Townie

Design Document - Team 27

Team 27:

- Jack Rookstool
- Jon Hurley
- Sai Girap
- Arnav Mehra
- Jamie Barrett

Purpose

Figuring out what to do with free time can be one of the toughest questions to answer, and so many people after working hard to earn a free weekend end up spending it doing the same things they would normally do. We believe that people love to be adventurous, but do not know where to begin.

The purpose of this project is to develop a web application that offloads the decision making burden and ensures an adventure. Townies are people that have lived in the same place for their whole life and know the area so well that they could provide a unique experience to any individual regardless of their preferences, time frame, budget, and more. This app looks to replicate that feeling and fuse it with a scavenger hunt style race through the city to find the experiences that have been collected for the day. Finding these locations, especially finding them quickly, are incentivized by the player receiving a point value for completing their object. The player's successes can be posted on social media, used to customize their profile, or used as an in-game perk that will help them complete more games quicker.

Also, like any adventure, the more the merrier. We will have a friend system that can suggest friends for users. These friends, as well as strangers, can join together to play these games together. They can stick together and explore as a block or split up and get through games with incredible speed. The choice is up to them. It is a great way to build connections with friends and locals alike.

1. User account

   As a user,
   a. I would like to create an account.
   b. I would like to sign into and manage my account.
   c. I would like the game to remember a previous login and start at the home page.
   d. I would like the option to reset my password if I forget it.
   e. I would like to know if anyone I might know also has an account and befriend them.
   f. I would like to provide two-factor authentication.
   g. I would like to verify my account email on creation.
   h. I would like to choose if my exact location remains private.
   i. As a player, I would like a viewable log of my trips.
2. Game creation

   As a player,
   a. I would like to input a location as a parameter so the system can narrow its focus in picking locations.
   b. I would like to input themes as a parameter so the system can parse through locations with an emphasis on those themes.
   c. I would like to have a random theme chosen that could be chosen as a result of satisfaction rates with previous themes.
   d. I would like to specify a budget so the system can steer away from locations that would violate that budget.
   e. I would like to specify a time frame so the game knows how many locations to pick and roughly how far apart those locations can be.
   f. I would like to specify a general radius.
   g. I would like to specify a mode of transportation.
   h. I would like the game to curate some fully premade levels.
   i. I would like for the game to have a tutorial with the ability to replay the tutorial using a help button.
3. Map interaction

   As a player,
   a. I would like the search radius to shrink proportional to the time frame.
   b. I would like to export the general radius of a location to google maps so I can use navigation.
   c. I would like the destination radius to shrink as I approach the destination.
   d. I would like to know an estimation of how long it would take to get to the first location.

    e. I would like to see the map with my current location and the destination radius.

4. Game play

   As a player,

    a. I would like a timer for how long it has taken both for the whole game and for the location at hand.
    b. I would like to be given the opportunity to “pause” the game to enjoy the location.
    c. I would like to be given the opportunity to skip a location.
    d. I would like the ability to change the time frame after starting the game.
    e. I would like to “peek” at a location in exchange for a number of points I have earned.
    f. I would like the ability to end the game early.
    g. I would like to know some fun facts about the location in which I have arrived.
    h. I would like a casual version which puts together a fun day with or without a theme, without hiding anything (like an impromptu itinerary for the day).

5. Group formation

   As a group player,

    a. I would be alerted if any friend, member from a previous group, or friend of friend happens to be on the app and would like to form a group.
    b. I would like to add other users as friends.
    c. I would like to form groups with other people looking to play the game.
    d. (If time allows) I would like to have a chat for me and my friends to communicate.
    e. I would like to be able to share a day with friends and family.
    f. I would like to be able to keep track of where my friends are.

6. Realtime notifications

   As a player,

    a. I would like the game to notify me when I have reached the location.
    b. I would like the game to form days for me without my instantiation, serving as an impromptu adventure. I would like the game to do this once a week.

7. Ranking System

   As a player,

    a. I would like to gain points for each successful guess.
    b. I would like points earned to scale with time spent searching (more points for quicker times) .
    c. I would like to spend points to add visual skins to the UI and display trophies of my games.
    d. I would like to have a ranking system based on how many games I have played.
    e. (If time allows) I would like to use my points to earn gift cards from locations.

8. Rating system

   As a player,

   a. I would like to rate specific themes.
   b. I would like to share my trips online (social media platforms).
   c. I would like to rate my satisfaction with the destinations.

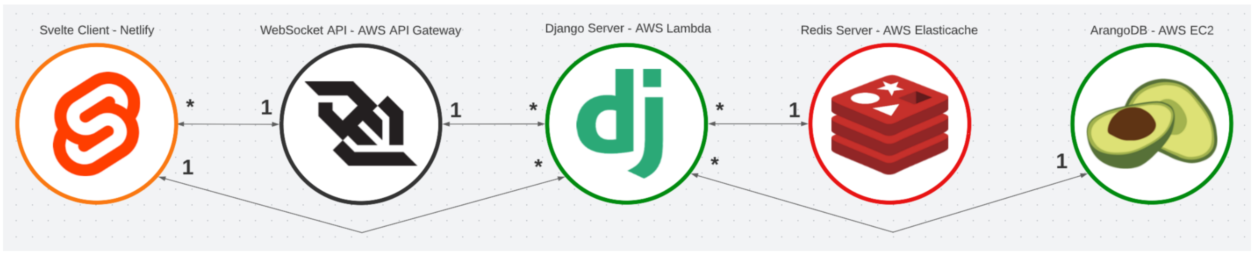
Non Functional Requirements

1. Performance
   a. I would like the application to run smoothly without crashing.
   b. I would like to handle 10,000 to 15,000 simultaneous requests.
2. Server
   a. I would like the server to store user data in a graph database.
   b. I would like client-side caching (local database) with Redis.
3. Appearance
   a. I would like an aesthetically pleasing user interface.
   b. I would like to allow users to customize their profiles.
4. Security
   a. I would like OAuth 2.0 authentication
   b. I would like to provide protections against cross site forgery.
   c. I would like to provide protections against cross site scripting.
   d. I would like to provide protections against SQL injection.
5. Usability
   a. I would like the game's UI to lead the users in the game
   b. I would like the app to be straightforward with a small learning curve

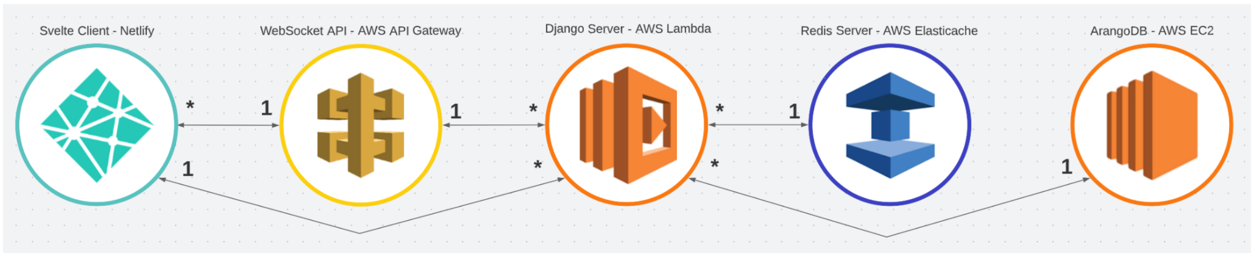
Design Outline

High Level Overview

Framework Stack:


Svelte Client - Netlify
WebSocket API - AWS API Gateway
Django Server - AWS Lambda
Redis Server - AWS Elasticache
ArangoDB - AWS EC2
*
1
1
*
*
1
1
*
*
1

Web-Service Stack:


Svelte Client - Netlify
WebSocket API - AWS API Gateway
Django Server - AWS Lambda
Redis Server - AWS Elasticache
ArangoDB - AWS EC2
*
1
1
*
*
1
1
*
*
1

