-be part of the integration process.
- Map Overlay Development: creating a system to display the information about parking zones on a map. The parking areas that each street belongs to would be indicated via a color-coding system (for example, blue for paid parking, green for residential parking, and white for free parking). Extensive GIS (Geographic Information System) capabilities are necessary to precisely illustrate the zones on the map.
- Interactive Pricing Information: enabling visitors to examine comprehensive details regarding parking regulations, costs, and time constraints by clicking on various zones. With the ability to access complete parking information quickly and easily, this feature would improve user convenience. To further enhance usability, a filtering

system that lets users choose their favorite parking zones according to price, time, or other factors should be created.

With this update, the application's usefulness would be much increased, and users would be able to find appropriate parking spots fast, taking into account their demands and budget. It would also help with extended stay planning by pointing out areas with cheaper rates or less stringent parking regulations.

### 7.2.3 PARTNERSHIPS WITH PARKING PAYMENT SERVICES

It is suggested that collaborations be established with parking payment providers like Telpark in order to provide a smooth parking experience. Through this integration, users would be able to pay for parking spaces straight through the WhereTo app in addition to finding them. The following actions would be taken as part of this integration:

- API Integracion: working together to incorporate payment service providers' APIs into the WhereTo app. To protect user transactions, secure data exchange protocols would be required. It will be crucial to provide a reliable payment gateway that accepts a variety of payment options, such as credit cards, digital wallets, and mobile payments.
- User Account Management: putting in place facilities for user accounts that let users manage their parking subscriptions, view transaction history, and preserve payment data. To protect user data, this would necessitate strong security measures like encryption and two-factor authentication.
- Seamless User Experience: creating a user interface that makes payments simple. With just a few clicks, users should be able to choose a parking space, see the prices, and finish the payment. It would also improve user experience to send out warnings and reminders regarding parking expiration dates. Including a loyalty or rewards program to encourage consistent app use may help boost user engagement even more.

WhereTo can become a full parking management platform and eliminate the headache of finding and paying for parking by integrating payment methods. Users would benefit greatly from this, since the app would become their one-stop shop for all parking-related needs.

### 7.2.4 ADVANCED MACHINE LEARNING CAPABILITIES

Real-time updates and predictive analytics can be obtained through the use of sophisticated machine learning algorithms, which will further improve the functionality of the program. This would involve:

- Predictive Modeling: creating algorithms that forecast parking availability in response to past performance, time of day, and nearby events. Users could more efficiently organize their parking with this assistance. More precise forecasts could be made by using big data analytics to find patterns and trends in parking utilization.
- Real-Time Data Integration: incorporating real-time information on parking availability from user reports, traffic cameras, and sensors. To handle real-time data, this functionality would need a strong data ingestion and processing pipeline. By using edge computing techniques to process data closer to the source, latency can be decreased, and application responsiveness can be increased.
- User Behavior Analysis: examining user behavior to offer recommendations for specific parking. Parking locations could be suggested based on trends found in user preferences by machine learning algorithms. It would improve the user experience to create a recommendation engine that considers things like the user's location, time of day, and previous parking decisions.usage.

With these sophisticated features, WhereTo might be used as a preventative parking helper in addition to a reactive one. The software can help users be more efficient overall and spend less time looking for parking by giving them timely and relevant information.

# CHAPTER 8: REFERENCES

[2] INRIX. (2017, june 12). Searching for parking costs Americans $73 billion a year. Inrix. https://inrix.com/press-releases/parking-pain-us/ 2.

[3] Hampshire, R., & Shoup, D. (2019, november 21). How Much Traffic is Cruising for Parking? Transfers Magazine. https://transfersmagazine.org/magazine-article/issue4/how-much-traffic-is-cruising-for-parking/ 3.

