WABBI: A Unified, Gamified Platform for Everything

Group 5
Niek Peters (s3126714), Junseo Kim (s2648687), Milan Oosterink (s3127168),
Matas Aizenas (s3178757), and Dio Nirwikara (s3176991)

University of Twente

M11 Design Project: Design Report

April 17, 2026

Table of Contents

1. Introduction

"Wabbi" is the company founded by our client, and this project represents a potential future product of the company. The motivation behind this project is to address the fact that, currently, there are a countless number of mobile apps that each provide their own distinct service to users. This vast catalogue of applications leads to users having to tediously download, organize, and maintain such applications on their phones. To solve this problem, Wabbi aims to provide a unified medium for information and services. This platform is designed for monetization through commissions, subscriptions, and one-time transactions, while incorporating gamification elements to drive virality and user engagement.

The project focuses on developing a mobile-first application, also available as a web application, that integrates the promotion of Points Of Interest (POIs) and experiences on a map with gamification. In this context, a "POI" represents a physical location such as a stadium, restaurant, house, or square. An "experience" represents any activity or event that can occur at a POI such as a football match, music festival, and a dinner service of a restaurant. The two goals of the project are, first, to provide a medium where POIs and experiences can be displayed and interacted with, and, second, to implement a gamification element to increase user engagement.

This project involves developing an initial, but easily extendable, prototype for the application. The ultimate goal of the client is to evolve this prototype into a Minimum Viable Product (MVP) following the conclusion of this project.

1.1 Report Overview

This report details the comprehensive design of the project, moving from initial conceptual requirements to final technical execution. The report begins by establishing the core functional and non-functional requirements in Section 2. Sections 3 and 4 outline the system's architecture, while Section 5 provides a technical explanation of the backend services. Then, the frontend is detailed in Section 6. Sections 7 and 8 address the underlying database structure and performance considerations necessary for the prototype. Finally, in the subsequent sections, the report documents the testing methodologies and development process, concluding with an evaluation of future work and the achieved results.

2. Requirements

This section outlines the primary requirements defined by the client that serve as the foundation for the project. While the comprehensive list of all functional and non-functional requirements is available in the Appendix (See Appendix 2.), the following overview focuses on the "must-have" features essential to the initial prototype, as these were the main goals of this project. The overview is accompanied by a justification, which details the client's specific motivations for the requirements. While our requirements list did change throughout our development time, shown below and in the appendix is our requirements list by the end of the project.

2.1 Functional Requirements

This section details the essential features the application must perform, categorized by their functional domain.

2.1.1. Map

- The system shall have a map of the whole world that can be divided into arbitrary sized hexes.
- The system shall have hexes that have other hexes within them.
- The system shall have hexes that can be captured.
- The system shall have a map with the hex system for the Netherlands.
- The system shall have the map with the hex system be theoretically scalable to the whole world.

The client required that a global map interface be used to display the POIs and experiences. A hexagonal grid system was also required to be used throughout the map. Moreover, the hex sizes needed to vary depending on the zoom level of the user such that they resembled territories in all zoom levels. These hexes were to be implemented in a way that they could be captured by groups of users. For the prototype, the map with the hex system just needed to be implemented for the Netherlands, but should be made to be easily extendable to the whole world.

2.1.2. Layers

- The system shall contain functionality for implementing map layers.
- The system shall allow experiences bound to a layer to be made by users.
- The system shall allow layers to be selected on the user interface.
- The system shall provide the user with a layer-specific set of options when creating experiences.
- The system shall allow experiences bound to a layer to be added from external sources via APIs or web scraping.
- The system shall contain an example layer with venues and events.

The client requires the app to have several layers, each focusing on a different type of experience (e.g., music concerts, sports events, and restaurant services). This distinction keeps the application organized and reduces visual clutter. The application must support a multi-channel approach for populating the map and its various layers. This requires the system to ingest data from two streams: user-created content and external data integration. Individual users must be able to

manually create experiences, and the automatic collection of experiences from third-party sources via web scraping or API integrations must be supported. For the scope of this project, it was agreed that the TicketMaster API be integrated into the general “Events” layer as a Proof of Concept (PoC) for external integration. User-created experiences will also be put on the “Events’ layer for the prototype, but the interface will allow different create configurations based on the specific layer being accessed.

2.1.3. Points of Interest (POIs)

- The system shall have POIs with latitude/longitude, name, description, (optional) pictures.
- The system shall allow users to create POIs.
- The system shall have POIs be attached to a hex.
- The system shall have user created POIs be linked to an account.
- The system shall have 2 types of POI visibility: temporary and permanent.
- The system shall make all user POI visibility temporary by default, so personal locations such as one’s house do not always show up on the map.
- The system shall allow users to request a POI’s visibility to become permanent.
- The system shall have POI popularity increase when users make contributions.
- The system shall base POI visibility in a layer on its popularity in a certain timeframe, relative to the popularity of other POIs in that layer.

Points of Interest (POIs) represent physical locations on the map, such as restaurants, venues, or shops. The system requires that POIs be either user-created or ingested from external sources. When a user creates a POI, they are automatically designated as the owner. POIs serve as the foundational pieces of the application. Since all experiences are attached to a POI, users can intuitively click a location on the map to view associated experiences.

Moreover, the system distinguishes between temporary and permanent POIs. User-created POIs are initially designated as temporary to account for their unverified status. In contrast, externally sourced POIs represent established public locations. Through the premium account feature, users can upgrade their POIs to permanent status or request ownership of an existing externally sourced POI.

2.1.4. Experiences

- The system shall have experiences with name, description, (optional) pictures.
- The system shall have experiences attached to a POI.
- The system shall allow users to host experiences at a POI.
- The system shall allow users to sign up for experiences.
- The system shall allow the owner of an experience to configure whether users can sign up and/or check in at the experience.
- The system shall track the popularity of experiences that are planned or happening.
- The system shall indicate on the map when a POI has an experience planned or happening.

Experiences represent the activities or events hosted at specific POIs. The system requires that every experience be linked to a parent POI and remain fully interactive for the user. Like POIs, these experiences are populated via two streams: user-generated content and external data sources. To maintain administrative control, a user-created experience can only be generated by the verified owner of the corresponding POI. Additionally, each experience includes a dedicated schedule to denote its specific occurrences and duration, allowing users to track events in real-time.

2.1.5. Contributions

- The system shall allow users to check in to experiences.

The client requires that users of the application be able to make contributions to POIs and experiences. These contributions take various forms, including the addition of images, reviews, or metadata updates such as opening times. Furthermore, the system must support presence tracking: users can "sign up" for an experience to indicate their intent to attend, or "check in" to denote that they are physically present at the location.

2.1.6 Users

- The system shall have the map be viewable without an account.
- The system shall have user accounts.
- The system shall have users verify their email before allowing account-only functionality.
- The system shall allow users to change their password via email.
- The system shall allow users to delete their accounts.
- The system shall have a map interface for users.
- The system shall allow users to create accounts and log in.
- The system shall allow users to gain experience points.
- The system shall allow users to refresh the map at any point.
- The system shall allow users to report POIs, experiences, other users, groups, reviews and photos for moderator review.

The client requires a robust account system while ensuring the map remains viewable for unauthenticated users. The system must support standard account management features, including email verification, password recovery, and account deletion. A clear distinction must be maintained between standard user accounts and paid premium accounts. Upon authentication, the map interface must display user-specific content, such as their own created POIs and experiences, and allow for manual data refreshing at any point. Finally, users must be able to gain and use experience points within the application.

2.1.7. Moderators

- The system shall have moderator accounts.
- The system shall include a dashboard where moderators can review reported content.
- The system shall hide content when it is deemed harmful by a moderator.
- The system shall allow moderators to delete contributions if need be.
- The system shall allow moderators to ban users (by email).
- The system shall allow moderators to review requests for POIs to become permanent.

The client requires moderator accounts to oversee and moderate user-generated content. An administrative dashboard must be provided to allow moderators to review reported items and subsequently hide or delete content deemed harmful. Moderators must also have the authority to ban or unban users via their linked email addresses. Additionally, moderators are responsible for reviewing and addressing user requests to transition POIs to permanent status.

2.1.8. Gamification

- The system shall allow users to earn experience points for contributions.

- The system shall allow users to view experience points earned by users on a global leaderboard.
- The system shall allow users to create one group each with a name, description and colour.
- The system shall allow users to join one group at a time.

The client requires the application to incorporate gamification elements to drive user engagement. Users earn experience points by making contributions to POIs or experiences, competing for the highest scores on a global leaderboard. Furthermore, the system must allow users to create and join groups to facilitate collaborative play. Groups compete to capture hexes by contributing more cumulative experience points than rival groups within a specific hex. To indicate successful capture, the hex takes the group's designated color and adopts the group's line style as an outline. This competitive mechanism serves to encourage frequent platform engagement and social interaction.

2.2 Non-Functional Requirements

This section details the requirements that define the quality attributes and constraints of the application.

2.2.1. Interface

- The system shall be usable without a tutorial.
- The system shall have a user interface that uses consistent design language.
- The system shall have the interface be usable for any device with a web browser.

The client requires the application to feature an intuitive User Experience (UX) supported by consistent design standards. Furthermore, the system must implement cross-platform responsiveness to ensure a seamless and functional experience across both mobile and web interfaces.

2.2.2. Security

- The system shall have passwords that are hashed and salted.

The client requires the implementation of secure password management protocols for the prototype. This requires the use of industry-standard encryption and hashing algorithms to ensure that user credentials remain protected.

2.2.3. Deployment

- The system shall be deployable without vendor lock-in.
- The system shall theoretically scale to any amount of users without changes to the implementation.

The client requires the prototype to be built using software that prevents vendor lock-in to ensure long-term flexibility. Open-source software should be utilized wherever possible, with a primary goal of self-hosted deployment over reliance on proprietary cloud services. Furthermore, all design and architectural choices must be prioritized for scalability to support the platform's eventual expansion from a regional prototype to a global application.

3. System Architecture

This section provides a conceptual overview of the system's architecture, with detail on the structural decisions made to satisfy the client's requirements. It outlines the integration between the mobile-first frontend, microservice-based backend, and scalable infrastructure.

3.1 Architectural Pattern: Microservices

To ensure the system remains easily extendable and inherently distributed, we adopted a microservice architecture. This design pattern splits the application into loosely coupled and independently deployable components called services. By isolating functionalities into distinct services, the system can be scaled granularly, and new features can be integrated without disrupting the existing services.

For this project, the backend was split up into four different services: the API, the Experience Scheduler, the Experience Scraper, and the Tileserver. As shown in *Figure 1*, these microservices operate as independent units that communicate with both the clients and one another via HTTP protocols.

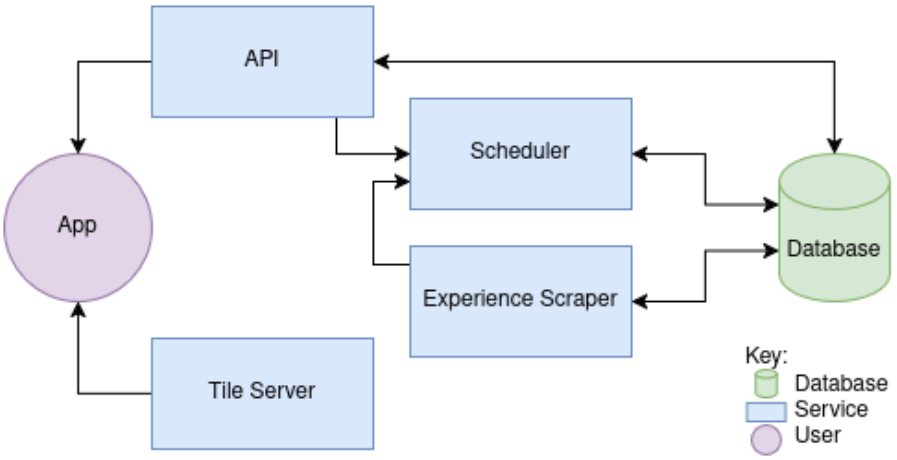


Figure 1: Architecture of different microservices with data flow visualized

The main reason for choosing a microservice architecture was to strategically separate the system into stateless and stateful services. The stateless services are the API and Tile Server and the stateful services are the Experience Scheduler and Experience Scraper.

This distinction is critical for scalability. Stateless services serve the users directly, and because each instance operates independently without needing to synchronize state nor communicate with others, they can be easily replicated. This allows the system to support a high volume of concurrent users by horizontally scaling these instances as needed. Conversely, if all of the services were combined into one application, the entire system would inherit the complexities of the stateful components. This would make replicating instances significantly more difficult and expensive, ultimately decreasing the system's ability to support a large user base.

Furthermore, this architecture allows for long-term extensibility through loose coupling. Since each component is completely independent from the rest, components can be replaced or upgraded

without impacting the rest of the system as long as the new component communicates in the same way. This modularity allows for significant future-proofing, such as changing the programming language of a component to one more suited for the task or fundamentally changing the functionality of a component without requiring major changes to the system.

3.2. Distributed Components

To realize this scalable and modular nature of using a microservice architecture and ensure that the application remains effectively distributed across multiple machines, the system is distributed into several components. This infrastructure ensures the platform can accommodate a growing user base while maintaining high performance. The following subsections detail the role and responsibility of each component within the distributed environment.

3.2.1. Kubernetes

Kubernetes serves as the orchestration layer that hosts and manages the lifecycle of all containers of the application. It acts as the central brain of the infrastructure, ensuring that each microservice is deployed correctly and remains in operation, which is essential for maintaining high availability (See Section 4.).

3.2.2. API

The API acts as the primary communication bridge between the Flutter clients and the backend logic, handling user authentication and data requests (See Section 5.1.).

3.2.3. Experience Scheduler

The Experience Scheduler is a background service responsible for transitioning experience states (e.g., past, ongoing, soon-upcoming, upcoming) based on the real-time scheduling logic of an experience (See Section 5.2.).

3.2.4. Experience Scraper

The Experience Scraper is an automated integration service that ingests data from external sources and third-party APIs to populate the POIs and experiences on the map (See Section 5.3.).

3.2.5. Tile Server

The Tile Server is a specialized service dedicated to storing and serving geospatial map tiles to the frontend for efficient map rendering. By delivering map data in small, pre-rendered tiles rather than a single large file, it ensures a high-performance experience when navigating the map (See Section 5.4.).

3.2.6. Web and Mobile Applications

The web and mobile applications provide identical functionality and user interaction patterns. The only difference is that the mobile application is downloaded and installed locally by users, whereas the web application is hosted on a server and accessed via a web browser. Both versions are developed using the Flutter cross-platform framework and the Dart programming language. This

allows the application to be compiled from a single codebase for both mobile and web applications, ensuring a consistent User Experience (UX) across all devices.

3.2.7. Nginx Load Balancer

Nginx is an open source application utilized as a load balancer within the system. It manages incoming traffic by distributing request loads across multiple service instances to ensure high availability and optimal performance. Additionally, Nginx is responsible for hosting and serving the compiled Flutter web application files to users accessing the platform via a browser.

3.2.8. YugabyteDB

YugabyteDB is a distributed SQL database chosen as the system's primary data storage. Its distributed architecture ensures that data remains resilient across multiple nodes. This choice specifically addresses the requirement for horizontal scalability, as it allows the platform to maintain data consistency across a distributed environment.

4. Infrastructure

This section details the underlying infrastructure and orchestration layer of the platform. To meet the client's requirements for a self-hosted and scalable system, the infrastructure was designed to ensure that the application remains functional across various hardware environments. The primary focus of this layer is to provide a stable environment for the distributed components defined in the previous section.

4.1 Kubernetes Environment

Kubernetes was chosen as the deployment target for this application due to its robust capabilities as a container orchestration platform. It enables the automatic scaling, deployment, and managing of containers across a cluster of machines. In this context, a container is a standardized, ready-to-run software package containing all the necessary application libraries and runtimes to run the software.

