Design Document for Geohunt

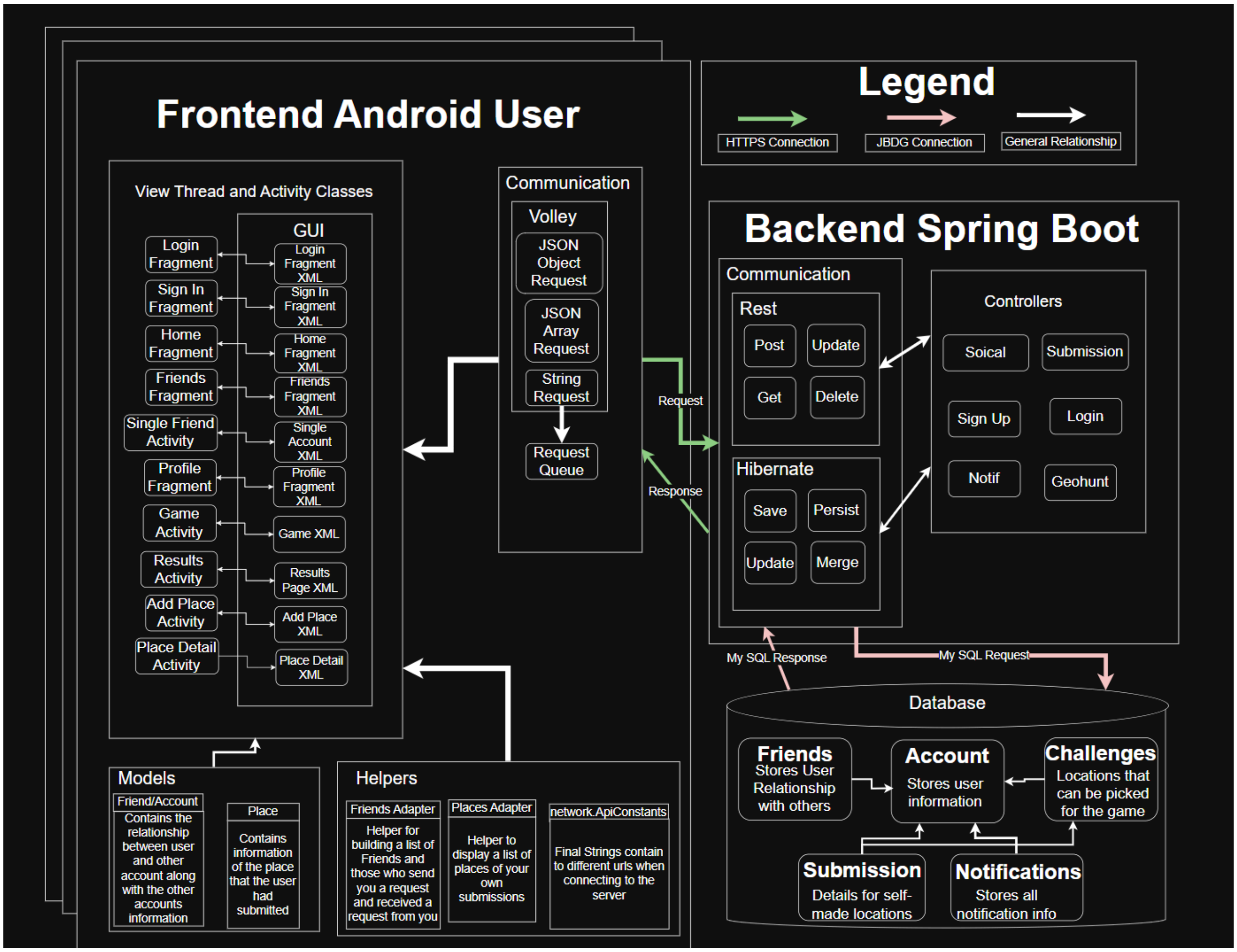
Group 2_Jubair_5

Nathan Imig: 35%

Alex Remiasz: 15%

AJ Tripathi: 35%

Evan Julson: 15 %

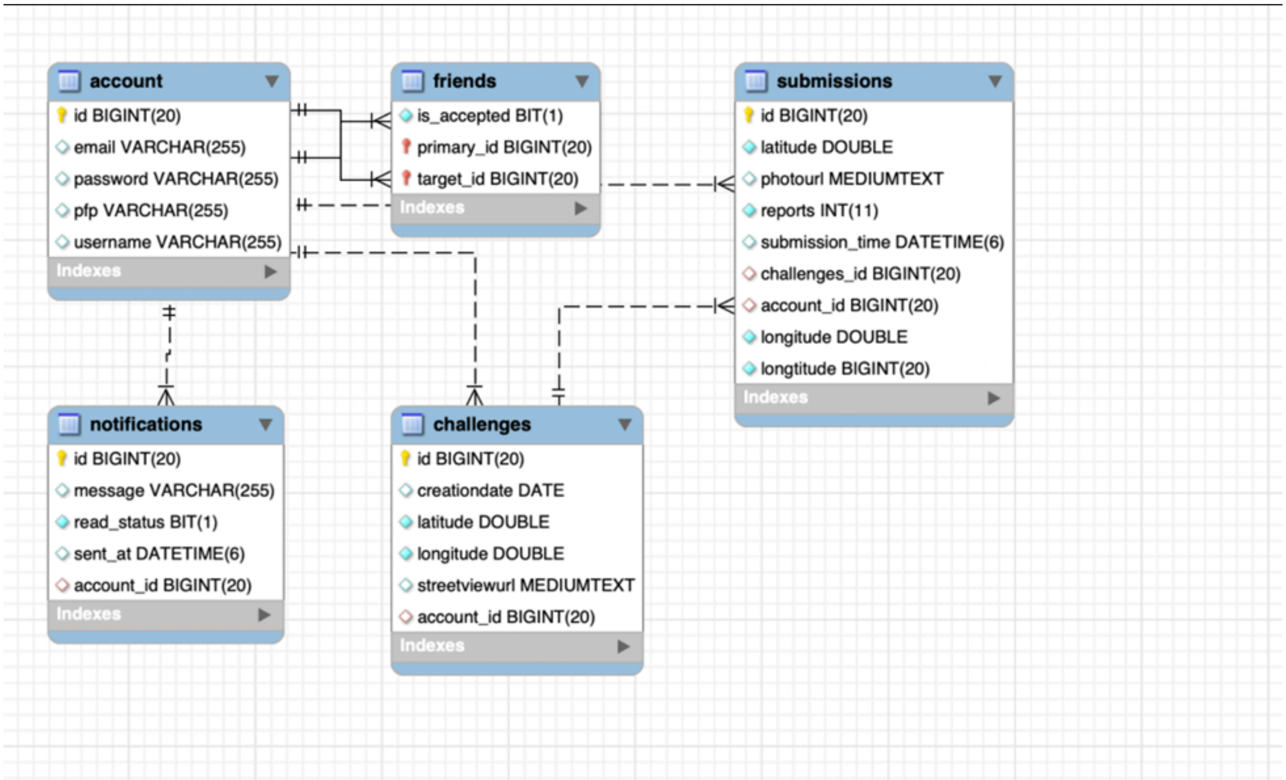

Frontend Android User
View Thread and Activity Classes
GUI
Login Fragment
Login Fragment XML
Sign In Fragment
Sign In Fragment XML
Home Fragment
Home Fragment XML
Friends Fragment
Friends Fragment XML
Single Friend Activity
Single Account XML
Profile Fragment
Profile Fragment XML
Game Activity
Game XML
Results Activity
Results Page XML
Add Place Activity
Add Place XML
Place Detail Activity
Place Detail XML
Communication
Volley
JSON Object Request
JSON Array Request
String Request
Request Queue
Models
Friend/Account
Contains the relationship between user and other account along with the other accounts information
Place
Contains information of the place that the user had submitted
Helpers
Friends Adapter
Helper for building a list of Friends and those who send you a request and received a request from you
Places Adapter
Helper to display a list of places of your own submissions
network.ApiConstants
Final Strings contain to different urls when connecting to the server
Legend
HTTPS Connection
JBDG Connection
General Relationship
Request
Response
Backend Spring Boot
Communication
Rest
Post
Update
Get
Delete
Hibernate
Save
Persist
Update
Merge
Controllers
Soical
Submission
Sign Up
Login
Notif
Geohunt
My SQL Response
My SQL Request
Database
Friends
Stores User Relationship with others
Account
Stores user information
Challenges
Locations that can be picked for the game
Submission
Details for self-made locations
Notifications
Stores all notification info

FrontEnd:

- ***Fragments and Activities***
    - Main Activity
        - Contains the home, friends, and profile fragment
    - Authentication Activity
        - Contains the login and sign up fragments
- ***Models***
    - Friends
        - Uses a friend class that will be used by the friends adapter. This adapter will put the users friends in a recycler viewer that the user can interact with
    - Place
        - Uses a place class that will be used by the place adapter. This adapter will display their own submission in a recycler viewer that the user can interact with

Backend:

- ***Generating/Fetching Images:***
    - For challenges, generating images is a slightly complex process. It utilizes the Google Maps API, specifically the Street View API, and the Places API.
    - This API requires a specific LAT and LON. Since our input is a latitude, a longitude, and a radius, I used math to find a random point within this circle. The math is explained below, written in *randomLocation()*
    - ***randomLocation() logic:***
        - Firstly, the math computes the radius given in radians. This is essential since our lat and lon are in degrees.
        - To generate a random point, we use two random numbers (u, v in the code)
        - v is used to compute a random angle, in radians, using the formula 2π*v.
        - u is used to compute a random distance, using radiusRadians * sqrt(u)
        - Next, the spherical law of cosines formula is used for latitude, and longitude uses a different formula that also corrects for the longitude shrinking as you move away from the equator, called the geodesic longitude formula.
        - These values are returned