[4] Nir, S. M. (2023, october 9). Parking in New York city really is worse than ever. The New York Times. https://www.nytimes.com/2023/10/09/nyregion/nycparking-cars.html

[5] Tahir, M. S. (2023, October 8). YOLO V8: A State-of-the-Art Object Detection Model with Significant Advantages over Previous YOLO Versions. *Medium*. https://medium.com/@salmantahir717/yolo-v8-a-state-of-the-art-object-detection-model-with-significant-advantages-over-previous-yolo-0e78cb971b37

[6] Keylabs. (2024, March 19). *Under the hood: YOLOV8 architecture explained*. Keylabs: Latest News and Updates. https://keylabs.ai/blog/under-the-hood-yolov8-architecture-explained/#:~:text=YOLOv8%20is%20a%20state%2Dof,objects%20in%20real%2Dtime%20scenarios.

[7] Mwaura, W. (2022, May 19). *Making HTTP requests with Axios*. CircleCI. https://circleci.com/blog/making-http-requests-with-axios/#:~:text=Axios%20is%20a%20promise%2Dbased,PUT%2FPATCH%20%2C%20and%20DELETE%20.

[8] City of New York, NYC Open Data. (s. f.). *NYC Open Data*. https://opendata.cityofnewyork.us/

[9] *Parking Meters Locations and Status (Map) | NYC Open Data*. (s. f.). https://data.cityofnewyork.us/Transportation/Parking-Meters-Locations-and-Status-Map-/mvib-nh9w

[10] *Distance on a sphere: The Haversine Formula*. (2021, 12 diciembre). Esri Community. https://community.esri.com/t5/coordinate-reference-systems-blog/distance-on-a-sphere-the-haversine-formula/ba-

p/902128#:~:text=All%20of%20these%20can%20be,longitude%20of%20the%20two%20points.

[11] Now, D. (s. f.). *moderno estacionamiento mapa alfiler icono. vector*. Vecteezy. https://es.vecteezy.com/arte-vectorial/26529451-moderno-estacionamiento-mapa-alfiler-icono-vector

[12] *Icono de parquímetro basic straight flat*. (2017, 15 diciembre). Freepik. https://www.freepik.es/icono/parquimetro_659569

# APPENDIX I: ALIGNMENT OF THE PROJECT WITH THE SDG

This project aligns with several United Nations Sustainable Development Goals (SGDs), focusing on urban sustainability, innovation, and quality of life improvements. Here´s how the project supports these goals in detail.

## SDG 11: Sustainable Cities and Communities

Target 11.2:

By providing real-time parking information through its mobile application, the project makes a substantial contribution to sustainable urban transportation. By streamlining the parking procedure, this feature lessens traffic congestion, and the amount of time drivers must spend looking for a place to park. In metropolitan regions where traffic congestion is a major problem that contributes to increased air pollution and a lower quality of life, effective parking management is essential. The program lessens the needless driving that causes these issues by giving drivers a tool to locate parking rapidly.

The efficiency of urban transportation networks is further increased by the integration of GPS and real-time data, which guarantees that vehicles receive the most up-to-date information regarding parking spaces that are available. This improvement in traffic flow contributes to the overall quality of urban life, making cities more livable and sustainable.

Target 11.6:

Additionally, the application is essential for enhancing urban areas' air quality. Significantly less time is spent traveling around looking for parking, which lowers automobile emissions. Urban air pollution is mostly caused by idling and driving in circles

while searching for parking spaces. The project contributes to reducing the number of pollutants, particulate matter, nitrogen oxides, and carbon monoxide released into the atmosphere by reducing these activities.

The project not only lowers emissions but also subtly promotes healthy urban life. Residents of cities who live in superior air quality have better respiratory and cardiovascular health. As a result, the application benefits public health in addition to addressing environmental and transportation-related challenges.

## SGD 9: Industry, Innovation, and Infrastructure

Target 9.1:

The creation of a sophisticated parking management system is a noteworthy breakthrough in urban infrastructure. The project makes use of cutting-edge technology like multiple APIs for real-time data collection and display and artificial intelligence (AI) for object detection. These creative solutions show a contemporary, technology-driven approach to solving urban problems.