The decision for Kubernetes was driven by the need for high-density scaling. As the user base grows, the application must distribute its workload across several machines. Kubernetes automatically scales containerized services to accommodate traffic spikes. In contrast, simpler runtimes such as Docker Compose lack native support of orchestration features and would require significant manual intervention to manage networking, updates, and load distribution. Kubernetes manages this complexity through its internal DNS service, which automatically routes traffic even as workloads shift across the cluster.

Other similar platforms to Kubernetes, such as Red Hat OpenShift and Docker Swarm, were also considered. The reason why OpenShift was not chosen was because it is a paid enterprise level application. Many other alternatives to Kubernetes were paid software and therefore not suitable for this project without a budget. We also considered Docker Swarm, but chose to not use it, as it is significantly less mature compared to Kubernetes.

Kubernetes splits the workload into basic units called Pods. A Pod is the smallest deployable unit, consisting of one or more containers performing a single task. Pods are then grouped into Deployments which are controller objects that supervise Pods and ensure that the desired number of instances are running and are healthy. Pods are frequently created and destroyed to balance the load equally across the cluster which, in turn, also changes their network addresses. In order to provide these Pods with stable network addresses and to ensure that they are accessible, they are grouped into abstraction layers called Services. A Service provides a stable network address which allows for load balanced access to the individual Pods managed by a Deployment.

4.2 Kubernetes Cluster Configuration

To ensure logical isolation and security, each microservice is deployed within its own Namespace. Stateful components, specifically the YugabyteDB nodes, utilize Persistent Volumes (PV) to ensure data persists across Pod restarts or migrations.

Security is managed through Kubernetes Secrets, which inject sensitive configuration data, such as API keys and database credentials, into the containers at runtime. This decoupled approach ensures that sensitive information is never hardcorded into the container images, mitigating the risk of data exposure if an image is leaked.

To manage the complexity of deploying third-party applications like YugabyteDB and Nginx, Helm Charts were utilized. As a package manager for Kubernetes, Helm automates the creation of Deployments, Services, and Persistent Volumes. This ensures that the infrastructure is set up correctly and follows best practices for configuration while remaining easy to update and maintain. *Figure 2* provides a visualization of the cluster's topology, illustrating the interaction between namespaces and services.

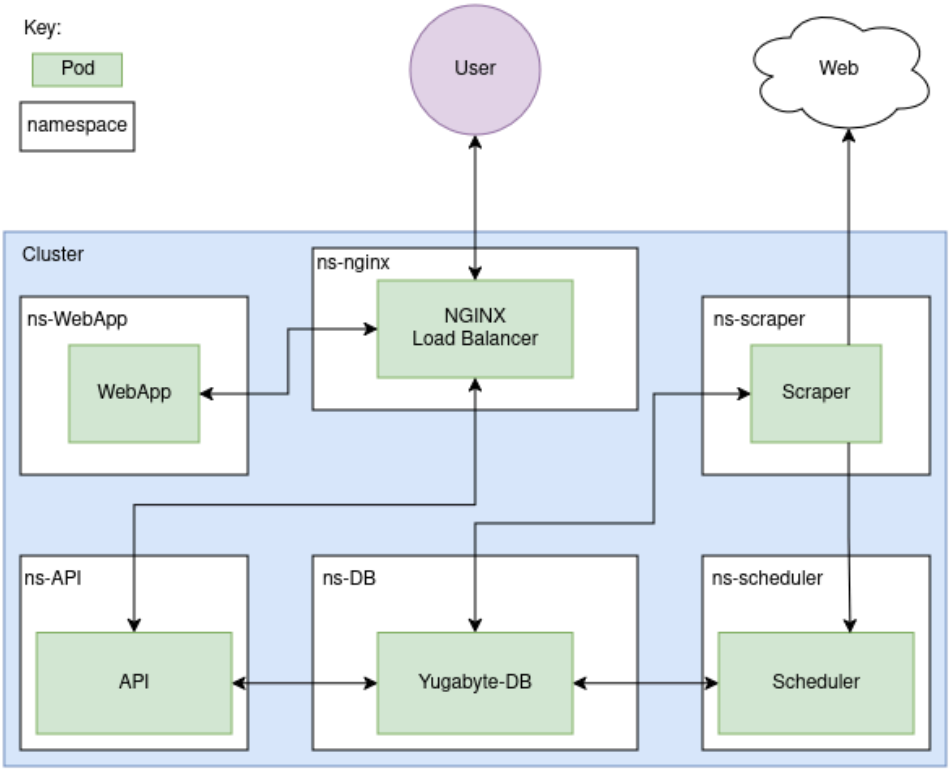


Figure 2: A visualisation of the Kubernetes Cluster

Each microservice is managed by a Deployment, allowing Kubernetes to automatically retract failed instances and maintain the desired system state. These Deployments are coupled with Services, which assign a unique persistent DNS name to each microservice within the cluster. The Service layer facilitates internal load balancing, distributing incoming requests to all available Pods in a given Deployment.

To further enhance scalability, this architecture can be combined with a Horizontal Pod Autoscaler (HPA), which dynamically increases the number of Pod replicas in response to high traffic, or a Vertical Pod Autoscaler (VPA), which adjusts the resource allocations for individual Pods. These automated scaling mechanisms ensure that the application remains responsive under varying load while optimizing resource utilization across the cluster.

5. Service Architecture

This section details the technical implementation and internal logic of the platform's core microservices. While the infrastructure provides the environment for these services to run, this layer contains the business logic necessary to fulfill the client's functional requirements. Each service is designed with a focus on modularity and high performance, utilizing standardized communication protocols and a consistent code architecture to ensure interoperability and ease of maintenance across the distributed system. The source code for the various services can be found in the Appendix (See Appendix 1.).

5.1. API

The API is the central core of the backend ecosystem, serving as the primary interface through which the Flutter clients interact with the system's data and logic. In addition to basic data retrieval, the API is also responsible for enforcing security protocols, managing user sessions, and executing logic that defines the user experience. By centralizing these responsibilities, the API ensures that business logic is applied consistently across all client platforms.

5.1.1. Code Architecture

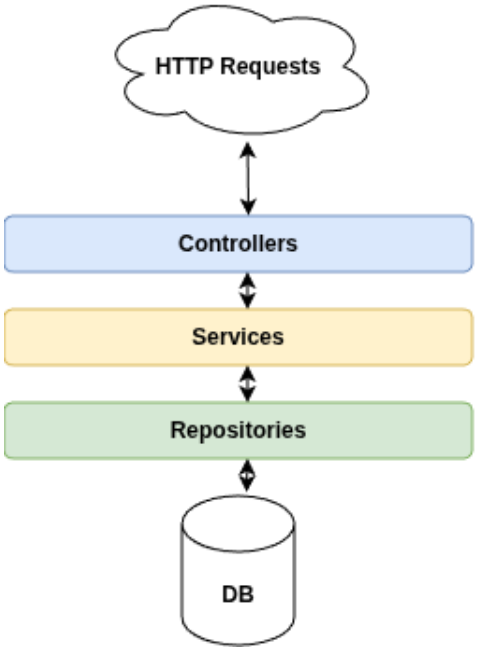


Figure 3: A visualisation of the backend code architecture

The API follows an N-Tier architectural pattern, which utilizes a strict separation of concerns to enhance maintainability. As illustrated in *Figure 3*, the codebase is organized into three primary layers:

- **Controllers:** This layer serves as the entry point for all incoming HTTP requests. Controllers are responsible for request routing, enforcing rate limits to prevent abuse, and verifying user authorization.

- **Services:** The service layer contains the “business logic” of the application. It receives data from the controllers to perform the complex operations required for the correct functionality of the application such as the validation and verification of request data and managing hex-based territory logic.
- **Repositories:** The repository layer abstracts the data access logic. It communicates directly with the database using SQL queries, ensuring that the rest of the application remains independent of the specific database implementation.

Once a request has been processed through these layers, the result is packaged into a standard response format and returned back up the chain to the client.

5.1.2. Authentication and Authorization System

To ensure data integrity and user privacy, the API implements a robust security framework focused on secure credential storage and stateless session management.

Password hashing is used to ensure that user passwords are never stored in plain text. Each password is “salted” with a unique random string before being hashed. This approach provides safety against security threats such as rainbow table attacks.

Once a user is authenticated with a username and password, the system issues a JSON Web Token (JWT) to the client. This token contains encrypted information about the user, such as the user ID and role, and is signed with a private key. Utilizing JWTs allows the API to remain stateless. The session data does not need to be stored, but instead is validated by the Controller layer using the digital signature of the token on every incoming request. This ensures that the user has the necessary permission to access specific resources.

5.1.3. Hex System

Since the hex system was a crucial requirement of the application, the core of the application’s spatial logic is built upon the H3 Library, a hexagonal hierarchical geospatial indexing system. This library provides unique 64-bit indices for every hexagonal cell globally and includes specialized functions to retrieve hexes by index, convert geographic coordinates (longitude and latitude) into grid cells, and navigate the hierarchy through parent-child relationships. These capabilities were essential for implementing complex, multi-layered logic, such as managing POI visibility across different zoom levels and accumulating group experience points from local “child” hexes to broader “parent” hexes.

5.1.4. Points System

Within the system, experience points are given to users for any contribution made, including uploading images, submitting reviews, or checking in at an experience. If a user is a member of a group, these experience points are simultaneously added to the group’s total. These group experience points are then used for the capturing of hexes. To maintain a responsive map experience, this influence over hexes by groups must be calculated and accumulated across the H3 hierarchical layers in accordance with the user’s current zoom level. For example, if there are nine hexes at a lower level and one group captured six of them, the parent hex of the nine hexes should be shown to be captured by that group with the accumulated experience points of that group shown. This hierarchical aggregation ensures that users can access accurate, up-to-date territory data across all layers of the map.

5.1.5. Popularity System

The popularity system serves as an internal metric to quantify user engagement with a Point of Interest (POI). Similar to the points system, popularity is incremented whenever a user contributes to a POI or its associated experiences. While this metric is not explicitly visible to users, it is the primary driver for determining a POI's visibility across different hexagonal layers. As specified in the requirements (See Section 2.1.3.), user-created POIs are categorized as "temporary" by default, meaning their visibility is initially restricted to only be shown if it gains popularity. The popularity system enables the POIs to be visible on higher layers as more users interact with them. To ensure the map remains dynamic and reflective of current trends, the system tracks popularity on an hourly basis, resetting the count at the start of each hour. This hourly popularity is used to calculate the visibilities of all user-created POIs on the application.

5.1.6. Email System

The email system is primarily responsible for user account verification and password recovery. For the purposes of this prototype, the system utilizes an SMTP sandbox provided by Mailtrap. This allows for the rigorous testing of automated workflows, such as verifying a new user's email address on account creation, within a controlled environment. By capturing all outbound traffic in a virtual inbox, the sandbox ensures that no actual emails are sent to real-world addresses during development.

5.1.7. Image Compression System

Since the prototype was required to avoid the use of external cloud storage options (See Section 2.2.3.), the prototype utilizes a local binary storage strategy where all user-uploaded media is stored directly within the database. To ensure this approach remains performant and does not lead to excessive storage overhead, the image compression system using SkiaSharp processes every file before storage. This system uses a lossy compression algorithm to significantly reduce the file size of high-resolution uploads while maintaining sufficient visual fidelity for a mobile-first interface.

5.2. Experience Scheduler

The Experience Scheduler is a dedicated state-management microservice responsible for tracking and updating the status of all platform experiences across four distinct states: past, ongoing, soon-upcoming, and upcoming. To ensure the API remains responsive, the scheduler performs a state-check cycle every five minutes and updates the database with the current status of each entry. This interval was selected because most experiences begin or end at time divisible by five minutes, and the processing of a large number of experiences can itself take several minutes. By pre-calculating these states, the API can instantly filter and serve data without the complexities associated with real-time state calculation during a user query.

To maintain high performance and avoid the overhead of global time management, the scheduler tracks all experiences using the UTC timezone, leaving local adjustments to be handled by the frontend. To optimize memory usage, only the experiences scheduled for the current day are kept in memory, as it is computationally inefficient to check past or far-future experiences every five minutes. This list is initialised during microservice startup and refreshed every day at midnight. To ensure the list remains accurate, the API notifies a specific HTTP endpoint in the scheduler whenever

a new experience is added to the database. A concurrency lock is utilized during these updates to ensure the list is not altered while a state-check is currently reading from it.

This architecture was chosen as a separate microservice because calculating states within the API would result in an extremely unresponsive user experience, as fetching all ongoing experiences would require iterating through the entire database for every request. While caching was considered, it would have significantly increased the API's memory usage and compromised the performance of the API. Similarly, performing these updates as a database function was rejected to prevent increasing the load on the database nodes. By utilizing the iCAL.NET library within a dedicated service, the system remains maintainable and compliant with modern calendar standards while ensuring the core database remains responsive for user queries. Alternative methods, such as database polling or triggers, were also dismissed. Polling is too resource-intensive, and YugabyteDB does not currently support the database triggers required for a more reactive implementation.

5.3. Experience Scraper

The Experience Scraper is an automated integration service designed to gather data from external sources, such as third-party APIs and web scraping. While the architecture supports multiple ingestion methods, this prototype focused on an API-driven implementation to populate the application with experiences and POIs. The Experience Scraper is configured to run once per day. This frequency was chosen because external event data typically remains static once published and is often added well in advance of the actual start date. Furthermore, limiting the scraping frequency ensures the system remains cost-effective by minimizing calls to external APIs that may charge per request. To ensure the service remains easily extendable, a set of interfaces was developed. Adding a new data source is as simple as creating a new implementation of these interfaces and scheduling a corresponding daily job.

To prevent data duplication, the scraper assigns a unique "external source ID" to every ingested experience. This identifier is either retrieved directly from the source API or generated as a hash of the experience's data. These IDs are stored in an in-memory dictionary, which is initialized upon service startup by querying the database for all existing external source records. Before any new entry is uploaded, the service performs a lookup against this dictionary to verify that the experience does not already exist. This approach ensures that the database remains clean and free of redundant entries while maintaining high performance during the ingestion process.

The service also manages the geographic relationship between experiences and physical locations by attaching scraped data to specific POIs. Before creating a new POI, the scraper first checks if a matching location already exists by searching for entries with the same name within a 10-meter radius of the provided longitude and latitude. This geographical verification ensures that experiences are correctly grouped under existing POIs rather than creating duplicate POIs for the same physical location. If no match is found within this radius, the microservice automatically generates a new POI entry with the relevant information provided by the API.

5.4. Tile Server

The Tile Server is a specialized service that provides small pre-rendered images, vector data, or map "tiles" to the frontend to ensure the efficient rendering of geographic information. This approach is essential for high-performance map navigation, as it allows the client application to request only the specific geographic area currently visible to the user rather than downloading a

single, large map file. By serving data in this format, the system maintains a responsive user interface even when handling dense datasets across varying zoom levels.

Due to the inherent complexity of developing a custom tile-rendering engine, creating a proprietary server was deemed out of scope for this project. Instead, the system uses tileserver-rs, a high-performance, open-source tile server. This specific solution was selected because it is highly efficient and easily containerized, allowing for seamless integration into the Kubernetes-managed infrastructure. Furthermore, tileserver-rs provides native support for PMTiles, the optimized, single-file tile format used by this application to store and serve map data. The main reason for selecting PMTiles, as opposed to alternatives like MBTiles, was the accessibility of high-quality map data in this format. Since Protomaps has open-sourced their map data and provides frequent updates, the system can rely on a consistent and accurate data source. Furthermore, the format offers seamless interoperability between tileserver-rs and MapLibre, ensuring that vector data can be efficiently served from the backend and rendered on the client side without compatibility issues.

