LabLocator - Technical Guide

Project Title	Lab Locator
Ethan Clarke	19372086
Evelina Prosyankina	19368013
Supervisor	Dr. Darragh O’Brien
Completion Date	06/05/2023

1. Introduction

1.1. Abstract

This project encompasses the mobile application “LabLocator” developed by Ethan Clarke and Evelina Prosyankina, under the supervision of Dr. Darragh O’Brien. It was developed for use on Android mobile devices, and intends to serve as a tool that can pinpoint the location of a user with a per-room accuracy while indoors. It does this by analysing the various Wi-Fi access points that are visible to the device, and passes that information through a machine-learning algorithm that has been trained to identify room locations based on that information. The application also monitors how many of its users currently reside in any given room, with the intent being to show all users a visualisation of the congestion in each room to allow them to make a more informed decision of what room they choose to visit for the purpose of doing work or study. The application was developed with the computer laboratories of the McNulty Building on Glasnevin Campus in mind, of which many of its labs can become quite noisy during peak times when there are many people frequenting them. The application presents itself as a proof of concept in the field of indoor localisation, a difficult technological issue involving the tracking of people and devices in an indoor environment where GPS is unreliable. The overall aim of LabLocator is to show the possibilities of a system where Wi-Fi access points are used to track general location on a per-room basis and how it can serve as a utility for monitoring the congestion of rooms, finding rooms, and finding people in rooms.

1.2. Motivation

Our interest in starting a project like this stems from multiple different converging reasons and motivations. We held prior interest in the fields of Indoor Localisation and Machine Learning which we had some prior experience with and wanted to expand our knowledge base and abilities on. We also sought to tie the project in with a major issue we experienced in our day to day studies.

Indoor Localisation, or Indoor Positioning Systems, are ways of tracking objects and people while indoors. While technology like GPS is fairly effective at pinpointing locations that are outside, it becomes much more difficult for them to resolve a location that is indoors or otherwise obscured from direct view of the sky. There are a number of different methods that exist for indoor tracking, including the Wi-Fi-based Positioning System (WPS), which uses access point signal strength, and Bluetooth, which is more concerned with general proximity of connected devices but has some potential for small-scale tracking of nearby objects. Indoor Localisation is a very complex problem, and while we did not set out to solve it we did wish to see if we could come up with an alternative angle on the problem.

Machine Learning is a subcategory of Artificial Intelligence that uses data and algorithms to create predictions that can be tuned and altered to what is needed. We had prior experience working with creating Machine Learning systems in some of our

modules and in our projects in third year. We were certain that Machine Learning is a field we could once again expand into and utilise to achieve something technically complex and useful.

A particular issue we've experienced in our time at DCU is in deciding which computer laboratory we should go to for work and study. There are a total of eight labs available for general use in the McNulty building on DCU's Glasnevin Campus, which is a lot to choose from. In addition to this, any room could be booked for classes at given times, which would be distracting to someone not part of that class if they are even permitted to stay in the room for the duration. Outside of being booked, these computer labs are often a place for computing students to meet up and converse and collaborate with each other about their classes and projects. While this is certainly a positive aspect of the labs, sometimes a person may desire less distractions to focus on some work, or they may be sensitive to noise. All of this is a lot of information to take into account when choosing a room to go to and culminated in us feeling like there is some need for the ability to see at a glance how congested each lab is, in order to make a more informed decision of which room we would prefer to attend for a given amount of time.

For these reasons, we were inspired to take up this project we called LabLocator in order to make it possible for us to see the general "business" of each lab and check quickly and easily what was the most optimal lab to go to for some quiet study.

2. Research

2.1. Indoor Localisation

Our research into localisation mainly focused on what possible methods to determine location were available to us, and how we could apply them within the context of searching for a computer lab within the McNulty building, such as via using WiFi access points or through Bluetooth. As indoor localisation is still an unsolved problem, it is difficult to find any one method that would truly be suitable for most indoor settings, which is why we had no intention of providing a true solution, and rather provide a possible use case on this issue on a small scale basis.

2.2. Feasibility

Our research began with determining the feasibility of making this application, with the main question being: what information about the area can our phone access without requiring a GPS? We found out that phones are able to see various WiFi access points, and that with each lab we went to, the visible access points would vary, allowing us to develop a prototype of the application that was able to directly scan and display DCU-Guest-WiFi and eduroam access points that would available to the device. This prototype showed that once we were able to collect and store the WiFi scans, we would be able to use that data to train a model to determine the general location of a user based on visible access points, and create a possibility to deal with the problem of indoor localisation on a small scale.

2.3. Machine Learning

As part of the basis for the application, we needed for devices to be able to determine their general location and send that information back to the server to determine the current capacity for a room, which required the use of machine learning to be able to view the visible access points and determine what room the device may be in. The format we chose to break down our data was to reformat it so that the columns were all possible access points, and the rows are each room and each value displaying “1” and “0”, to signify what access points are visible or not, and choose an algorithm that is able to predict the room based on the format of the data.

We chose to use K-Nearest Neighbour (KNN) as it can be used for classification of defined groups, such as image classification with the MNIST dataset, and required preprocessing the data into vectors of numbers, which suited our dataset.

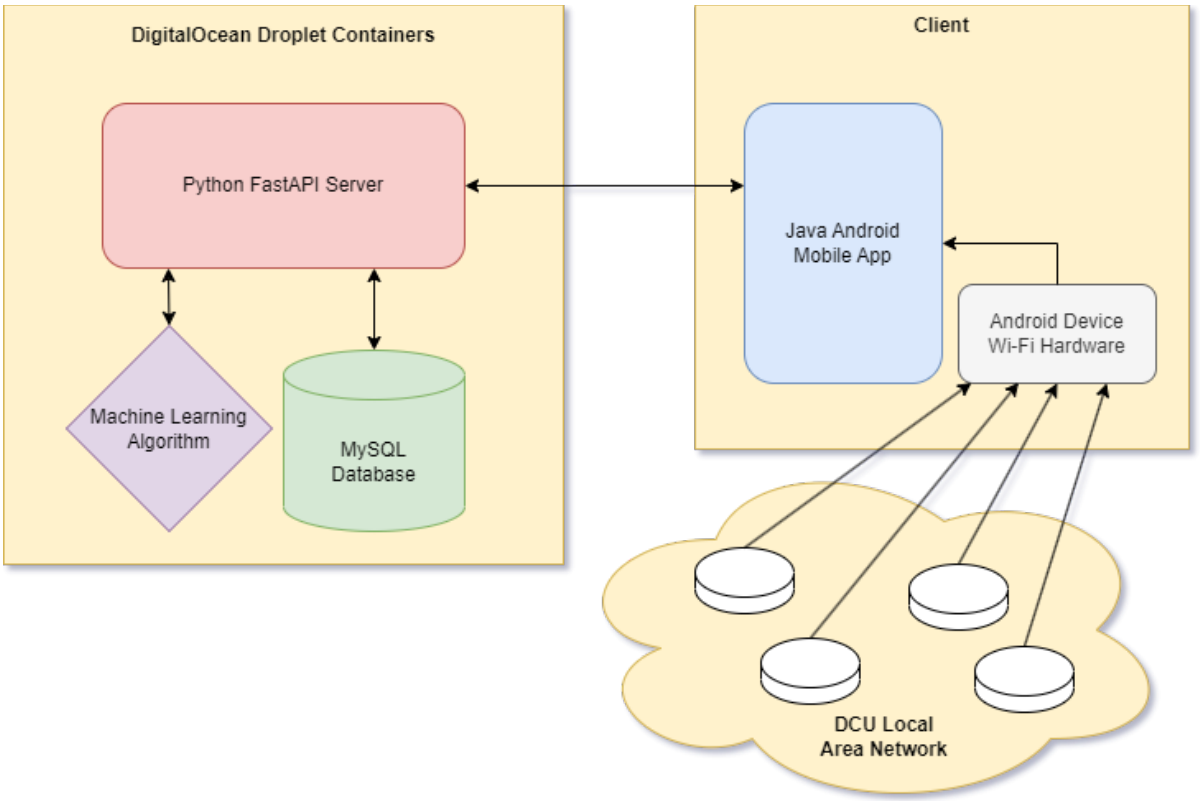
At a later point, we chose to test KNN against other classification algorithms, such as Support Vector Classification (SVC), Linear SVC, and Random Forests, to determine whether KNN was best suited for our purposes by measuring the resulting score of the test dataset against KNN, and found that the differences between each score and KNN was too marginal to justify a change to another classifier.

