

Cyber final project

Project’s subject: HeyPhineas – Location based recreation social network

Student’s name: Dan Lvov

Student’s ID:

School and city: De-Shalit high school, Rehovot

Teacher’s name: Eran Bineth

Hand-in date: 07/06/22

Table of Contents

[1 Introduction 3](#_Toc103678272)

[2 Theory 5](#_Toc103678273)

[3 Final Product 12](#_Toc103678274)

[4 The project-writing process 25](#_Toc103678275)

[5 Solution Components 29](#_Toc103678276)

[6 Testing Scenarios 34](#_Toc103678277)

[7 Reflection 35](#_Toc103678278)

[8 Instructions for installation and operation 35](#_Toc103678279)

[9 Bibliography 36](#_Toc103678280)

[10 Appendix 36](#_Toc103678281)

# 1 Introduction

## The subject

*HeyPhineas* is a unique social network application, aimed at connecting friends with mutual interests and geographical proximity. The application integrates a K-Means algorithm to compute the best location to meet, taking into consideration close-by relevant activities. *HeyPhineas* uses a K-means implementation along with KNN, to classify a type of activity for a group of people and makes use of Google Maps API to draw a route and apply directions. It also employs an online database of thousands of locations (parks, restaurants, etc.) and traffic data to generate the optimal route for the user.

The name *HeyPhineas*, comes from the popular TV-show Phineas and Ferb. In the show, the main character *Phineas* tells ferb in the beginning of each episode: “Ferb, I know what we’re going to to today!”. Drawing inspiration from this, I envisioned the user as Ferb, asking: “Hey Phineas, what will we be doing today?” – Thus, *HeyPhineas*.

## Main Goals

The main goals of the project

* Operate smoothly with Google maps, traffic data, and directions. In terms, this would mean the map element of the website should display routes, people, and markers properly, and the movements of said items must be smooth.
* Coordinate multiple groups of people simultaneously. The server runs on several threads which work in parallel to “combine” together the user’s interface, making several “events” happen at once (e.g., handling multiple users moving at once, multiple chats, groups, and the like).
* Provide a friend/activities suggestion algorithm that gives reliable results. As a use case, with user-provided interests (i.e., hand-inputted interest values like sport, food, parks, on a scale of 1-5) – the clustering algorithm, together with the classifier should provide a recommendation that the actual person in question will approve of.

My personal goals are to:

* Delve into K-means, KNN, and the various data clustering / classifying algorithms to understand their usages, pros and cons. Gain an understanding of various recommendation algorithms and their usage, and implementation
* Achieve a solid grasp on dynamic websites using sockets (mainly socket.io, flask-io) and various other features such as Flasks’ html code syntax, understand how cookies / sessions work on browsers, etc.
* Get experiences with Google’s premium APIs and services (e.g Google Maps API, location matrix, distance calculation, etc.).
* Gain more experience with building projects on the level of and above 5 units programming within the Cyber course.

## The Rationale

The rationale behind making *HeyPhineas* is to solve the often-arising difficulty of choosing activities to spend free time, while saving time in the process – making everything accessible within the press of a button.

I often find myself at a loss when choosing pastime activity, baffled at the number of options. It was specifically at a time like this when a friend and I came up with the idea of *HeyPhineas*; *HeyPhineas* promises to be the solution to organizing meetups with friends and getting everyone on the same page.

The UI of many such websites is riddled with features and complexity that I have come to find unnecessary. I wanted to build a simplistic website that would answer to my needs while at the same time not going too overboard with over-the-top functionality and graphics.

## The reference to school material

The project relates to the learned materials by implementing encrypted socket communication between a server and its clients over TCP, which was a large part of this, and last year’s material. The server-based communication in this project provides synchronization between users allowing for messaging and a dynamic map showing everyone’s location.

The project also includes thread management. As a user-socket emits an event, the server must open a thread for the function associated with that event and close it upon completion. This makes the server processor heavy, requiring many threads.

# 2 Theory

## Theory

The project covers four key components: web programming using flask, client-server communication management, a suggestion algorithm, and the use of Google map APIs.

1. **Web Programming using Flask**

Flask is a (micro) framework that allows for dynamically generated HTML, and its serving to a network client. Able to host a relatively small server, Flask has a number of tools to manage a website, such as redirects, hosting files, and the ability to parse if-statements, for loops, and other functions. A web application framework is a collection of libraries and modules which helps developers write the business layer without worrying about the protocol, thread management, etc. And Python’s Django follows a traditional web framework which is also called an enterprise framework.

In contrast to a traditional web framework, a micro-framework is a minimalistic framework where developers are provided with a lot of freedom to build the web application layer. As compared to an enterprise framework, a developer would not need to set up many things in a micro-framework to get the web app hosted and running. This is incredibly useful in cases of small web app development where the needs are not the same as an enterprise-level framework which would save lots of development ~ maintenance time and money consequently.

The implementation of Flask web programming is built amongst others on several ideas, including WSGI and Jinja2

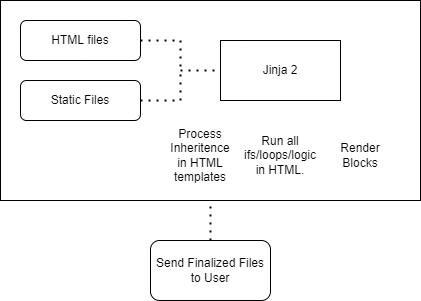
Web Server Gateway Interface (WSGI) is the standard for Python web application development. WSGI is a specification for a universal interface between the web server and the web applications. Werkzeug is one of the most advanced WSGI modules that contain various tools and utilities that facilitate web application development. Flask implements Werkzeug.

Jinja 2 is a template rendering engine. It renders the web pages for the server with any specified custom content given to it by the webserver. Flask renders its HTML based templates using Jinja 2.

The major advantages of Flask are:

* Ease of setup and use.
* Freedom to build the structure of the web application.
* Stemming from its popularity, large communities such as StackOverflow and others have solutions to common (and uncommon) problems.

With freedom comes responsibility, similarly, Flask needs the developers to carefully structure it, since Flask doesn’t have “flask rules” to follow as compared to frameworks like Django. As the web app increases in complexity, the user needs to maintain a consistent structure.



1. **The client-server communication management**

While WebSocket is clearly the best way to establish a bidirectional communication, experience has shown that it is not always possible to establish a WebSocket connection, due to corporate proxies, personal firewall, antivirus software. From the user perspective, an unsuccessful WebSocket connection can translate in up to at least 10 seconds of waiting for the real-time application to begin exchanging data. This perceptively hurts user experience.

**Socket.IO**

The solution comes in the form of Socket.IO. Socket.IO focuses on reliability and user experience first, marginal potential UX improvements and increased server performance second.

Socket.IO is not a WebSocket implementation. Although Socket.IO indeed uses WebSocket for transport, when possible, it adds additional metadata to each packet. That is why a WebSocket client will not be able to successfully connect to a Socket.IO server, and a Socket.IO client will not be able to connect to a plain WebSocket server either.

On the sever side, I used Flask-Socket IO, which gives Flask applications access to low latency bi-directional communications between the clients and the server. The client-side application is written in Python and is socket-event-emission based. Every event that the server picks up on opens a thread that closes when the function associated with the event has finished executing. Most of the communication is done through JSON for developer readability and ease of use.

On the user side, I used Socket.IO (JS), which is a library that enables low-latency, bidirectional and event-based communication between a client and a server. It is built on top of the WebSocket protocol and provides additional guarantees like fallback to HTTP long polling or automatic reconnection.

Although originally planned to be a mobile application, in this project the client side is built on Socket.IO and is not meant to be used in a background service for mobile applications. The Socket.IO library keeps an open TCP connection to the server, which may result in a high battery drain.

Socket.IO has a few features that made me choose it over regular web-sockets.

* By default, clients establish the connection with the HTTP long-polling transport.

Web applications were originally developed around a client/server model, where the Web client is always the initiator of transactions, requesting data from the server. Thus, there was no mechanism for the server to independently send, or push, data to the client without the client first making a request.

To overcome this deficiency, Web app developers can implement a technique called HTTP long polling, where the client polls the server requesting new information. The server holds the request open until new data is available. Once available, the server responds and sends the new information. When the client receives the new information, it immediately sends another request, and the operation is repeated. This effectively emulates a server push feature

The connection will fall back to HTTP long-polling in case the WebSocket connection cannot be established. Even if most browsers now support Web Sockets (more than 97%), it is still a great feature as we still receive reports from users that cannot establish a WebSocket connection because they are behind some misconfigured proxy.