6. User Application Design

The following section covers important design choices made regarding the user application design. This covers both the visual design of the user interface and the decisions made for its implementation. While this section contains the design of the views of the main user flows, additional figures are listed in the Appendix (See Appendix 4.).

6.1. User Interface Design

The design decisions taken for the user interface were mainly taken for clarity or simplicity purposes. One of the goals was for the interface to be easily understandable for users. Therefore, we tried keeping things clear and avoided adding clutter to the different interfaces. Another important aspect to it was simplicity of development. This project was aimed mostly towards implementing a prototype with as much functionality as possible, rather than focussing on its aesthetics. This was done by request of the client.

To simplify development, Google's Material Design suite was used, which provides many ready-to-use elements that already implement some styling, accessibility features and interaction functionality. This way, the focus could be placed more on the layouts and user flows, rather than the exact look of the elements themselves, enabling us to implement more features within the time we were given.

As a prototype, the main purpose of these interfaces is to give an example of what the application *could* look like. What features does the application need? What elements are needed? Where should elements be placed? What user flows exist? These are the main questions that the developed prototype should allow to be answered after performing user testing, as described under the Future Work section (See Section 11.).

For the future product, the client expressed interest in using the *liquid glass* style. In *Figure 4*, an example is shown of what liquid glass style buttons would look like. From this suggestion, some of the elements shown in the coming wireframes are semi-transparent. Changing the implemented interfaces to use this style was considered a low-priority requirement and was not completed within the span of this project. Therefore, some elements that look semi-transparent in the wireframes do not have this transparency in the implemented interfaces.

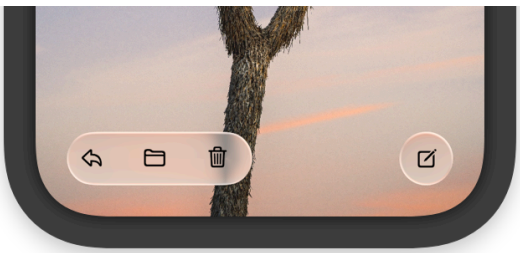
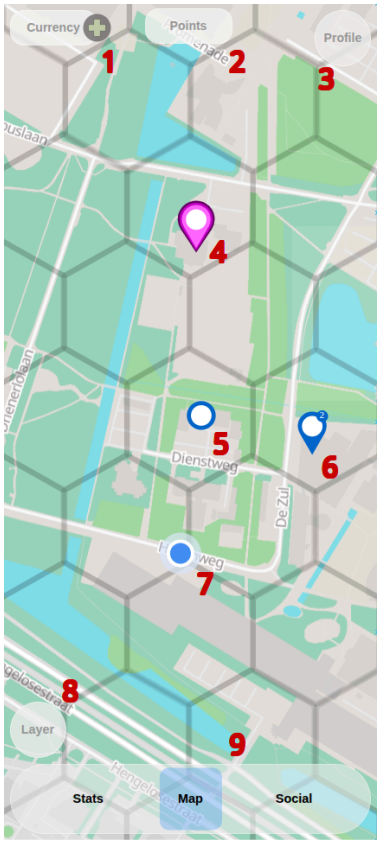


Figure 4: Example of Apple's liquid glass style[1]

[1] *Liquid Glass*, n.d.

6.1.1. Map View Design

The most important view of the user application interface is the view of the map, where the user is shown a map that spans the whole world. Like any map application, the user can pan around by dragging, and zoom in and out by using pinch gestures on touchscreens, or scrolling with a computer mouse. *Figure 5* shows the wireframe that was made to show what was envisioned for this view. The red numbers labeled 1 through 9 mark different elements of the wireframe.



Description of number labels

1	Currency counter
2	Experience point counter
3	Profile button
4	Map marker indicating POI has a *trending* experience
5	Map marker indicating POI has no upcoming/soon upcoming experiences
6	Map marker indicating POI has 2 upcoming/soon upcoming experiences
7	Marker of user's current location
8	Layer selector button
9	Bottom navigation bar

Figure 5: Wireframe of the main map view

This wireframe shown in *Figure 5* alone defines many different features of the application. Perhaps the most interesting one is the hex grid overlay displayed on top of the map. This is how the hex grid for the gamification aspect of the application was envisioned. This wireframe did not yet show what hexes would look like when *captured* by different groups.

Another point to address is the set of different POI markers on the map. These markers, labeled '4', '5' and '6', are used to differentiate between POIs that are likely less or more interesting. For this, a distinction was made for the following types of POI: ones with a *trending* experience, ones with some ongoing/soon upcoming experiences and ones without any ongoing/soon upcoming experiences. The idea is that POIs with a trending experience should stand out the most from other map markers, as these trending experiences are likely more interesting than the rest (an experience becomes trending when many people contribute to it in a set period of time). Furthermore, a POI with nothing going on is likely less interesting than one with ongoing/soon upcoming experiences the user can attend. An experience is considered 'soon upcoming' whenever it is set to start within 3 days.

The layer selector button labeled with ‘8’ is another important element of the map view. Clicking this button would bring up a menu, where users can select different map layers to view different types of experiences. Imagine having a map layer where users can find concerts, another where they can find restaurants and another where they can find sport events. Through this separation, users can more easily find exactly the types of experiences they are looking for, instead of cluttering the screen with many experiences that serve completely different needs.

The elements marked ‘2’ and ‘3’ were implemented almost exactly as shown in the wireframe. The only real difference being that element ‘1’ was removed (more on this down below), allowing element ‘2’ to be displayed on the top left instead. The experience points indicator simply shows how many experience points the user has earned on their account. These points are earned through making contributions to different POIs and experiences throughout the application. The profile button will bring up a menu where users can either go to their profile page or log out.

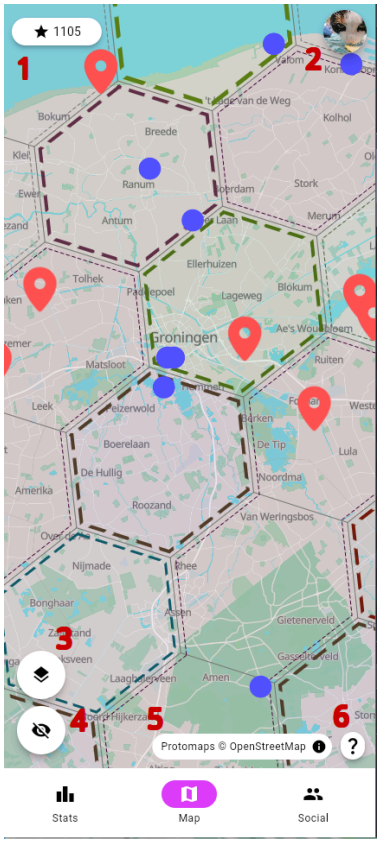
Finally, we will cover the features that did not make it in the final prototype.

Originally, the idea was to have an in-game currency akin to many modern live-service games. This is what the counter marked with ‘1’ would indicate. It would also allow the user to go to a store page where they can buy more of said currency by clicking the plus icon. Integrating with payment services was deemed out of scope, however, and has been designated to a future extension of the application, as described under the Future Work section (See Section 11.).

Similarly, as location-based services were also delegated to a future extension of the application, the user location marker labeled with ‘7’ was also not implemented (See Section 11.).

A small detail that was not implemented due to prioritization of other features was the number indicator shown on the marker labeled with ‘6’. So, in the current implementation there is still a separate marker to indicate there are some ongoing/soon upcoming experiences at a POI, but no number is shown next to it.

Now, we cover how the envisioned interface for the map view was actually implemented. *Figure 6* shows the actual implemented map view. Again, different elements are labeled with red numbers. The markers visible on the map are either randomly generated data used as mock data during development or data ingested from the TicketMaster API via the Experience Scraper (See Section 5.3.).



Description of number labels

1	Experience point counter
2	Profile button
3	Layer selector button
4	Hex ownership visibility toggle
5	OpenStreetMap source attribution
6	Help button

Figure 6: Implemented map view

Firstly, the hex grid. As stated before, an element that was not considered in the wireframe was that groups can *capture* hexes on the map. In the final implementation, this group *ownership* of different hexes is shown with coloured, striped borders and a light background infill. This was inspired by the popular video game *Civilization VI*, where the borders between territory of different nations are also shown with coloured hex borders (see *Figure 7*). This way of differentiating territory felt like a perfect fit for this application. The light background infill was added to better differentiate hexes that are claimed and unclaimed. Groups can choose these colours separately and however they want when creating a new group or editing their existing one.

Figure 7: Territorial borders in Civilization VI[2]

[2] Sebayang, n.d.

Now onto the different labeled elements. As stated before, the elements labeled ‘1’ and ‘2’ were implemented almost exactly like envisioned in the wireframe from *Figure 5* (where they were labeled ‘2’ and ‘3’). Another element that was copied over is element ‘3’, the layer selector button. An important element that was missing from the wireframe is element ‘5’. The map data used by the application comes from *OpenStreetMap*, which requires that a source attribution is placed and visible on the map.[3]

In addition to these elements, though, some others were added as well. Element ‘4’ is a button that allows the user to toggle the group hex ownership visibility on or off. This feature is intended for users who do not really care to partake in the game elements of the application and simply want to find some experiences to attend. The screenshot shown on the left in *Figure 8*, shows what the map looks like with this visibility toggled off.

Clicking the button marked with ‘6’ in *Figure 6* brings up the menu shown in the middle screenshot in *Figure 8*. This is a simple legend that explains what the different kinds of map markers mean. No *trending* map markers are shown on the map as of now, as the data in our prototype contains no trending experiences.

Finally, the screenshot on the right in *Figure 8* shows what the opened layer selector menu looks like, which appears when the user clicks element ‘3’ from *Figure 6*. The menu shows all the available map layers, in addition to an option to select ‘No layer’. This option will simply display the map without any markers, allowing the user to focus just on the group hex ownership if they want. The currently selected layer is also marked with the ‘(current)’ text in its name.

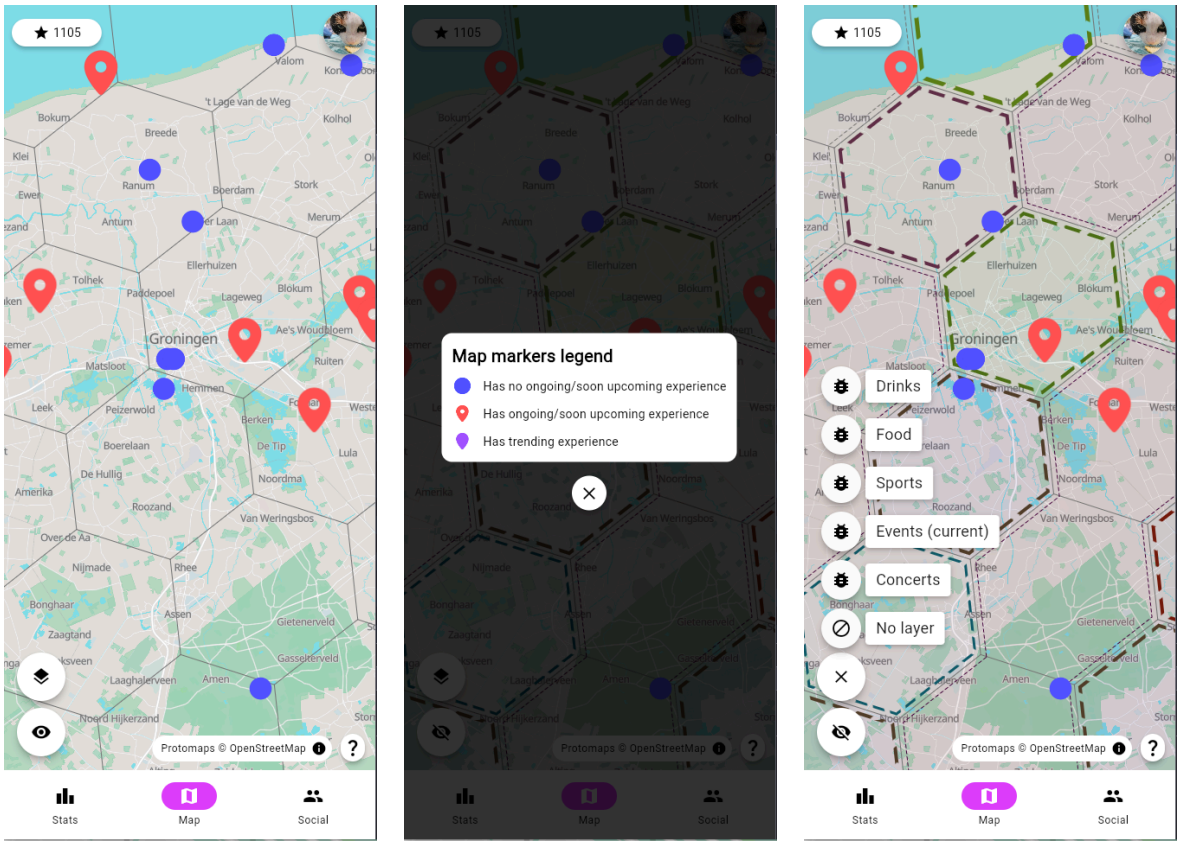


Figure 8: Additional features of the implemented map view

[3] *Licence/Attribution Guidelines*, 2022

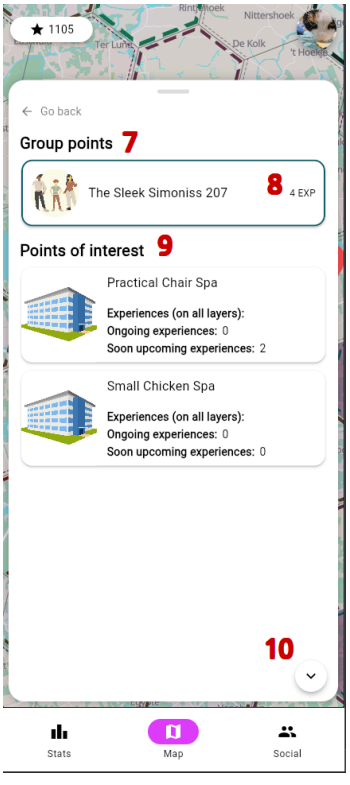
Apart from these extra features shown on the map itself, another important feature was implemented, which allows users to click on a hex. *Figure 9* shows elements of the interface after the user has clicked a hex. Clicking a hex does a few things: it highlights the hex with a white border (see label ‘1’), it increases the opacity of the background infill colour of hexes owned by the same group (see purple hex above label ‘1’) and it brings up an overlay (see label ‘4’) displaying both the amount of experience points different groups have earned in this hex (label ‘7’ and ‘8’) and a list of all POIs in this hex (label ‘9’). Through these views, users can easily see what groups own which hexes, and get a concise list of all POIs in a hex, which is nice in areas with many POIs close together, like a city center with many pubs, bars and clubs.

Initially, this map overlay is minimized, as shown on the left in *Figure 9*. From this state, the user can dismiss it by either clicking the back button labeled ‘2’, dragging down the menu using the handle labeled ‘3’, or clicking the dismiss overlay button labeled ‘6’. Dismissing the overlay brings the user back to the default map view like in *Figure 6*. Alternatively, the user can expand the overlay by either dragging up the menu using the handle labeled ‘3’, or clicking the expand overlay button labeled ‘5’. This brings the menu to the state shown on the right in *Figure 9*. Once expanded, the user can once again shrink the menu by dragging down or clicking the shrink overlay button labeled ‘10’. All these different ways of shrinking and expanding the menu were added to accommodate users with different devices. On a touch screen, drag gestures feel natural, while with a computer mouse, using a button is easier.

