Buddy Walk Project Report

Introduction

The Buddy Walk project aimed to create a Flask application offering a buddy system with an integrated mapping tool that connects users to one another in the interest of safety when walking trips alone in the city. This report will outline the key planning, features and structure of the project as well as its outcomes and reflections on its usage or usability.

Background

Responding to a growing public mood of a lack of safety when walking in the city, 'Buddy Walk' was designed to allow a greater security - both real and sensed - to users during trips made after dark. After entering their details, users are able to input their planned location, route and start time. The application then connects them to another registered user taking the same route, along with their contact details and a planned meeting point that is limited to a 10 minute walking distance from their current location. Current mapping tools often offer route advice including main roads as a safety option. As opposed to highlighting the dangers of a quieter street, Buddy Walk aims to offer a more empowered solution. Users are not required to moderate their routines based on possible dangers but instead find safety in numbers.

Specifications

We have actively encouraged the aforementioned ethos to influence how we considered and applied both the technical and non-technical features of our system. When initially prioritising the requirements and specifications the MoSCoW method was used to decide how these would be divided between three project sprints. After careful analysis of these factors the requirements were consolidated and is summarised below.

Technical Requirements

- A 'users' database written in SQL to store the users' information as pulled from the frontend input fields and provide access to the location and time information needed for the find_buddy() routing functions.

- A 'find_buddy()' function: a function that matches two users together based on the proximity of their start and end locations along with their desired journey start times. This function pulls user data from the 'users' database.

- A routing function: a function that returns a route for the pair to take once matched. This uses data points already retrieved for the 'find_buddy()' function. The route is then returned to the user.

- A Python-SQL database connection: In the backend, the find_buddy() and routing functions require a connection to the 'users' database, as well as the API.

- An API: a self-written API to push outputs from our find_buddy() and routing functions to the user, and to pull inputs for start and end location and journey start time. API endpoints must include a landing page, input pages and an output page.

- Connection to a mapping API: for the functions mentioned above, to match buddies by distance and then return a suitable route to them will require a connection to a free mapping API. For the Buddy Walk project the Google Maps Distance Matrix, Google Maps Directions, and Google Maps Geocoding APIs were used.

- A simple front-end interface: for the experience of the user a simple front-end interface is required for the submission of their journey information.

Non-Technical Requirements

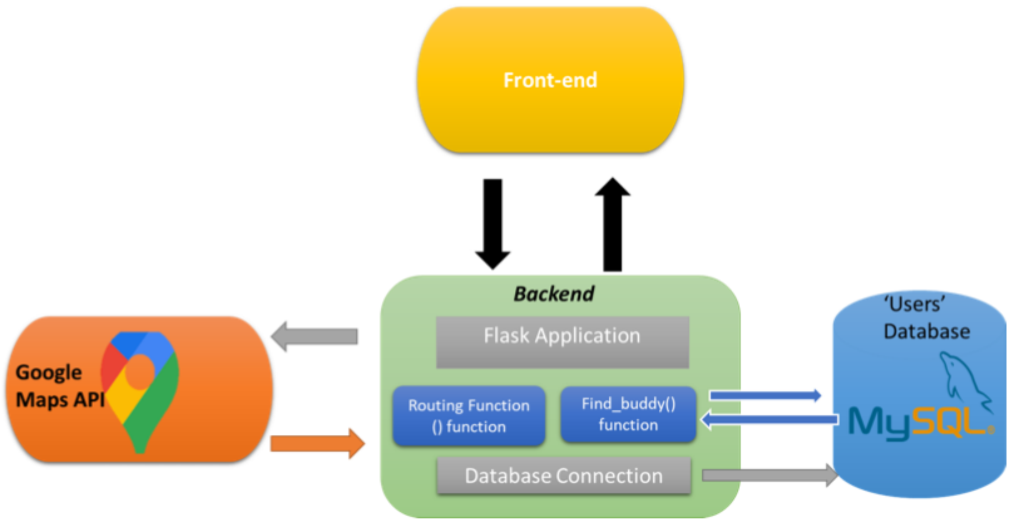
- Integrity: Users must have a sense of trust in the technology they are interacting with as it is responsible for their personal safety - the presentation of the application as well as its functionality and security should reflect this.