* Automatic reconnection - Under some particular conditions, the WebSocket connection between the server and the client can be interrupted with both sides being unaware of the broken state of the link. That's why Socket.IO includes a heartbeat mechanism, which periodically checks the status of the connection. When the client eventually gets disconnected, it automatically reconnects with an exponential back-off delay, in order not to overwhelm the server.
* Packet buffering - The packets are automatically buffered when the client is disconnected and will be sent upon reconnection. This is very useful in the use-case of a user with an unstable connection. This reduces the amount of “confirmation” packets that are required to assure the server that what it’s sending is being received by the user.

1. **The decision-based algorithms - K-Nearest Neighbors algorithm**

Diagram

Description automatically generatedThe decision-based algorithms rely on the K-Nearest Neighbors algorithm. This is a [machine learning](https://www.unite.ai/what-is-machine-learning/) technique that[can be used for both regression and classification tasks](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=6&cad=rja&uact=8&ved=2ahUKEwjT8emWsubnAhUHKKwKHbeYCKIQFjAFegQIAxAB&url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FK-nearest_neighbors_algorithm&usg=AOvVaw2YaNgXyrE3Vga4aiLiYcGm). The algorithm examines [the labels of a chosen number of data points](https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761) surrounding a target data point, in order to make a prediction about the class that the data point falls into. K-Nearest Neighbors is a conceptually simple yet very powerful algorithm, and for those reasons, it’s one of the most popular machine learning algorithms.

In plain language, the objective of K-Means is to put data points with similar characteristics in the same cluster (i.e., internal cohesion) and separate data points with different characteristics into different clusters (i.e., external separation).

Technically, we need mathematical formulas to quantify both internal cohesion and external separation.

Intra-cluster variance (a.k.a., the squared error function or sum of squares within (SSW) or sum of squares error (SSE)) is used to quantify internal cohesion. It is defined as the sum of the squared distance between the average point (called Centroid) and each point of the cluster. The smaller the value, the better the clustering is.

Diagram

Description automatically generated with medium confidence

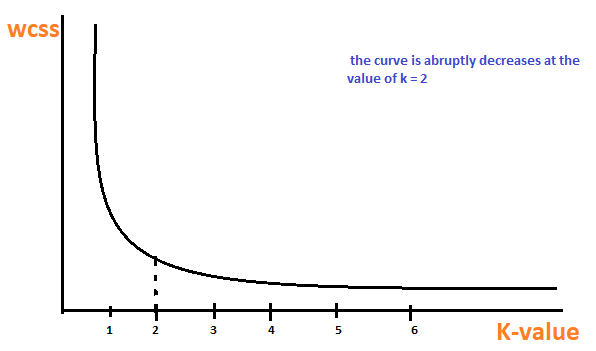
K-means clustering tries to group similar kinds of items in form of clusters. It finds the similarity between the items and groups them into the clusters. K-means clustering algorithm works in three steps.

1. Select the k values.
2. Initialize the centroids.
3. Select the group and find the average.
4. Move cluster to center of mass of group
5. Repeat until no points change clusters

One of the most challenging tasks in this clustering algorithm is to choose the right values of k. What should be the right k-value? When choosing the k values randomly, it might be correct or may be wrong. If wrong, it will directly affect the model performance. Accordingly, there are two methods by which the right value of k is selected:

1. Elbow Method, that is used in this project.
2. Silhouette Method

According to the Elbow Method, when the value of k is 1, the within-cluster sum of the square will be high. As the value of k increases, the within-cluster sum of square value will decrease.

When a graph is plotted (for visualization purposes) between k-values and the within-cluster sum of the square. at some point, the derivative will get closer to 0 abruptly. That point will be considered as the “Elbow Point”; Since the graph resembles a hand (and an elbow). This is the best K value.

Using the **Silhouette Method**, we calculate the s(i), (the Silhouette score) for each point in the dataset.

To do this, we calculate 2 values. A(i), and B(i).

* A(i) is the distance to each point in the same cluster.
* B(i) is the distance to each point in the nearest cluster.

The silhouette score will be equal to:

Scores closer to 1 are ideal. The closer a score is to -1, the worse it is. Naturally, we will pick the number of clusters that result in the highest scores.

Advantages of K-means

* It is very simple to implement.
* It is scalable to a huge data set and faster to large datasets.
* It adapts the new examples very frequently.
* Generalization of clusters for different shapes and sizes.

Disadvantages of K-means

* It is sensitive to the outliers.
* Choosing the k values manually is a tough job.
* As the number of dimensions increases its scalability decreases.

Running this algorithm only once, with random centroid placement might yield varying results. To avoid this, the fitting function runs N (20-100) repetitions, and take the only with the lowest Internal Cohesion.

1. **The use of Google maps APIs**

Google Maps essentially uses two Graph algorithms – Dijkstra’s algorithm and A\* algorithm, to calculate the shortest distance from point A (source) to point B (destination). A graph data structure is essentially a collection of nodes that are defined by edges and vertices.

In order to use these, I make GET requests to Google Maps’ API end points. These include the locations API, distance matrix, directions API. The response from the API is in JSON format, including fields (information) about the location requested (e.g., the locations API).

## Existing similar products

*HeyPhineas* (No domain yet) takes the good parts of social media and map services and combines them together.

GoogleMaps - <https://www.google.com/maps>

Seeing as *HeyPhineas* uses Google Maps’ services, it is fair to assume that Google Maps has many of the features offered by this product. However, what I felt was lacking in Google Maps is the “get together” feeling that *HeyPhineas* provides.

Being a social network, *HeyPhineas* allows for communication between users and the organization of said users together, while letting data clustering algorithms handle the decision making – Which is overseen by the users.

Waze - <https://www.waze.com/>

Waze is similar to Google Maps; However, it does not have an easily accessible API so I did not incorporate it into the final product.

Both Waze and Google Maps also lack a chat feature, forcing users to rely on other communication methods in order to go together.

SnapChat - <https://www.snapchat.com/>

As users of SnapChat will know, SnapChat has a map feature that allows users to see where other users are in real time, and itself being a social network at heart, allows for organization of said users.

What *HeyPhineas* holds over SnapChat in comparison, is the automatic nature of user grouping and suggestions.

Whatsapp - <https://www.whatsapp.com/?lang=en>

Whatsapp is a chat app with a location share feature. This feature allows users to track other users’ location in real time on the map (which I believe to be Google Maps based).

Where Whatsapp loses to *HeyPhineas*, or rather, where the purposes of the app diverge: *HeyPhineas* shows user paths; Has the map and the chat visible at the same time for more convenient chatting.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Map/Location | Points of Interest | Generates plan for you | Party Travel (Groups) |
| HeyPhineas | supported | supported | supported | supported |
| GoogleMaps | supported | supported | Cannot suggest a location | Lack of ability to display multiple users at once |
| Waze | supported | supported | Cannot suggest a location | Lack of ability to display multiple users at once |
| SnapChat | supported | supported | Cannot suggest a location | Lack of ability to display multiple users at once |
| Most social media |  | supported | Cannot suggest a location | Lack of ability to display multiple users at once |
| Whatsapp | supported | Does not have points of interest for users to browse | Cannot suggest a location | supported |

# 3 Final **Product**

## Project description

*HeyPhineas* allows users to coordinate hangouts based on proximity, interests, and traffic along their route. The application is accessible through its website on PCs and mobile devices. Following registration, the user gains access to the network and can start planning hangouts with friends.

When logging in a new user, a user must input their initial location. By pressing a single button, the user is provided with a few different suggestions, as to what to do / where to go / with whom to go with. The user can choose an option and send an invitation to friends, they will be able to see the said invitation and join.

*HeyPhineas* can adjust routes and recommendations based on traffic data (leveraging Google API capabilities) to find places that are less packed, and the optimal way to get to them. Users might not always want to go exactly how *HeyPhineas* recommends, so the application entrusts them with data to make their own decision. Users will have the choice to plan their routes to meet up before the destination, to walk/drive together. Naturally, with more user data (preferences, etc.) *HeyPhineas* can have a better picture of who wants to do what.