This same overlay screen is used for multiple different views related to the map. There are also user flows with multiple levels of navigation within this overlay screen. Therefore, the back button labeled ‘2’ was added, to simply navigate back to the previous view inside this overlay screen. Additionally, if there is no previous view, it acts as another way to dismiss the overlay menu.

Minimized hex view

Maximized hex view



Description of number labels

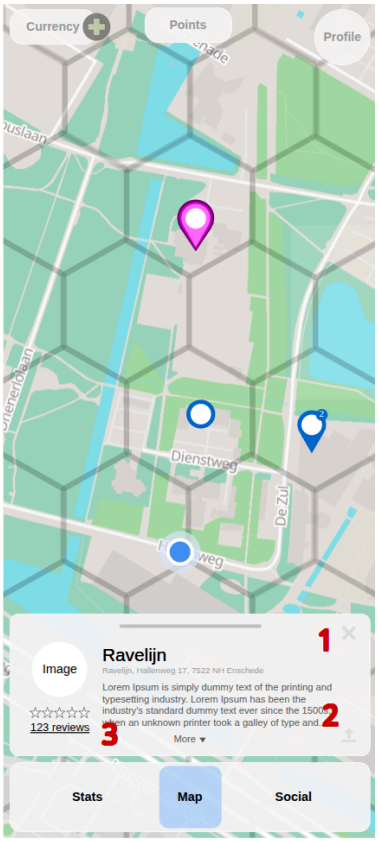
1	Selected hex
2	Back button
3	Drag handle
4	Overlay screen
5	Expand overlay button
6	Dismiss overlay button
7	Group points list
8	Group experience points in this hex
9	POI list
10	Shrink overlay button

Figure 9: Implemented hex clicking map views

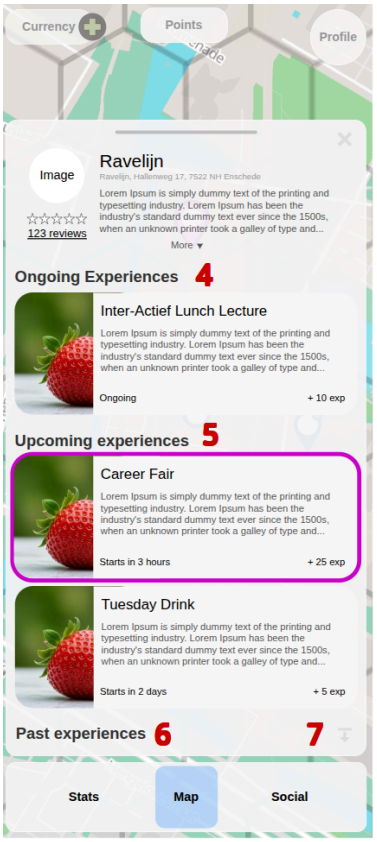
6.1.2. POI and Experience View Design

When the user clicks a POI marker on the map (see *Figure 6*), or selects a POI from the list of POIs inside a hex (see *Figure 9*), another map overlay menu comes up that shows information on the POI, allows adding and viewing contributions to it, as well as any experiences attached to that POI. *Figure 10* shows the wireframes that were made to show what was envisioned for these views.

Minimized POI view



Maximized POI view



Description of number labels

1	Dismiss overlay button
2	Expand overlay button
3	Link to reviews view
4	Ongoing experiences list
5	Upcoming experiences list
6	Past experiences list
7	Shrink overlay button

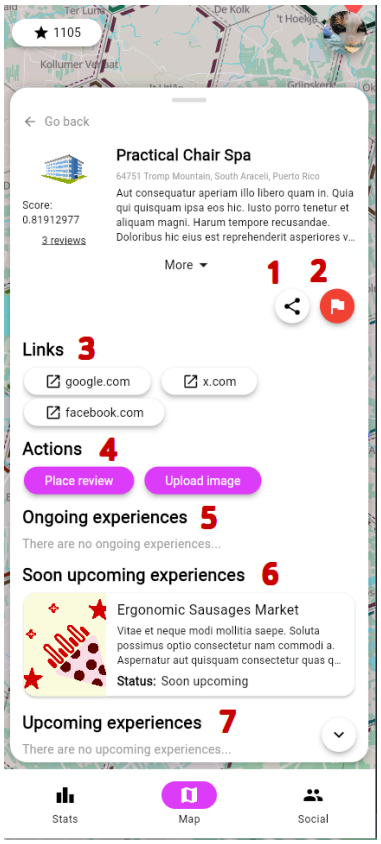
Figure 10: Wireframes of POI views

On the POI views' wireframes, as shown in *Figure 10*, it was envisioned that the user should be able to view its name, address, description, cover image, review score and count (labeled '3') and list of experiences (labeled '4', '5' and '6'). The experience with the pink border (labeled '5') signifies that this specific experience is either trending or sponsored. Clicking the review count would bring users to a view with a list of reviews for the POI, as envisioned in the wireframe shown in *Figure 13*. The list of experiences is split into different categories, differentiating between experiences that are ongoing, upcoming or past.

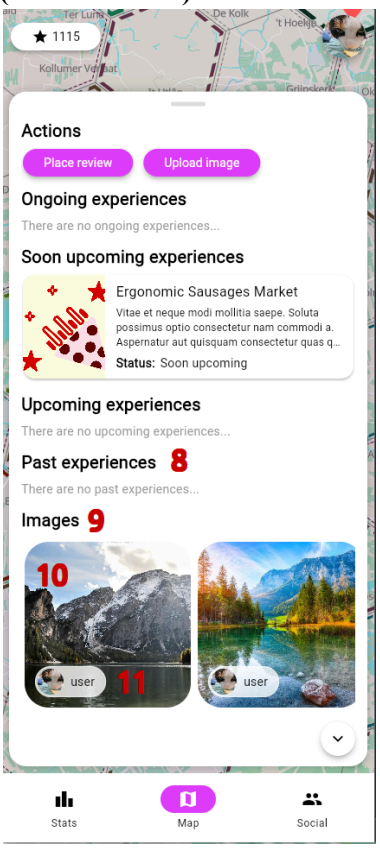
Just as described before *Figure 9*, these POI views use the map overlay screen that users can dismiss, shrink or expand. Some relevant differences to note between the envisioned version in the wireframes in *Figure 10* and the finally implemented version as shown in *Figure 9* and *Figure 11*, include the positioning of the dismiss, shrink and expand buttons (labeled '1', '7' and '2' in *Figure 9* respectively) and the inclusion of a back button in the implemented version. In the implemented version, the dismiss, shrink and expand buttons are all placed together, in the bottom right of the overlay. By being placed together, users will not have to move their finger or computer mouse as much when using the different buttons. By being placed in the bottom right, it is easier for mobile

users to reach the button with their thumb, especially compared to the dismiss button being in the top right when the overlay menu is expanded. The missing back button in the wireframes is a simple oversight. While a back button was planned on views in deeper navigation levels within the overlay, we found that adding and removing it between navigations resulted in the views shifting, which felt clunky. Therefore, in the implemented version, we decided to have the back button always be visible on any navigation level within the map overlay screen.

Maximized POI view



Maximized POI view (scrolled down)



Description of number labels

1	Share button
2	Report button
3	Links section
4	Action buttons
5	Ongoing experiences list
6	Soon upcoming experiences list
7	Upcoming experiences list
8	Past experiences list
9	Image carousel section
10	Image card
11	Uploader information

Figure 11: Implemented POI view

On the implemented POI view, shown in *Figure 11*, multiple items were added that were not in the original wireframes. Firstly, the share and report buttons were added (labeled ‘1’ and ‘2’ respectively). Clicking the share button will either bring up a platform-specific share menu, or fall back to simply copying the URL. This button was added to enable easier sharing of various important views, so that users can show their friends what POIs, experiences, profiles or groups they are talking about. The report button was added to any user contributions, as users need a way to report harmful or inappropriate content. Secondly, the ‘links’,‘actions’ and ‘images’ sections were added (labeled ‘3’, ‘4’ and ‘9’ respectively). While creating the wireframes, we did not account for users needing the ability to follow external links from a POI (like a link to buying concert tickets) and that sections were needed for making and viewing different contributions.

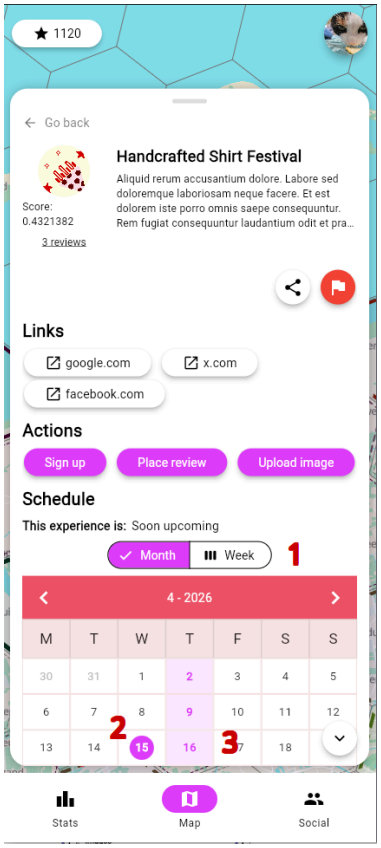
Additionally, a ‘soon upcoming experiences’ section was added (labeled ‘6’), which shows experiences with the ‘soon upcoming’ state. Experiences get this state when they are set to start within three days. This status was added as we realized for most experiences it did not make sense to only give them a special marker as soon as they are ongoing (see the red markers in *Figure 8*). For many of

them in the real world, you would have to reserve or buy tickets for it. Therefore, we decided to add this extra status.

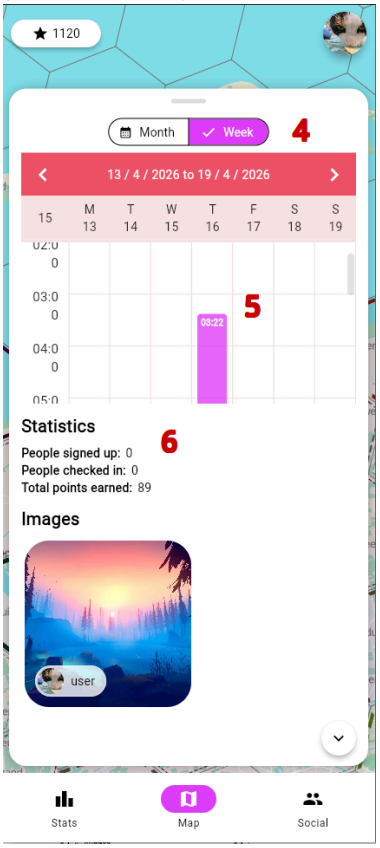
In the images section (labeled ‘9’), an image carousel is shown, which displays a card for each uploaded image (labeled ‘10’). The user can then drag horizontally to scroll and view other images. Every card also shows the user who uploaded the image (labeled ‘11’).

Now, after clicking on an experience preview, the user is brought to the experience view, shown in *Figure 12*. The experience view shows details on the experience, including its name, description, cover image, review score and count, external links, available actions, experience schedule, statistics (labeled ‘6’) and uploaded images.

Maximized experience view



Maximized experience view (scrolled down)



Description of number labels

1	Calendar view mode button (on month)
2	Current day indicator
3	Days of the experience
4	Calendar view mode button (on week)
5	Experience schedule time
6	Experience statistics

Figure 12: Implemented experience view

Like on the POI view, the buttons in the ‘actions’ section allow users to make contributions to the experience. These actions include signing up for an experience, checking in, placing reviews and uploading images. The idea behind signing up is that an experience will gain popularity before it starts if it gets many sign-ups. This way, if many people sign up for a party, other users know it will be a big party, incentivizing them to also attend. It would then also be possible to check in to an experience while it is ongoing, where you would get bonus experience points if you also signed up beforehand. Like POIs, the other contributions include reviews and images.

The schedule shown under the ‘schedule’ section displays exactly when the experience will happen. It supports two separate views: the month view and the week view. These can be selected by using the buttons shown in *Figure 12* labeled with ‘1’ and ‘4’, where they are shown in the two

different states. On the month view, shown in the left screenshot of *Figure 12*, the current day is highlighted (see label ‘2’) and any day on which the experience happens has a slight coloured background (see label ‘3’). We can see this experience happens weekly on Thursdays. Clicking on a day on the month view will also bring you to the week view. On the week view, the exact start and end times of the experience are shown (see label ‘5’). We see this experience starts at 3:22 AM.

Finally, the experience view includes a ‘statistics’ section, labeled ‘6’ in *Figure 12*. This section shows the amount of people signed up, amount of people checked in and the total amount of experience points earned at this experience.

For both POIs and experiences, users can post reviews. These reviews are separate: one could place independent reviews on both a POI and its experiences. For both cases however, the same view layout is used, with the only difference being what the reviews are fetched/placed for.

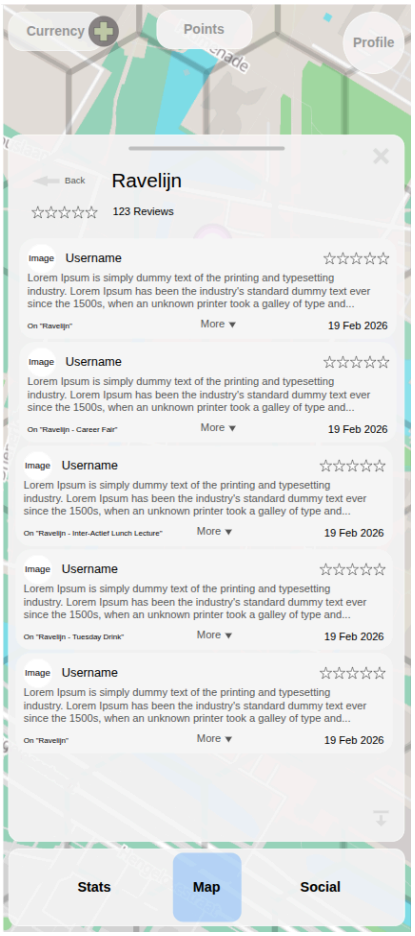
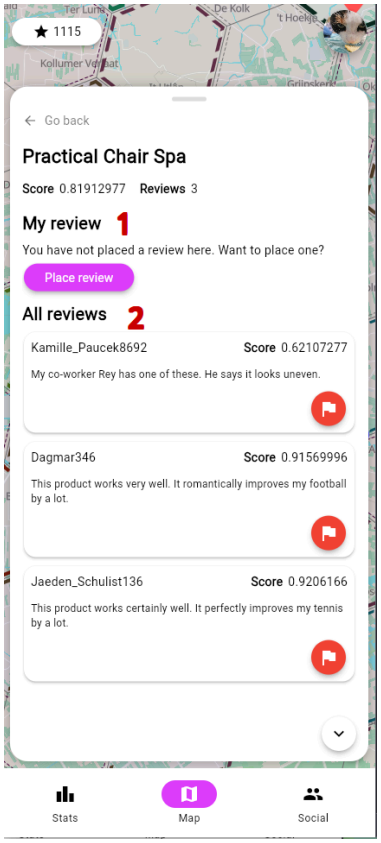


Figure 13: Wireframe of reviews view

In *Figure 13*, a simple wireframe of the envisioned reviews view is shown. The view displays the name of the POI or experience, average review score and count, plus the list of all reviews.