    - After we have this random point, we use the Google Maps API to get an image in this location. However, this is achieved through two functions: *generateChallenge()* and *fallbackGenerate()*.
    - ***generateChallenge()***
        - generateChallenge() uses the Google Maps Places API. This ensures that the image we get is recognizable (such as a landmark or something similar). Although this means that images will be easily guessable, it also reduces the total number of possible images, lowering the total number of times this function can return an image. When no new images can be generated, *fallbackGenerate()* is called.
    - ***fallbackGenerate()***
        - fallbackGenerate() uses the regular streetview API, which means that images can contain very hard-to-identify objects, such as cornfields.
    - **Database Fetches:**
        - To return all challenges that fall within the user's current latitude, longitude, and radius constraints, the Haversine formula is used. This ensures that the distance between two sets of (lat, lon) coordinates is within the radius constraints.



account
id BIGINT(20)
email VARCHAR(255)
password VARCHAR(255)
pfp VARCHAR(255)
username VARCHAR(255)
Indexes

friends
is_accepted BIT(1)
primary_id BIGINT(20)
target_id BIGINT(20)
Indexes

submissions
id BIGINT(20)
latitude DOUBLE
photourl MEDIUMTEXT
reports INT(11)
submission_time DATETIME(6)
challenges_id BIGINT(20)
account_id BIGINT(20)
longitude DOUBLE
longtitude BIGINT(20)
Indexes

notifications
id BIGINT(20)
message VARCHAR(255)
read_status BIT(1)
sent_at DATETIME(6)
account_id BIGINT(20)
Indexes

challenges
id BIGINT(20)
creationdate DATE
latitude DOUBLE
longitude DOUBLE
streetviewurl MEDIUMTEXT
account_id BIGINT(20)
Indexes