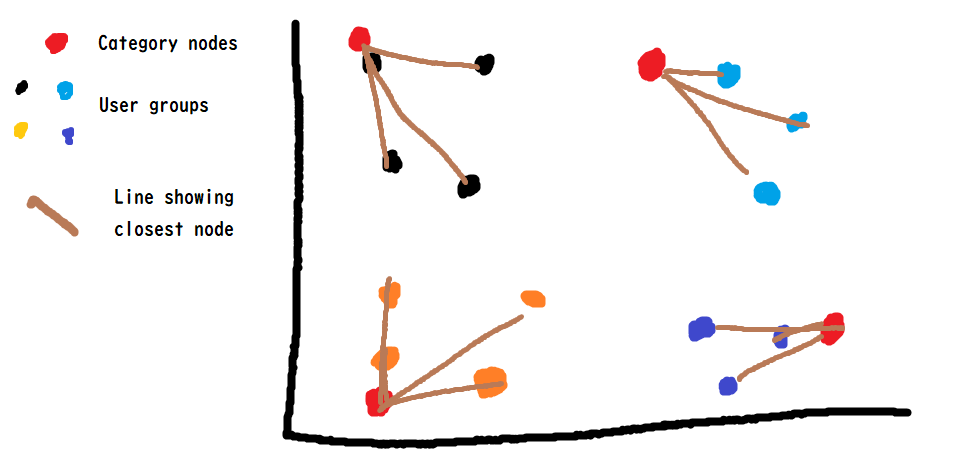
The interactive map, as well as routes and traffic data are made available by using Google Maps API. The map shows points of interest, user added locations, and other users, allowing for an interactive experience.

The admin can see the network logs (history), but not the user chats (as they are encrypted and private). The users can only see the chats they have access to (Party, global).

## Main Algorithms

### **Utilizing the K-Means algorithm**

User data in HeyPhineas is stored as ‘interest’ values for various qualities such as sports, types of food, etc. Based on those values, every user is placed on a point in a multi-dimensional vector space. We can calculate the “distance” between the various users, group, and cluster them together. In doing so, we can find people who are close to each other interest-wise, have mutual friends, and the like.

When a party requests a destination from the server, it needs to know what type of place to suggest. I have created what I call “nodes”, which are basically points in the interests-vector space that represent categories of places, like parks, restaurants, etc. To assign a category to a group, I take the interest values of each member in the group and check what node they are closest to. The algorithm is a simple distance comparison.

### **Communication**

As explained in the theory chapter, the communication between user and server in HeyPhineas is accomplished by TCP sockets (socket.io) opened in the browser. This user-server allows, by proxy, for communication between users – making an essential part of HeyPhineas possible.

In order to make sure that all socket event listeners have been connected properly to the html (e.g., using jquery or document.getElementById) they are created only after the page has loaded. Static folder -> Folder containing the HTML/CSS/JS files for the page.

* “Sockets perform handshake” refers to the TCP handshake the sockets perform when connecting.
* “User is signed as a connected user” refers to the user being added to the online users’ dictionary server-side. This is used to track user data such as currently party members, socket id, and location.
* Communication begins -> sockets have fully loaded.

**The Handshake**

At the beginning of the Engine.IO connection, the server sends some information:

{  
 "sid": "FSDjX-WRwSA4zTZMALqx",  
 "upgrades": ["websocket"],  
 "pingInterval": 25000,  
 "pingTimeout": 20000  
}

* The sid is the ID of the session, it must be included in the sid query parameter in all subsequent HTTP requests
* The upgrades array contains the list of all "better" transports that are supported by the server
* The pingInterval and pingTimeout values are used in the heartbeat mechanism

**The Upgrade**

HTTP Upgrade is used to indicate a preference or requirement to switch to a different version of HTTP or to another protocol, if possible:

The Upgrade general header allows the client to specify what additional communication protocols it supports and would like to use if the server finds it appropriate to switch protocols. The server MUST use the Upgrade header field within a 101 (Switching Protocols) response to indicate which protocol(s) are being switched.

For example,

Upgrade: HTTP/2.0, SHTTP/1.3, IRC/6.9, RTA/x11

The Upgrade header field is intended to provide a simple mechanism for transition from HTTP/1.1 to some other, incompatible protocol.

To upgrade, the client will:

* Ensure its outgoing buffer is empty
* Put the current transport in read-only mode
* Try to establish a connection with the other transport
* If successful, close the first transport

**The end-to-end process**

The overall process can be summed at 5 steps:

* handshake (contains the session ID — here, zBjrh...AAAK — that is used in subsequent requests)
* send data (HTTP long-polling)
* receive data (HTTP long-polling)
* upgrade (WebSocket)
* receive data (HTTP long-polling, closed once the WebSocket connection in 4. is successfully established)

HeyPhineas communicates through secure sockets. Passwords in HeyPhineas are encrypted (salt & peppering) and not stored in plain text.

### **Error Management**

Flask incorporates an error handler event, which is triggered whenever a flask-based (network or web endpoint) function throws an error.

This will display a small popup on the users screen, showing that an error has occurred.

**Disconnection detection**

The Socket.IO connection is considered as closed when:

* One HTTP request (either GET or POST) fails (for example, when the server is shutdown)
* The WebSocket connection is closed (for example, when the user closes the tab in its browser)
* socket.disconnect() is called on the server-side or on the client-side

There is also a heartbeat mechanism which checks that the connection between the server and the client is still up and running. At a given interval (the pingInterval value sent in the handshake) the server sends a PING packet and the client has a few seconds (the pingTimeout value) to send a PONG packet back. If the server does not receive a PONG packet back, it will consider that the connection is closed. Conversely, if the client does not receive a PING packet within pingInterval + pingTimeout, it will consider that the connection is closed.

The userside disconnection reasons are the following:

|  |  |
| --- | --- |
| Reason | Description |
| io server disconnect | The server has forcefully disconnect the socket with socket.disconnect() |
| io client disconnect | The socket was manually disconnected using socket.disconnect() |
| ping timeout | The server did not send a PING within the pingInterval + pingTimeout range |
| Transport close | The connection was closed (example: the user has lost connection, or the network was changed from WiFi to 4G) |
| transport error | The connection has encountered an error (example: the server was killed during a HTTP long-polling cycle) |

In the first two cases (explicit disconnection), the client will not try to reconnect and you need to manually call socket.connect(). In all other cases, the client will wait for a small random delay and then try to reconnect (Default 1000ms~)

In the case of a user-disconnect, once reconnected the socket will be updated in the communication tables. If user does not reconnect, it is displayed to the other users (e.g., Icon is displayed in gray on the map instead of full color)

### **Data management**

The runtime data/session is managed by the browser, which uses an SQLite file in the %appdata% folder. Server-side user data is stored in the main directory as an SQLite database consisting of a “Users” table, and a “Messages” table, to manage the user inbox. Users’ interests, locations, and inbox are stored in said tables.

All interactions with the database are made through a python wrapper I made to make editing/adding/deleting records easier.

### **Registration**

The registration in HeyPhineas is very simple, and only requires the user to input a username, and password. Usernames are unique.

### **Chat Functionality**

The users and server keep a log of all chat messages. Using the socket communication, a user can send a message to the server, and it will send the same message to all the users in the chatroom the user is in.

Every chatroom has an id, a name, its members, and its chat history.

* On the client side:

The user client is waiting on socket events from the server, to create new tabs / delete tabs. The user client keeps a copy of chat\_history for the channels that it is allowed to see. The chat history is simply a dictionary of lists, with keys being the chat ids; Values being arrays of objects in this format:

{

name: str,

message: str

}

* On the server side:

Listening for new message events from users, which include the chat id + content. The server keeps the entire chat\_histories dictionary:

chat\_rooms[id] = {  
 "name": str,  
 "members": {name: confirmed(bool), name: confirmed(bool), ...},  
 "history" : [  
 {"author": name, "message": message},  
 ...   
 ]  
 }

To make sure that users have created all of the tabs they have, the server sends create\_chat packets until the user sends a confirm (like a SYN-ACK) packet.

The server sends the entire history of the chat that the user can see, instead of singular messages in case that a user has disconnected for some time. This way the user can be sure they are seeing the entire chat and are not missing out because of connection issues.

### **Google Maps APIs**

To display an interactive map in HeyPhineas along with all the routes/markers/etc., I use Google Maps’ API.

To add markers and draw paths I use the google maps Marker object. The google.maps.Marker constructor takes a single Marker options object literal, specifying the initial properties of the marker.

The following fields are particularly important and commonly set when constructing a marker:

* position (required) specifies a LatLng identifying the initial location of the marker. One way of retrieving a LatLng is by using the Geocoding service.
* map (optional) specifies the Map on which to place the marker. If you do not specify the map on construction of the marker, the marker is created but is not attached to (or displayed on) the map. You may add the marker later by calling the marker's setMap() method.

Drawing paths on the google maps element using JS consists of two parts:

* Creating a list of google.maps.LatLng elements, which will serve as the points to connect.
* Creating a path object with said list and setting it to a map object.