- The usability of the front-end input page should reflect the inclusive ethos of the app in being easy to read and return clear information to the user.

- Unit testing to prevent errors that would inhibit the ability of the app to allow greater safety for the user e.g. checking that the distance limit from the journey start location is under a 10 minute walk from the user.

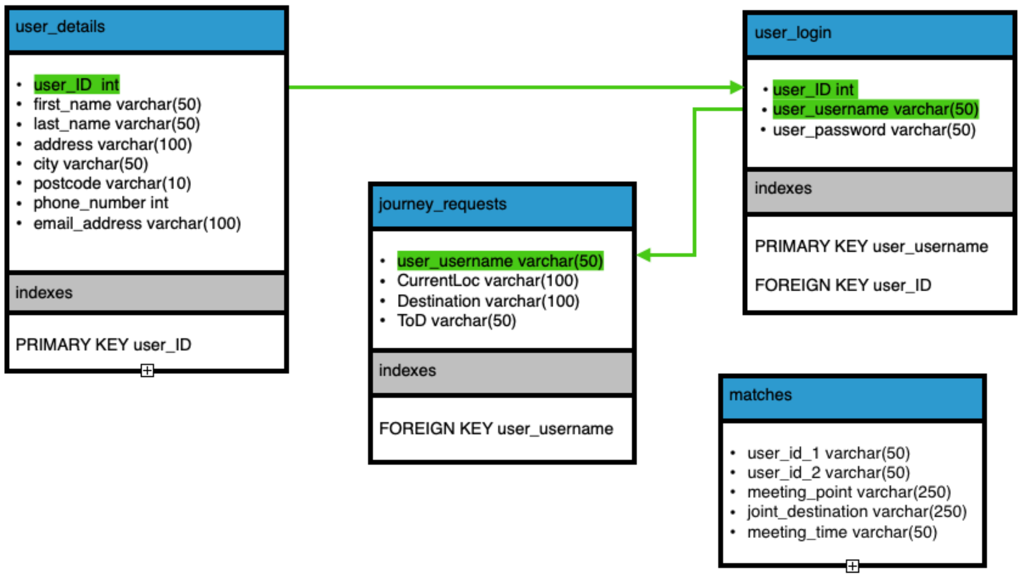
Architecture

Given the technical requirements mentioned above, the system that presented the best choice in terms of design architecture for the system was a client-server model. Once the user inputs their journey details through an internet server, there are two distinct processes that are called via the flask application and back end logic. Through connection with the 'users' SQL database a 'buddy' is returned based on proximity to the user via the 'find_buddy' function. The routing function then calls the Google Maps API to return a mapped route of the journey ahead and returns this to the user. This process is highlighted in the visualisation below:



Database Design

The design of the database was contingent on several prioritised needs of its use; the ability to hold the location of users, to do so securely as well as the need to input journey information from the live user. The user_details and user_login perform the needs of the former, with the user details stored separately to the other tables and accessible via a user ID number. A primary key is placed on this column to connect between the two tables. The journey requests table is required to store the current location, destination and time of day inputted by the user through the front-end of the application. It is then stored here for further analysis via the utils file. Here the user_username column connects via foreign key to the stored information in the user_login and user_details tables. The ERD or entity relationship diagram below displays further the relationships between the tables.



User Personas

In the interests of keeping the user and their needs as a priority when considering the design and specifications of our application, we created user personas to consolidate our aims and objectives. We then extracted key desirable features as user stories, and named them as priorities in the technical and non-technical functionalities of the project. Three examples of our chosen user personas are given below.

Name: Agnes McLeod
Age: 78
As an elderly person I want to be able to go home from the shops in the mornings without fearing that I'll be mugged because I'm walking by myself ever since my dear husband Alistair passed away 15 years ago. I want something that is easy to use that my grandchildren can explain to me easily.