Description of number labels

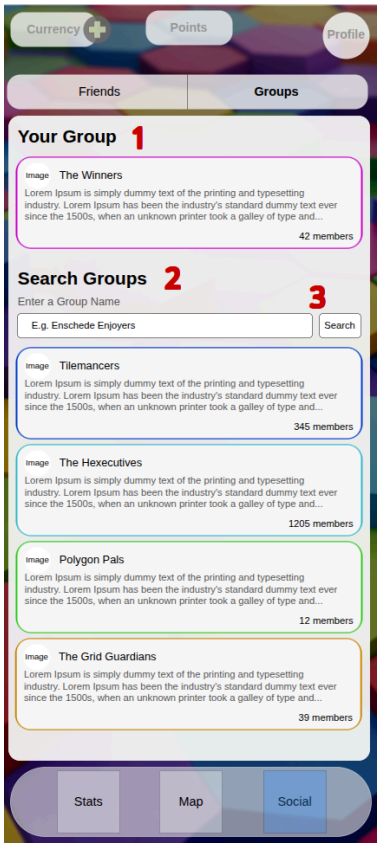
1	My review section
2	All reviews section

Figure 14: Implemented reviews view

In the implemented reviews view, shown in *Figure 14*, one extra section was added that was not in the wireframe. This section, labeled ‘1’, shows your review for this POI or experience, if you have posted one. If not, it shows a button to place a review. With this, it becomes easy for the user to quickly see if they have already posted a review and post one if they have not done so yet. The section labeled ‘2’ is almost the same as in the wireframe, with the difference being the look of the reviews themselves. Like any contribution, the reviews feature a report button. It is to be noted that the POI name and corresponding reviews in the figure are randomly generated.

6.1.3. Social View Design

When the user navigates to the ‘Social’ section through the bottom navigation bar shown in all previous screens, a new set of views becomes accessible. In *Figure 15*, the wireframe created for the ‘Groups’ view is shown. Apart from the groups view, it was originally also the idea to implement a set of ‘Friends’ views in this ‘Social’ section. However, due to time constraints, the friends system was not implemented in the User Application. Therefore, no views concerning the friends system are covered here. This limitation is explained in detail in the Future Work section (See Section 11.).

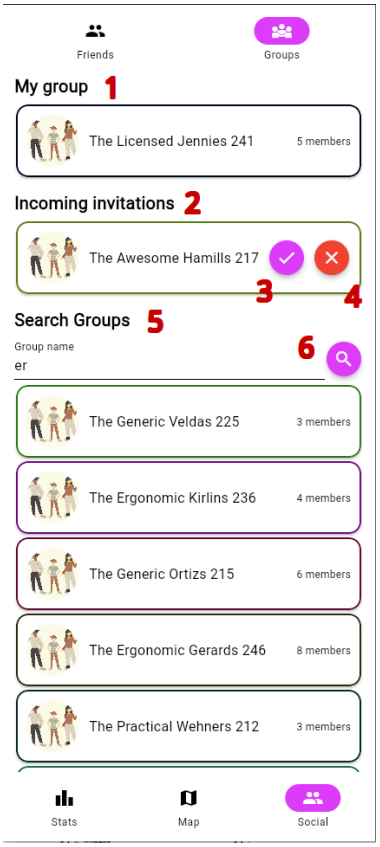


Description of number labels

1	Section showing the group you are in
2	Search groups section
3	Group search button

Figure 15: Wireframe of groups view

On the groups view, shown in *Figure 15*, it was envisioned users would be able to view the group they are currently in (labeled ‘1’) and search for other groups, in the section labeled ‘2’. The user would then be able to enter (part of) a group name in the search box and press the search button labeled ‘3’ to search. Then, a list of matching results would be shown in this section. Another idea that surfaced here was to have each group preview the border colour that is the same as that group’s border colour on the map. Furthermore, every group preview has the group’s cover image, name, description and member count.



Description of number labels

1	Section showing the group you are in
2	Section showing incoming group invitations
3	Accept invitation button
4	Decline invitation button
5	Search groups section
6	Group search button

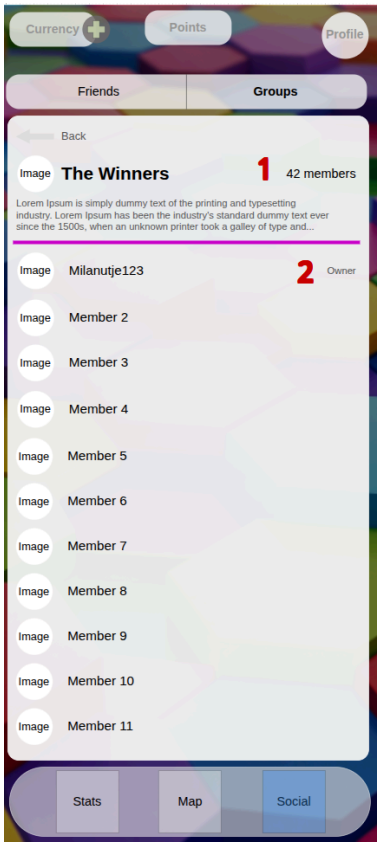
Figure 16: Implemented groups view

Now, on the implemented view shown in *Figure 16*, some changes from the wireframe can be seen. Most notably, a new section, labeled ‘2’ was added. This section shows any incoming invitations from different groups to the current user. Joining groups being invite-based was not yet considered while creating the wireframes. So, through this section, the user sees group previews for all incoming invitations, with two buttons on each: a button to accept the invitation (labeled ‘3’) and one to decline it (labeled ‘4’). Apart from that, it should be noted that group previews no longer show a description. When implementing the interface, we found it made the group previews too cluttered and that it was not very useful in this context.

The other elements, namely the section showing the group you are in (labeled ‘1’) and the search groups section (labeled ‘5’) with the search button (labeled ‘6’), were largely carried over from the wireframe. A small change being that the search button now uses an icon instead of text, as we found this to be clearer.

One more notable change from the wireframes is that we decided not to display the top bar showing experience points and the user’s profile button on views outside the map view anymore. In the implemented version, they are only present on the map view, which was shown before in *Figure 6*, particularly the elements labeled ‘1’ and ‘2’. This was chosen as the non-map views already display a lot of information, where those extra top bar elements would just be more visual clutter. It also did not seem too cumbersome to have to navigate back to the map view to view and interact with these elements.

Now, whenever the user selects one of these group previews, a new view is shown. In *Figure 17*, the envisioned wireframe for this view is shown. We refer to this as the ‘group’ view, as it shows information on a single group.

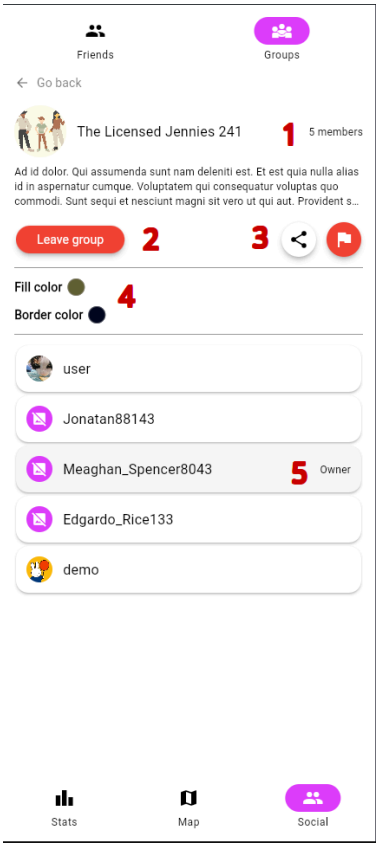


Description of number labels

1	Group member count
2	Group owner indicator

Figure 17: Wireframe of group view

In the wireframe shown in *Figure 17*, we see the view is relatively simple. It contains information like the group cover image, name, description and members list, plus two other helpful elements: the first being the member count (labeled ‘1’) and the second being an indicator on the user preview, showing which user is the group’s owner (labeled ‘2’). These two elements were added to provide extra helpful information to the user.



Description of number labels

1	Group member count
2	Leave group button
3	Share & report buttons
4	Group colour indicators
5	Group owner indicator

Figure 18: Implemented group view

Now, in the implemented group view, shown in *Figure 18*, a few changes from the wireframe can be seen. Firstly, there is a button to leave the group (labeled ‘2’), which appears if you are currently in this group. On the right-hand side of that button is the regular set of share and report buttons (labeled ‘3’), as this is another interesting view a user might want to share with others. A group should be reportable, as users could upload an inappropriate picture for the cover image, or put offensive text in the group name and description.

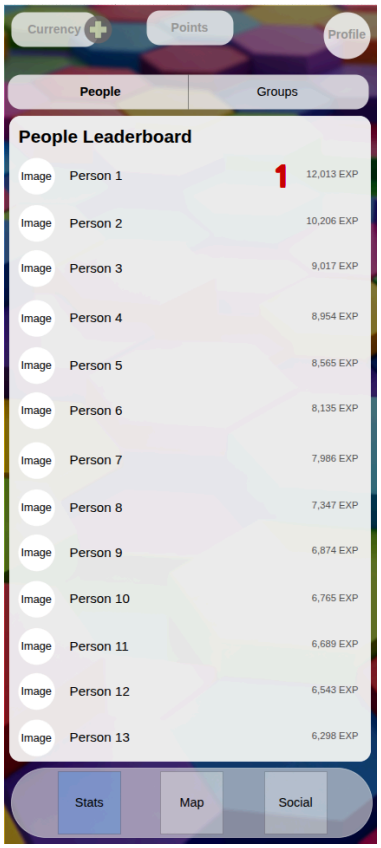
Another set of elements that was added is the set of colour indicators labeled ‘4’. These show the two colours of the group that also show up on their claimed hexes: the fill colour (the hex infill) and the border colour (the hex border colour).

The member count labeled ‘1’ and the owner indicator labeled ‘5’ were carried over from the wireframe.

6.1.4. Leaderboard View Design

The final set of views accessible through the bottom navigation bar, is the set of leaderboard views. In *Figure 19*, the wireframes for the envisioned views are shown.

Users leaderboard



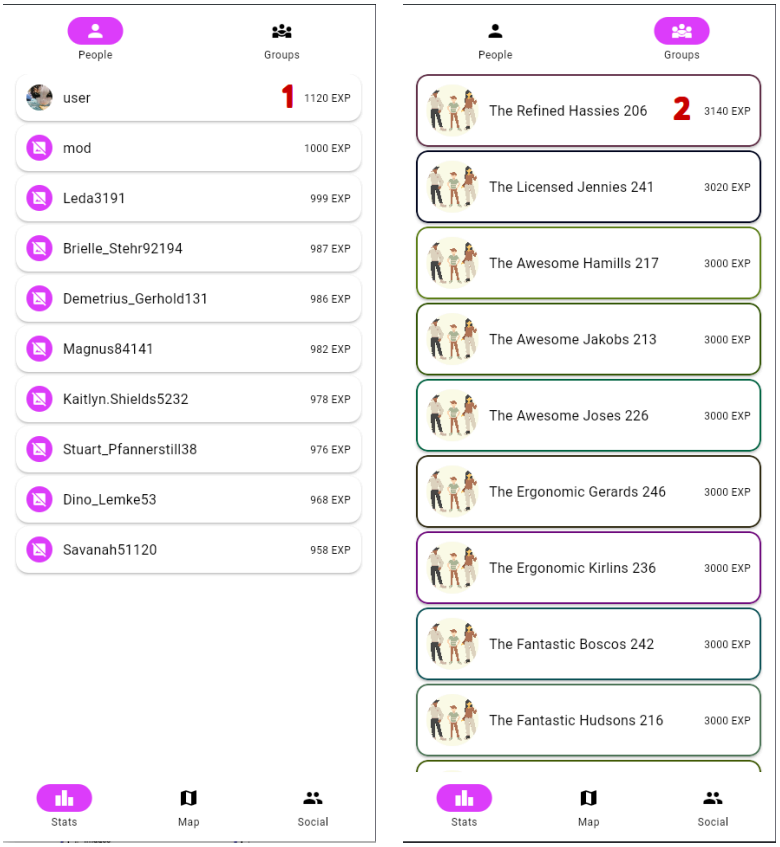
Groups leaderboard

Description of number labels

1	User experience points
2	Group experience points

Figure 19: Wireframes of leaderboard views

There are two separate leaderboard views: one for individual users (shown on the left) and one for groups (shown on the right). Apart from the difference in whether the views list users or groups, the layout is the same. One more noteworthy point being the difference in where experience points for users (labeled ‘1’) and groups (labeled ‘2’) come from. It should be noted that a group’s experience points is the sum of all experience points of all members.



Description of number labels

1	User experience points
2	Group experience points

Figure 20: Implemented leaderboard views

In the implemented interfaces for the leaderboard views, shown in *Figure 20*, it can be seen that at least the user leaderboard, shown on the left, looks almost exactly as in the wireframe. The groups leaderboard (shown on the right), however, uses some different elements to display the different groups. To simplify development and keep visual consistency across the application, we decided to reuse the same group preview elements used in the groups and hex overlay views (shown before in *Figure 16* and *Figure 9* respectively). Therefore, the elements shown in the list look quite different from those envisioned in the wireframe.

Another point worth mentioning is that these leaderboards only show the top 10 users and groups. This was done out of time constraints: there was unfortunately not enough to add pagination features to the API server and an infinitely scrollable or paginated list in the user interface. In a future version of the project, we recommend this to be implemented as the goal of the leaderboards was to add another element of competition to the game side of the application. With only 10 users and groups being shown, though, there is no such competition for all other users and groups of the application. This is explained in more detail in the Future Work section (See Section 11.).

6.2. User Application Technology Choices

The following section describes the technologies chosen to implement the user application. The table in *Figure 21* shows a list of the chosen technologies along with their purpose.

Purpose	**Chosen Technology**
Programming language	Dart
App Framework	Flutter
Client-Side Routing	GoRouter
Context/State Management	Provider
Map System	MapLibre
Hex System	H3
Calendar Format	iCal - RFC 5545
Automated Testing	flutter_test

Figure 21: Chosen User Application technologies

As the application is mainly meant for mobile and web, our choice in technology was mainly between a mobile-first web application or a true native mobile application with web capabilities. The prototype is meant to be extended into a true product later. The main use case will be people interacting with the interface on their phones, as the application has many outdoor use cases. Therefore, going with a true native mobile app with web capabilities on the side seemed like the better choice. As this app should be cross-platform, Flutter was the first solution that came to mind. Alternatives to this would be solutions such as React Native, NativeScript, Ionic, Kotlin Multiplatform or .NET MAUI. The first three of these alternatives rely on running JavaScript on the client, which is slower than running the native assembly that Flutter produces for those platforms, meaning Flutter will perform better than them. The last two options, Kotlin Multiplatform and .NET MAUI, do seem like solid contenders, but MapLibre is not officially supported at this moment. As this prototype is meant to be expanded into a real product later, we wanted its dependencies to be stable and complete. It also appeared that compilation to native assembly for these last two options was not well-supported for all platforms. Therefore, Flutter seemed like the best option and was chosen for implementing the user application.

A widely-used solution for client-side routing is GoRouter, which is developed by the Flutter team itself. This library allows declarative configuration of a page hierarchy, while also adding deep-linking support for the application. This means that users can share a link to a specific page, and other users will open that specific page when using that link. As this is an application that relies heavily on social interaction, we felt link-sharing capabilities like this were crucial.

To greatly reduce code repetition and “prop-drilling” (having to pass properties many levels down the widget tree), we use the Provider library, which enables adding references to an object that is accessible to all widgets. Additionally, the library allows declaring a view’s dependency on some

data, which will automatically re-render the view whenever the data changes. This library is also recommended by the Flutter team, and used in some of their Flutter example applications.

To display a map, we used an open-source solution that supports rich customization for styling and high performance through hardware acceleration. The only mature, free map rendering framework with a Flutter library that meets these requirements was MapLibre, which was what we worked with. The most prominent alternatives were flutter_map and Mapbox. However, flutter_map uses the Flutter rendering engine itself to draw the map, requiring the Flutter engine to do many extra calculations, resulting in more load on the CPU. MapLibre, on the other hand, uses a native rendering engine and delegates almost all work to the GPU, giving it better performance. The other alternative, Mapbox, is a paid framework.