When requesting direction from the Google Maps API, by looking in the right part of the resulting JSON, we can find the start and end of each “line”. Using these points’ latitude and longitude values we can create a list of points to draw the path through.

A path can have multiple properties, like color, text next to its points, and a few more.

## Constrains and requirements

Constraints:

* Since the project will not be tested on real mobile locations, the app will not have location tracking, but rather a script that will simulate people walking around.
* The app will only work consistently on Windows OS, functionality may vary on other operating systems.
* HeyPhineas is limited by the data GoogleMaps has. If an establishment has moved / has recently been created and has not been manually added to the database of locations HeyPhineas has no way of knowing about the updated location.
* The design is built for the desktop aspect ratio. While it is possible to add CSS/HTML for mobile users, as was the original plan for this project (to be a mobile app/accessible from the mobile device), since this idea is no longer within the scope of the project the HeyPhineas website is accessible and built specifically for the wider screen, and functionality/visibility of certain elements may be affected by the device’s aspect ratio.

Requirements:

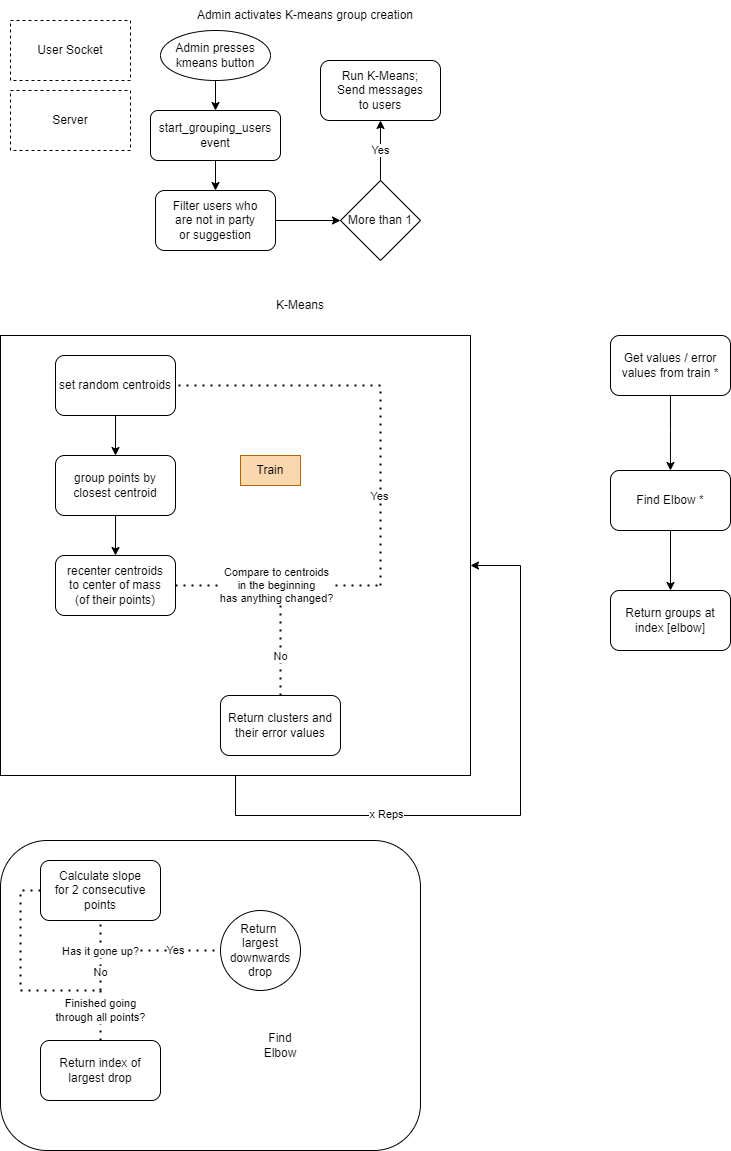
* *HeyPhineas* must be able to handle unintended use such as invalid input in forms, and unexpected user actions.
* Likewise, *HeyPhineas* must be able to handle network issues and disconnects, after all not everyone has a perfect internet connection out there, and the main goal (even though this won’t be tested, it can be simulated) is to allow users to go outside with *HeyPhineas* and reach their destination.
* Although less important to the functionality of *HeyPhineas*, something that can’t be overlooked is the graphics. A good website has good, visually appealing graphics, and *HeyPhineas* is no different.