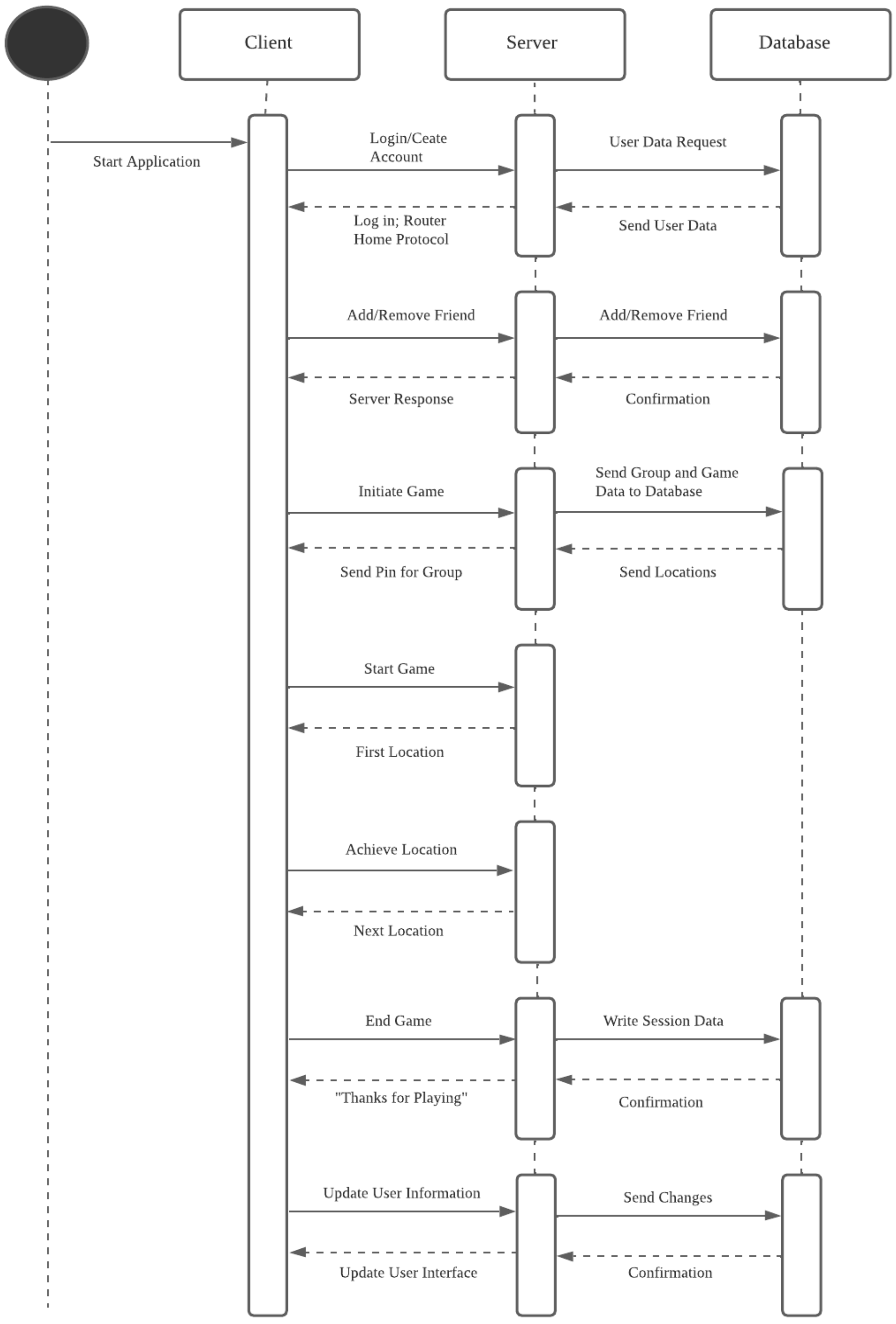
To provide an overview of our system design, I have provided both a version displaying our choice of frameworks and a version displaying our choice of web-services.

For our client-side, we will be using Svelte hosted on Netlify. It will communicate with our backend in 1 of 2 ways. The first is sending HTTP requests directly to our Django app server running on AWS Lambda. This connection will be 1 to many as multiple lambdas, by nature, may be involved in completing the requests of a single user. The second is for establishing a WebSocket connection with AWS API Gateway. This will be a many to 1 relation as multiple clients may be connected to the same AWS API Gateway instance. From there, the Gateway will trigger those same lambda functions, once again providing a 1 to many relation.

Our Lambdas will then fetch and/or send necessary data to our Redis instance running on AWS Elasticache. If the needed data is found within this cache, it can be sent back to the client. Otherwise, we must search for this data in the ArangoDB instance we plan to have running on an AWS EC2. Both of these relations (between lambdas and Redis, between lambdas and ArangoDB) will be many to 1 as multiple lambda functions will attempt to retrieve or write data from single Redis and ArangoDB instances.

Sequence of Events Overview

The sequence diagram below shows the typical interaction between clients, the server, and the database. This sequence begins when a user starts the web app. When the user logs in/creates an account, the client sends a request to the server. The servers handles the request be sending a query to the database and receives the user's data from the database. After logging in, the client can send requests to the server for other functions, such as add/removing friends, updating user information, and initiating a game. The server responds to these requests by sending data and queries to the database. The database will create and update necessary fields for the data sent and return a confirmation. Additionally, it may return a list of popular locations that are relevant to the query if the query made is to initiate a game, which means to form a group and begin picking location preferences. These locations would be passed on to all clients in the group that was formed. If the user starts a game, the server generates a map for the clients and returns it to the client. If the clients reach the destination the map describes, the client will send a request for another map that the server will compute and return. This continues until the clients give the endgame request, which will either be manually requested or automatically requested when all the destinations have been reached. This will send a request to the server. In turn, the server will send a request to the database along with data to store from the game session. The database will respond with a confirmation of completion, at which point the server will send a response to the client notifying and congratulating the users for a successful game.


Client
Server
Database
Start Application
Login/Ceate Account
Log in; Router Home Protocol
User Data Request
Send User Data
Add/Remove Friend
Server Response
Add/Remove Friend
Confirmation
Initiate Game
Send Pin for Group
Send Group and Game Data to Database
Send Locations
Start Game
First Location
Achieve Location
Next Location
End Game
"Thanks for Playing"
Write Session Data
Confirmation
Update User Information
Update User Interface
Send Changes
Confirmation

Design Issues

Functional Issues:

1. What kind of information do we need for account creation?

- Choice 1: Username only.
- Choice 2: Username and password only.
- Choice 3: Username, password and email
- Choice 4: Username, password and phone number.

Decision: Choice 4.

Justification:

We have chosen choice four, because when creating an account, a username and password are mandatory. We could use simply just a username, though that provides minimal security, as we will likely share usernames as a form of friend request system. A username and password combination is likely secure enough, though in the event of the user wanting to change their password via reset, a phone number is imperative to have on file. This is likely all we would need from the user for account creation, thus, we have chosen to use a combination of username, password and phone number for our account creation.

2. How do we implement the selection of location attractions?

- Choice 1: Using preselected locations.
- Choice 2: Web-scraping from Yelp and Tripadvisor.
- Choice 3: Querying Google Maps API.

Decision: Choice 2 and Choice 3.

Justification:

We have chosen to web scrape from sites like Yelp and Tripadvisor, as well as the Google Maps API to select our locations for our players. We could use the preselected locations option, though that would get incredibly cost-expensive, both in the aspect of time and the aspect of storage. We would have to hand-choose locations for every possible city/town, and also store all of them in our database at all times – no, this is not the correct solution. Though, we are planning to add primary attractions for a location into our database. This will help avoid web-scraping unless we absolutely have to. This would also improve response time for the request and give consistency between results. Choosing between Tripadvisor and Yelp would be an arbitrary choice, so we have chosen to use both of them for web-scraping purposes – this lets us generate solutions on the fly, not having to keep all locations for all cities at once in our database. We have also chosen to use the Google Maps API as it also has the ability to query for nearby attractions.

3. How do we handle random group formation?

- Choice 1: Matchmaking based on skill.
- Choice 2: Random association.
- Choice 3: Prioritize friend and/or family networks

Decision: Choice 1 and Choice 3

Justification:

We have chosen to implement matchmaking based on skill and prioritizing family/friend networks instead of random association. We made this choice because the alternative of random association could create uncomfortable situations between random strangers. Matchmaking, or alternatively, skill-based matchmaking would be a slightly more safe alternative, because the people in the groups would be matched up with people of similar "skill level", which would be determined via a point system we will give based on how quick you find the destinations in the game. We will also consider friend/family networks in creating groups by checking to see if members in the user's friend/family network are online and looking for a game, and would match them together.

4. How do we collect the location data of the user?

- Choice 1: Constantly poll for their location, even idly in the background.
- Choice 2: Poll for their location once every predetermined timestep while the application is open.
- Choice 3: Have the user manually enter their location each time.

Decision: Choice 2 and a little of Choice 3.

Justification:

We have chosen to use polling their location during the gameplay with an initial entry of their location. This idea best fits our project's design, as we will use the initial entry of their location to generate their sequence of locations to search for, but we'll poll their location frequently to update the map accordingly – akin to how Google Maps updates the user's location on the map. This is a much better solution than the first choice of constantly polling for their location, because this would become strenuous on the power consumption of our product. We want to create an energy efficient product while still producing effective results, which is what our selection of second choice can produce.

5. How should we communicate how close the player is to their location?

- Choice 1: By showing the radius of the circle shrink as they approach their destination.
- Choice 2: By posting a message to the screen when they are 1000 feet, 500 feet, etc. away from their location.
- Choice 3: By implementing a hot-or-cold system.

Decision: Choice 1, maybe Choice 3 if time permits.

Justification:

We have chosen to implement the circle shrinking as the user gets closer to their destination, because the concept of popping messages up on the screen could get annoying to users over time. This could also be difficult, due to the distance being one of the features the user can select in creating their curated adventure through the app. If we have the radius of the circle shrink, this allows for a good sign to the player that they are getting close to the objective without giving them multiple pop-ups on their screen. It should be a relatively quiet change to the map UI, meaning that the players can spend more time looking around at the city/town they are exploring. If we have time, though, we would like to implement a hot-or-cold system where we have 5 levels – freezing, cold, moderate, hot, burning – to determine how close the player is to their destination. This is an additional feature we would like to add once we have the core implementation finished.