2.4. Android Studio

We had to choose between building the application using Flutter or Android Studio, or as a web application. As a web application was not indicative of what we wanted our app to do, which was to retrieve results once a user was in a lab via their device, which the web application would have difficulty dealing with certain aspects of the process, it was quickly scrapped. Flutter is a framework that can be used to make Android, iOS and web applications, although it is not specialised for our needs and so we ultimately chose to design LabLocator as an Android application using Android Studio, as we both had Android phones, and both had a good understanding of Java, which is one of the languages that can be used to build Android apps.

3. Design

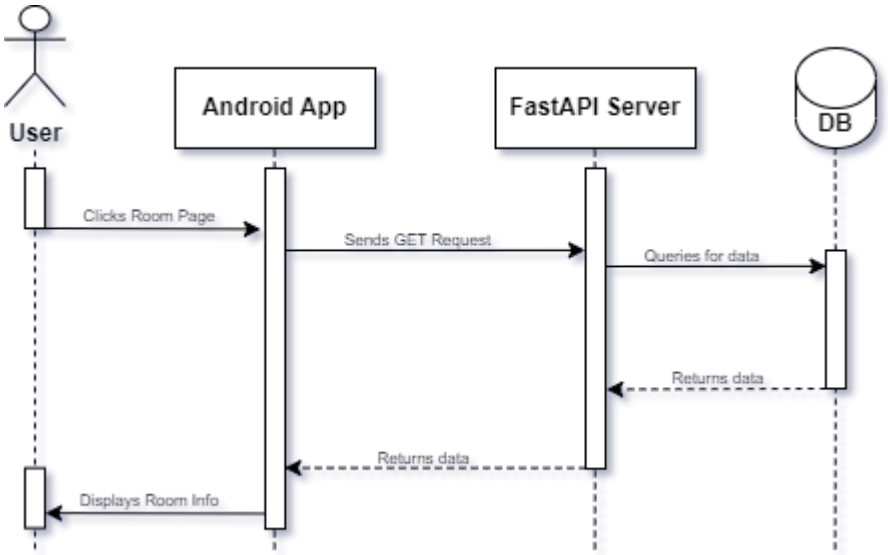
3.1. System Architecture



The system architecture shows at a high level how the system is built. Users interact with the system through the Android Application written in Java, which they would install on their personal Android smartphone device. When a user is present on the DCU Glasnevin Campus, their devices should be able to view the various routers that are part of the eduroam and DCU Guest WiFi networks. After collecting the data from each wifi access point, the application sends it over to the python-FastAPI based web server that is hosted on DigitalOcean. The endpoints integrate with a MySQL database that is also hosted on DigitalOcean, and a SciKit-Learn based machine learning programme that we use to predict the device's location. The MySQL database stores the location log of each device and this is used by the FastAPI app to determine the population of users per room.

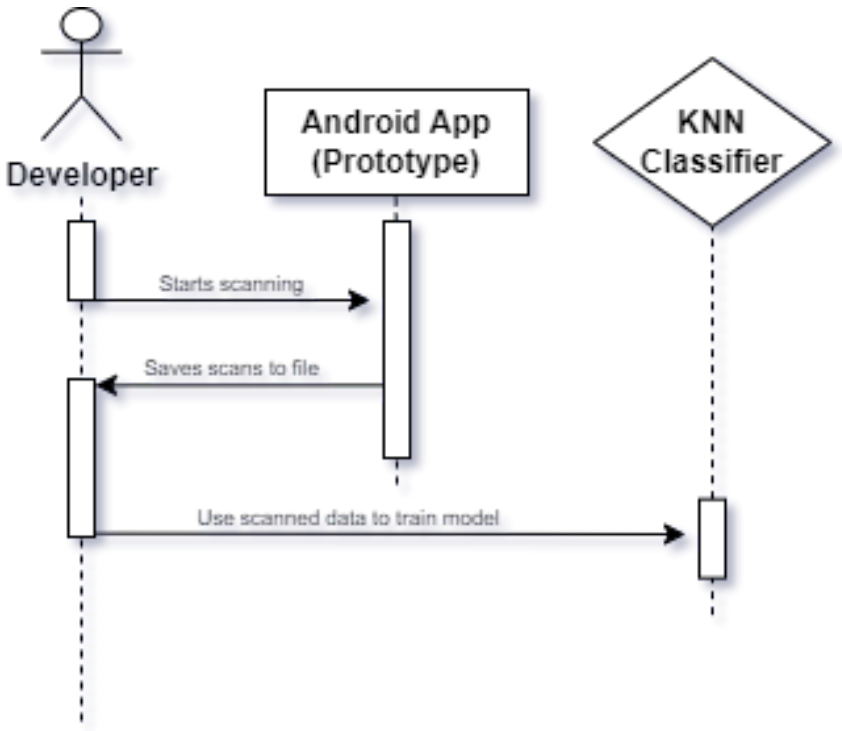
3.2. Components

3.2.1. Backend Server



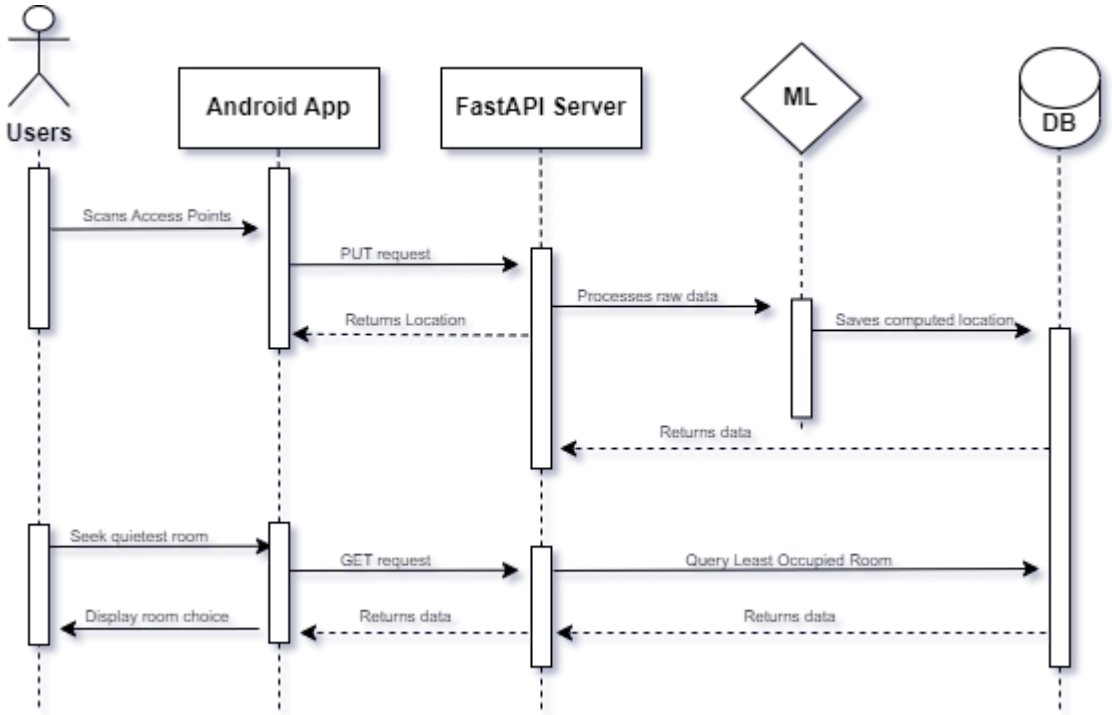
Here we see the intended events that occur when a user wants to retrieve information about a particular room while using the android application. When a user taps into the page of an individual room in order to get information about it, the application will send a GET request to the FastAPI backend. This in turn, queries the database for information which is sent back along the chain of connections until the application displays it on screen for the user to read.