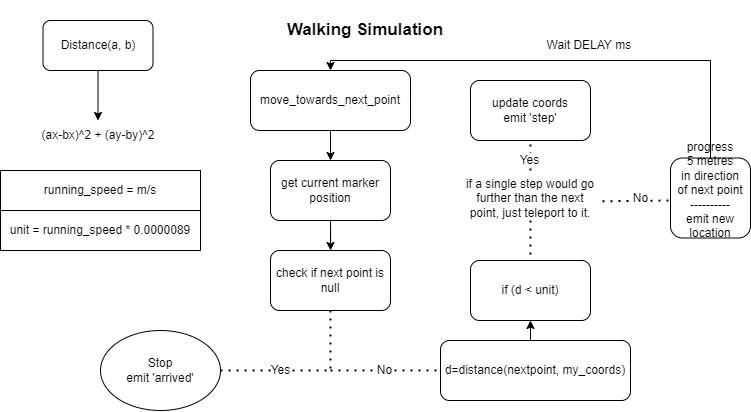
## Use cases

1. User Joining

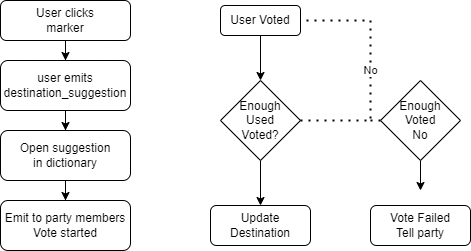
Diagram

Description automatically generated

1. Admin activates K-means group creation
2. Walking Simulation

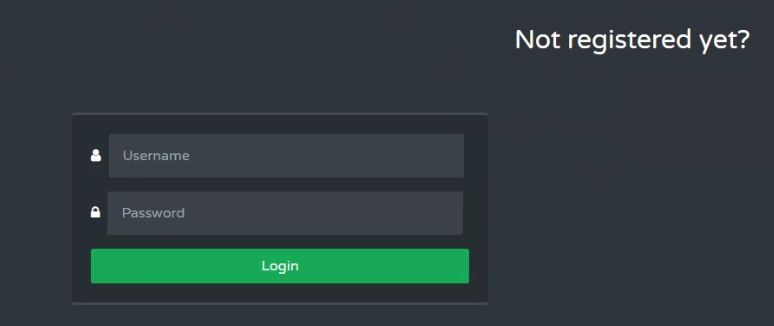


1. User Controlled Choice

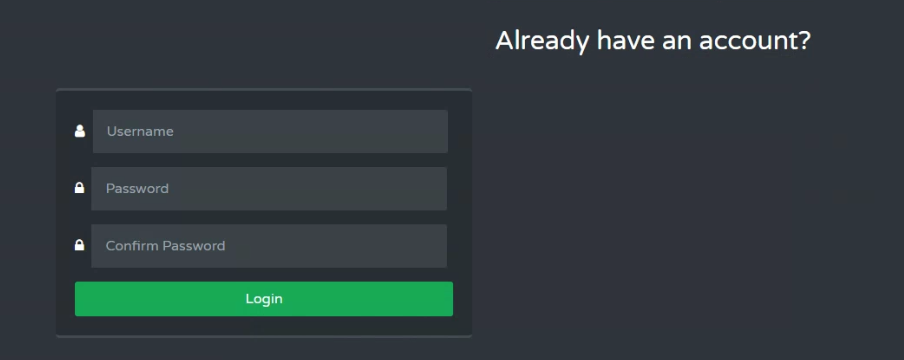


## User interface

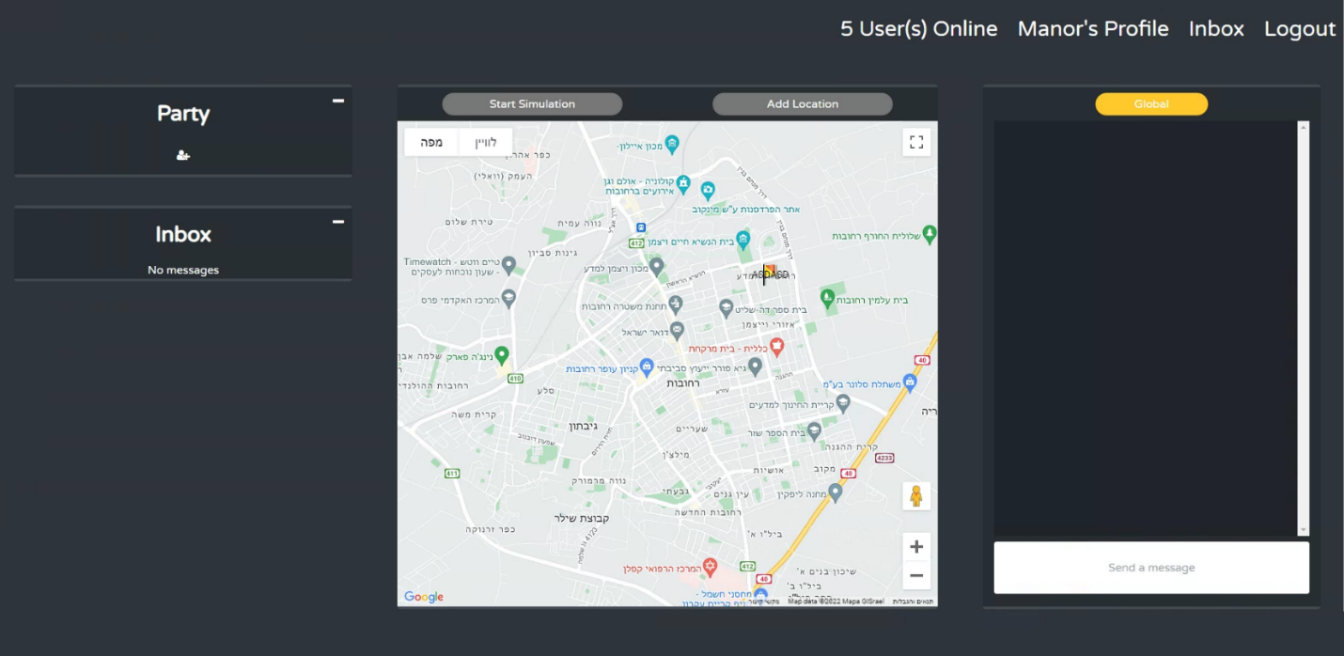
1. Login Interface



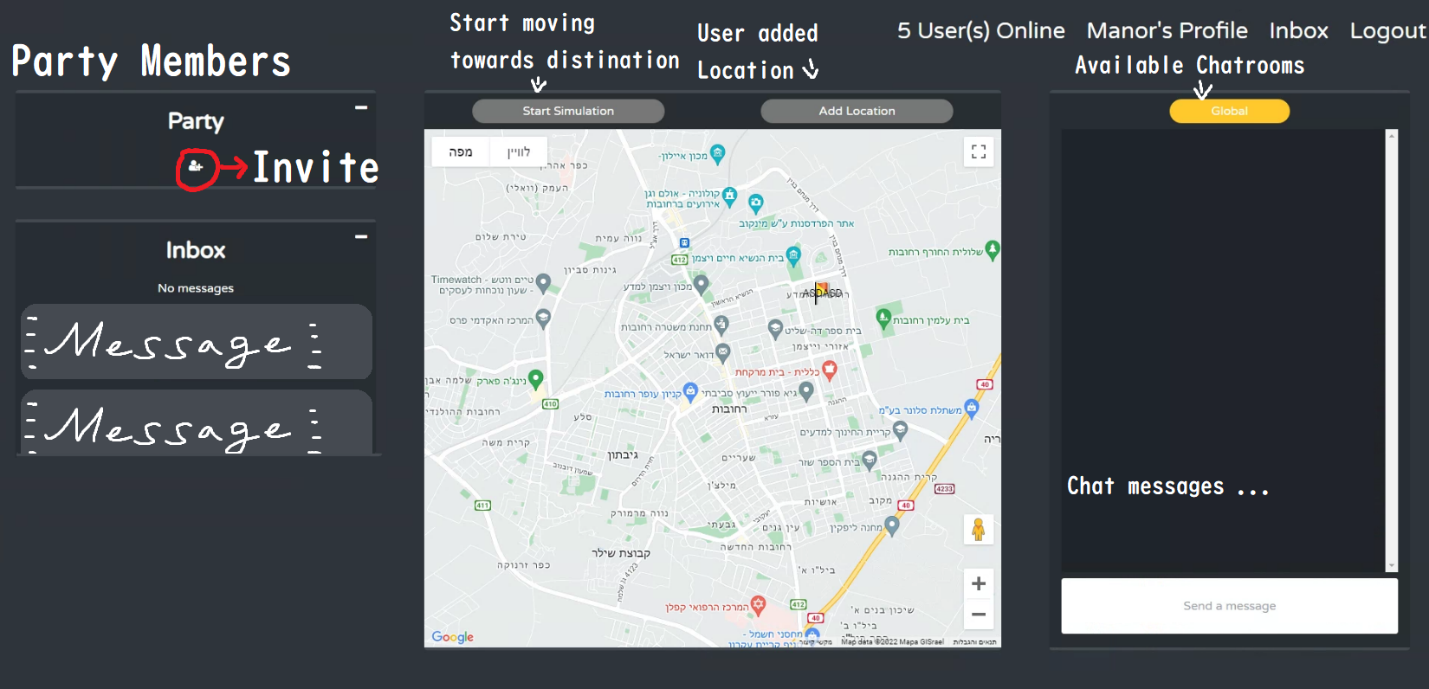
1. Register Interface



1. User map

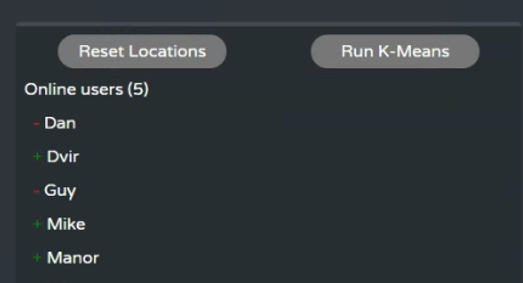


Now with explanations:

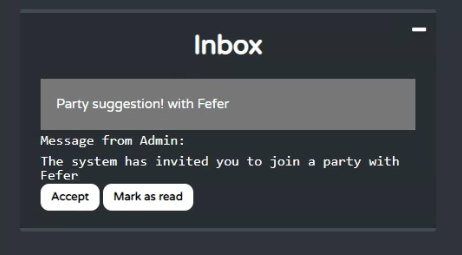


1. Admin panel

Admin does not have chatrooms, instead they have the Admin Panel:



1. Party suggestion from the server:



1. Admin view:

When there are multiple groups (1+), the admin sees them distinguished by color

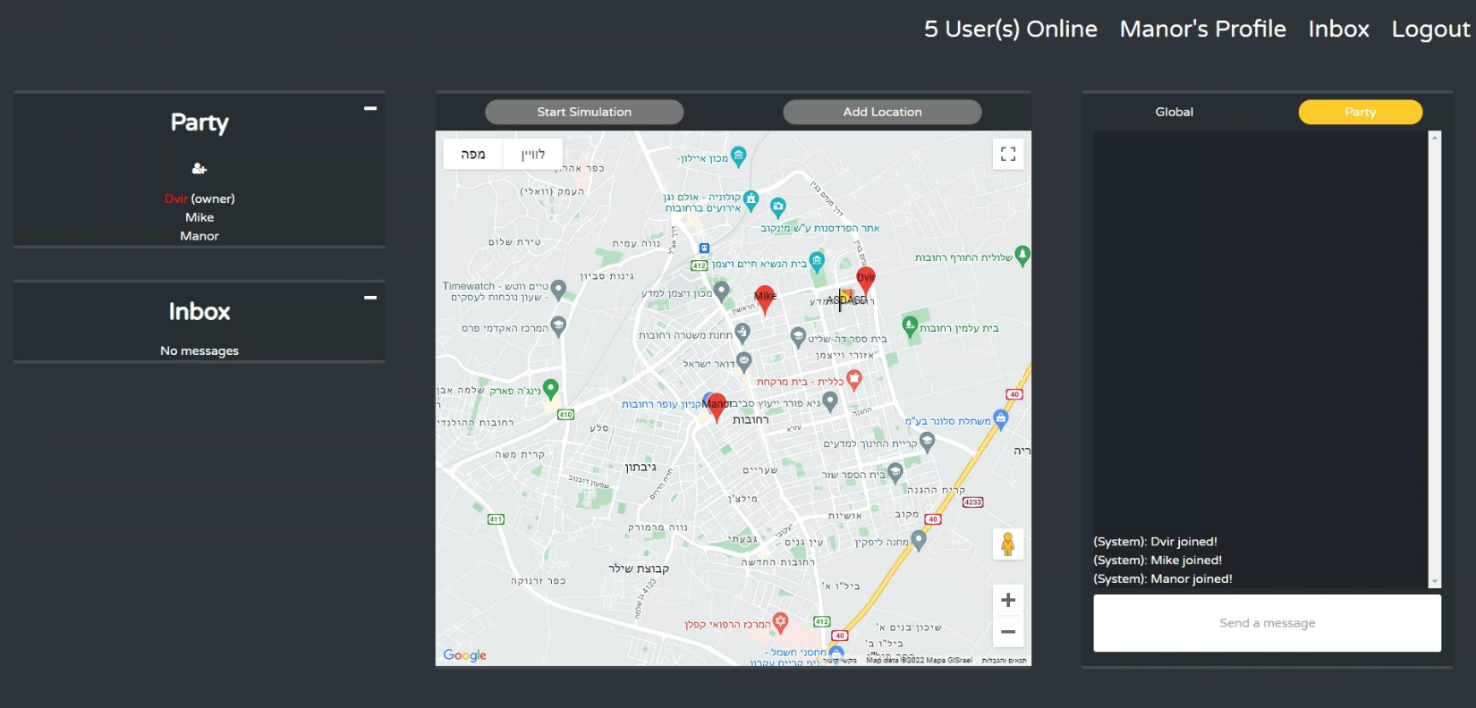


(In this image, the groups are Maya+Fefer Manor+Dvir+Mike)

1. User POV of being in a group

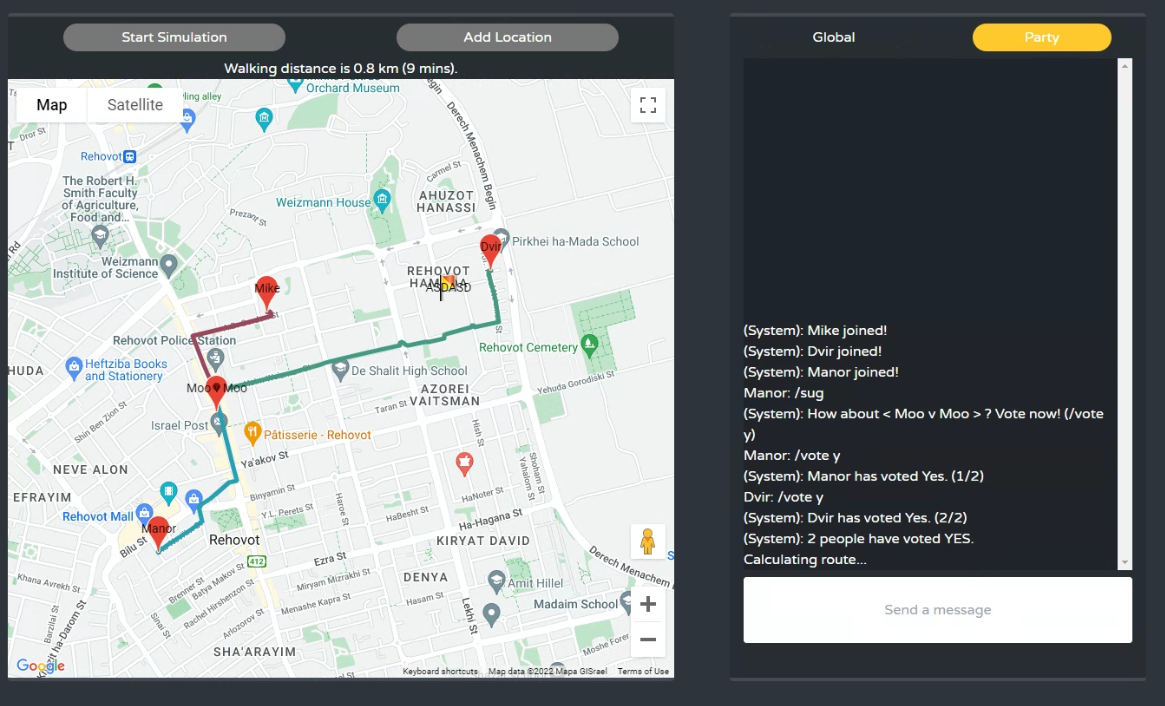
In this image you can see the user’s screen after they have joined a party .

A “Party” chat tab has opened, and the Party tab on the left now has the party’s members. The members’ markers are now visible on the map.



The party has decided on a destination, and the system has calculated their routes to it.

Each user’s route is displayed with a different colour.



Having voted on a location, received directions

# 4 The project-writing process

## The process

The process of making this project from start to finish breaks up into a few phases:

### **Coming up with the idea**

What it took to come up with the idea for *HeyPhineas* is a single walk with a friend on a quiet Thursday night. Wanting to make something together, we brainstormed problems that we can try to solve.

Together, we came up with the idea for an app (originally, designed for the mobile device) that would make plan-making the easiest part of hanging out. The original idea is not much different from the final version of the project, although the original idea was not for a project of this scale, but rather as a commercial product – It included things like monetization and actual location tracking.

### **Selecting the environment and resources**

Having changed my project from cluster computing to building and realizing the vision for *HeyPhineas*, I had to choose my work environment.

Depending on what I would build my UI in, I would have to work in vastly varying programs, and languages. A few options I considered were Windows Forms, along with the built-in GUI maker of Visual Studio. I also considered staying true to the original concept and basing my project in Android Studio, and its Java components. In the end, after consideration of the scope of the project, and other factors, I ultimately chose to build the project in Web, with Python (Flask) as its back end.

After choosing the basis for my GUI, I had to choose and pick what resources I would work with. I needed:

* A customizable and interactive map
* A way to group users into groups of similar interests
* A way to decide on a location based on said interests
* A database

After viewing multiple options, including OpenMapAPI, Google Maps, mapbox API, geoapify, api-map, and many more, in the end I chose to work with the well documented Google Maps API. Google Maps has a very detailed and thorough documentation, with a thriving community in websites such as Stackoverflow. Though not being free, I have not been charged a single penny for my thousands for requests over the year working on this project.

After researching data clustering (ironically, very close to my original idea of cluster programing), I found out about KNN and K-means. Both algorithms are explored in detail in the theory section of this paper.

For the database, I struggled to choose between MongoDB and SQLite. I have already made a neat wrapper for SQLite, but I also wanted to go into uncharted area with MongoDB seeing as it is used a lot in the marker. In the end I settled on using SQLite, and to explore MongoDB/NoSQL in my free time and in other projects.

Having already made a very useful wrapper for SQLite, I only made slight adjustments to make it fit my needs this time.

### **Researching solutions to the problem**

At first, I misunderstood the purpose of KNN, and imagined that it can also be used for data clustering. This caused me to not have a proper data clustering function for a long time. After properly researching the subject, I came across K-Means.

### **Implementing the solutions in the code**

Incorporating the algorithms ingo my code was really simple, as I’ve already laid all the groundwork for them to be put right in.

### **Mashing everything together into a finalized product**

After slowly building up the website half of the website, and the KNN/KMEANS module separately, I had to import the module at some point. Of course, it didn’t work right away, but after only one bug-fixing session, it worked as expected. This was a big surprise, and a very welcomed one.

## Challenges and different implementation options

1. **SQLite vs MangoDB**

SQLite is a C-based library that allows applications to store and retrieve data from local files on devices via a SQL interface. SQLite was started as an open-source project in Sep 2000 and is supported on a few platforms, mostly mobile devices, and mobile operating systems.

MongoDB Server is a general-purpose document database which supports different platforms. MongoDB’s core concept is a NoSQL flexible schema that stores data in BSON (Binary JSON-like) documents grouped in “collections” and “databases.” MongoDB can potentially run on small devices but its main power is being a backend database for your applications. The best and most convenient way to run a MongoDB Server is with MongoDB Atlas, our Database-as-a-Service platform.

MongoDB Atlas is scalable and highly available by design cloud platform, allowing multiple clients and languages to connect and power different applications. If the framework is C#, Java (Android), iOS, Node.js, or Python.

Many of the comparisons between MongoDB and SQLite tend to favor MongoDB. Indeed, I attempted to use it for a short time, but the setup appeared troublesome. Having already made a very simple to use and developer friendly wrapper for SQL, sparing myself the need to remake the database and input all the data again; I decided to use SQLite.