Non-functional Issues:

1. What backend language should we use for our project?

- Choice 1: Flask
- Choice 2: Express.JS
- Choice 3: Next.JS
- Choice 4: Django

Decision: Choice 4

Justification:

We have chosen to use Django for our backend for a couple of reasons. When we were designing the idea for Townie, we decided quickly that we'd need to be doing web-scraping to implement our scavenger hunt idea, which we then realized that it would make the most sense to store our web-scraper on the same server as our backend, which means we'd want to look at something Python-based, which narrowed our choices down to Flask or Django. Because Flask's security is deprecated, and we need to have the best security for our project due to working with user locations and login information, we have decided to use Django, because it is more secure and reliable than Flask.

2. What frontend framework should we use for our project?

- Choice 1: ReactJS
- Choice 2: Svelte
- Choice 3: Flutter
- Choice 4: React Native

Decision: Choice 2

Justification:

Two members of our group are experienced with React, though we believe that a framework like Svelte will be a little more intuitive for us to work with. We plan on creating a website, so a framework like Flutter wouldn't make sense because it is not a very good framework for web development. Similarly, React Native would not make as much sense, because we're planning on focusing more on the website aspect, so it does not make sense to elect to use the framework that's specifically used for multi-platform software. Thus, we were left between ReactJS and Svelte, and we elected to use Svelte because it will be more intuitive and familiar to programming languages that we have worked with in the past.

3. What database should we use to store our data in?

- Choice 1: Redis
- Choice 2: MySQL
- Choice 3: ArangoDB
- Choice 4: MongoDB

Decision: Choice 1 and Choice 3

Justification:

We have chosen to use ArangoDB as our primary database, but will also use Redis to store lobby data for group sessions. We were considering MongoDB due to some of our members having experience with it, though we decided that a graph-based model offers a lot of flexibility for the social aspects of our application. This narrowed our choice of DB engine down to Neo4j or ArangoDB. Ultimately, we decided to go with ArangoDB because, unlike Neo4j, it offers multi-model support (in case we need to switch to a document model) while providing comparable, if not better, performance.

In addition to ArangoDB, we are considering implementing Redis to store lobby data for group sessions. Because this data will have a short lifespan (probably a couple hours at most) and needs to be as fast as possible to ensure data synchronization between group members, we believe that Redis, being an in-memory store, would be the best option.

4. What web services company should we use for hosting?

- Choice 1: AWS
- Choice 2: Azure
- Choice 4: GCP

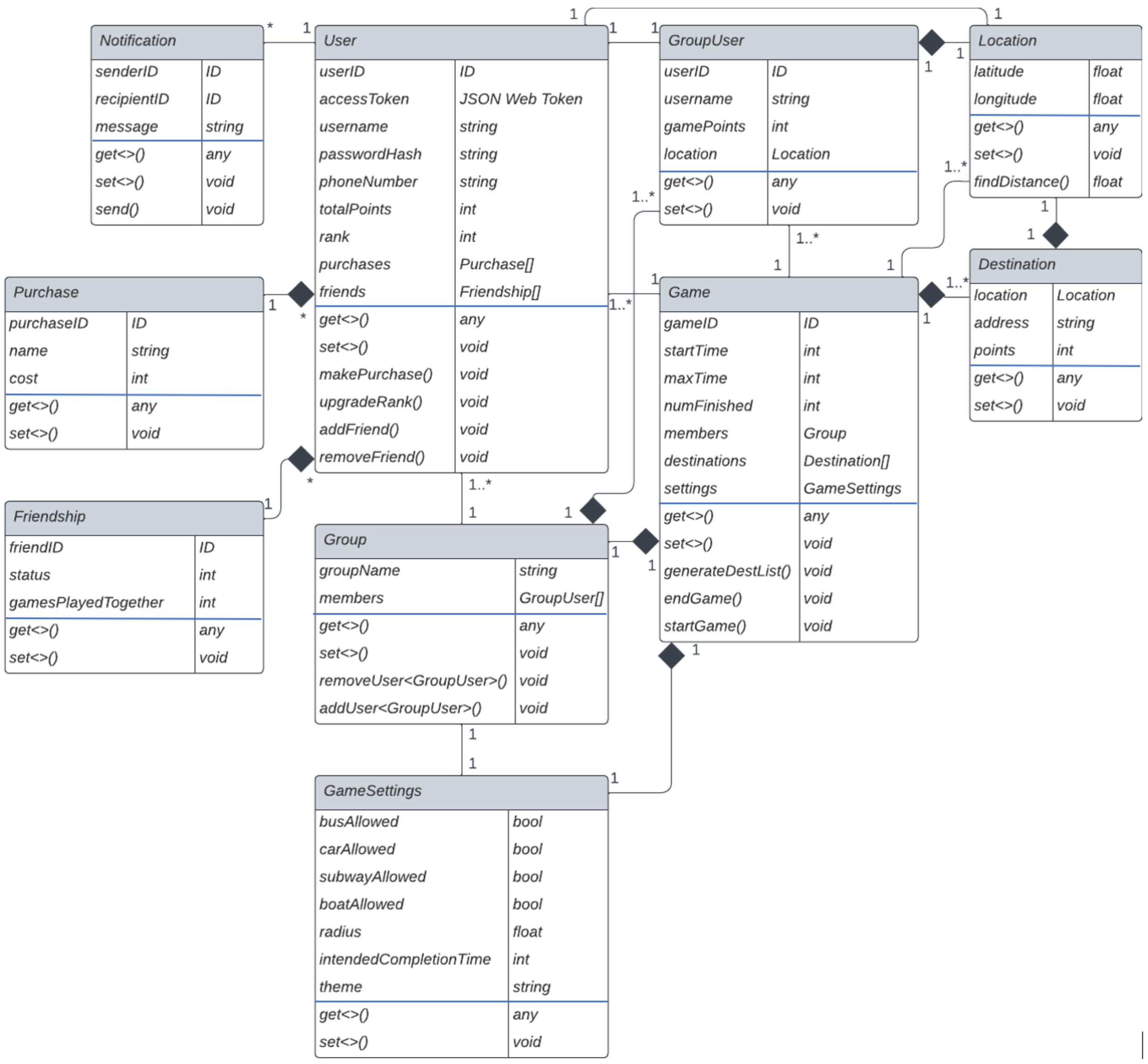
Decision: Choice 1

Justification:

Ultimately, this decision came down to which cloud provider we wanted to use for the entire backend. All companies provide free-tiers that will be good enough to satisfy our basic needs to start, and switching between them should not be overly difficult as all services we require (serverless functions, in-memory data stores, support for building websocket APIs, and something to run ArangoDB off of) are present and easily scalable on all platforms. The reason why we are choosing AWS over the other cloud providers out there is because it is what we have the most experience with. (It also gives us more to talk about in an Amazon interview, lol).

Design Details

Class Design



Descriptions of Classes and Interactions between Classes

These classes were selected with the idea of our game in mind – we wanted to choose our classes or objects carefully so we can apply them efficiently to our application. Each of our classes have a list of attributes and functions, which represent traits and operations that belong to that type of object.

- User Description:
  - A User object is created upon the creation of an account, which will store the username, password, and phone number of the owner.
  - Upon account creation, we will generate a unique identification number to refer to this User object by. This will be used to make storage and retrieval of user information in a database easier.
  - In accordance with OAuth 2.0, we will also store access tokens for each user. As a JSON web token, it will contain, most importantly, the token and an expiry time.
  - We will also store the total points the User has accrued over their previous games. These points will enable users to upgrade their rank (upgradeRank) and make other purchases (makePurchase).
  - We will also have functions to get attributes of the User object and set them, which will primarily be used in the event of updating the information of a User.

  Relations:
  - Each user corresponds to a respective GroupUser, which is essentially a more secure version of the User object for participating in groups.
  - Because multiple users can be associated with a group (via GroupUser), there is also a many-to-one relation with the Group and Game classes.
  - Users also have a one-to-one association with the Location class (via GroupUser) as users can only exist at a single location.
  - Each user can also send notifications to other users, so users will have a one-to-many relationship with the Notification class.
  - As described, users can also make any number of purchases with their points which we will store, giving a contained one-to-many relation with the Purchase class.

    - We also store a list of the user’s friends for ease of joining groups with their friends or family. So, users will have a one-to-many relationship with the Friendship class.

- Notification Description:
    - Because user actions can trigger notifications via signup, friend requests, friend acceptance, friend removals, and game invitations, we must have a way to structure notifications between the client and app server connected to our texting API.
    - For any notifications, we will have the user ids of the sender and receiver and the message being sent.
    - Notifications will have a send() function so they are propagated to the correct recipient.

  Relations:
    - As previously described, a notification will have a many-to-one relation with the User class and users can end up sending any number of notifications for a variety of reasons.

- Purchase Description:
    - Because users can purchase any number of themes, badges, etc. that we decide to create, we must store these purchases.
    - Purchases will have an id for indexing, a name for the item being purchased, and a cost associated with buying a purchasable item.

  Relations:
    - A user can have many purchases, giving a contained many-to-one relation with the User class.

- Friendship Description:
    - In order to keep track of the user’s friendships and their status, we will connect users via this object.
    - As part of this relationship, we will store the friend’s ID

Relations:

  - Because users can have friendships with any number of other users, there is a many-to-one relationship contained with the User Class.