3.2.2. Predicting Locations



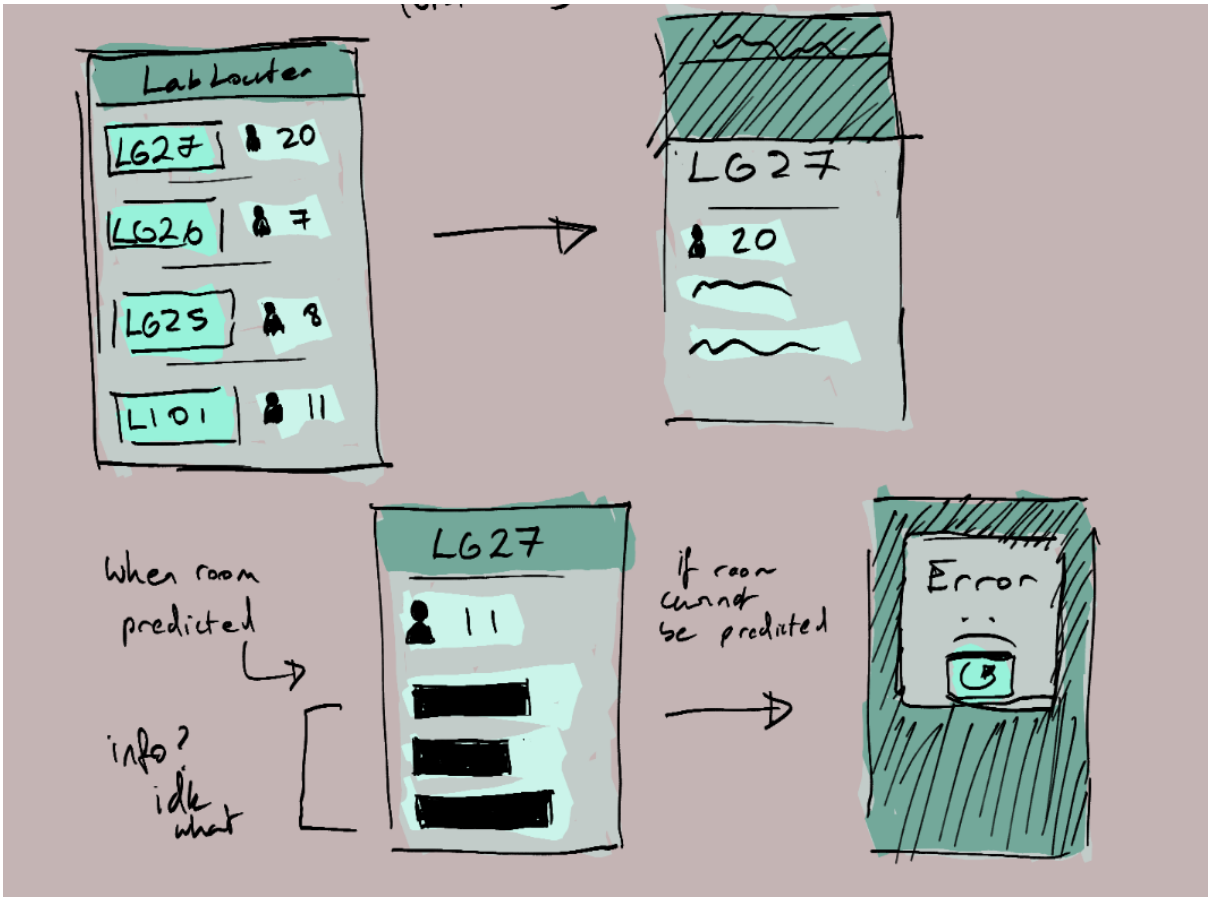
Above shows our intended methodology for collecting Wi-Fi access point data. As developers we would use our Android device's hardware to acquire lists of access points in accordance with whatever room we were currently present in. Using this, we could develop a table of data to feed into a KNN Classifier. We would be able to use this process to develop our machine learning algorithm, and we could easily recount these steps to add more rooms or reinforce existing room data when we see fit.

3.2.3. Tracking Room Congestion



As previously depicted, we intend to acquire and store the locations of app users. While this data is stored, one of the primary functions we desire is to then use all of this room location data to suggest to all users what might be the quietest and most optimal laboratory room for them to attend and do some quiet study. With the location data stored in the database, we can process the possibilities and show a user where they might intend to want to go to.

3.2.4. User Interface



We created mockup designs to support our ideas for what we wanted the mobile application to look like. This would be a vital piece of the project as the end result of the app design becomes what all users interact with. We wanted to create a good user experience, with an interface that makes intuitive sense to the user. As such we saw it as an important step to try and set out a design philosophy beforehand. We then used the mockup designs we created as a baseline when developing the real app interface.

4. Implementation

4.1. Training Data Collector

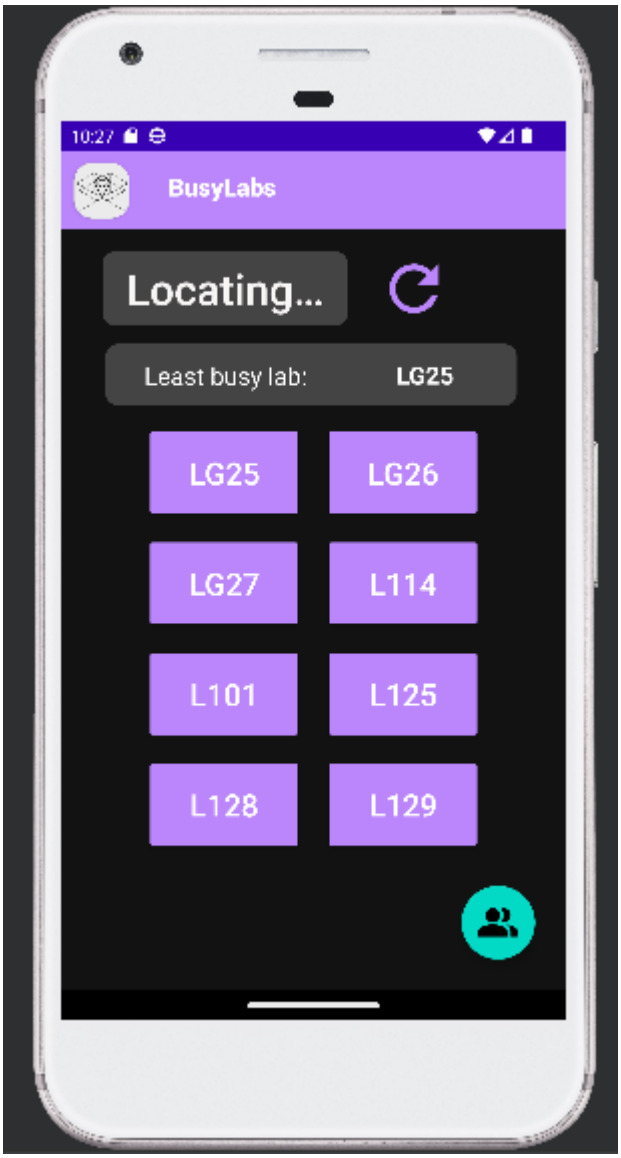
We spent some time researching the feasibility of a system that attempts to predict a user's relative location based on what WiFi access points were visible to it. Once we were satisfied that we could achieve this, we set about collecting data for building a Machine Learning model. We achieved this directly through the Android application we were developing, as it is our intent to use the very same methods to find the user's location later on.