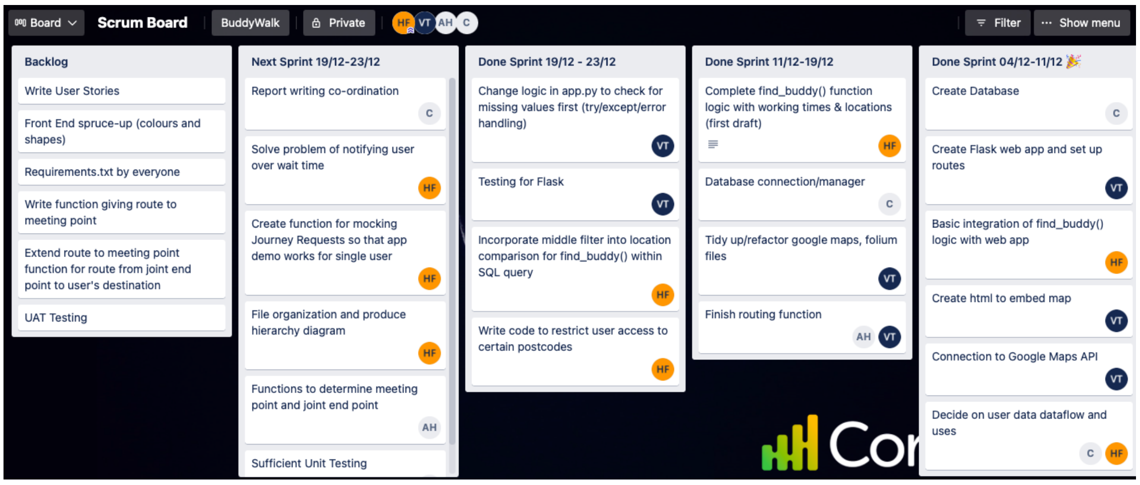
Name: Phoebe Moreton
Age: 25
I often work late in the evenings, meaning that during the winter months it is often dark on my journey home. As a young woman I want the reassurance that I can return home safely, without having to question my route or take long detours in the interests of safety.

Name: Henry Gonzalez
Age: 53
My son has recently moved to London as an international student and I often worry about him getting lost in the city. I want peace of mind that he doesn't take a wrong turn, but I don't want him to stay at home and limit his experiences on account of my safety concerns.

Implementation and Execution

Development approach/Agile development

In order to maintain an Agile approach we set up a sprint planning board on Trello. This helped to maintain an overview of all aspects of the project, the responsibilities of each team member and their personal deadlines. We divided our time into three week-long sprints with three sprint meetings. Through communicating our progress to the team at these sessions we were able to confirm tasks completed or refactor tasks based on challenges or co-operation from other members based on time or other constraints. A key benefit of the process was the ability to prioritise the process flow as to which aspects of the application needed to be built in which order, so that the project could progress. Each team member worked on their own branch of the same Github repo, and added all members as reviewers for pull requests. As such we were able to offer feedback and make adjustments to each other's code throughout the process.



Backlog
- Write User Stories
- Front End spruce-up (colours and shapes)
- Requirements.txt by everyone
- Write function giving route to meeting point
- Extend route to meeting point function for route from joint end point to user's destination
- UAT Testing

Next Sprint 19/12-23/12
- Report writing co-ordination — C
- Solve problem of notifying user over wait time — HF
- Create function for mocking Journey Requests so that app demo works for single user — HF
- File organization and produce hierarchy diagram — HF
- Functions to determine meeting point and joint end point — AH
- Sufficient Unit Testing

Done Sprint 19/12 - 23/12
- Change logic in app.py to check for missing values first (try/except/error handling) — VT
- Testing for Flask — VT
- Incorporate middle filter into location comparison for find_buddy() within SQL query — HF
- Write code to restrict user access to certain postcodes — HF

Done Sprint 11/12-19/12
- Complete find_buddy() function logic with working times & locations (first draft) — HF
- Database connection/manager — C
- Tidy up/refactor google maps, folium files — VT
- Finish routing function — AH VT

Done Sprint 04/12-11/12 🎉
- Create Database — C
- Create Flask web app and set up routes — VT
- Basic integration of find_buddy() logic with web app — HF
- Create html to embed map — VT
- Connection to Google Maps API — VT
- Decide on user data dataflow and uses — C HF

Team member roles

The team member roles were oriented around personal strengths as well as specific areas of interest. Claudia worked on the user database, connecting it to the application and writing the report, Aishah and Verena worked on the maps integration and route calculations, Verena worked on the flask app and Holly worked on the code that matches users. At the end, Verena wrote unit tests for the different parts of the project.