- GroupUser
  - GroupUsers are essentially users without any of the information that is not needed by a group. These special users are used only for the duration a group exists and contain a few special attributes that are useful for the group's game.
  - They contain the same unique ID that their corresponding user has to make adding points to the user database entry simpler.
  - GroupUsers contain a gamePoints attribute that keeps track of how many points the user has earned throughout a game. These points will be added either when the user leaves the group or the game ends.
  - GroupUsers have a location attribute. This will be used to detect how close the player is to the destination, which will update the map in the UI accordingly by shrinking the radius. The getLocation function will provide the GroupUser's location for a method that generates the distance between a GroupUser and a destination.
  - set<> will be used primarily to update the location of the user periodically.

  Relations:
  - Because each GroupUser corresponds directly to a User of the application, there is a one-to-one relation with the User class.
  - Because each Group (and Game) will have a list of GroupUsers, which will be expanded upon later, there will be a many-to-one relation with the Group and Game classes.
  - Lastly, because GroupUser stores a location for the group user, there will be a one-to-one relation with the Location class.

- Group Description
  - A group object is created when a user elects to create a game. It facilitates that game by keeping track of the user involved.

  - Each group will have a name in case the group wishes to name their group.

  Relations:

  - Each group will contain a list of GroupUsers called members that will be accessible to all the users in the group, giving a one-to-many relationship with the GroupUser class (also giving a one-to-many relationship with the User class). Using addUser() and removeUser(), groups have the functionality to add and remove GroupUsers from the group.
  - Because each group was formed to play a game stored in the Game class, a group will actually be stored within the game class, giving a one-to-one relation with the Game class (and the GameSettings class by default).

- Game Description:
  - A game object is central to playing functionality. It keeps track of its live state and is shared by all GroupUsers.
  - Each game will have a unique id, making database storage and retrieval simpler.
  - Each game will have settings stored within a GameSettings object, enabling customizable input for the list of destinations generated by the web scraper.
  - Each game will have a list of destinations that is accessible to all group members. Each GroupUser will be able to progress through this list one at a time. Only one, starting with the first for everyone, will be viewable to the player. Once a player has reached that location, the next destination will be viewable to them.
  - Each game has a creationTime that is set at the beginning of the start of the game.
  - Each game has a timeout time that is set by calculating a multiple of the expected time to reach all the locations. This is a necessary component because it works with creationTime to create an event that forces the game to be ended and the group to be disbanded. This feature exists to prevent games that group elect not to finish from running indefinitely. This condition calls endGame(), which ends the game and disbands the groups.
  - numFinished is used to keep track of all the GroupUsers that have either finished or left the group. This is important because when this number is equal to the number of GroupUsers in the list, endGame() will be called.

  - startGame() is called to initialize the game and route each GroupUser to the group page where the map will be.
  - endGame() ends the game and disbands the group. It also kicks off giving each user the points they are owed both for their rank and their total points.

  Relations:

  - Games will hold a GameSettings object, giving a one-to-one relation with the GameSettings class.
  - Games will hold a Group object, giving a one-to-one relation with the Group class (and therefore a one-to-many relation with the GroupUser and User classes).
  - The list of destinations stored by a Game indicates a one-to-many relation with the Destination Class (along with a one-to-many relation with the Location class).

- GameSettings:
  - This object will hold all the selected settings of a game. These will be used by the web-scraper to determine the most desirable set of destinations to visit.
  - This object will allow users to modify things such as a theme, acceptable transportation methods, and the intended amount of time the set of destinations should take to visit.
  - Additionally, we will store the mask radius of the destination. This means the user will not be able to see the exact location of the destination, but a rough estimate of where it should be.

  Relations:

  - Games will hold a GameSettings object, giving a one-to-one relation with the Game class (and therefore also the Group class).

- Destination
  - Destination has a location attribute, which is of type Location. This is used to generate and interact with a map with a general radius posted.
  - The destination has an address attribute. This attribute is used to allow more casual versions of the game that simply curate a list of destinations and reveal them to the group at once.

    - Destination has a point value associated with it. This is to be used at the end of a game when players need to be rewarded points for playing the game and finding this destination.
    - The points field will dictate the number of points to users when they find the destination.

  Relations:

    - Because a destination has a corresponding location, allowing us to determine if a user has reached the destination, there will be a one-to-one relation with locations.
    - As previously described, the Game class will store the list of destinations in the game, meaning there will be a many-to-one relation with the Game class.

- Location
    - Locations contain longitude and latitude. These give the coordinates necessary to pinpoint both where GroupUsers are and where destinations are so their distances from each other may be compared and the map can be altered to reflect a shrinking distance by creating a new map with a smaller radius.
    - Locations contain an accuracy attribute. This is used to denote how accurate the location provided by Google Maps API is and will be used to make computing distance from a destination more accurate.

  Relations:

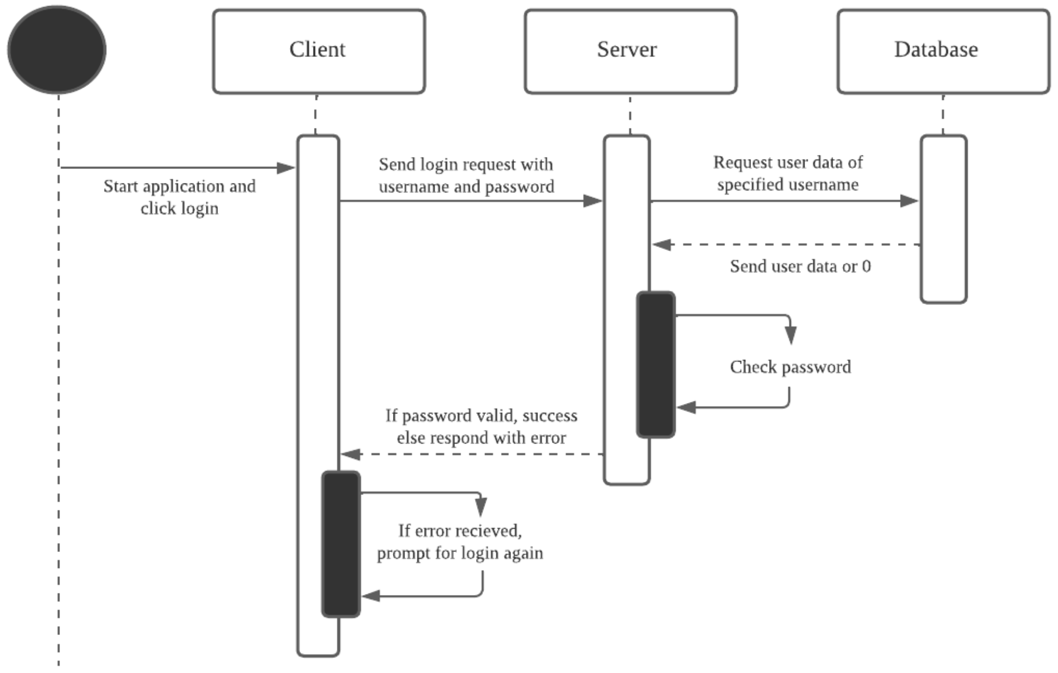
    - Because each destination has a location, there will be a one-to-one relation with the Destination class.
    - Also, because each user can have a location (stored in GroupUser), there will be a one-to-one relation with the GroupUser class (and also User class).

Sequence Diagram:

The diagrams below demonstrate the sequence of events in the major portions of our application, which involve logging in, the creation of an account, initiating a group for a game, starting a game, ending a game, adding and removing friends, and updating user information. The sequence of events diagrams display how our messages will exchange between our client, server and database. If we perform an action as a user on the client side, it will send a message to the server, and the server will then query our database to receive the necessary data – whether it may be a User item or a set of locations. The server will then process the data on the backend and then return the result to the client.

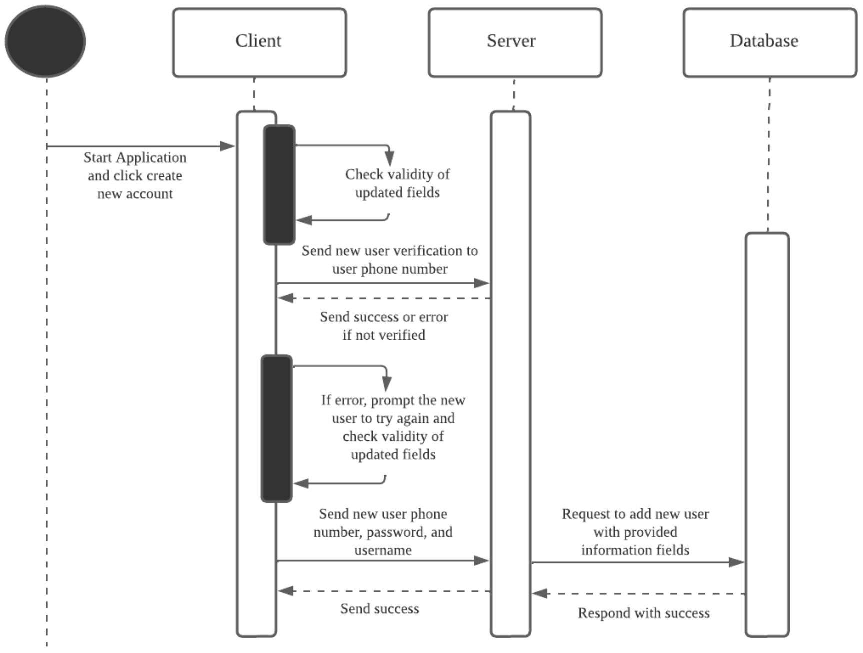
1. Sequence of events when the user attempts to login:

When the user tries to log in, the request is sent to the server, where a request to the database for that User object is made. The database will either send the user data back, or a 0 indicating a failure. After that occurs, the server will check the password, and if valid, the user has gained access. If a failure, the user is prompted to enter a new password.