Location	Time	SSID	BSSID	RSSI	AP List
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	eduroam;9c:8c:d8:a1:1d:21;-78|DCU-Guest-WiFi;9c:8c:d8:a1:1d:20;-78|DCU-Guest-WiFi;9c:8c:d8:a1:1d:30;-82|eduroam;9c:8c:d8:a1:1d:31;-82|DCU-Guest-WiFi;70:3
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-46|eduroam;70:3a:0e:62:e0:41;-46|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-53|eduroam;70:3a:0e:62:e0:51;-53|DCU-Guest-WiFi;cc:
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-41|eduroam;70:3a:0e:62:e0:41;-42|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-54|eduroam;70:3a:0e:62:e0:51;-54|DCU-Guest-WiFi;24:
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-44|eduroam;70:3a:0e:62:e0:41;-42|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-52|eduroam;70:3a:0e:62:e0:51;-52|DCU-Guest-WiFi;cc:
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-41|eduroam;70:3a:0e:62:e0:41;-42|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-55|eduroam;70:3a:0e:62:e0:51;-56|eduroam;24:f2:7f:ac
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-43|eduroam;70:3a:0e:62:e0:41;-43|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-55|eduroam;70:3a:0e:62:e0:51;-55|eduroam;24:f2:7f:ac
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-41|eduroam;70:3a:0e:62:e0:41;-41|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-54|eduroam;70:3a:0e:62:e0:51;-54|DCU-Guest-WiFi;cc:
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-43|eduroam;70:3a:0e:62:e0:41;-41|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-54|eduroam;70:3a:0e:62:e0:51;-54|eduroam;24:f2:7f:ac
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-42|eduroam;70:3a:0e:62:e0:41;-41|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-57|eduroam;70:3a:0e:62:e0:51;-57|eduroam;24:f2:7f:ac
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-47|eduroam;70:3a:0e:62:e0:41;-47|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-48|eduroam;70:3a:0e:62:e0:51;-48|DCU-Guest-WiFi;cc:
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	eduroam;70:3a:0e:62:e0:41;-60|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-66|eduroam;70:3a:0e:62:e0:51;-66|DCU-Guest-WiFi;24:f2:7f:ac:d4:e0;-73|DCU-Guest-WiFi;24:f
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-62|eduroam;70:3a:0e:62:e0:41;-62|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-65|eduroam;70:3a:0e:62:e0:51;-64|eduroam;cc:88:c7:8
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-62|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-59|eduroam;70:3a:0e:62:e0:51;-58|eduroam;24:f2:7f:ac:d4:e1;-72|eduroam;cc:88:c7:80
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-64|eduroam;70:3a:0e:62:e0:41;-65|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-57|eduroam;70:3a:0e:62:e0:51;-58|eduroam;24:f2:7f:ac
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-61|eduroam;70:3a:0e:62:e0:41;-61|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-61|eduroam;70:3a:0e:62:e0:51;-61|eduroam;24:f2:7f:ac
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-60|eduroam;70:3a:0e:62:e0:41;-61|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-61|eduroam;70:3a:0e:62:e0:51;-61|eduroam;c8:b5:ad:4
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-59|eduroam;70:3a:0e:62:e0:41;-60|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-60|eduroam;70:3a:0e:62:e0:51;-61|eduroam;cc:88:c7:8
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-64|eduroam;70:3a:0e:62:e0:41;-63|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-64|eduroam;70:3a:0e:62:e0:51;-64|DCU-Guest-WiFi;cc:
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-61|eduroam;70:3a:0e:62:e0:41;-60|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-49|eduroam;70:3a:0e:62:e0:51;-49|eduroam;cc:88:c7:8
LG25	1.68E+09	eduroam	70:3a:0e:6	-66	DCU-Guest-WiFi;70:3a:0e:62:e0:40;-40|eduroam;70:3a:0e:62:e0:41;-49|DCU-Guest-WiFi;70:3a:0e:62:e0:50;-43|eduroam;70:3a:0e:62:e0:51;-43|eduroam;24:f2:7f:ac

Above is a sample output from our data collection. You can see the various fields we collected from at the top. We set the room we were currently taking scans from, in the case of the sample data shown this is LG25. The application outputs the current time, the SSID, BSSID, and RSSI of the network they are currently connected to. After this, the APList is a representation of every other Access Point visible to the device that it is not connected to, including alternative Access Points that make up the same network that the device is currently connected to. In order to keep our data consistent, we limited the valid Access Points to the "eduroam" and "DCU-Guest-WiFi" networks as these are part of the University's WiFi network and we know we can depend on them to be relatively static, and we did not want to create any biases by including things like Mobile Hotspots that can be deployed in any location. Below is an example of what that scanned data gets processed into to become training data for the Machine Learning model. The table contains a column for every possible access point that could be observed by a user's device. If an access point is visible from a particular room during a scan, it gets marked with a "1" for that field, otherwise it receives a "0" instead.

	Room	20:a6:cd:i	24:f2:7f:a	c8:b5:ad:(	24:f2:7f:a	20:a6:cd:!	c8:b5:ad:(	20:a6:cd:(	c8:b5:ad:(	34:fc:b9:1	c8:b5:ad:(	c8:b5:ad:(	34:fc:b9:1	24:f2:7f:a	70:3a:0e:	24:f2:7f:a	20:a6:cd:!
0	LG25	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1
1	LG25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
2	LG25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
3	LG25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
4	LG25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
5	LG25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
6	LG25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
7	LG25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
8	LG25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
9	LG25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
10	LG25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
11	LG25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
12	LG25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
13	LG25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
14	LG25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
15	LG25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
16	LG25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1