AI technology enables the program to precisely recognize and analyze traffic signs and parking meters, especially when using models such as YOLOv8 for object detection. This feature is necessary to give consumers accurate parking information. The incorporation of these cutting-edge technology is a prime example of how innovation can propel advancements in urban infrastructure, increasing its effectiveness and adaptability to the demands of urban residents.

Target 9.4:

Through process optimization, the initiative promotes resource efficiency and sustainability in urban infrastructure renovation. Large parking signs and complex signage systems are among the physical infrastructures that the program helps to eliminate. Rather,

it uses digital technology to deliver information that is needed, saving material resources and lessening its impact on the environment.

Additionally, the project encourages the use of real-time information systems and data analytics, both of which are essential elements of smart city programs. With the use of these technologies, cities can better serve their citizens, react swiftly to changing circumstances, and manage their resources more efficiently. These kinds of technologies will be essential to maintaining the sustainability and resilience of cities as they expand and change.

## SGD 3: Good Health and Well-being

Target 3.9:

By using more effective parking alternatives to reduce urban air pollution, the project directly improves public health. Numerous health problems, including respiratory and cardiovascular disorders, are associated with poor air quality. The concept helps reduce the amount of dangerous pollutants in the air by reducing the amount of time cars spend idling or driving around looking for parking.

Better air quality leads to better health outcomes for those living in cities. Lowering the exposure to pollutants like particulate matter and nitrogen dioxide can help lower the prevalence of lung cancer, asthma, and other respiratory disorders. In this sense, by tackling one of the primary causes of air pollution in cities, the project advances more general goals related to public health.

Additionally, the software promotes more environmentally friendly driving practices, which may have long-term positive effects on health. The project contributes to the creation of a healthier urban environment where inhabitants can benefit from cleaner air and improved general well-being by encouraging effective parking practices.

## SGD 13: Climate Action

Target 13.2:

Through decreased automobile emissions, the project mitigates the effects of climate change and promotes climate action. Greenhouse gas emissions and fuel use are reduced by effective parking management. The software helps drivers find parking more quickly, which cuts down on the amount of time cars are on the road and thus, carbon dioxide emissions.

This decrease in emissions helps the worldwide effort to fight climate change. Transportation emits a significant amount of greenhouse gasses, therefore a workable solution to lessen the industry's carbon footprint is to increase the efficiency of urban parking facilities. The initiative supports global warming mitigation and sustainable development objectives.

Apart from the immediate advantages it provides for the environment, the project also serves to highlight the significance of sustainable transportation techniques. It further supports climate action activities by encouraging users to adopt more environmentally friendly driving habits by showcasing the observable benefits of effective parking management.

## SGD 8: Decent Work and Economic Growth

Target 8.2:

The project promotes economic growth by using technology to increase productivity in urban areas. It opens new doors for the data management, urban planning, and tech development industries. Professionals with expertise in software engineering, machine learning, and geographic analysis are needed for the application's development and implementation, which will boost the economy and create jobs.

By making urban transportation networks more efficient, the project also boosts local economy. Smoother and more consistent traffic flows can result from effective parking management, which also helps to lessen traffic congestion. Businesses and individuals who depend on urban transit for their daily operations may find that their productivity is increased as a result.

Furthermore, by displaying creative applications of AI and real-time data systems, the application can help the tech sector flourish. Further economic growth and development will be fueled by the rising need for knowledge in these fields as more cities and regions embrace comparable technologies.

## SGD 17: Partnerships for the Goals

Target 17.16:

The project, which integrates many APIs and works with technology suppliers like Google, is an example of a successful public-private relationship. The sustainability and scalability of urban solutions depend on these collaborations. Instead of needing to create every component from scratch, the project may offer a high-quality service by utilizing current technology and working with reputable IT businesses.