Tools and Libraries

uuid
This module allowed the creation of unique user IDs (UUIDs) for each user of the app, used to uniquely identify each record in our database tables, and to allow the retrieval of user-specific data across app routes.

math
We used the math module for math.pi, which is used to convert Latitude and Longitude coordinates from radians to degrees or back, by helper functions in db_utils.py.

googlemaps
We used the googlemaps module along with a Google Maps API key to access the methods of the various Google Maps APIs used.

datetime
We chose to use datetime objects to manipulate date values in our backend, whilst using the module to convert between datetime objects and strings for ease of value comparison.

haversine
The haversine module proved useful for increasing the efficiency of the find_buddy() function, as it was used to calculate the distance between two sets of points, in terms of their Latitude and Longitude coordinates, according to the Haversine formula for the distances between points on a sphere.

mysql.connector
We used mysql.connector to make the connection to the MySQL database, and to write database utility functions as an interface with the backend.

flask
Flask provided the web framework of our app.

> render_template
> We used this to render html forms that are shown in the browser and post or get information
>
> redirect
> Redirect acted to send the user of the app to the next route when invoking the post method
>
> url_for
> This was used to retrieve the app.route url of a function
>
> send_file
> Send_file sent a file to the browser - in this case the html map that is generated for the user
>
> json
> We used json for simple conversion between Python dictionaries and strings, in order to store entire dictionaries of addresses in a database column for later retrieval of address information.

Implementation process

Flask

The basic flask app was one of the first features to be created with the aim to have an empty app skeleton that was required to be populated with backend logic, for example finding a buddy, dealing with the routing logic and creating a map. User input flows through the different parts of the app and backend logic to create the wanted output.
The main parts of the app are:

Start page/user_input:
- Grabbing and storing user input
- Add user journey to database
- Look for similar journeys in database
- Match with buddy based on current location, destination, time of departure
- Return match

Your_buddy:
- Display buddies to each other: username, phone number, joint start point, joint end point, time of departure

Show_map:
- Map to show the route from joint start point to joint end point

Receiving user input from the browser instead of the pycharm terminal required very basic html code, which unfortunately no team member had. Verena here learned how to use the flask.render_template() and basic html files to generate web pages and grab user inputs. This presented a huge success as it went beyond the scope of the course material. The map was initially intended to be displayed together with the buddy info, but it was difficult to discern if this was possible to display both on the same page with an html template. Implementing flask.send_file() allowed for the map to be shown as a new page.

Route function

The Google maps directions API was used in the route.py file to generate directions of a route and coordinates for the start and end location as well as for each step. The map module folium was used to generate a map with markers for the current location and destination with a line for the route between these points generated with the Google maps API. Pprint was used here for the purpose of printing responses in a more reader-friendly format to the benefit of developers as opposed to the user functionality.

Matching users and buddies

As we began to develop BuddyWalk, though the ideas present in our initial requirements remained, some subtleties changed in response to development needs and challenges. Whilst the 'users' database held user information for intended login and registration functionalities, we introduced a 'journey_requests' table to store the data necessary to match users to their buddies. Each set of form inputs by a user was characterised as a 'journey request'. The find_buddy() function was then developed to implement the matching of journey requests.

The main challenge in the successful development of find_buddy() was the question of how to search the journey requests database efficiently for the best match. Initially, a dummy find_buddy() function was coded that verified some simple requirements: 1) are the journeys close enough together in time?, and 2) of all journeys close enough together in time, which is the best match in terms of locations?

The Google Maps Distance Matrix API was initially used to calculate the distance between the user and the candidate's meeting points and destinations, but this proved too slow. We realised that we could use mySQL to our benefit, and perform the distance calculations inside SQL queries. Doing so would provide an exponential speedup in the case of a large amount of data to search.