4.2. Android Application

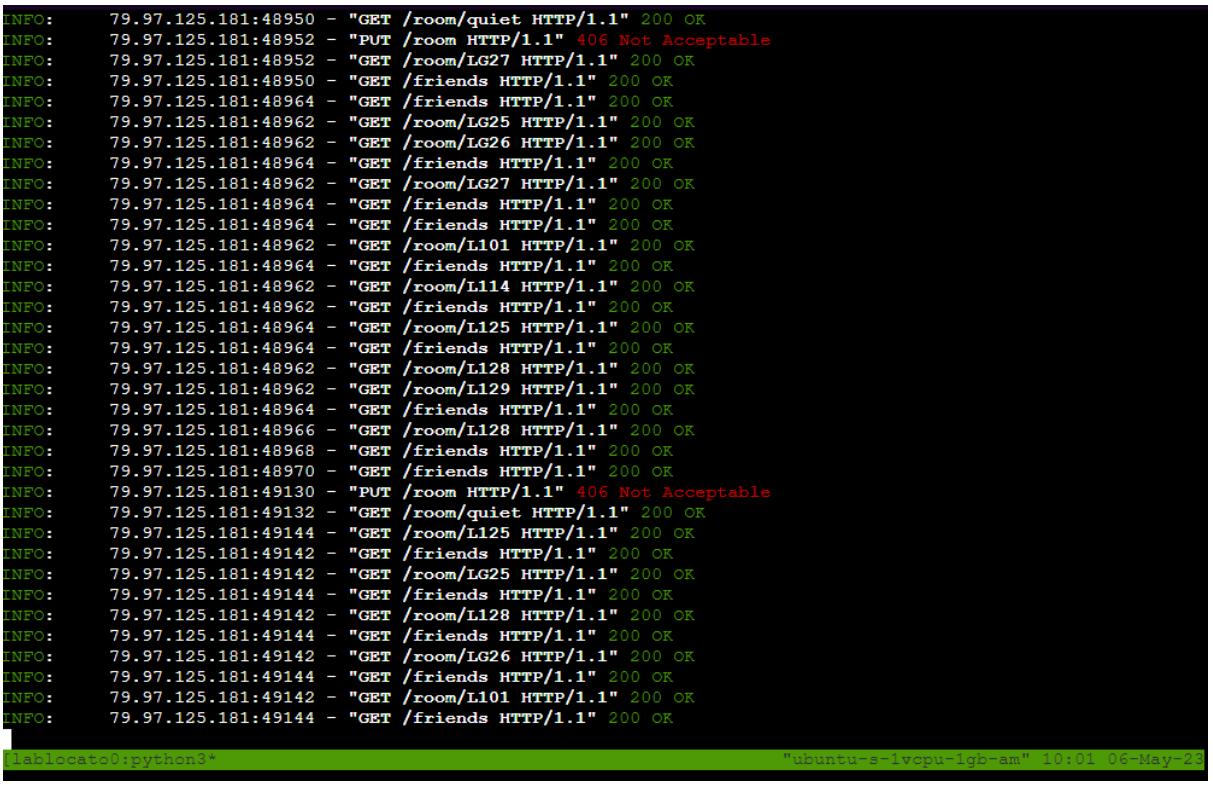


We used the Android Studio development environment to create our mobile application. This is Google's official integrated development environment for creating Android applications, and can be used to programme apps in either Java, Kotlin, or both. Application UI layout is made in XML and can be configured with a visual editor as well as a text editor. We used a

series of Activity models to portray our functionalities on several pages within the app. The initial scan of the user's location is run automatically, but the user can press a refresh button to update this if they wish. The user can also press buttons to navigate through information about each individual room, where the population for those rooms will be displayed alongside a quick map to show them how to find that room. Lastly, there is a "Friends" button in the bottom right corner that the user can use to view their friends list, along with an indication of what room those friends are currently in, if they are in a lab at that moment.

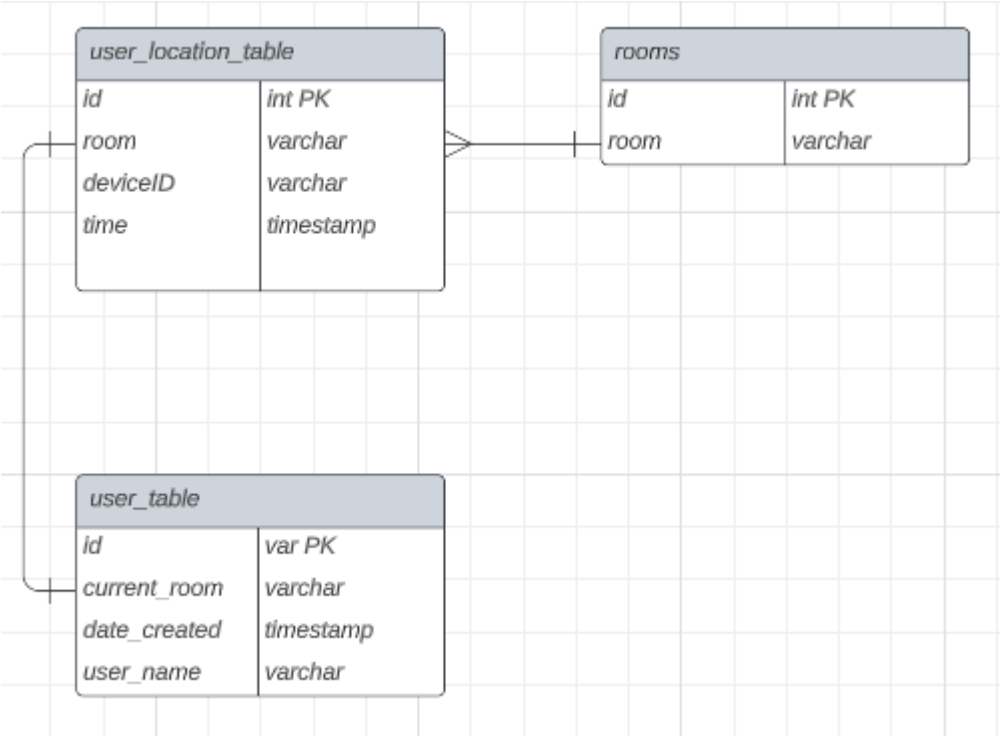
4.3. Backend API

We hosted our backend on the DigitalOcean platform, which allowed us to spin up virtual machines in containers called "Droplets" quickly and easily. We used a Ubuntu VM to run our FastAPI python client. Since this VM was configured for networking, we had a public facing static IP address that we could point our application and testing materials at, and receive appropriate responses as needed. We used "tmux" to allow the client to run indefinitely without needing developer control, but we could update the application as needed by pulling the changes made in the repo. Below is a screenshot of the terminal shell, running FastAPI and serving the incoming requests.



4.4. Database

We also hosted our MySQL database on DigitalOcean. This had some major security bonuses, as we could restrict access to the database to only the Droplet running our FastAPI client. That way there could be no mishandling of the tables and data. The database was used to store information about user's locations and feed it back when necessary to display the current population of any given room.


user_location_table
id
int PK
room
varchar
deviceID
varchar
time
timestamp
rooms
id
int PK
room
varchar
user_table
id
var PK
current_room
varchar
date_created
timestamp
user_name
varchar

5. Testing

5.1. Background

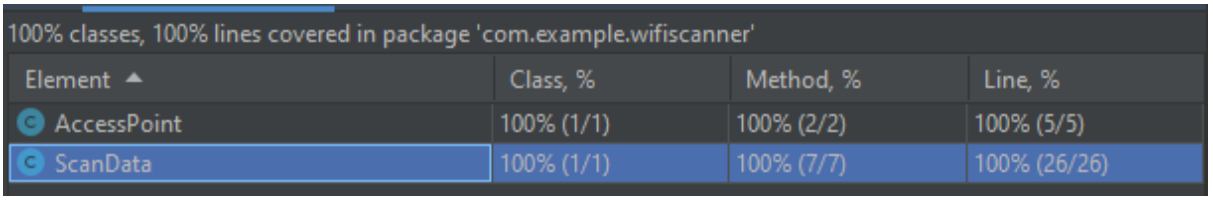
It was important for us to have a good standard of code to be able to maintain our project throughout the time of development. We wanted to be able to refactor and expand upon our code whenever necessary and without getting into issues. For these reasons we maintained several testing methods and development standards as we worked on this project.

5.2. Unit Testing

We employed some unit testing to ensure that parts of our code would maintain the same functionality throughout the entire process of development on the application. This came in two separate parts to test the different compartments of the project. We employed unit testing on both the python based FastAPI backend and the Java based Android frontend. For the backend, we made use of the pytest library to test our API endpoints. It was vital that each endpoint maintained the same outputs given particular inputs so that there would never be any mishaps down the line.

Element	Statistics, %
2023-ca400-clarke42-prosyae2	0% files, 70% lines covered
src	100% files, 70% lines cove...
backend	100% files, 70% lines cove...
main.py	83% lines covered
model.py	36% lines covered
test_main.py	89% lines covered

Above is the coverage report for our python scripts. Our unit testing was focused primarily on the “main.py” script which housed our API endpoint functionalities, these we prioritised because of its crucial nature. We achieved an overall average coverage of 70% of lines across the backend. We also created a separate Test Database so we could perform tests on queries without interfering with our live data.