Client
Server
Database
Start application and click login
Send login request with username and password
Request user data of specified username
Send user data or 0
Check password
If password valid, success else respond with error
If error recieved, prompt for login again

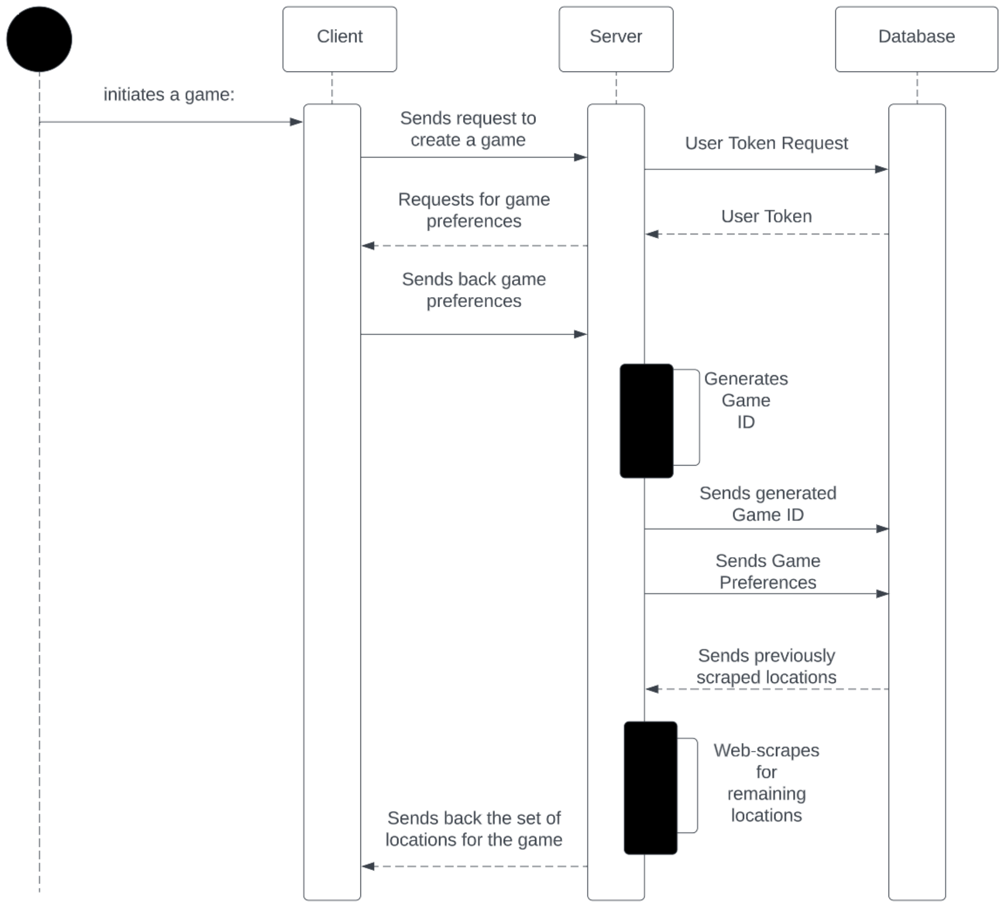
2. Sequence of events when the user creates an account:

When the user creates an account, they are prompted to enter their appropriate fields. Once they have been verified by the user, they are sent to the server, where it will send a text message to the phone number to indicate the account has been made. If a text is not sent, there would be an indication that the account creation failed. After correctly inputting the data, the server sends that user data to the database, and sends back a success once the user has been saved.


Client
Server
Database
Start Application and click create new account
Check validity of updated fields
Send new user verification to user phone number
Send success or error if not verified
If error, prompt the new user to try again and check validity of updated fields
Send new user phone number, password, and username
Request to add new user with provided information fields
Send success
Respond with success

3. Sequence of events when the user initiates a game:

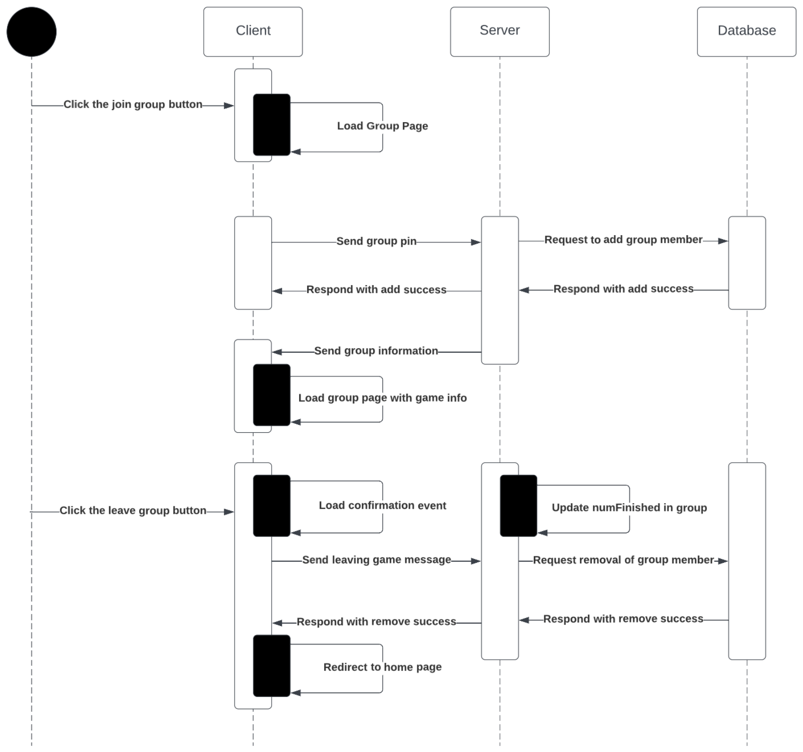
When the user initiates a game, the server authenticates the token that the user has, assuring that our users are valid. After the authentication, the client can input their preferences, which are sent back to the server. The server would then generate a game ID. After that, the server sends that information to the database to extract previously saved scraped locations. The server web-scrapes to gain the remaining locations to reach the location max, and then returns the group.


Client
Server
Database
initiates a game:
Sends request to create a game
User Token Request
Requests for game preferences
User Token
Sends back game preferences
Generates Game ID
Sends generated Game ID
Sends Game Preferences
Sends previously scraped locations
Web-scrapes for remaining locations
Sends back the set of locations for the game

4. Sequence of events for when the user tries to join or leave a group:

When the client chooses to join a group, they can send the group pin to the server, who then sends a poll to the database to add that user to the group. The server would then send group information to the client, which would populate with the group.

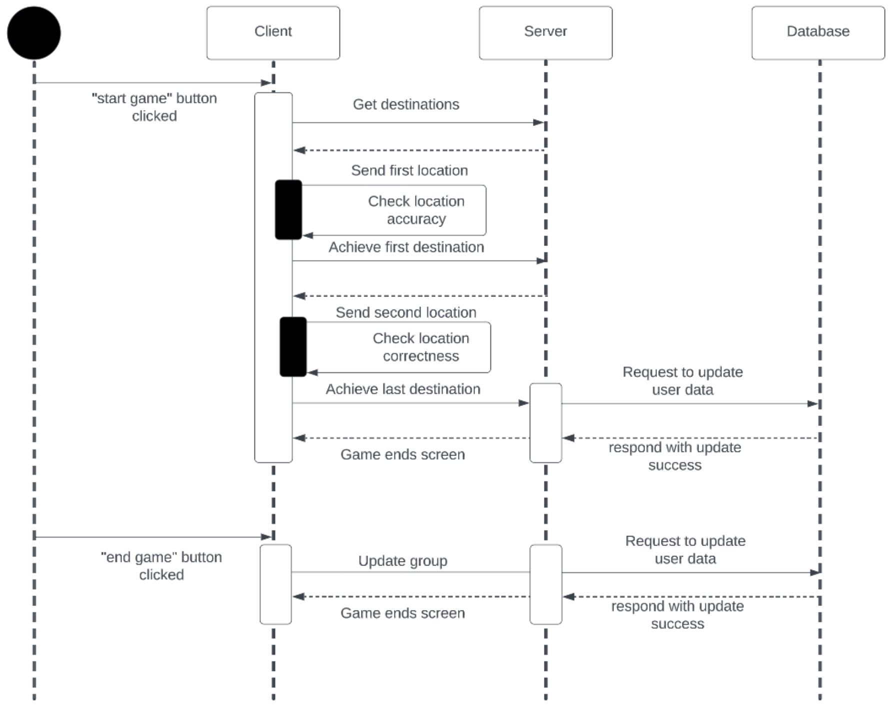
When they try to leave the group, the client would have a confirmation event and the server would update numFinished to keep the game flowing. The client would send a poll to remove the user from the group to the server, and the server passes that to the database. The client would then be sent to the homepage.