Through these collaborations, the project is also able to stay abreast of the most recent developments in technology and industry best practices. This guarantees the application's continued efficacy and relevance in a quickly evolving technical environment. Working with reputable partners can further increase the project's visibility and reputation, drawing in more users and possible contributors.

All things considered, the project shows how smart alliances can spur sustainability and innovation in urban planning. Together, public and private organizations may address complex urban issues in a more efficient and scalable manner, thereby assisting in the accomplishment of the SDGs.

# APPENDIX II

## *STYLES*

Css from Main Screen:

```
const styles = StyleSheet.create({
  backgroundImage: {
    flex: 1,
    justifyContent: 'center',
  },
  overlay: {
    flex: 1,
    backgroundColor: 'rgba(0, 0, 0, 0.5)',
    alignItems: 'center',
    justifyContent: 'flex-start', // Ensures content starts from the top
    paddingTop: 40,
  },
  title: {
    fontSize: 64,
    fontWeight: 'bold',
    color: '#FFFFFF',
    marginTop:157,
    marginBottom: 20,
  },
  input: {
    width: '90%',
    height: 50,
    backgroundColor: '#f8f9fa',
    borderRadius: 25,
    paddingHorizontal: 20,
    fontSize: 16,
    //zIndex: -1000,
    color: '#333333',
    marginVertical: 10,
    opacity: 0.9,
  },
  input1: {
    width: '90%',
    height: 50,
    marginLeft: 20,
    backgroundColor: '#f8f9fa',
    borderRadius: 25,
    paddingHorizontal: 20,
    fontSize: 16,
```

```
    zIndex: -1000,
    color: '#333333',
    marginVertical: 10,
    opacity: 0.9,
  },
  googlePlacesInputContainer: {
    width: '90%', // Ensure the container is wide enough
    backgroundColor: 'transparent',
    zIndex: 5,
    color: '#333333',
    borderTopWidth: 0,
    borderBottomWidth: 0,
  },
  googlePlacesInput: {
    width: '90%',
    height: 50,
    backgroundColor: '#f8f9fa',
    borderRadius: 25,
    zIndex: 1000,
    paddingHorizontal: 20,
    fontSize: 16,
    color: '#333333',
    marginVertical: 10,
    opacity: 0.9,
  },
  listView: {
    position: 'absolute',
    top:65, // Adjust this value based on your layout
    width: '90%',
    borderRadius: 25,
    zIndex: 100, // Ensure this is very high to bring in front
    backgroundColor: 'white',
    elevation: 7, // For Android to ensure the shadow and elevation
  },
  button: {
    width: '90%',
    backgroundColor: '#4ECCA3',
    padding: 15,
    borderRadius: 25,
    alignItems: 'center',
    justifyContent: 'center',
    marginVertical: 10,
  },
  buttonuser1: {//current location button
    width: '45%',
    backgroundColor: '#FFFFFF',
    padding: 15,
    borderRadius: 25,
    zIndex: -1000, // Ensure this is very high to bring in front
    alignItems: 'center',
    justifyContent: 'center',
    marginBottom: 300,
```

```
  },
  buttonuser2: {//find parking button
    width: '45%',
    backgroundColor: '#38a681',
    padding: 15,
    borderRadius: 25,
    alignItems: 'center',
    zIndex: -1000,
    justifyContent: 'center',
    marginVertical: 10,
  },
  buttonText1: {
    color: '#FFFFFF',
    fontSize: 18,
    fontWeight: 'bold',
  },
  buttonText2: {
    color: '#38a681',
    fontSize: 18,
    fontWeight: 'bold',
  },
  errorText: {
    color: '#FF0000',
    marginTop: -38,
    marginBottom: 20,

  },
  // New styles for help modal and button
  helpButton: {
    position: 'absolute',
    right: 20,
    top: 50,
    backgroundColor: '#38a681',
    borderRadius: 20,
    padding: 10,
  },
  helpButtonText: {
    fontSize: 24,
    color: '#FFFFFF',
  },
  centeredModalView: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    marginTop: 22,
    backgroundColor: 'rgba(0, 0, 0, 0.8)',
  },
  modalContent: {
    margin: 20,
    backgroundColor: "white",
    borderRadius: 20,
    padding: 35,
```

```
    alignItems: "center",
    shadowColor: "#000",
    shadowOffset: {
      width: 0,
      height: 2,
    },
    shadowOpacity: 0.25,
    shadowRadius: 4,
    elevation: 5,
  },
  modalExample: {
    marginBottom: 15,
    textAlign: "center",
    fontSize: 16,
    color: '#007bff', // Example color for differentiation
    fontWeight: '500', // Slightly bolder than normal text for emphasis
  },
  modalTitle: {
    marginBottom: 15,
    textAlign: "center",
    fontSize: 20,
    fontWeight: 'bold',
  },
  modalText: {
    marginBottom: 15,
    textAlign: "center",
    fontSize: 16,
  },
  buttonClose: {
    backgroundColor: "#2196F3",
    marginTop: 15,
    borderRadius: 20,
    padding: 10,
  },
  textStyle: {
    color: "white",
    fontWeight: "bold",
    textAlign: "center",
  },
});
```