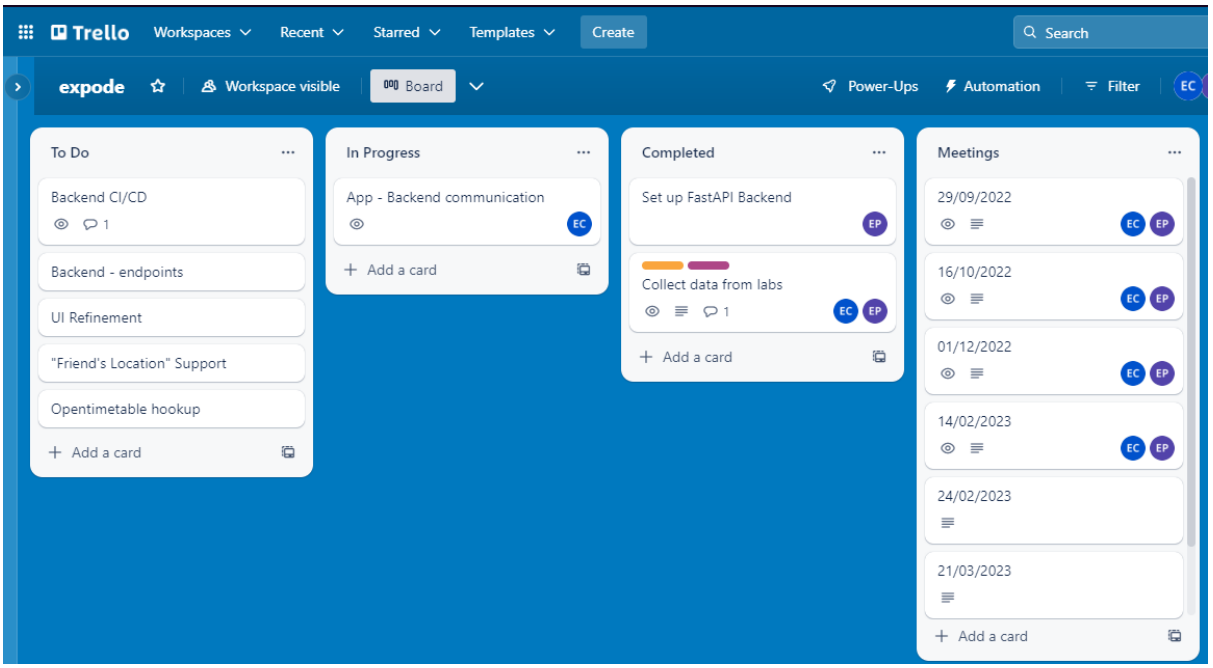
100% classes, 100% lines covered in package 'com.example.wifiscanner'

Element	Class, %	Method, %	Line, %
AccessPoint	100% (1/1)	100% (2/2)	100% (5/5)
ScanData	100% (1/1)	100% (7/7)	100% (26/26)

In our Android application, we used JUnit and Mockito to perform our unit tests. We did not perform any unit testing on the Activity classes of our application as these were better suited for testing with other methods. However, with our functional Java

classes we did achieve 100% testing with the methods used for scanning and storing WiFi information, as can be seen above.

5.3. Project Management with Trello



We used a Trello board to manage all tasks pertaining to our project's development cycle. We treated each individual task like a user story. We would create a backlog of tasks and decide what should be worked on. Each task would move between the stages of "To Do", "In Progress" and "Completed" as necessary, and we would assign the tasks to either member of the group. We would have weekly check-ins with each other to ensure progress was going smoothly, and if anything needed to be re-assessed or changed. We also logged the various meetings we had with our supervisor. In addition to the regular development process and tasks, we would include bugs and issues that came up in this list of tasks on the Trello board. These bugs would be considered alongside all other tasks and fixes in accordance with their severity relative to the importance of all the tasks we have in the backlog.

5.4. Git Branches and Code Reviews

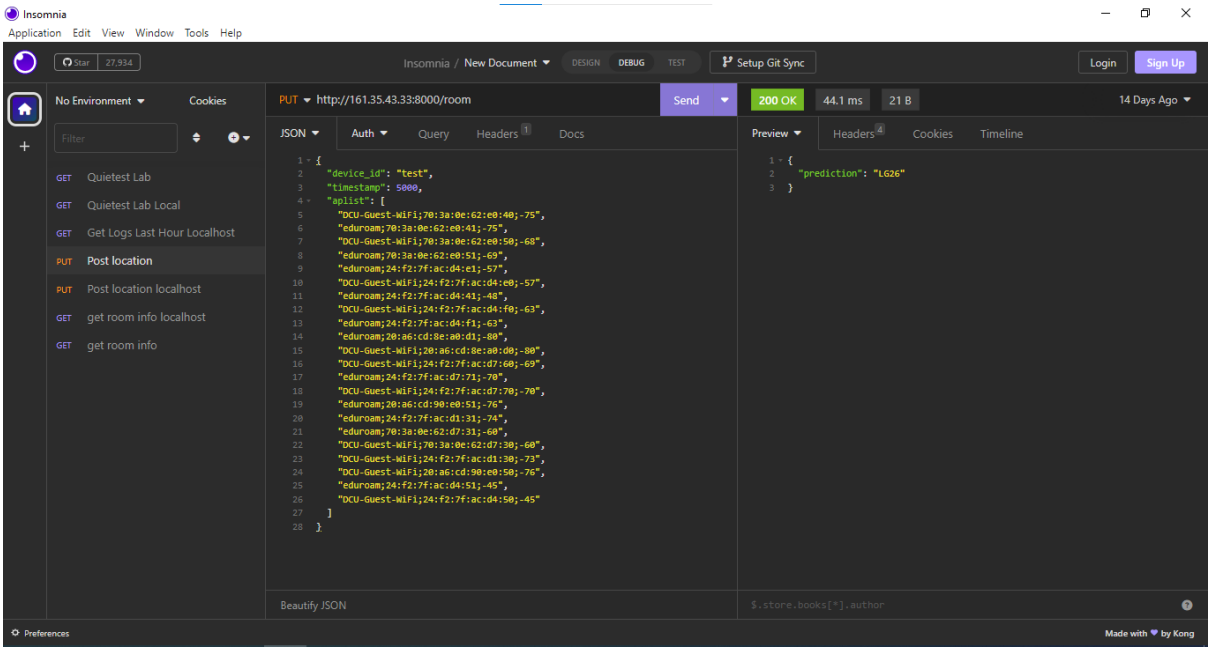
When working on any given task or bug, we employed good git etiquette by making sure to work on individual goals in their own branches. By doing this, we never get mixed up in what we are working on and submit several convoluted updates at once and obfuscate the development process. We would commit code regularly and sensibly, as we made progress on our code. When we had checked our functionality to ensure that we had completed a given task, we would push and submit a merge request through Gitlab. We would make sure to always assign each other to review the code that was being submitted. This served as a great way to review each other's code regularly and give feedback when necessary. Once any issues and suggestions were cleared with the submitted code, we would approve the merge and allow it to go through to the master branch.

5.5. Manual Testing

With every new development we made, we made sure to review both code and physical functionality of the application and the backend before allowing a merge to be completed. This involved going through the application on both the built-in Android emulator that is packaged with Android Studio, and testing our endpoints with Insomnia.

Android Studio's emulator allows us to compile and load our application without needing a physical device hooked up and ready to accept and run an in-development test application. Using such a method we can easily run through our UI functionality and confirm that it has not been altered where it has not intended to be. We can change Android versions and screen sizes to experiment with how the UI appears on many different devices, without actually needing to own them all.

Insomnia is an open source desktop application that provides functionality for making HTTP requests, which is very useful for development and testing. We can use it to make requests to endpoints by supplying a web address, and choosing the appropriate request protocol. Insomnia also supports the ability to supply a JSON payload on a request, which is vital for testing out more complex endpoints.



5.6. Anonymous User Feedback

We obtained ethical approval to reach out to students in the college and ask them to leave anonymous feedback on our application. We invited students in the school of computing to test out the functionality of our application. They were given access to our Android application and asked to use it for as long as they desired to. Once they had completed their utilisation of the application, we asked them to fill out a review

form so that we could obtain knowledge on what the general consensus of our application design was like. Through this feedback we were able to get great insights into what people liked about LabLocator. But more importantly, we also learned about what people would have liked to have changed about their experience. We took all of the feedback into consideration and used it to alter parts of the app's functionality and the placement of some UI elements so that we would end up with a more effective user experience.