Client
Server
Database
Click the join group button
Load Group Page
Send group pin
Request to add group member
Respond with add success
Respond with add success
Send group information
Load group page with game info
Load confirmation event
Update numFinished in group
Click the leave group button
Send leaving game message
Request removal of group member
Respond with remove success
Respond with remove success
Redirect to home page

5. Sequence of events for when the user starts and ends the game:

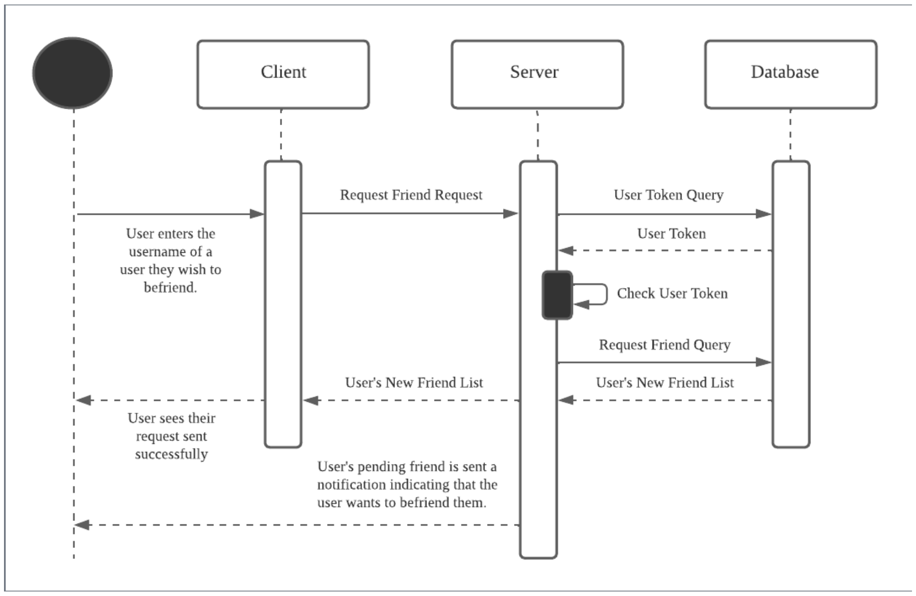
When the start game button is selected, the server sends the first location to the client. Then the client would repeatedly check for the location accuracy until they have reached the first location. Upon reaching the first location, the second location is returned. This process repeats over and over again until the game has ended, where the client sends their groupUser data to the server, and the server sends that to the database.

If the end game button is selected, the server makes the action to update the group to remove the user from the group, which that poll is sent to the database. During this call, the user data is updated with the new point total they have. A confirmation is sent soon after, and they are sent to the “end game” screen.


Client
Server
Database
"start game" button clicked
Get destinations
Send first location
Check location accuracy
Achieve first destination
Send second location
Check location correctness
Achieve last destination
Request to update user data
Game ends screen
respond with update success
"end game" button clicked
Update group
Request to update user data
Game ends screen
respond with update success

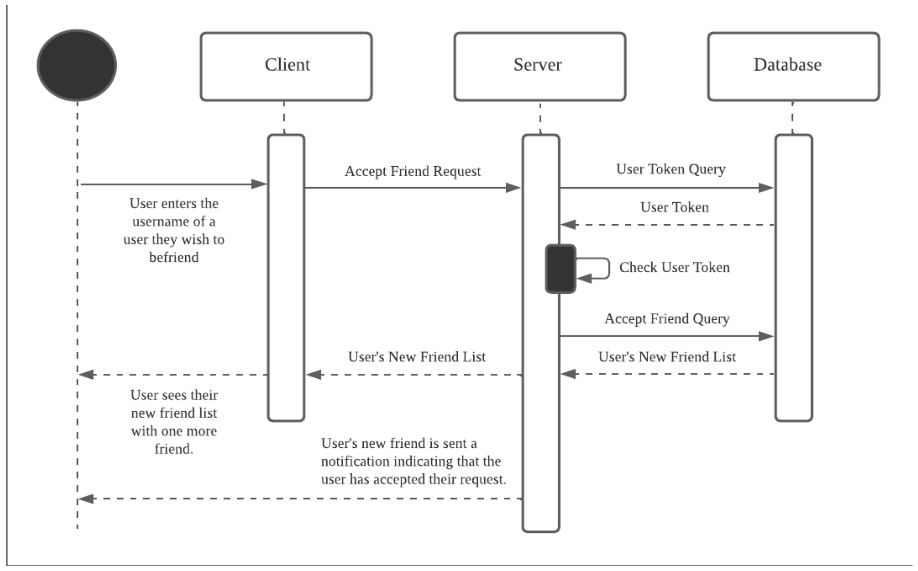
6. Sequence of events for when the user wants to request another user to be their friend:

When the user attempts to request another user as a friend, the client requests a friend request on the server side. The server queries for the user token, and then proceeds to verify the user token. The server then requests the friend request to the database, updating the friends list of the user. Then, the server sends a notification to the client showing that the request was updated successfully, and a notification is sent to the other user's phone number to show they have a friend request.


Client
Server
Database
Request Friend Request
User Token Query
User Token
User enters the username of a user they wish to befriend.
Check User Token
Request Friend Query
User's New Friend List
User's New Friend List
User sees their request sent successfully
User's pending friend is sent a notification indicating that the user wants to befriend them.

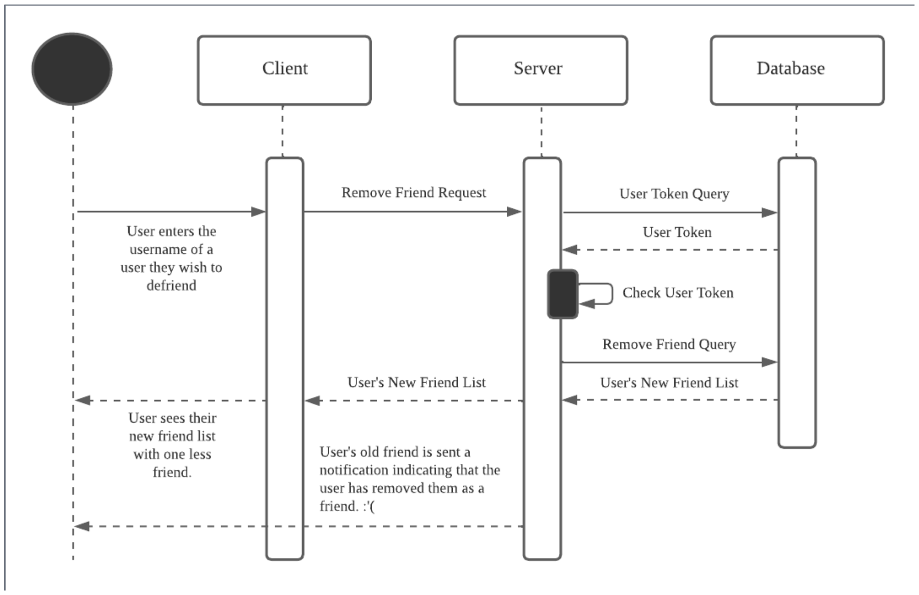
7. Sequence of events for when the user wants to accept another user’s friend request:

When the user attempts to accept a friend request, the client accepts a friend request on the server side. The server queries for the user token, and then proceeds to verify the user token. The server then queries to accept the friend request to the database, updating the friends list of the user. Then, the server sends a notification to the client showing that the request was accepted successfully, and a notification is sent to the other user’s phone number to show that the request was accepted.


Client
Server
Database
Accept Friend Request
User Token Query
User Token
User enters the username of a user they wish to befriend
Check User Token
Accept Friend Query
User's New Friend List
User's New Friend List
User sees their new friend list with one more friend.
User's new friend is sent a notification indicating that the user has accepted their request.

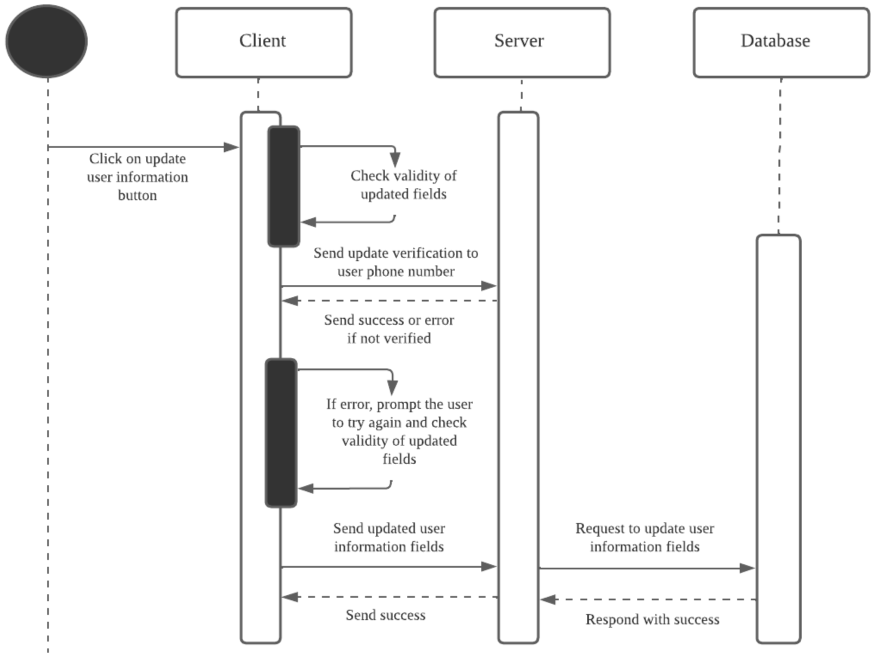
8. Sequence of events for when the user wants to remove a user as a friend:

When the user attempts to remove a friend, the client requests a friend removal on the server side. The server queries for the user token, and then proceeds to verify the user token. The server then requests the friend removal to the database, updating the friends list of the user. Then, the server sends a notification to the client showing that the request was updated successfully, and a notification is sent to the other user's phone number to show that the friend was removed.