It was challenging to execute mathematical statements within SQL code. The Pythagorean formula with converging meridians was chosen, which calculates the distance between two points of latitude and longitude with less precision than the Haversine formula, but still accounts for the converging meridians of longitude:

$$D = R\sqrt{(\Delta\phi)^2 + (\cos(\phi_m)\Delta\lambda)^2};$$

where D is the distance in miles, R is the earth's radius in miles, and the deltaed expressions are the distances between the current locations and the destinations respectively.

The Pythagorean formula inside the SQL query allowed us to extract all journey requests with current location and destination points within a given radius of the user's. Following this step, we could refine our search, and select the optimal journey. This step was achieved using the haversine module, utilising the higher-accuracy formula at this critical step. It was applied iteratively over the list of candidates for matching journeys.

The resulting buddy function proceeds as follows:

```
1. Take in and process the user's journey request.
2. Set minimum and maximum times to search for.
3. Query the database for matching journeys, using WHERE clauses for the following:
        3a. time of Departure is between the minimum and maximum times;
        3b. the username is not the same as the user's (so you can't get yourself as a buddy);
        3c. the journey request has not yet been matched with any other user,
        3d. the current locations of the user and candidate are less than a given distance apart;
        3e. the destinations of the user and candidate are less than a given distance apart.
4. From the matching journeys, select the optimal journey:
        4a. loop over the matching journeys, calculating the distance between current locations and destinations for each candidate;
        4b. sum the distances for a total distance measure;
        4c. select the journey request with the minimum total distance measure.
5. Return the buddy's journey request.
```

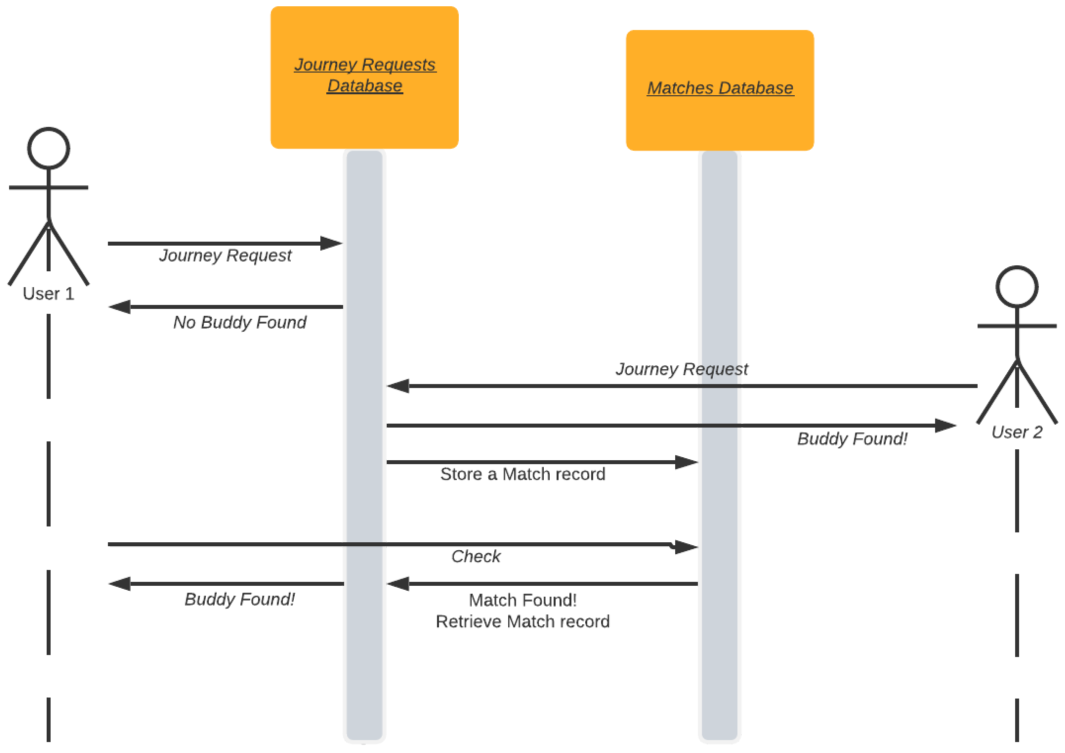
Multiple-user functionality

BuddyWalk was envisaged for use by multiple users simultaneously, so to reach its potential it would need to be deployed, which is far outside the scope of this project. However, we considered a two-user app a great first step. Development was initially approached from the perspective of one user, with mock entries in the journey requests table allowing the user to receive a match despite the absence of fellow users. Once this was complete, we needed a means of testing the app for use by two different users.