6. Sample Code

6.1. Scanning Access Points

```java
@Override
public void run() {
    int i = 0;
    while (this.keepRunning) {

        if (++i > MAX_ITERATIONS) {
            this.keepRunning = false;
            break;
        }

        ScanData scanData = this.getData();

        if (!scanData.apList.isEmpty()) {
            String apDataString = scanData.toString();
            writeToFile(apDataString);
            mainActivity.updateTextViewCurrentLocation(apDataString);
        }

        try {
            Thread.sleep(30000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

The above function is a method of the ScanThread class, which extends Java's Thread object. It is an override of Thread's run() method, for running a forked thread. This thread controls the acquisition of Wi-Fi Access Point data. It is a thread that runs until we call it to stop, or it reaches the maximum number of iterations in the case of it being left to run for too long. It calls our ScanData object to collect the access point information, which is collected using Android's net.wifi.WifiManager library. This collects the information from the OS itself. Once we have this, we write the information to a file named "output_scan.csv". Android's OS has a built-in limitation that only lets us ping for a new scan once every thirty seconds, this is a feature intended to help save on battery life. We worked around this by leaving our phones running the thread while we did other work in the laboratories, making good use of the time while also collecting data.

6.2. Converting Scan Data to ML Training data

```python
def transform_data(data):
    locations = data['Location']
    ap_list = data['AP List'].values.tolist()

    ap_list_split = [i.split('|') for i in ap_list]

    bssid_list = []
    for room in ap_list_split:
        bssid_list.append(get_bssid_list(room))

    bssid_columns = get_unique_bssids(bssid_list)

    zipped_data = list(zip(locations, bssid_list))

    return create_dataframe(zipped_data, bssid_columns)
```

After acquiring the data in the form of a file named “output_scan.csv”, we pass it over to our python script to parse the information within it. It was important to automate this phase because we were collecting large amounts of data. We work with the strings in the file to get the information we need, extracting the lists of SSIDs and BSSIDs, before passing it onto the next stage of the process in training our machine learning model.

6.3. Create Dataframe and Model

```python
# format the data to train the model
def create_dataframe(data, columns):
    ap_visibility = []
    # create dictionary for each room containing visible and non visible access points
    for room in data:
        ap_dict = {access_point: 1 for access_point in room[1]}
        ap_visibility.append(ap_dict)

        for access_point in columns:
            if access_point not in ap_dict:
                ap_dict[access_point] = 0

    concatted = []
    # compile rooms and visible access points to dataframe
    for i in range(len(ap_visibility)):
        room = pd.Series({'Room': data[i][0]})
        aps = pd.Series(ap_visibility[i])
        concatted.append(pd.concat([room, aps]))

    df = pd.DataFrame(concatted, columns=['Room'] + columns)

    df.to_csv("ml_data.csv")

    return df
```

The create_dataframe() method formats all the data to be packaged in a way that can be read by our K-Nearest Neighbours Classifier. The individual laboratory room names are iterated through, and for each one of them we fill in a 1 if that room has a scan where a certain BSSID column would be found there from a scan, and a 0 if it would not be. All of this is concatenated into a list and fed into a DataFrame object. The DataFrame object is then exported to CSV format and returned.

```python
def create_model(data):
    data_dropped = data.drop('Room', axis=1)

    df_feat = pd.DataFrame(data_dropped, columns=data.columns[1:])

    X = df_feat
    y = data['Room']

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=139)

    knn = KNeighborsClassifier(n_neighbors=13)
    knn.fit(X_train, y_train)

    return knn
```

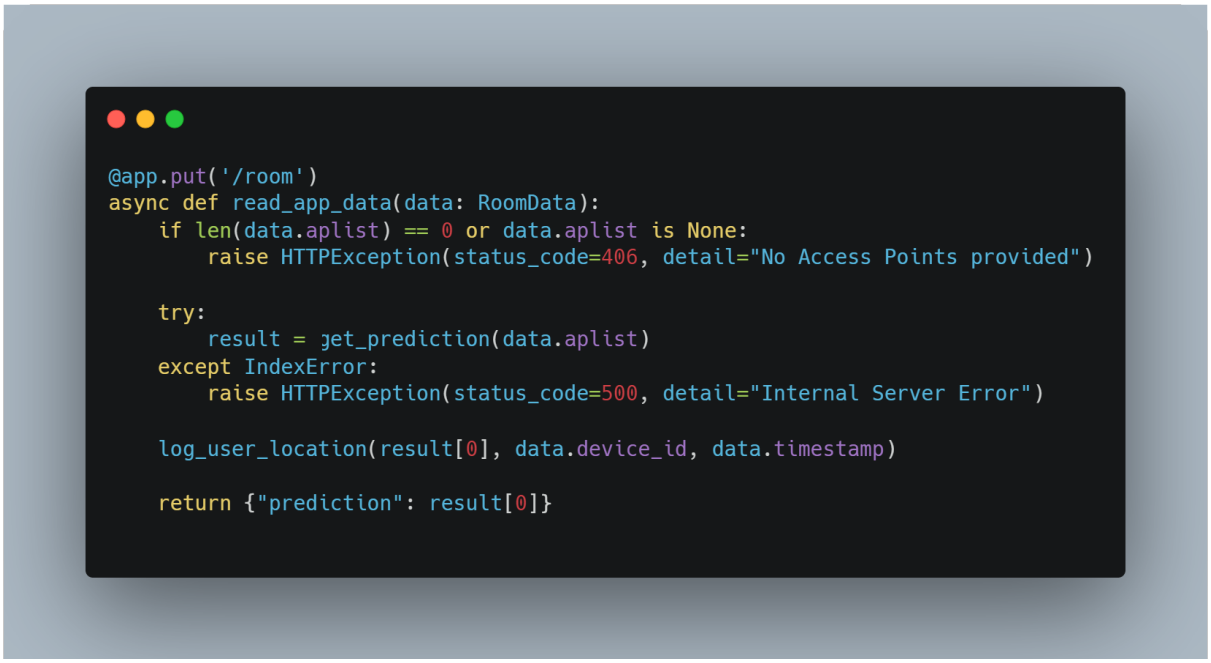
Once we have our data collected, processed, and formatted from our previous methods we can run the actual creation of our machine learning model. We run through Sci-Kit Learn to create our classifier model. With all the data previously gathered we easily train our model on all of it to create a reliable system for predicting our room locations.

6.4. Requesting a Room Prediction - Frontend

```
public void updateUserCurrentRoom(ScanData scanData) throws IOException, JSONException {
    String ROOM_REQUEST_URL = "http://161.35.43.33:8000/room";
    List<String> aplist = scanData.toApListString();

    JSONObject jsonBody = new JSONObject();
    jsonBody.put("device_id", this.androidId);
    jsonBody.put("timestamp", System.currentTimeMillis() / 1000);

    JSONArray jsonArray = new JSONArray();
    for (int i = 0; i < aplist.size(); i++) {
        jsonArray.put(aplist.get(i));
    }
    jsonBody.put("aplist", jsonArray);

    JsonObjectRequest request = new JsonObjectRequest(
            Request.Method.PUT,
            ROOM_REQUEST_URL,
            jsonBody,
            response -> {
                try {
                        String room = response.getString("prediction");

                        updateTextViewCurrentLocation(room);
                        this.currentRoom = room;
                } catch (JSONException e) {
                    e.printStackTrace();
                }
            },
            Throwable::printStackTrace
    );