We used the H3 library developed by Uber Technologies for the hex-grid system. H3 is a global grid system for indexing geographies into a hexagonal grid. This global hex grid can be directly used by our application to present users with a multi-layered hex grid of various sizes. H3 is the only hex-based open source library for maps and geospatial indexing. Other alternatives such as S2 Geometry by Google exist, but it uses square cells. The grid consisting of hexes is one of the requirements of the client. Thus, H3 was the best choice for our product.

To represent schedules we used the standard iCal RFC 5545 format. This standard is used by popular calendar apps such as Google Calendar, and there are many libraries available that can parse and generate it. By using this standard, we forgo the difficulty of coming up with a format ourselves and make future integration with calendar apps possible.

For automated testing, we used Flutter's 'flutter_test' package, which was developed by the Flutter team. It is the de facto solution for testing Flutter code and supports all the features we need. We were not able to find any alternatives.

6.3. User Application Code Architecture Design

To design the code architecture for the user application, we looked at a case study by Google, from the Flutter documentation. In the case study, Google defines a pattern implementing the MVVM (Model, View, ViewModel) architecture to separate different parts of a Flutter app's codebase, into two distinct layers: the UI layer and data layer. The UI layer is responsible for managing and displaying UI state. The data layer is responsible for sending and receiving data to/from external APIs and caching. It should serve as a 'single source of truth' for application data.[4] [5]

Further inspired from the case study, the code architecture of the user application uses the following folders per feature:

- widgets/ (views)
- view_models/
- data/models/
- data/repositories/
- data/services/

The key terms are described as follows:

[4] *UI Layer Case Study - Architecture*, 2026
[5] *Data Layer - Architecture*, 2025

- **Widgets:** Widgets define the UI based on current state. They may contain simple UI-only logic.
- **View Models:** View models handle view-related business logic and coordinate data access through repositories.
- **Repositories:** Repositories sit between view models and services. Can be stateful and are used for caching, composition, and domain-level computation.
- **Services:** Services are thin API/system adapters that are (almost) completely stateless.

Furthermore, to streamline external API server access by the different services, a single shared ‘ApiClient’ instance is used to handle all communications with the external API server. This ‘ApiClient’ handles different HTTP response codes, automatically refreshes authentication tokens and handles response parsing.

Another important element to address is how data loading is handled across the application. Much of the displayed information in the various views comes from the external API server. Such external requests take time to complete though (so should show a loading indicator) and sometimes can fail with errors. To handle these hurdles consistently, we developed a data loader pattern through which view models can define several ‘DataLoader’ fields, which can then be consumed in the view by a ‘ConsumeLoader’ widget. These two elements together with using the aforementioned Provider library for state management and automatic UI updates, handle all loading and error handling logic across the application in a consistent way.

The defined data loading API also allows declaring different loading strategies, namely ‘Immediate’, ‘After’ and ‘Manual’, where:

- ‘Immediate’ starts fetching data as soon as the view model is instantiated
- ‘After’ starts fetching as soon as some other data loader finished, also having access to the value produced by the previous data loader
- ‘Manual’ has to be manually triggered by some action from the view

Additionally, data loaders can listen to other ‘Listenable’ objects (like authentication state) and rerun whenever these send a notification. These different loading strategies together with the ability to add ‘listenables’, allow declarative declaration of data dependencies and automatic reloading of content when states change. For example, a pattern that is common in the codebase is to add the ‘authRepository’ as a ‘listenable’, meaning the data loader will rerun whenever the user logs in or out. Through this, one can easily hide or show elements automatically only for logged in users or owners of a certain POI/experience/group/profile.

A full description of how to add new features following this code architecture and how to use the ‘ApiClient’ and data loader patterns is given in the *README.md* file of the user application codebase. The source code of the user application can be found in the Appendix (See Appendix 1.).

6.4. Map System Architecture Design

An integral part of the User Application is the map system. As such an important part of the application, we explain here how this specific part of the code is architected using core repositories and what happens during a map update cycle.

6.4.1. Core Repositories

The system follows a hierarchical structure, with the MapViewModel serving as the main orchestrator. It manages the state of the map and it coordinates between several specialised repositories that handle specific aspects of map rendering.

Global Repositories

- **VisibilityManagerRepository**: Manages the hex visibility cache. Calculates the set of H3 cells currently within the viewport and identifies additions or removals during movement.
- **OwnershipCacheRepository**: Manages the hex ownership (by group ID) cache.

Sub-Orchestrators

- **HexGridRepository**: Manages the topological and geometrical construction and rendering of the base hex grid, polygon fills for individual cells, and selection highlighting. Also manages hex fill geometry cache.
- **MarkerRepository**: Handles the fetching, caching, and rendering of POI markers.
- **GroupBorderRepository**: Orchestrator for the border system; coordinates topology, geometry, and rendering repositories.

Group Border Sub-Repositories

- **BorderTopologyRepository**: Contains and manages topology of hex edges, keeping track of “border” edges; edges between hexes that do not have the same owner.
- **BorderGeometryRepository**: Converts topology representation into geometric borders, handling miter joins (corners) and anti-meridian wrapping.
- **BorderRendererRepository**: Manages group style information cache and GeoJSON batching.

Utilities

- **H3Repository**: A wrapper for H3 library calls; handles conversions between H3 indexes, coordinates, and string representations.
- **GeometryRepository**: Provides low-level geometric math and utilities, such as path splitting and coordinate distance calculations.
- **GeoJsonRepository**: Responsible for the construction of GeoJSON strings.

6.4.2. Map Update Cycle

The map system uses an update cycle triggered by camera movements outside pre-loaded bounds, or map initialisation. To prevent race conditions with multiple updates running at the same time, a ‘locking’-strategy is employed. When a new update is triggered while an update is still in progress, a flag is set so one more update cycle is completed after the current one. The following lists the components of the cycle in detail, in chronological order:

Cache Update

- The MapViewModel calculates and caches the current H3 resolution and bounds around the viewport.

Visibility Calculation

- The set of visible H3 cells is determined using the H3Repository based on the newly cached viewport bounds, and the VisibilityManagerRepository caches this and identifies which hexes have been added or removed since the last update.

Ownership Query

- For newly added hexes, the OwnershipCacheRepository fetches and caches new or expired group ownership data from the API.

Group Border

- The GroupBorderRepository updates its BorderTopologyRepository with the new hex ownership data, marking the changed adjacency graphs as “dirty”.
- For “dirty” graphs, the BorderGeometryRepository uses a path walk algorithm to build continuous coordinate lists (borders) that can be converted into geoJson.
- The BorderRendererRepository requests new or expired style information of groups from the API.
- The BorderRendererRepository uses the GeoJsonRepository to build a GeoJSON string for group borders with the correct style attributes.
- The BorderRendererRepository updates the GeoJSON source for its MapLibre LineStyleLayer.

Hex Grid

- The HexGridRepository calculates new polygon geometries for hex fills of added hexes, handling anti-meridian wrapping where necessary.
- The HexGridRepository uses the GeoJsonRepository to build a GeoJSON string for the hex grid and group fills with the correct style attributes.
- The HexGridRepository updates the GeoJSON source for its MapLibre LineStyleLayer and FillStyleLayer.

Marker Layer

- The MarkerLayerRepository fetches and caches new or expired marker data for the currently selected layer from the API.
- The MarkerLayerRepository gets the markers that are in the currently visible hexes and uses the GeoJsonRepository to build a GeoJSON string for the markers.
- The MarkerLayerRepository updates the GeoJSON source for its MapLibre SymbolStyleLayer.

7. Database Design

The following section aims to give an explanation for the design choices of the database. Specifically, the representation of each concept within the application, such as users, groups, and experiences, as database tables are covered in detail. While this section explains the justification for the design, the full design diagram and structure can be found in the Appendix (See Appendix 7.).

7.1. Users

Users are reportable entities so the “user_id” is a foreign key from the reportables table. In terms of constraints, the username of an account must be unique to not have overlapping names and the email connected to that account must also be unique to ensure users can not make multiple accounts from the same email. Moreover, user accounts need to have their emails confirmed before a user can login which is why we have the “email_confirmed” column. Users can also elect to join a group or add a profile picture to their account, so we have the “group_id” and “profile_picture_id” respectively to enable this. These ids are nullable as it is possible for users to not be part of a group or have a profile picture.

This user table represents both regular users of the application and moderators. Originally, we had an account table that represented overlapping information between a user and a moderator. Then we had separate user and moderator tables. We decided to change this to only have one table that represents both types of accounts, as we decided that moderators should have the same capability as regular users. This means creating POIs and experiences, signing up for experiences, adding reviews, and other features that users can interact with. As we removed the separate tables to then only have one table with added the “is_moderator” column to determine if the account is a moderator account.

While we were unable to implement the friends system, we still designed and implemented the database logic for what a friends system may look like in the future. Our friends table serves two purposes, the first is to determine what friends a user has and the second is to determine any friend requests a user has or has sent out. A row in the friends table can either be a row that determines two users as friends or a row that is for a friend request. The purpose of a row is differentiated by the value of the “accepted” column. If it is false, then the row represents a friend request and if it is true then the row represents two users as friends. We elected for this friends table to represent both friends and friend requests as we realised that separate tables would have the same columns, namely two columns that are foreign keys of a “user_id”. Therefore it would be simpler to have one table with an extra column to differentiate the purpose of a row.

Users are reportable entities meaning a user has the ability to report another user for a moderator to review. Like other reportable entities, when it is reported a report is made and a new row is added to the reports table. For users specifically, if a user is banned then we will store their email, hashed, in our banned_emails table. We store their email, to ensure that a banned user cannot just simply make a new account using the same email. Therefore, if someone were to use a banned email to make a new account they will be unable to do so. The email is stored after it has been hashed to keep in place the privacy and security of the user.

7.2. Groups

Groups are another type of reportable entities in our application, meaning just like other reportable entities they too can be reported by users and reviewed by moderators. The table itself keeps track of the id of the owner, id of the group's logo, the group's name, a description, and the style of their borders. For our application a user can only be part of one group, therefore if they are an owner of a group then that group that they own is the only group they can be a part of. This is currently just how the database works, as the "group_id" column is of an int type and not a list of ints therefore the table can only store the id of one group. As for the border style, this determines how hexes will look if a group controls it. Controlling a hex means having the most points in that hex if it's at the lowest resolution, at higher resolution it is determined by what groups control the most smaller hexes that make up that hex. Currently, the only customisation a group has over their border style is the color of their hex fill and the colour of their border line. The return type in the backend logic is a list of JSON nodes, we use this return as it ensures that more customisation options can be easily implemented for future development.

Currently for users to join a group, the owner of the group must send them an invite request. This was a decision made to ensure group members cannot just spam invites to other users or for users not in the group to spam the owner with requests to join the group. When an owner sends a user an invite request, a new row is added to the group_invites table. This table keeps track of the id of the user being invited and the id of the group that has sent the request. When a user accepts an invite, the request is deleted from the table and the "group_id" column of the user is changed to the id of the group.

7.3. Points of Interest (POIs)

Our POIs table represents all points of interest that an experience can be present on. A POI is a contribution, which means it is also a reportable entity and just like other reportable entities, it too can be reported by a user. A POI can either be user created or come from an external source, if it comes from an external source it does not have an owner thus the "contributor_id" column in the contributions table stays empty. As POIs need to be seen on the map, the longitude and latitude is stored in the database. The longitude and latitude are used to create the value of the "origin_internal_hex_id" column. POIs can also have different restrictions on what users can contribute too. There are two restrictions, whether a user is allowed to upload a schedule for the POI and whether a user is allowed to upload photos for the POI.

We have two more POI-related tables in the database. The first table relates to the popularity of a POI . The popularity of a POI relates to the contributions made to the POI, this includes sign ups, check ins, reviews, and contribution images. The popularity of a POI is also dependent on the time. So when a contribution is made to a POI, we first look at the "created_at" column. If less than an hour has passed from the time stored in that column, the "popularity" column is increased based on the amount of points the contribution provides. If more than an hour has passed from the time stored in that column, then a new row is created, with the "created_at" column set to the current time, the "points" column reset and set to the amount of points the contribution provides. We do this to ensure that the most relevant POI is shown as popular to the users, as if it has been a long time since a contribution has been made that it is unlikely to be a very popular POI in recent hours. The other table relates to the visibility of a POI on the map. We have this table to ensure we keep track of the layer that the POI is a part of and the resolution that a POI should be visible at.

7.4. Experiences

The experience table represents all experiences of all layers and POIs in our application. The id of the layer and POI that the experience is attached to, is stored in this table and used to correctly display the experiences on the frontend. The amount of points an experience has is updated each time a contribution is made to the experience, these contributions include a user signing up for the experience, checking into the experience, making a review for the experience and adding an image to the experience's image carousel. The status of the experience refers to how soon it is going to start, if it has gone by, or if it is currently ongoing. This is to allow our frontend to properly notify users on the status of an experience. Moreover, past experiences are shown to users as it will allow them to see previous reviews and images. We also have restrictions that can be made for an experience, these restrictions decide whether users can sign up, check in, or upload photos. The "is_sponsored" column refers to a premium feature where creators of an experience can pay to have their experience be the first to show up when a user clicks on a POI. As payment integration is not part of the final prototype, this feature is currently unavailable to users.

7.5. Images

For all image-related entities such as the images users add to POIs, profile pictures, and cover images, the data of those images are added to the image_data table. All images are reportable entities so when a user adds an image to the system, we first add a new row into the reportables table and use that id as the id for a new row in the image_data table. This id is also then used as a foreign key for other rows that have image-related entities, this is to ensure we know what image data that id is related to. We elected to have this separate image_data table to make reporting images simple in the backend. If we just stored the data of the image in the entities they are a part of, for example storing the image data as part of a user's row in the users table, then the deletion for the images would be a bit more complex. With this image_data table, we can just delete the row in this table and any reference of the id of that row in other tables will be set to null thus deleting the image from other tables.

7.6. Hexes

To store information related to hexes in the database, we decided to have two tables. One that is used to determine the group that controls the hex and the other that is used to determine the amount of points a group has in a hex. Both tables use "internal_hex_id" as a primary key and a "group_id" as a foreign key. While the key constraints are similar in both tables, we decided to have two tables as we needed to query who controls a hex multiple times over a small period of time. Due to the large amount of queries needed, the queries needed to be quick which is why we created the controlled_hexes table. This table stores the group that has the most points in a hex and thus controls the hex.

7.7 Layers

Our client wanted different POIs and experiences to show on the map depending on what layer the user selected. These layers group similar types of experiences together. For example, experiences related to music will be part of a concert layer and experiences related to food will be part of a food layer. Our client wanted a way for new layers to be created easily and so we created the layers table to enable future developers to add new layers into the application. Moreover, the id of each layer row is then used as a foreign key for experiences and POIs to know what layer an

experience and a POI is part of. When a layer is selected by the user in the frontend, the experiences that are shown on the POI screen will be experiences that are a part of that layer. For example, there could be a POI for a stadium. This stadium can have experiences related to sporting events but potentially music concerts too. If a user selects a “sports” layer, only sports related experiences will show on the stadium’s POI screen, and if a user selects a “concert” layer, only concert related experiences will show on the stadium’s POI screen. This is to ensure users see the experiences that they expect to see on a given layer.

7.8. Reports

To enable reports in our application, we created a reportables table where all entities that can be reported by a user will get their id from. Reporting an entity means that the entity in the system is put into the reports table and a moderator will decide whether to delete the entity from the system or not. While the process of reporting an entity is available in our prototype, we unfortunately do not have a moderator dashboard to view and delete reported entities.

For each time a new reportable entity is added to the database, we first add a new row into the reportables table. The id of this new row will then be used as the id of the reportable entity we want to add to the database. This is why for all reportable entities, the main id is a foreign key of the id of a row in the reportables table. Afterwards, if a reportable entity is reported by a user, a new row is added into the reports table where “reported_id” is the id of the reported item.