1. **Windows form vs HTML CSS**

Originally the idea was to build an app, and at first, I wondered if I should delve into making UIs with it. Firstly, because at the time I was fresh out of a project based in web and had no experience with windows forms – I leaned towards making it in Flask, and the UI with HTML/CSS.

Before starting to work on this project, I had little experience in making dynamic (JS heavy) pages with forms, sockets, and buttons that work without reloading the page. This seemed like a barrier and a clear reason to use windows form, but I managed to learn it from scratch and build a product I’m proud of.

1. **Flask vs Django**

Django is a high-level, full-stack framework used for quickly developing clean-looking apps. "High-level" means that Django is designed to minimize actual coding during the app design process. Instead of writing code, developers who use high-level frameworks can define options with forms, tables, and other interfaces. However, Django is not for small projects, and Django is opinionated. While it's not impossible to use Django in a way other than what was intended by its designers, doing so will lead to a lot of extra work. This can make Django seem monolithic.

Flask is a micro-framework, meaning that it's designed to perform a more limited role than a full-stack framework like Django. But this isn't necessarily a bad thing. As a micro-framework, Flask has few dependencies on external libraries compared to the full-stack Django framework. But there are two big dependencies with Flask: Werkzeug and jinja2.

As a micro-framework, Flask is designed to perform a few tasks extremely well. There are also other reasons why Web Developers choose Flask over other frameworks like Django. Let's take a look at some of Flask's biggest advantages.

* Flask offers more flexibility - Compared to the opinionated Django framework, Flask is more flexible to different working styles and approaches to web app development. Programmers with more coding experience or who need more control of the app design prefer Flask for this reason.
* Flask has more database support - Because Flask has no default model, it inherently supports multiple types of databases. This also makes databases easier to integrate into a Flask application.
* Flask is easier to use for smaller applications - If you're looking to design a relatively simple web app with a few static pages, Flask will make your life easier than Django. For smaller web applications, many programmers find that Flask is easily scalable as well.

Most of the disadvantages of Flask, being mostly used for small applications, are rather advantages for this specific project. This project does not call for scalability, or awe-inspiring flexibility. I don’t require complex features, and I’m not building a complex system. With this being said: I chose Flask.

1. **How to handle data, when to save to database?**

Throughout the making of this project, I dabbed in many ways to handle data, and various ways of storing it. As an example, when a user is moving towards their destination, they rapidly send their destination to the server, and the other users in their party. There’s no reason to update their location in the SQLite database every time since the location is updated several times a second. Instead, I opted for saving said data only occasionally, and when the user disconnects.

With multiple threads trying to read and write to the DB at the same time, I implemented a thread lock in the database – To avoid multiple threads are accessing the SQLite engine multiple times a second and waiting for each other to finish, I decrease the number of accesses to the database.

1. **Tailwind.vs. Normal css**

As per official Tailwind CSS documentation, Tailwind CSS is a highly customizable, low-level CSS framework that allows you to custom-build designs eliminating opinionated component styles that you would anyway want to override.

Advantages of Tailwind over plain CSS:

* Tailwind CSS is a highly customizable framework. Although it comes with a default configuration, it is simple to override it with a tailwind.config.js file. The configuration file enables easy customization of color palettes, styling, spacing, themes, etc.
* Tailwind eliminates the need for long and confusing css files. It keeps all the css in a very readable format in the class names of divs

Although plain CSS is a powerful tool, maintaining it is a tough job. It’s not hard to read but it is hard to look through hundreds of lines of code to debug issues.

# 5 Solution **Components**

## Project disciplines

* Networking – Server/Client, TCP protocol, Secure Communication
* Operating systems - Proven to run on Windows, functionality may vary on other operating systems stemming from the different ways different operating systems handle threads, especially many of them.
* Documentation will be fully available in the Github page
* UI – HTML/CSS served by Flask
* Every module is in a separate file, all documented.
* Databases
* Security

## Environmental requirements

* Programming Language:

Python 3.9

JS

Flask

gmaps

googlemaps

numpy

pymongo

requests

Flask-SocketIO

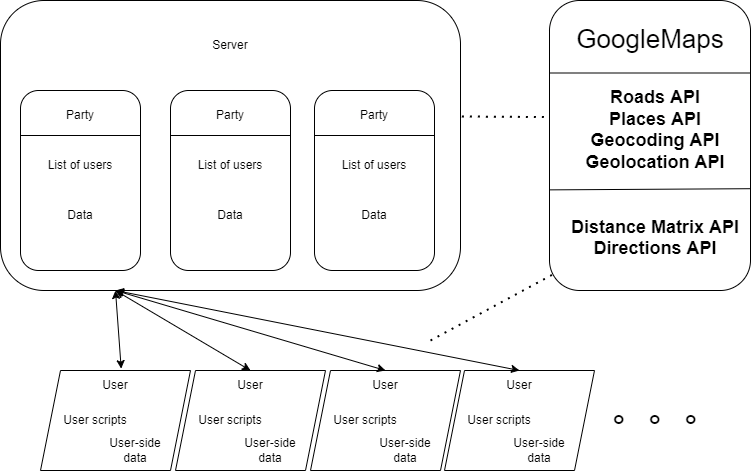
python-engineio

python-socketio

* Built in:

Pycharm

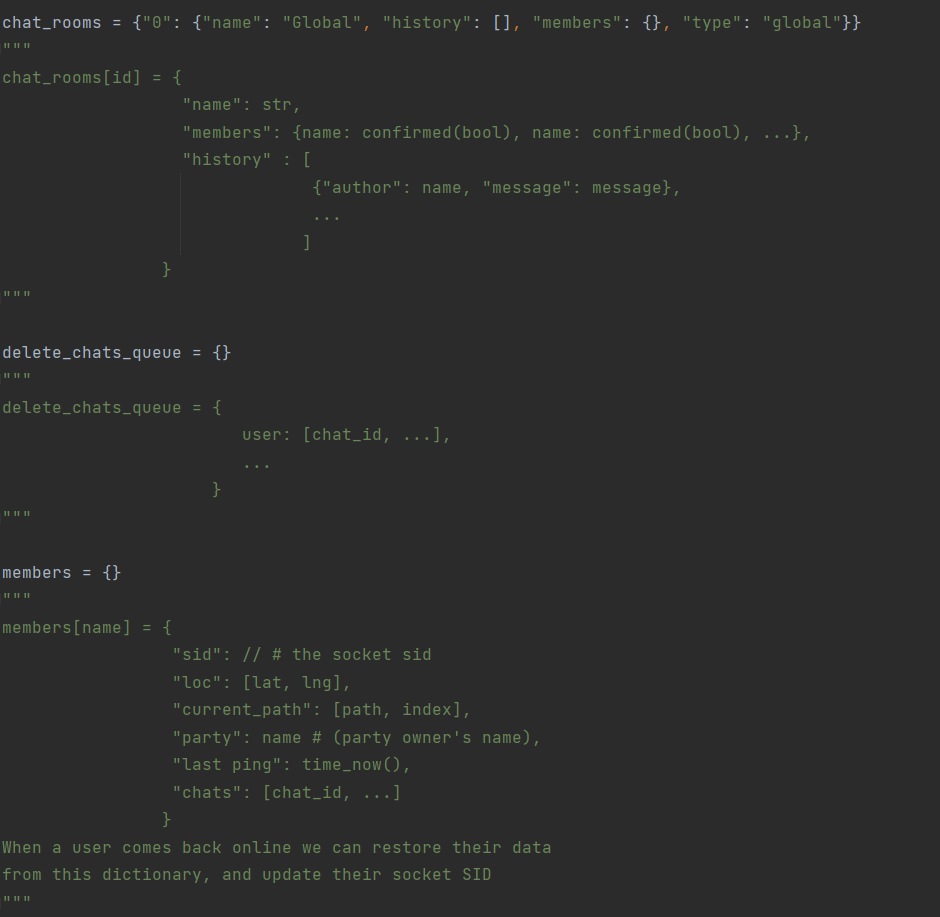
## Topological view

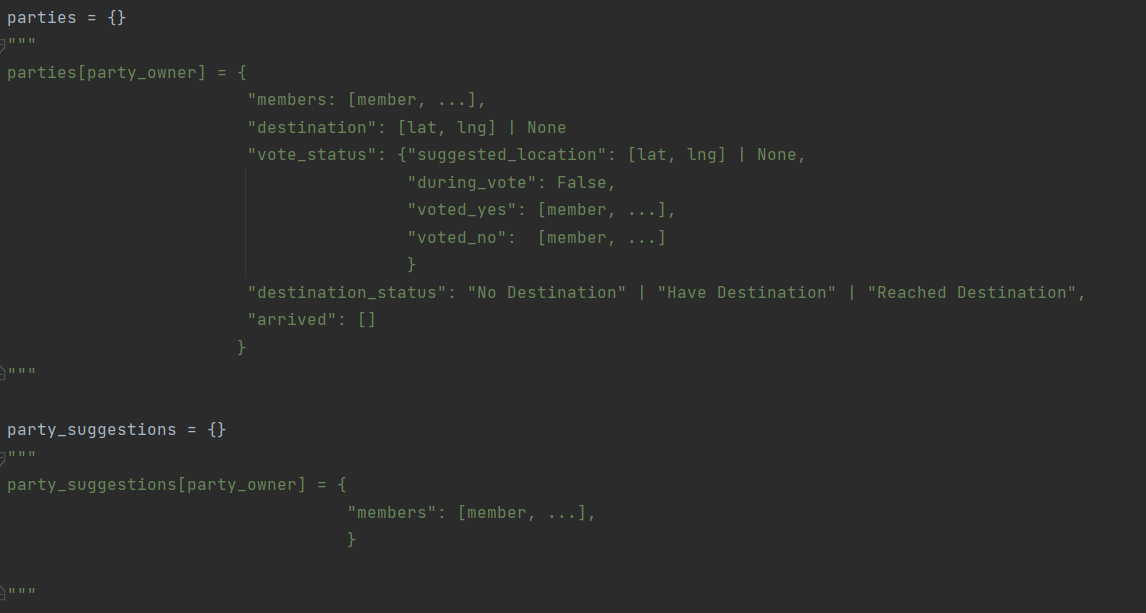


* Server holds dictionaries for the parties
* Users connected through sockets to the server with client-side scripts controlling behavior
* Users using distance API and direction API.
* Server using Geolocation, Roads API, Places API, and Geocoding API.

## Data structure definition

The user is sent a subset of the chat\_rooms dictionary, in order to render the chatrooms the user is allowed to see.





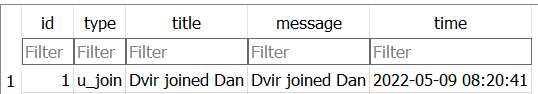
## Database definition

In the Database there are multiple tables.

* **admin\_history**

This table keeps the admin messages history, with each message having an ID, Type, Title, Content (message), and the time it was logged.

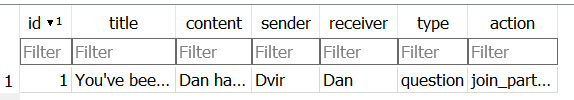
These are displayed as a list on the admin UI.



* **messages**

This table keeps the messages for all users on the platform. Messages of course have a sender and a receiver. If the type of the message is “question”, it will have an action which will be parsed by a parser function in the server when reacted to.

A reaction consists of pressing “Accept” “Decline” or “Mark as read”.

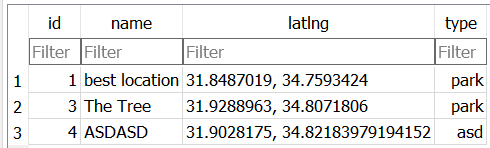


* **user\_added\_locations**

Each location has an ID, a name, its coordinates, and a type.

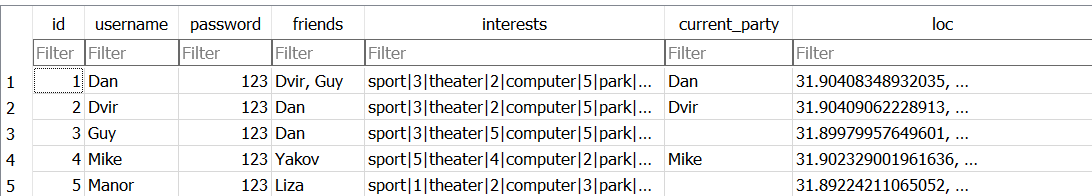
The type will decide its icon on the map.

The name and latlng define the marker’s properties on the map.



* **users**

Each user has a username, passwords, a list of friends, interests, current party, and location



The admin user has only a username and password. 

## Interface

Interface to Google Map API, as explained above

## Main modules

|  |  |
| --- | --- |
| Function | Description |
| get\_all\_user\_chats | Loop over the chat\_rooms dictionary and return all of the rooms in which the user is present |
| suggest\_party | Send a message suggesting a party to a list of users |
| get\_place\_  recommendation\_location | Request to locations API at a radius around the center of the group (coordinate-wise) |
| create\_chat | Create a chat in the chat\_rooms dictionary; Send create\_chat event to users |
| separate\_into\_colours | Given a list of parties, gives each party a color |
| create\_party | Create an entry in the parties dict, send create\_chat to all members, update party members, update markers |
| get\_messages | Fetch the messages of a user |
| get\_party\_members | Fetch the party members of a user |
| split\_interests | Fetch user interests from the database and upload them to the runtime variables |
| emit\_to\_everyone | Emit to all connected users (broadcast) |
| emit\_to\_party | Emit to all party members (multicast) |
| send\_user\_added\_locations | Broadcast user added locations |
| party\_coords | Multicast cords to party members |
| disconnect\_user\_from\_party | Update dictionaries and send “user left” message. Update markers. |
| send\_message\_to\_party | Multicast chat message to party members |
| parse\_action | Parse action on an inbox message |
| send\_path\_to\_party | Multicast user path to party members |
| join\_party | Multicast new member data to party members and update dictionaries, markers |
| broadcast\_user\_difference | Broadcast user difference |

# 6 Testing Scenarios

## Main Testing scenarios

Scenario 1: 4-2

* 6 users connect, along with an admin (users with specific interest values)
* Admin activates K-means, users separate into a group of 4 and a group of 2.
* The group of 2 both accept
* 3 out of the 4 accept in the latter group.
* Group of 2 invites user that declined in latter group
* Both groups (now both consisting of 3 users) try to find a place:
* Group 1 requests suggestion and begin vote
* Group 2's member clicks on user added location and begins vote
* Vote in group 1 succeeds, begin moving towards destination
* Vote in group 2 fails, they ask for suggestion and repeat process
* Vote in group 2 succeeds; begin to move towards destination
* Someone along the process a user disconnects and reconnects at a later stage.
* Admin history logs everything.

# 7 Reflection

## Time table

|  |  |
| --- | --- |
| Month | Monthly mission |
| November | Integrate Google API and KNN on randomly generated userbase |
| December | Finish the network side of the project (communication between users, parties, ect.) , 3 users (signin+move) userDB, simulation of user locations & uploaded place-attractions |
| January | Integrate traffic data into distance calculations |
| February | Make the actual website (design) Here all the scenarios must be completed. |
| March | Make the travel-simulation / scrip that fakes location data to simulate a trip to a location. (before this point it’ll be manual input) |
| April | Handle communication abruptions |
| May | Finishing touches and make everything work completely together |

## Insights

Throughout the creation of *HeyPhineas* I learned about many frameworks and ways of working with Web, including but not limited to tailwind, Flask.IO, and many other client and server-side tools.

I learned about many algorithms such as K-means (++) and KNN, I learned how to make graph-based friend networks and delved in genetic algorithms, but wanting consistent results, I ended up not using genetic algorithms in the final product.

# 8 Instructions for installation and operation

## Configuration and prerequisites

Python 3.9~

Flask==1.1.2  
gmaps==0.9.0  
googlemaps==4.5.3  
numpy==1.19.5  
pymongo==3.12.1  
requests==2.25.1  
Flask-SocketIO==4.3.1  
python-engineio==3.13.2  
python-socketio==4.6.0

The project runs on the user’s browser. A computer that serves as a server hosts the website on the local network, and users can connect and view the website by inputting the IPv4 of the server computer and the port 8080 (e.g., 10.0.0.12:8080) into the browser. Running the project on WAN, meaning allowing users outside the local network requires the network operator to open port forwarding on the aforementioned port.

## Installation

Download the repository from

<https://github.com/IMakeBotsForYou/HeyPhinis/archive/refs/heads/main.zip>

run command pip install -u requirements.txt

run command python main.py

# 9 Bibliography

https://www.cse.unr.edu/~sushil/class/gas/papers/GARecommnder.pdf

<https://softwareengineering.stackexchange.com/questions/419007/k-nearest-neighbors-for-friend-recommendation-system>

<https://www.codeforests.com/2021/01/23/plot-route-on-google-maps/>

<https://flask.palletsprojects.com/en/2.0.x/>

https://pypi.org/project/googlemaps/

# 10 Appendix