    Volley.newRequestQueue(getApplicationContext()).add(request);
}
```

From our end-user oriented mobile application, we make the acquisition of a room prediction an automatic process. It happens as soon as the user opens the app and allows for location permissions to be used. To collect a set of WiFi Access Points, we use the exact same process as we used before to collect machine learning data. But this time we gather a singular set of Access Points consisting of whatever “eduroam” and “DCU-Guest-WiFi” network signals the android device can see at that given moment. We then use the Volley library to send a HTTP PUT request to our backend, with the access points in tow as a JSON payload. The request is an asynchronous process, so it does not interfere with the user’s ability to use the application. Once the request is complete, it should successfully return a room prediction to the user given they are actually residing in a lab room at that moment in time. Once it does, it updates a TextView object representing part of the visual display in the app’s UI, notifying the user that it has successfully predicted their location.

6.5. Requesting a Room Prediction - Backend



```python
@app.put('/room')
async def read_app_data(data: RoomData):
    if len(data.aplist) == 0 or data.aplist is None:
        raise HTTPException(status_code=406, detail="No Access Points provided")

    try:
        result = get_prediction(data.aplist)
    except IndexError:
        raise HTTPException(status_code=500, detail="Internal Server Error")

    log_user_location(result[0], data.device_id, data.timestamp)

    return {"prediction": result[0]}
```

The above python code shows the FastAPI endpoint where the backend deals with a request for a room prediction. It has some validation code to ensure that a valid JSON body has been provided, and if not it will return a 406 - Not Acceptable code. In the event of a successful prediction, the room code will instead be returned as a JSON object to the recipient. And the user's location, device ID, and the current time will all be logged into the MySQL database.

6.6. Getting Information About Rooms

```python
@app.get("/room/{room_id}")
async def get_room_data(room_id):
    if room_id == "quiet":
        room_data = get_all_devices_this_hour()
        if not room_data:
            return "LG25"
        return least_populated_room(room_data)
    room_data = get_device_locations_in_room_this_hour(room_id)
    return room_population(room_data, room_id)


def get_device_locations_in_room_this_hour(room):
    ping_db()
    sql = """
    SELECT id, room, deviceID, time
FROM lablocator_db.user_location_table
WHERE `time` >= DATE_SUB(NOW(), INTERVAL 1 HOUR) AND room = %s
ORDER BY time
;
    """
    db_cursor.execute(sql, [room])

    result = db_cursor.fetchall()
    return result
```

When the user's device running our application seeks information about the population of rooms that exist presently, it makes a call to another endpoint in our FastAPI server. This endpoint can check for individual rooms, or it can query for the quietest room at the current time. To get the current quietest room, it runs a query to the MySQL database to get each room's current population, and takes the room with the lowest number value. Other than this quiet parameter, the endpoint can query for individual rooms. Displayed above also is the SQL query used to perform this action, in the get_device_locations_in_room_this_hour() method. The endpoint then returns the number of LabLocator users who have been logged in that room in the last hour. Users are differentiated by their Android device's unique hardware ID, and only the latest log from each individual device is considered valid.

7. Problems Solved

7.1. Predicting User Location

One of the first and most important problems to solve was to create a way for user location to be predicted by the application, as this would serve as the basis for the application's function as well as be important for collecting the data we would need to use to train our ML model. We chose to do this via creating a prototype of the app that was able to scan the current location of the device and output the WiFi access point scans. We found that a mobile device is able to view and output the current visible WiFi access points and used that to create a model using a classifier such as KNN, as it is able to work from a limited amount of data, which was ideal within this context as we had to gather the data and the amount would be comparably smaller than a premade dataset.

The WiFi scans could not be used in their default state, as due to the nature of the classifier, it was required to have the data preprocessed into a vector that would be usable. As we needed the data to be shaped in a way that shows what access points are visible and are not, the data had to be transformed and processed by taking every unique access point that appears within the access point list portion of the scan and create a list of columns with the values within them showing the visibility of the room using either 1 or 0, while adding a column for "rooms" which shows which room these access points are visible from.

7.2. What Counts as a Person in the Labs?

When determining what counted as a person within the context of the application, and how to determine an approximation of the current occupancy of a lab, we decided to begin by creating a table within the database that would be used to log a user's location, device ID and what time the data is logged once the room they are located in is predicted, which can then be used to "count" people, or device locations, via the use of the */room/* endpoint to retrieve all unique device IDs located within a lab in the past hour and display that information once the user chooses to check the occupancy of a specific lab.

7.3. Suggesting the Most Optimal Room

It can be difficult for a student to find a room that is either quiet or not busy, and they may not even find one at all depending on how busy the McNulty is during a particular day, and going through the effort of checking multiple labs across two floors to find most of them full or near full can be considered a notable enough that is worth mitigating when possible. We decided to deal with this by using the */room/quiet* endpoint to find the quietest room over the past hour, and display this information on the main screen, ensuring that it is quickly accessible once a user opens the app.

7.4. Showing Lab Locations

The application had no visual indicators of where the labs could be located, which can be inconvenient for users unfamiliar with the layout of the McNulty building, and to deal with this we chose to create simple floor diagrams for the 1st and ground floor of the building, with the selected room being highlighted, giving a general idea of the room locations to the users.

8. Results

8.1. Working Predictions

We were able to create a model that can output predictions with an 89% accuracy score and use that in conjunction with the application to determine the approximate location of the device based on visible access points. Despite not expanding to use access point signal strengths to determine location, the model remains capable of being able to approximate which lab the user is located in, and display that this application can be potentially usable on a small scale.

8.2. Social Concept of App Shows Potential

The social concept of the application shows promise in incorporating a social aspect within a location-based application, and can be expanded on over time to incorporate more than the potential use case displayed within the application. The social aspect can be branched off into multiple directions, such as room-specific chat rooms, adding and removing friends, and friends being able to “signal” where they are or where they want to go.

8.3. Success of LabLocator

Through anonymous user feedback via online forms, we were able to determine issues and improve the app based on the received feedback, but we were able to measure overall user of the application. From our findings, we found that the anonymous feedback rated the application well from a total of 5 responses, which while not many responses, still reflects positively on the app as it can be considered to be well built and structured.

9. Future Work

9.1. Wider Campus Integration

As the app is designed to be used exclusively within a set series of labs within the McNulty, it is only functional within that specific area, yet there is potential to expand on this framework and create an app that covers more sections of the campus, which could prove to be beneficial for students and lecturers in determining current room capacities and may be useful for lecturers wanting to determine lab attendance.

9.2. OpenTimetable Integration

A future possibility for the app, and one of the initial proposed elements, was to integrate OpenTimetable and use it to determine if a lab was booked at a particular time slot around the current time indicated by the device. As OpenTimetable has no known available API and the only method of retrieving relevant information was through scraping the site and parsing the output, the concept was eventually scrapped, although we believe that if given more time to develop the app, then this would be a valuable component and would be beneficial to students who want check room availability for the labs yet do not want to use OpenTimetable to do so.

9.3. Refine Dataset

Another area we would like to refine if we work on this in the future is creating a more sophisticated version of the model, using the signal strength of visible access points to determine room position rather than simply using their current visibility, as access points can span across multiple rooms and affect the output of the model and signal strengths can be used as a way to mitigate this and ensure that the accuracy of the model improves.

9.4. Map Directions from Current Position

Another component that we would like to focus on in the future is creating a component that is able to create directions to a lab from the current position of a device, which would be beneficial to users who are not familiar with the McNulty and need help to find certain rooms. The current version of the app includes a basic map of the McNulty with the selected room being highlighted, although in the future we would be interested in expanding this feature to determine the directions to a lab based on a user’s current location.

9.5. Predicting Future Room Occupancy

A component that was part of the initial design of the app was the predictions of future occupancy of a room through gathering data of room occupancy over a period of time. This component was scrapped as it did not fit in with the design of the app, and required a large amount of data that we would not be able to produce with the time we had, and if we did choose to train this model, then a significant amount of time and effort would be put into simply refining it, which would be less than ideal for creating the rest of the application. If given more time, we would be able to create a model that could predict future occupancy and believe that it would useful to students who want to avoid rooms that may get full occupancy over a period of a few hours.