Client
Server
Database
Remove Friend Request
User Token Query
User enters the username of a user they wish to defriend
User Token
Check User Token
Remove Friend Query
User's New Friend List
User's New Friend List
User sees their new friend list with one less friend.
User's old friend is sent a notification indicating that the user has removed them as a friend. :'(

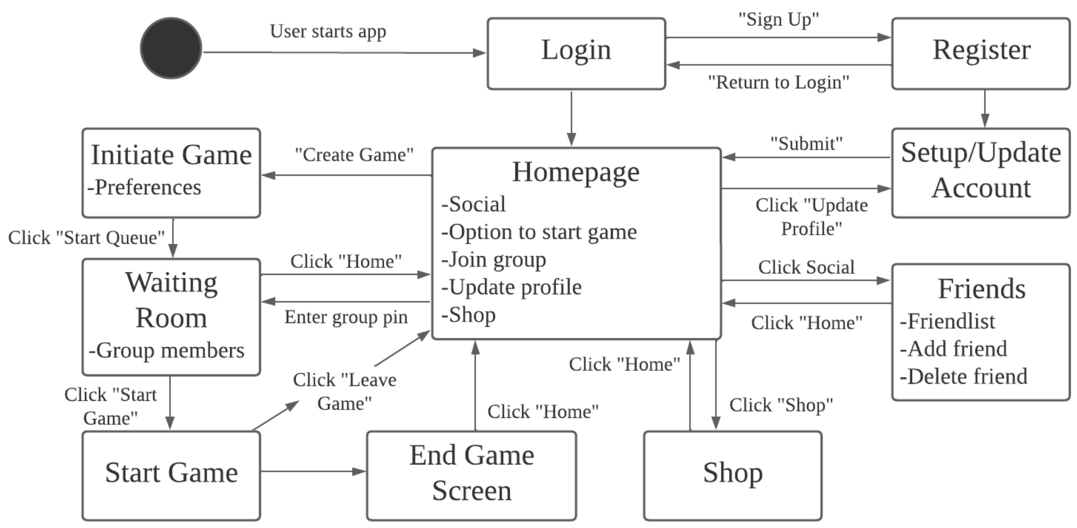
9. Sequence of events for when the user wants to update their information:

When the user wants to update their information, they input new fields into the text boxes. Assuming these fields are valid, the client sends the information to the server. If they are invalid, they are given another opportunity to update their fields. The server sends a request to the database to update the user information, and the database responds with a confirmation. The server then finally sends back either success or error to the client.


Client
Server
Database
Click on update user information button
Check validity of updated fields
Send update verification to user phone number
Send success or error if not verified
If error, prompt the user to try again and check validity of updated fields
Send updated user information fields
Request to update user information fields
Send success
Respond with success

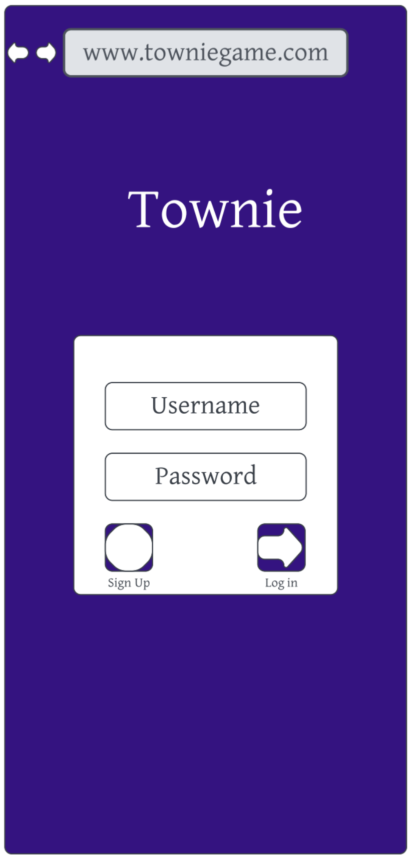
Navigation Flow Diagram

The design of our program navigation is designed for user accessibility and efficiency – our login screen can simply guide the user to the homepage or to the account creation menu. Once they have created an account and logged in, they can initiate a game, which involves giving the user's preference for the game creation; updating account information with a new username, password or phone number; entering a waiting room for a group by inputting the group's PIN; accessing the user's friends list by going to the Social menu; and going to the shop to buy UI upgrades or new ranks. Our navigation tries to allow users to easily flow from one part of our application to another without any difficulty.

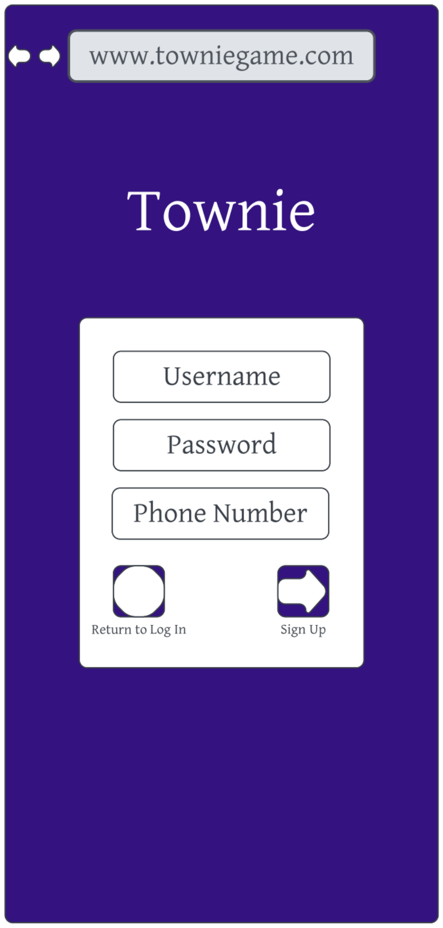


UI Mockup:

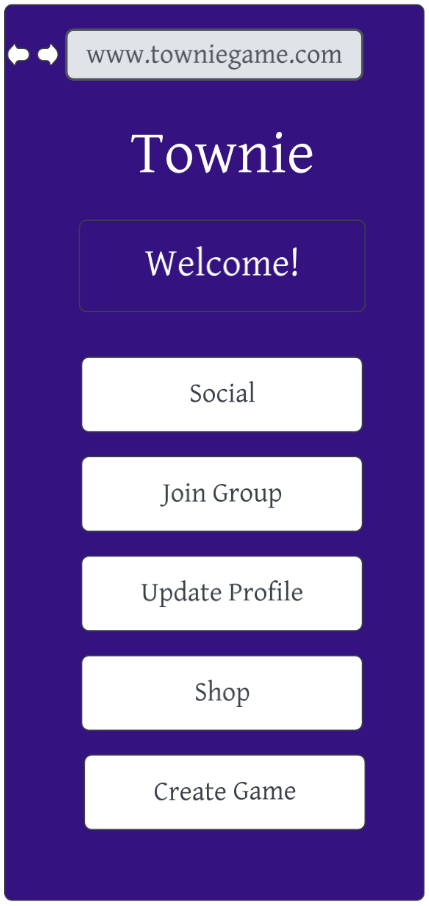
We have designed our UI to have a search bar at the top to represent the location to enter the URL of the website. We have tried to mimic a mobile web browser because we assume this is the intended audience of our game. We have kept all related functions on the same page to avoid any misunderstandings for our users to have to put up with. We have chosen simple boxes for buttons on our UI, though we may end up changing them as we design the product.



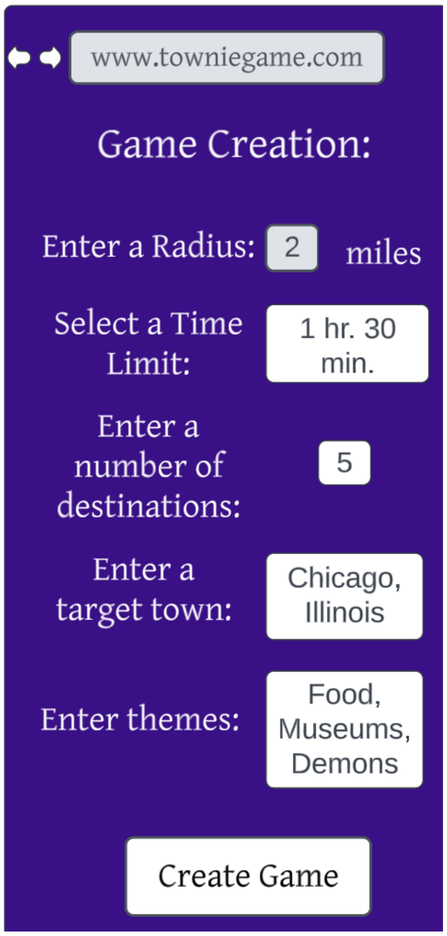
This is our login screen. Here, we prompt the user for their username and password, and then they can attempt to login. If they do not have an account, they can select the “Sign Up” button, which will take them to the account creation menu.