The following are all the reportable entities:

- Users
- Groups
- Image Data
- Contributions

7.9. Contributions

Contribution is the term we use to define all entities that a user can add to our applications. This includes entities such as user-created POIs and experiences, but also when users sign up and check into an experience. These entities can be reported by other users, and so the id of a contribution is a foreign key of the id of a row in the reportables table. We decided to group these different entities together under the umbrella-term contributions, as all these entities can be created by users and they all have a datetime that they were created at. This is why our contributions table has a “contributor_id” column to save the id of the user that created the entity and a “created_at” to save the datetime that the entity was created at. Furthermore, the “contributor_id” column can be null as we wanted to ensure contribution entities that come from external sources and not users can still be represented in the database.

The following are all the contribution entities:

- Experiences
- POIs
- Reviews
- Images
- Sign Ups

- Check Ins
- Opening Times

Unfortunately, for opening times, while they are considered as contributions in the databases and the backend logic for users to add new opening times exists, we were unable to integrate it with the frontend. So, in the final prototype users are unable to suggest new operating hours for a POI or an experience. This aspect is explained in more detail in the Future Work section (See Section 11.).

8. Performance Considerations

This section aims to explain all the performance considerations that were made throughout the application. Since the performance of the prototype was stressed beginning with the requirements, it had substantial effects on many aspects of the system.

8.1. Frontend

Frontend performance is especially important for the map system. This part of the application displays POI markers and hex ownership visualisation, both of which require a significant amount of data from the server. As users pan around the map, these data requests can happen frequently and in quick succession. To keep the system practically scalable to a very large number of users, it is important to minimize the number of API calls. Reducing unnecessary requests lowers server load and improves responsiveness. Additionally, Flutter web performance is relatively lacking compared to, for example, a traditional Javascript approach, so it is important to avoid as much complex client-side computation as possible to keep the web version of the application performant.

8.1.1. Differential Visibility Management

To prevent the system from having to process data outside the map's viewport, a "Visibility Manager" pattern is used. This visibility manager keeps track of the set of hexes that are currently visible on screen, also pre-loading a certain area around the viewport.

Delta Processing

When a certain movement threshold is reached, i.e. when a viewport border comes sufficiently close to unloaded hexes or the user has zoomed to a point where a different hex resolution should be shown, the visibility manager calculates the difference between the set of hexagons previously in view and the new set, only requesting uncached data from the API for the 'added' hexes.

Resource Disposal

All Dart objects related to 'removed' hexes are unloaded, to keep memory usage to a minimum.

8.1.2 Caching and TTL Policies

The system uses a distributed caching strategy with TTL (Time To Live) policies to minimise the number of API calls made, while keeping cached data relatively up-to-date.

Hex Ownership, Group Style and Marker Caching

The three types of data the map system needs to function are hex ownership records (which hexes are owned by which groups), group style records (what the border and hex fill for a specific group should look like), and POI marker records (where markers should be displayed on the map). These three types of records are stored in three separate caches, for the following reasons:

- POI markers are layer-dependent, while hex ownership and group style are not. When the layer is changed, we do not want to re-fetch hex ownership and style records, so these caches do not react to layer changes.
- The volume of group styles requested from the API will always be smaller than the volume of hex ownership records. To mitigate client-side data duplication and unnecessary memory usage, we cache group styles separately and do local lookups on the client-side, instead of encoding the group style in every ownership record separately.

These caches all use a configurable TTL, currently set to 5 minutes, that ensures that if the client tries to load cached data of which the age exceeds a certain limit, the data is re-fetched.

Geometry Caching

Additionally, the system maintains caches for polygon fill geometries and group border LineStrings. Since H3 coordinates for a specific resolution are static, these geometries are calculated once and stored, allowing the system to rebuild previously visited parts of the hex grid via a lookup, instead of having to recompute a large amount of trigonometric projections.

8.1.3 Lazy Geometry Reconstruction

A significant performance bottleneck in our system is the reconstruction of complex borders. Our system uses a “dirty state” mechanism to optimize this process and prevent unnecessary recalculations.

State Tracking

Internally, an adjacency graph per group is constructed, keeping track of all visible borders around that group’s territory. This adjacency graph tracks changes to its topology. If no edges have been added to or removed from the graph within an update cycle, the graph is marked as “clean”.

Selective Recalculation

During the update cycle, the geometry engine skips any group that is not marked dirty. This ensures that a change in one small region of the map does not trigger a global recalculation of all visible borders.

8.1.4 Efficient Border Computation

The group border rendering system internally relies on a graph-based representation of directed H3 edges to generate continuous LineString paths that can be rendered by MapLibre.

Adjacency Graph Structure

By organizing borders into an adjacency graph (mapping vertex keys to sets of half-edges), the system reduces the complexity of finding the next segment in a path to O(1).

Efficient Path Assembly

The “graph walk” algorithm is designed to assemble the longest possible continuous paths for a single GeoJSON feature. This significantly reduces the total number of features sent to the GPU, as rendering one long LineString is more efficient than rendering many individual edge segments.

8.1.5 Efficient GeoJSON Serialisation

A limitation of the Dart Maplibre library is that it only accepts GeoJSON input in string format. This means that, during every update, the source GeoJSON string must be rebuilt. To maximise the performance of string construction, the following method is used:

Buffered String Construction

The system uses a StringBuffer to manually assemble GeoJSON strings. This avoids the high memory allocation overhead of repeated string concatenation.

Batching

Partial GeoJSON data is batched by group ID. This allows the system to concatenate pre-serialized “GeoJSON batches” into a single feature collection. Using the same “dirty state” mechanism as used in lazy geometry construction, only “dirty” GeoJSON batches are recalculated on each update cycle. This provides a significant improvement in performance compared to rebuilding the entire string on every update cycle, as string building is relatively expensive.

8.2. Backend

Performance considerations for the backend were crucial for the efficient returning of responses to user requests. For the smooth experience of the user on the client, efficiency, especially for database queries, was a primary concern for backend development.

8.2.1. Database

For our database, there are certain data that the frontend needs in large quantities and in short intervals. These data that are required quickly and many times are ones that relate to what the user sees. A user can move around the map in the application many times and very quickly in a short period of time, so the database must be able to provide the required data to allow the frontend to correctly show the information the user needs. Therefore the queries to get these data must be very quick, to ensure the frontend gets the required data. This is why we made certain tables in the database, as the data in these tables are required many times and in a short interval.

The following tables have critically important data:

- controlled_hexes
- poi_popularities
- poi_visibilities

The controlled_hexes table stores data of what group controls a certain hex. Knowing this information is critical for the frontend to properly show the hexes and who controls the hexes on the map. With this table we can just do a simple SELECT query. If we only had the hex_points table, which stores the points a group has in a hex, then the query to find the group that controls the hex, i.e. the group with the most points in the hex, will be much more complex and take longer.

Both the poi_popularities and poi_visibilities tables relate to critical information about a POI, mainly its popularity and its visibility respectively. These terms have been defined previously in the Database Design section (See Section 7.3.). As a user can be moving around the map quickly, the popularity and visibility of a POI needs to be queried quickly to ensure the frontend provides the

correct information to the user in a timely manner. With these two tables we can do that through a SELECT query.

8.3. Hardware Recommendations

Storage

For fast database responses it is extremely important to have fast storage. Therefore fast SSD storage is recommended for the nodes.

Furthermore the nodes should be able to hold all of the data. The map file itself is 150GB and the seeded database is approximately 5GB, which is currently very small and is only expected to grow. We therefore recommend 1Tb of storage as a starting point so that the application has space to expand into.

Memory

To ensure a responsive application, it is important that the amount of memory is sufficient so that data does not need to be loaded as often from the disk. Optimally, each node would be able to fit the entire database in RAM. Furthermore it can be expected that each pod will assume at least 1GB of memory. As a result, the amount of memory should be based on the amount of pods running on the node and any overhead that is needed for the DB and operating system. We therefore recommend an absolute minimum of 16GB of memory but this will most likely increase as the user base increases.

CPU

Similarly, the amount of CPU cores needed is based on the amount of pods that will be running. This is because each pod can be expected to consume at least 1 vCPU (virtual CPU) core. Furthermore, YugabyteDB is a heavy application and is expected to need at least 4 CPU cores, but 8 to 16 are recommended for a production environment. We therefore recommend a CPU with a minimum of 16 cores.

Networking

The main consideration when it comes to networking is latency between nodes. This is because YugabyteDB needs to sync all writes across all nodes. As a result, a maximum latency of 5ms is recommended.

Network throughput depends a lot on the size of the user base. As a starting point, 1Gbps will be enough, but as each node starts to serve more traffic, it may become insufficient.

9. Testing

This section details the testing methodologies employed to ensure the platforms' reliability, security, and performance. These tests ensure that individual components function correctly in isolation and that the integrated system maintains data integrity across the mobile and web interfaces.The source code for the tests can be found in the Appendix (See Appendix 1.).

9.1 Backend Testing

The backend testing suite focuses on testing the Service and Repository layers of the N-tier architecture of the backend code architecture (See Section 5.1.1.). By isolating these layers, the testing process ensures that the core business logic and data access patterns remain reliable.

9.1.1 Service Layer Testing

Unit tests were used to validate the core business logic of the backend, focusing on the Service layer of the N-tier architecture. The primary objective was to achieve 100% code coverage across all classes and methods responsible for the logic of the API routes. By using mock objects to isolate the Service layer from external dependencies like the database, we were able to verify that all Service logic functioned correctly under various edge cases. The results of these tests confirmed that the internal logic met the project requirements, with all methods passing their respective test suites. It is to be noted that while 100% code coverage was the target, some Service classes, such as the EmailService, were left out due to their inherent nature of being not as straightforward to unit test. Instead, they were all manually tested through various user and data flows.

These tests can be found in the `Services/Implementations` directory of the `API_Server.Tests` project of the source code.

9.1.2 Repository Layer Testing

Integration tests were utilized to verify the accuracy and performance of the SQL queries within the repository layer. To ensure data integrity, these tests were executed against a dedicated testing schema in the database, which remains entirely separate from the production environment. This isolation allowed for the execution of complex operations and queries without affecting live data. The results of this testing phase demonstrated that all repository methods interacted correctly with the database.

These tests can be found in the `Data/Repositories` directory of the `API_Server.Tests` project of the source code.

9.2 Frontend Testing

The client application consists mostly of code displaying and managing the different visual interfaces. Flutter, with its 'flutter_test' package, provides functionality for simulating button clicks and gestures, with which you can test whether user interactions trigger the correct functionality. It also supports checking whether widgets are displayed, and where they are displayed. During the development of this prototype, much functionality changed and elements were moved. Automating tests like these would take a significant amount of time, however, which would take away time for

implementing features. For now, we have instead manually tested the different interfaces during development and they seem to all work as expected. In any case, for properly testing whether an interface works, user testing seems like a better strategy as mentioned in the Future Works section (See Section 11.).

For some of the core logic that is widely used throughout the application, however, automated unit tests were written. This includes tests for code responsible for client-server-communication, the data loading pattern (See Section 6.3.) and for validating user input.

These tests can be found in the `test/utils` directory, named `api_client_test.dart`, `data_loader_test.dart` and `validator_test.dart`. The api client tests will test all the different HTTP methods, authentication token refresh logic and JSON response parsing from the ApiClient. The data loader tests test the different loading strategies, states and updates. The validator tests test that all the form validation functions correctly allow/disallow certain inputs. All tests defined in these files pass.

10. Process

This section outlines the management framework and collaborative workflows utilized by the group throughout the project timeline. This approach enabled the continuous integration of feedback from both the client and supervisor, while also maintaining transparent task distribution and regular synchronization within the group.

10.1. SCRUM

For the development of the product, we decided to use the SCRUM Agile-framework. We chose to use this framework to allow for the flexibility to keep up with the demands of the client, and to ensure proper communication and transparency between team members, as task isolation was quite high. We chose a 1-week sprint duration, with a new SCRUM master each week.

10.2. Planning

At the beginning of the project, the planning was quite strict. We had in-person meetings on almost every weekday of the first two weeks. Given the complexity of the project, the initial two weeks were important to plan out the design and scope of the project, make a task division, and schedule weekly meetings with the supervisor and client.

When it came to implementing the project, we chose not to set strict deadlines for specific features. We did this because we had to work with unfamiliar tools, so we found it hard to make an accurate estimate of when features could be completed. Instead, we made a priority ranking, and simply started working on features in order of priority. During each SCRUM meeting, we talked about what we would be doing for the upcoming week and any problems that we may have. For the most part, the frontend and backend work was done separately. When it came time to integrate the backend to the frontend, we communicated any issue we faced to each other over discord to resolve the issue.

Nearing the end of the project, we started to keep track of the deadlines for the final deliverables of the project. This was to ensure we understood the amount of time we had left to provide the final deliverables, so we could change the priority of certain features if deemed necessary due to time constraints.

10.3. Division of Tasks

The general task division decided on as a group after the design stage of the project is as follows:

Frontend

- **Niek**: General user interfaces, Client architecture, Client-server integration
- **Milan**: Map, iCal schedule builder, Creation/editing forms

Backend

- **Junseo**: API routing, Backend service logic, Database seeding
- **Dio**: Database queries

- **Matas**: Deployment environment, Development environment, Experience scheduler, Experience scraper

10.4. Supervisor Meetings

One meeting with the supervisor was held every week. These were used to present our progress to the supervisor and receive feedback on our work. Before each meeting, the team would get together to make a short agenda for the meeting. The meeting was always led by the current week's SCRUM master. This has worked quite well, and the meetings have gone smoothly.

10.5. Client Meetings

One meeting with the client was held every week. These meetings mainly functioned to make sure what we were implementing stayed in alignment with what the client wanted. This frequency of meetings turned out to be necessary, as some of the client's requirements changed throughout the course of the design and implementation of the product. The requirements list in Appendix 2 is the final requirements list we procured by the end of the project.

10.6 Green Card

Due to his outstanding contributions and dedication to the project while also leading the communication with both the client and the supervisor, we have decided to award Junseo Kim (s2648687) the green card for this project. His high availability and quick responses were greatly appreciated by the rest of the team.

11. Future Work

The goal of this project was to develop a functional prototype, and in this capacity, the project has been a success. While certain “must-have” requirements were not fully realized within the given timeframe, the core infrastructure has been implemented. To transition this system from a prototype to a production-ready Minimum Viable Product (MVP), several key areas of development must be addressed. Should a future team continue this project, the following considerations are essential.

11.1. Extensions

While the prototype successfully demonstrates the core functionality of the platform, certain “must-have” requirements identified in the initial planning phase (See Section 2.1.) were not fully realized due to time constraints. Addressing these missing core features is the primary priority for future development, followed by the implementation of broader system extensions.

11.1.1. Moderation

An important set of must-have requirements that we did not manage to implement are related to the moderation system. It is currently possible to report contributions, but there is no dashboard where moderators can review reports and resolve them. For an MVP, content moderation is crucial. There will be a lot of user-generated content on the app, meaning there is a lot of possibility for uploading harmful or inappropriate content. Without a complete moderation system, the platform could quickly become a toxic space.

11.1.2. Payment

Payment integration was one of the core requirements of the client, as they wanted to monetize this product. Our prototype did not have any monetization, but for the MVP, various payment options and corresponding rewards and actions must be considered.

11.1.3. Friend System

A friend system was another requirement that we were not able to implement. While the database structure and API supports such logic, the user interface was not realized due to time constraints. The client requested such a feature to increase user engagement within the platform. For an MVP, users must be able to add friends and be able to interact with them.

11.1.4. Privacy

Another important set of features are related to visibility of user-generated content. For privacy reasons, a user might only want certain people to be able to see their POIs or information on their profile. A feature that goes hand-in-hand with this is the friend system. With a friend system, users could invite only certain people to their experiences, or share their location with them.