## Css from Map Display Map

```
const styles = StyleSheet.create({
     backgroundImage: {
       flex: 1,
       width: '100%',
       height: '100%',
    },
```

```
    container: {
      flex: 1,
      justifyContent: 'flex-end',

    },
    homeButton: {
      position: 'absolute',
      width: 50,
      height: 50,
      top: 50, // Adjust as needed
      left: 20, // Adjust as needed
      elevation: 5,
      backgroundColor: '#38a681',
      borderRadius: 20,
      padding: 10,
  },
    map: {
      width: width,
      height: height ,
    },
    centeredView: {
      flex: 1,
      justifyContent: "center",
      alignItems: "center",
      marginTop: 22,
    },
    modalView: {
      margin: 20,
      backgroundColor: "white",
      borderRadius: 20,
      padding: 35,
      alignItems: "center",
      shadowColor: "#000",
      shadowOffset: {
        width: 0,
        height: 2,
      },
      shadowOpacity: 0.25,
      shadowRadius: 4,
      elevation: 5,
    },
    parkingImage: {
      width: 300,
      height: 200,
      borderRadius: 10,
      marginBottom: 15,
    },
    mapTypeButton: {
      position: 'absolute',
      backgroundColor: '#38a681',
      bottom: 20, // Adjust as necessary
      right: 20, // Adjust as necessary
```

```
    //backgroundColor: 'rgba(0,0,0,0.7)',
    borderRadius: 20,
    padding: 10,
  },
  mapTypeButtonText: {
    color: 'white',
    fontSize: 18,
    //nothing for commit
  },
  backButton: {
    backgroundColor: "#4ECCA3",
    paddingVertical: 12,
    paddingHorizontal: 20,
    borderRadius: 25,
    marginTop: 10,
    marginBottom: 25,
    alignSelf: 'center',
  },
  backButtonText: {
    color: "white",
    fontSize: 18,
    fontWeight: "bold",
  },
  button: {
    borderRadius: 20,
    padding: 10,
    elevation: 2,
  },
  buttonClose: {
    backgroundColor: "#2196F3",
    marginTop: 15,
  },
  textStyle: {
    color: "white",
    fontWeight: "bold",
    textAlign: "center",
  },
  helpButton: {
    position: 'absolute',
    right: 20,
    top: 50,
    backgroundColor: '#38a681',
    borderRadius: 20,
    padding: 10,
  },
  helpButtonText: {
    fontSize: 24,
    color: '#FFFFFF',
  },
  centeredModalView: {
    flex: 1,
    justifyContent: "center",
```

```
        alignItems: "center",
        marginTop: 22,
        backgroundColor: 'rgba(0, 0, 0, 0.8)',
      },
      modalContent: {
        margin: 20,
        backgroundColor: "white",
        borderRadius: 20,
        padding: 35,
        alignItems: "center",
        shadowColor: "#000",
        shadowOffset: {
          width: 0,
          height: 2,
        },
        shadowOpacity: 0.25,
        shadowRadius: 4,
        elevation: 5,
      },
      modalTitle: {
        marginBottom: 15,
        textAlign: "center",
        fontSize: 20,
        fontWeight: 'bold',
      },
      modalText: {
        marginBottom: 15,
        textAlign: "center",
        fontSize: 16,
      },
      modalText1: {
        marginBottom: 15,
        textAlign: "center",
        fontSize: 16,
      }
    });
```