This is the Sign Up menu. Here, users can create their accounts by entering their username, password and their phone number. Once they have submitted correct information, the user can select “Sign Up” to be taken to the home screen, as well as creating the User object in the database. If they do not wish to create an account, they can return to the login menu by using the button “Return to Log In”.



This is the homepage of our site. It is a hub from which users can navigate to every part of our website with just a single click. The user can access their friends list by choosing “Social”, join a group, update their profile information, go to the shop, or create a game.



www.towniegame.com

Game Creation:

Enter a Radius: 2 miles

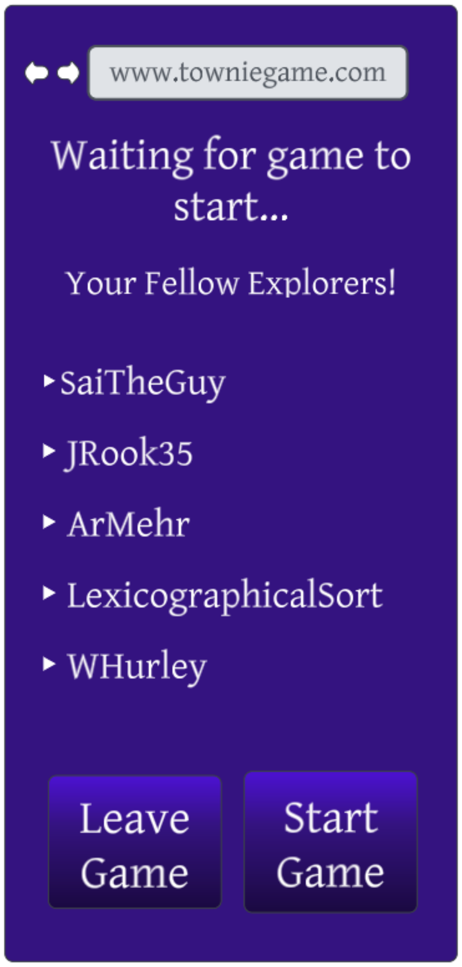
Select a Time Limit: 1 hr. 30 min.

Enter a number of destinations: 5

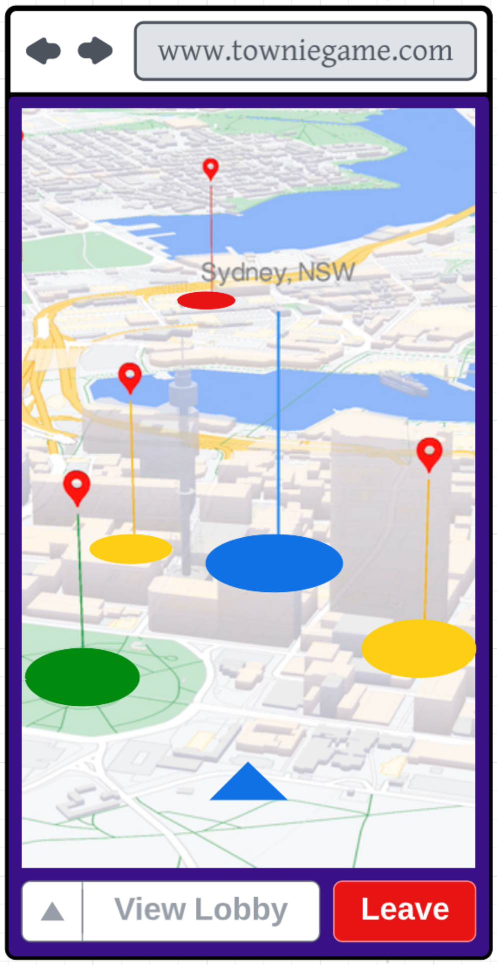
Enter a target town: Chicago, Illinois

Enter themes: Food, Museums, Demons

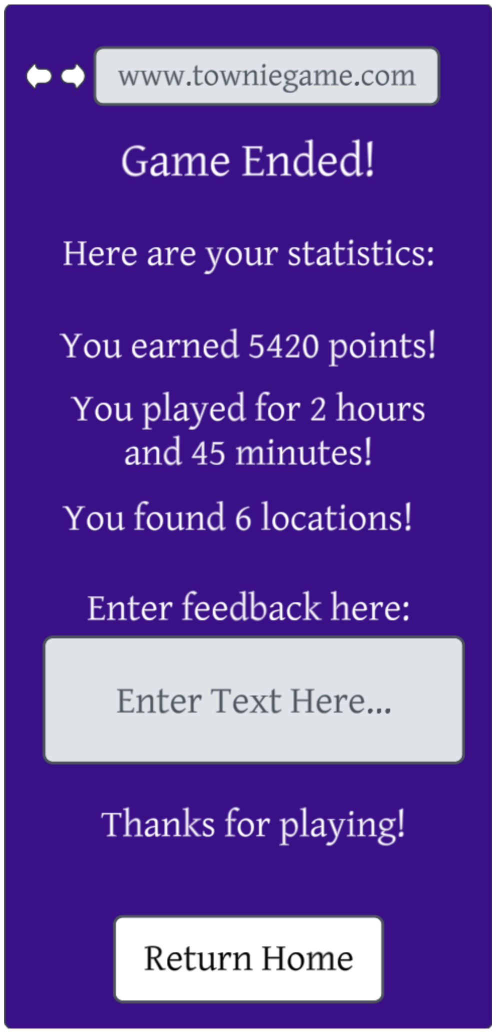
In our game creation menu, we prompt the user for crucial data – how far away should the locations be, how long should the time limit of the game be, how many destinations, their starting town, and any themes they are interested in.



This is our waiting room where users will sit in queue, waiting for their group mates to join before beginning a game.



This is the game screen. This is what the users will see as they are actively playing the game. We are using the Google Maps API to show the map, as well as their destinations with a radius around them. We can see that the user can view the current lobby as well as leave the game.



www.towniegame.com

Game Ended!

Here are your statistics:

You earned 5420 points!

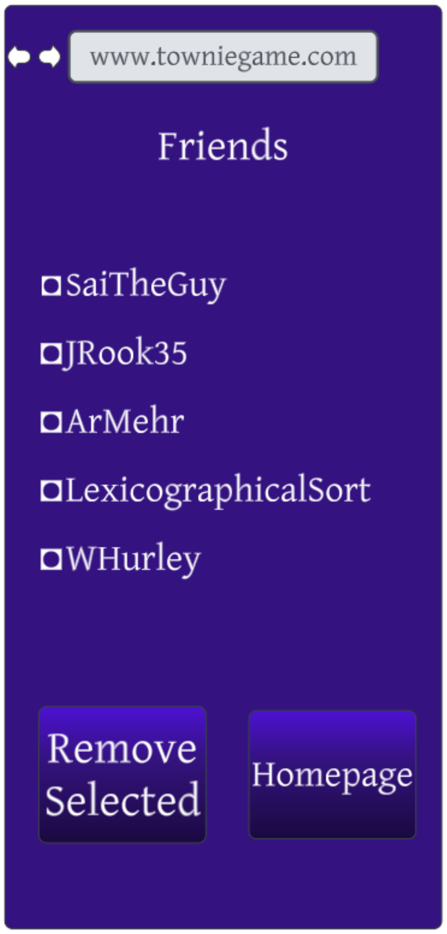
You played for 2 hours and 45 minutes!

You found 6 locations!

Enter feedback here:

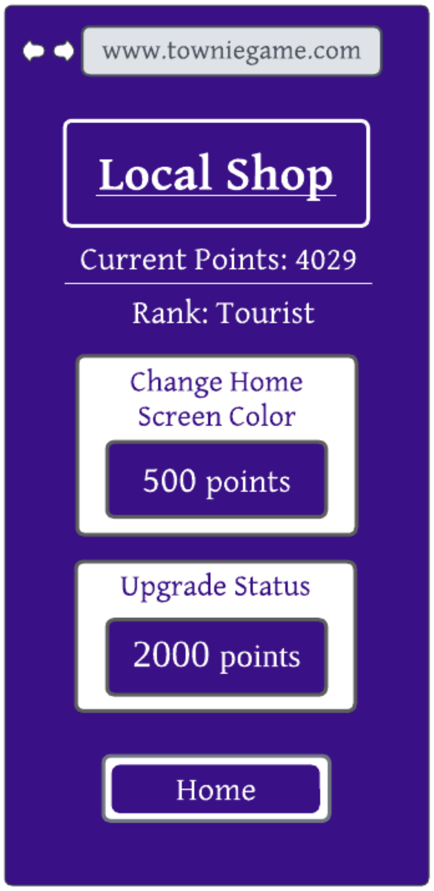
Thanks for playing!

This is the end game screen, which occurs when the group visits all destinations, or the party is empty. It lists the total number of points, the total time played, and the total number of destinations reached. This has a button that returns the user to the homepage.



□SaiTheGuy
□JRook35
□ArMehr
□LexicographicalSort
□WHurley

This is the social screen that shows the user their friends list. Here, you can remove friends, see any new friends you have made, or return to the homepage.



This is the shop screen. Here, users can spend their points to buy different homepage/UI colors, and different ranks for the ranking system.