We considered the possibility of one team member running the server, altering their router permissions, and other team members accessing the app from other computers and networks. However, the success of demonstrating this would be hardware-dependent. Further, we used Flask sessions to store user data across different requests, precluding us from impersonating two users with the same browser. Within the Flask development environment, we found a workaround which made it possible to do this on the same computer. We were able to impersonate another user by using a browser in Incognito or Private mode.

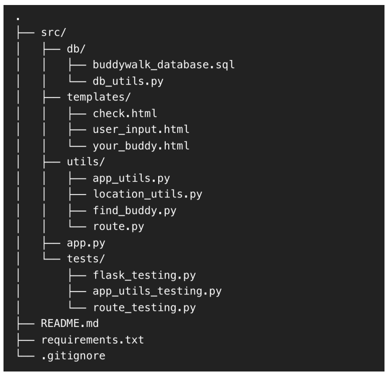
Achieving multiple-user functionality was a challenge in Flask routing logic and in the management of data. However, by creating unique user IDs for each user upon their arrival at the root page, and using a database of matches to store user-independent 'match' records, we succeeded in creating an app which could handle, and differentiate between, two users at once without a login functionality.

One aspect of this was the fact that users would submit journey requests asynchronously, so that the first user to submit one (User 1) does not find a match, and has to wait. When User 2 submits a journey request that matches User 1, they get an instant match. User 1 is then able to check to see whether they have been matched with a 'Check' button, and if they have, they will be returned their buddy's data in the same format as User 2 received, but with the correct user's contact details. This process is illustrated in the sequence diagram below.



Project Structure

Given all of the considerations above, we were able to discern a clearer structure and division for the project itself in the process of the project implementation. This is outlined further in the diagram.



```
.
├── src/
│   ├── db/
│   │   ├── buddywalk_database.sql
│   │   └── db_utils.py
│   ├── templates/
│   │   ├── check.html
│   │   ├── user_input.html
│   │   └── your_buddy.html
│   ├── utils/
│   │   ├── app_utils.py
│   │   ├── location_utils.py
│   │   ├── find_buddy.py
│   │   └── route.py
│   ├── app.py
│   └── tests/
│       ├── flask_testing.py
│       ├── app_utils_testing.py
│       └── route_testing.py
├── README.md
├── requirements.txt
└── .gitignore
```

Testing and Evaluation

Unit testing using Python's unittest module was used for testing most utils, route and flask app functions. As the utils functions are small functions with a clear data flow (data input → data manipulation → data output), unit testing was easy to implement. We tried to create testing functions that would pass, fail, or raise certain exceptions as expected.

However, as the flask app functions are larger functions that call on lots of other functions, connect to databases and require larger user input, unit testing proved to be difficult. Further, the return type of the app functions are redirect, render_templates and send_file functions rather than manipulated data. In hindsight, unittest was possibly the wrong choice of module and instead a module more suitable for testing flask apps should have been chosen such as pytest. However, as unittest was within the scope of the course material, this was used to write the test functions.

System limitations

A key limitation of our system was that as we did not have live users we had to create a live buddy for the purpose of the functioning of the app. On a larger scale this would need to be refactored to assume a larger pool of active users.

Scalability also proved a topic of limitation as with a growing number of users API calls would become significantly slower to process. Given that the server is currently run via our own computers and MySQL this would also be a point worth noting if scaled to a wider audience.

Conclusion

The Buddy Walk project successfully developed a flask application enabling users to input journey details and receive a matching buddy along with a planned route and visualisation. Included within this is a database of users, a python to SQL connection to this with both a routing and buddy matching function. A self written API presents a simple front-end to allow for the user input, while a connection to the Google maps API allows for the presentation of a map visualisation of the chosen route or journey details. The result is an application that provides a hopeful starting (and end) point to a safe and explorative life in the city. Both in the project team and in it's future use, friends will be made along the way.