Css from Detail Modal

```
const styles = StyleSheet.create({
  card: {
    position: 'absolute', // Adjust positioning as needed
    bottom: 20, // Example placement at the bottom
    left: 20,
    right: 20,
    backgroundColor: 'white',
    padding: 20,
    borderRadius: 8,
    shadowOpacity: 0.25,
```

```
    shadowRadius: 5,
    shadowColor: 'black',
    shadowOffset: { height: 0, width: 0 },
    elevation: 5, // for Android
  },
  image: {
    width: '100%',
    height: 200, // Adjust as needed
    borderRadius: 4,
  },
  text: {
    marginTop: 10,
    fontSize: 16,
  },
});
```

## *REQUIREMENTS*

The requirements for the Project are specified on a file called requirements.txt and it is used in Python projects to specify the dependencies needed for the project. Each line in the file specifies a package and its version that the project relies on.

```
aniso8601==9.0.1
bcrypt==4.1.2
beautifulsoup4==4.12.3
blinker==1.7.0
cachetools==5.3.3
certifi==2024.2.2
charset-normalizer==3.3.2
click==8.1.7
contourpy==1.2.0
cycler==0.12.1
exceptiongroup==1.2.0
filelock==3.13.1
Flask==3.0.2
Flask-RESTful==0.3.10
Flask-SQLAlchemy==3.1.1
fonttools==4.49.0
fsspec==2024.2.0
google==3.0.0
google-api-core==2.17.1
google-api-python-client==2.121.0
google-auth==2.28.1
google-auth-httplib2==0.2.0
google-cloud==0.34.0
google-cloud-vision==3.7.2
```

```
googleapis-common-protos==1.62.0
grpcio==1.62.0
grpcio-status==1.62.0
httplib2==0.22.0
idna==3.6
iniconfig==2.0.0
itsdangerous==2.1.2
Jinja2==3.1.3
kiwisolver==1.4.5
MarkupSafe==2.1.5
matplotlib==3.8.3
mpmath==1.3.0
networkx==3.2.1
numpy==1.26.4
opencv-python==4.9.0.80
packaging==23.2
pandas==2.2.1
pillow==10.2.0
pluggy==1.4.0
proto-plus==1.23.0
protobuf==4.25.3
psutil==5.9.8
py-cpuinfo==9.0.0
pyasn1==0.5.1
pyasn1-modules==0.3.0
pyparsing==3.1.1
pytest==8.0.2
python-dateutil==2.8.2
pytz==2024.1
PyYAML==6.0.1
requests==2.31.0
rsa==4.9
scipy==1.12.0
seaborn==0.13.2
six==1.16.0
soupsieve==2.5
SQLAlchemy==2.0.27
sympy==1.12
thop==0.1.1.post2209072238
tomli==2.0.1
torch==2.2.1
torchvision==0.17.1
tqdm==4.66.2
typing_extensions==4.9.0
tzdata==2024.1
ultralytics==8.1.18
uritemplate==4.1.1
urllib3==2.2.1
Werkzeug==3.0.1
```