11.1.5. Opening Time Contributions

An initial requirement that was set was for users to be able to upload “opening times” for POIs to gain points. This would require the system to enable any user to upload such contributions

and also enable the owner of a POI to accept the most appropriate contribution. While this could not be done due to time constraints, an MVP would need to implement this feature.

11.1.6. Leaderboard

The current points leaderboard for both users and groups are minimal. The client wanted a leaderboard that would increase competition between users and groups, and also one that could be used to gauge where a user or group stands within the platform. For an MVP, the leaderboard must be made more engaging, with pagination, so that any user can navigate through it to find their relative positions.

11.1.7. Gamification

The gamification side of the application also has potential for extension. An example would be a system where different groups can form alliances. A problem with the current group system is that it will be nearly impossible for a local group to have influence on a larger scale. A system where groups can collaborate in alliances and earn points collectively, on a larger scale, would allow for players to feel like they are contributing to something bigger. Additionally, you could have different levels of alliances. Think of having an alliance within a city, which is part of an alliance at provincial scale, which is part of one at country scale, which is part of one at continent scale.

Much inspiration can be taken from live service games. Common features found in those games include quests, seasonal events and cosmetics. To make the application a viable business option, payment integration could be a great addition as well. Businesses could use the platform as an advertising platform, paying money to sponsor their experiences, increasing their exposure. Players could spend money to get in-game currency, which they could then spend on cosmetics and boosts.

Preventing cheating is another area that needs more development. Currently, players are able to check in to an event from any location, even if they are not at the event. One of the planned features was to add a GPS-based anticheat, which would require you to verify your location to be close to the POI. Such a GPS-based anticheat does bring about questions of user privacy and security. Under the EU's GDPR regulation, GPS tracking is allowed but under strict circumstances. User permission will be required before the application can start tracking GPS location however it may also be possible to justify GPS tracking on the grounds of legitimate interest, as GPS tracking is required to prevent cheating in the application. Legitimate interest means data processing that is required for the application to function correctly. Nevertheless, no matter the justification for GPS tracking, tracking a user's GPS location must only be conducted when the user opens the application. For the anticheat, this is not a problem as to verify that the user is close to the POI, the application must be open anyways. Other checks could be added too, like needing to be checked in to an experience to be able to upload photos for it, or needing to have been near a POI to place a review. Location could also be used to limit the area where you can create your own POIs and experiences.

11.1.8. External Integrations

As a tech demo, a layer containing events retrieved from the TicketMaster API, in addition to randomly generated content, was added. Many other integrations could be imagined in this project's future. The broad vision of this application is to have it be the 'Map of Everything'. It should be the app people go to whenever they want to find anything in the area. Imagine being able to find fun activities to do with friends, ongoing discounts at various shops, sports events, festivals and more.

More complete integrations could be imagined too, like the possibility of making restaurant reservations through the app, calling taxis and ordering food. Anything that would make sense to find on a map could be added to the system. Some external integrations also feature affiliate programs. This could be another great additional source of income.

11.2. Verification and Revisions

While the functional logic of the platform has been validated through the prototype, the transition to an MVP would require a phase of verification and aesthetic refinement. This process focuses on transforming the current “function-first” implementation into a user-centric application.

11.2.1. Branding and Styling

For the project to get to an MVP state, it is important for the application to get a compelling name and logo, plus a consistent design language that comes with a brand identity. Currently, default styles were used, from Google’s Material Design suite. This was done to greatly speed up development, as the aim of this project was to value function over form. Eventually though, the product does need its own brand identity. The idea currently is for the application to use a liquid glass style, fitting in well with the modern design that is popular right now. The clean aesthetic of this style would fit in well with the colourful map.

11.2.2. User Experience and Testing

As function was prioritized over form, the aim was mostly to add user interface elements for their functionality, without necessarily investigating the implications for the user experience of the positioning of interface elements and the different flows of the application. For a minimum viable product, user testing should be carried out to find problems with the current interface. Likely, many elements need to be changed, moved, or even completely replaced. It could very well be that the different flows the user frequents are not intuitive, requiring a restructuring of the navigation of the application.

User testing will likely also help discover many different features that would improve the application as a whole. As of now, only a handful of people have had the ability to suggest different features, which is likely insufficient for the scale this project aims to reach. We imagine that for this project to become an actual product, many rounds of user testing, feedback sessions, and design revisions need to be performed.

12. Conclusion

From the start of the project, we understood that the application our client was asking for would be quite complex and have a lot of moving parts. Which is why at the beginning of the project the planning was strict and there were many in-person meetings. We wanted to ensure when we started implementation we had ironed out most of the requirements and features that the final prototype would have. For the most part the initial planning phase helped quite a lot to help our direction during implementation. However, it was still difficult to predict how long some requirements and features would take. Many of the tools we used during the project were the first time we used them, so there was a lot of learning during the project that added to the time it took to implement certain requirements and features.

The difficulty in predicting how long some implementation would take also extended to the internal deadlines we had with our client. We had an internal midway point meeting with our client (See Appendix 6.). By the time of this meeting we wanted to get most of our ‘must-have’ requirements completed. Unfortunately, this did not happen and so we had to push back some ‘must-have’ requirements to the second half of the project timeline. Even then, there were still some ‘must-have’ requirements that we were working on until our final prototype meeting with the client. Some features that we elected to remove from the final prototype were payment integration, the friends system, and a moderator dashboard. The exact requirements that were implemented and not implemented are denoted in the Appendix (See Appendix 2.).

Overall, the ambitious scope for the project did bring a lot of difficulties during implementation. However due to this ambitious scope it forced us to reevaluate the scope and the requirements, which gave good insight into the difficulties in such a project and how to solve such problems. The final prototype we delivered to our client does meet their basic vision. It is an application in which users can find events and places to go in their area combined with the gamification elements our client requested.

13. Statement on the Use of AI

No AI was used at any point in the process of writing the design report, manual and creating the poster. Neither was any AI used in the design process of the application.

For implementation/coding, with permission of our supervisor and client, AI was used to aid in learning how to use the chosen technologies, which we had all not worked with before. Additionally, it was used to help with library integration and writing code documentation and tests. In all cases, output from the LLMs was carefully audited and modified as needed before accepting it.

14. Sources

Data layer - Architecture. (2025, September 5). Flutter documentation.

https://docs.flutter.dev/app-architecture/case-study/data-layer

Licence/Attribution Guidelines. (2022, March 16). OpenStreetMap Foundation.

https://osmfoundation.org/wiki/Licence/Attribution_Guidelines

Liquid Glass. (n.d.). Apple Developer.

https://developer.apple.com/documentation/TechnologyOverviews/liquid-glass

Sebayang, E. A. (n.d.). *Civilization 6: Best City-States To Suzerain*.

https://gamerant.com/civilization-6-best-city-states-to-suzerain/

UI layer case study - Architecture. (2026, January 14). Flutter documentation.

https://docs.flutter.dev/app-architecture/case-study/ui-layer

Appendix

Appendix 1. Source Code

The source code can be found at the following link:
https://drive.google.com/drive/folders/1wFOI5blPj4Cij6k_c9GBnHVrPOyYFwru?usp=sharing

Appendix 2. Full Requirements List

The complete requirements list given by the client is listed below. Requirements with checkmarks (✔) at the end are requirements we have fully implemented in the final prototype. Requirements with crosses (✘) at the end are requirements that we were unable to implement in the final prototype.

Green - Must have
Yellow - Should have
Red - Could have

Functional Requirements

Map

1. The system shall have a map of the whole world that can be divided into arbitrary sized hexes - ✔
2. The system shall have hexes that have other hexes within them - ✔
3. The system shall have hexes that can be captured - ✔
4. The system shall have a map with the hex system for the Netherlands - ✔
5. The system shall have the map with the hex system be theoretically scalable to the whole world - ✔
6. The system shall allow users to set hexes on the map they want to be notified about - ✘

Layers

1. The system shall contain functionality for implementing map layers - ✔
2. The system shall allow experiences bound to a layer to be made by users - ✔
3. The system shall allow layers to be selected on the user interface - ✔
4. The system shall provide the user with a layer-specific set of options when creating experiences - ✘
5. The system shall allow experiences bound to a layer to be added from external sources via APIs or web scraping - ✔
6. The system shall contain an example layer with venues and events - ✔
7. The system shall allow developing new map layers without changes to the core codebase - ✘

Points of Interest (POIs)

1. The system shall have POIs with latitude/longitude, name, description, (optional) pictures - ✔
2. The system shall allow users to create POIs - ✔
3. The system shall have POIs be attached to a hex - ✔
4. The system shall have user created POIs be linked to an account - ✔
5. The system shall have 2 types of POI visibility: temporary and permanent - ✔

6. The system shall make all user POI visibility temporary by default, so personal locations such as one's house do not always show up on the map - ✔
7. The system shall allow users to request a POI's visibility to become permanent - ✘
8. The system shall have POI popularity increase when users make contributions - ✔
9. The system shall base POI visibility in a layer on its popularity in a certain timeframe, relative to the popularity of other POIs in that layer - ✔
10. The system shall allow premium users to request becoming owner of an externally sourced POI - ✘
11. The system shall allow the owner of a POI to change the name, descriptions and (optional) pictures - ✔
12. The system shall allow POIs to be externally sourced via APIs or web scraping
13. The system shall keep all externally sourced POIs permanently visible - ✔

Experiences

1. The system shall have experiences with name, description, (optional) pictures - ✔
2. The system shall have experiences attached to a POI - ✔
3. The system shall allow users to host experiences at a POI - ✔
4. The system shall allow users to sign up for experiences - ✔
5. The system shall allow the owner of an experience to configure whether users can sign up and/or check in at the experience - ✔
6. The system shall track the popularity of experiences that are planned or happening - ✔
7. The system shall indicate on the map when a POI has an experience planned or happening - ✔
8. The system shall allow any user to add experiences to a POI - ✘
9. The system shall distinguish experiences added by the owner of a POI from those added by other users - ✘
10. The system shall allow the owner of an experience to edit the name, description and (optional) pictures - ✔
11. The system shall allow users to set the visibility of the experiences they host (invite-only, friends, public) - ✘
12. The system shall allow users to sponsor their experiences - ✘
13. The system shall indicate an experience is sponsored in the client interface - ✘
14. The system shall award players with more experience points when interacting with sponsored experiences - ✘

Contributions

1. The system shall allow users to check in to experiences – ✔
2. The system shall give more experience points to users that checked in to an experience after signing up - ✔
3. The system shall allow users to upload photos for POIs and experiences - ✔
4. The system shall allow users to leave reviews for POIs and experiences - ✔
5. The system shall allow users to add opening times to POIs and experiences - ✘

Users

1. The system shall have the map be viewable without an account - ✔
2. The system shall have user accounts - ✔
3. The system shall have users verify their email before allowing account-only functionality - ✔
4. The system shall allow users to change their password via email - ✔
5. The system shall allow users to delete their accounts - ✔

6. The system shall have a map interface for users - ✔
7. The system shall allow users to create accounts and log in - ✔
8. The system shall allow users to gain experience points - ✔
9. The system shall allow users to refresh the map at any point - ✘
10. The system shall allow users to report POIs, experiences, other users, groups, reviews and photos for moderator review - ✔
11. The system shall allow users to pay to upgrade to a premium account - ✘
12. The system shall allow users to add others as friends - ✘
13. The system shall allow users to see themselves on the map based on their actual location - ✘
14. The system shall allow users to see other users in their friendlist on the map - ✘

Moderation

1. The system shall have moderator accounts - ✔
2. The system shall include a dashboard where moderators can review reported content - ✘
3. The system shall hide content when it is deemed harmful by a moderator - ✘
4. The system shall allow moderators to delete contributions if need be - ✘
5. The system shall allow moderators to ban users (by email) - ✘
6. The system shall allow moderators to review requests for POIs to become permanent - ✘

Location History

1. The system shall allow users to opt-in to long-term tracking of GPS location - ✘
2. The system shall allow users to view their complete history of contributions and location - ✘

Gamification

1. The system shall allow users to earn experience points for contributions - ✔
2. The system shall allow users to view experience points earned by users on a global leaderboard - ✔
3. The system shall allow users to create one group each with a name, description and colour - ✔
4. The system shall allow users to join one group at a time - ✔
5. The system shall allow the owner of a group to edit the name, description and colour - ✔
6. The system shall allow users to leave groups - ✔
7. The system shall track experience points earned per group - ✔
8. The system shall allow users to view experience points earned by groups on a global leaderboard - ✔
9. The system shall track experience points earned for each group for each hex - ✔
10. The system shall have the group with most experience points earned on a hex ‘own’ that hex - ✔
11. The system shall allow users to earn in-game currency for contributions - ✘
12. The system shall allow users to buy in-game currency - ✘
13. The system shall allow users to spend in-game currency on cosmetics - ✘
14. The system allows the owner of a POI to add a quest to it - ✘
15. The system shall reward experience points to users for completing quests - ✘
16. The system shall allow the creator of a group to change its visibility (public, invite-only, friends) - ✘
17. The system shall reset experience points earned on a seasonal basis - ✘
18. The system shall allow users to view leaderboards scoped to different region sizes - ✘

Non-functional Requirements

Interface

1. The system shall be usable without a tutorial - ✔
2. The system shall have a user interface that uses consistent design language - ✔
3. The system shall have the interface be usable for any device with a web browser - ✔
4. The system shall have a UI that 80% of users find easy to navigate - ✘
5. The system shall have a moderator interface that uses consistent design language - ✘
6. The system shall have a user interface that uses the liquid glass design style - ✘

Security

1. The system shall have passwords that are hashed and salted - ✔
2. The system shall have GPS-based or QR code-based anti-cheat for check ins - ✘
3. The system shall be GDPR compliant - ✘

Deployment

1. The system shall be deployable without vendor lock-in - ✔
2. The system shall theoretically scale to any amount of users without changes to the implementation - ✔

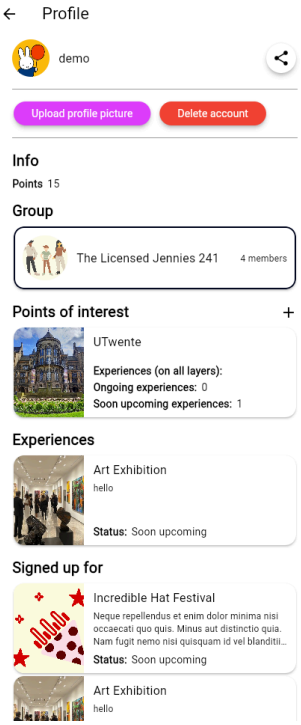
Appendix 3. Client Manual

The client manual can be found at the following link:
https://drive.google.com/file/d/1uKRX4YuFrQowJg8PdRavhgK9HqqDVny8/view?usp=drive_link

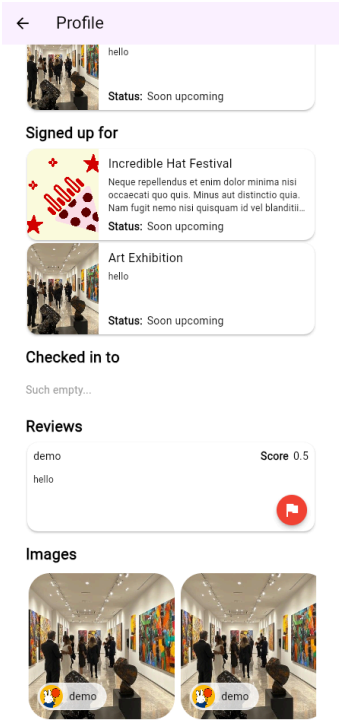
Appendix 4. UI Views

This section contains a set of screenshots of different UI views that were not covered in Section 6.1.

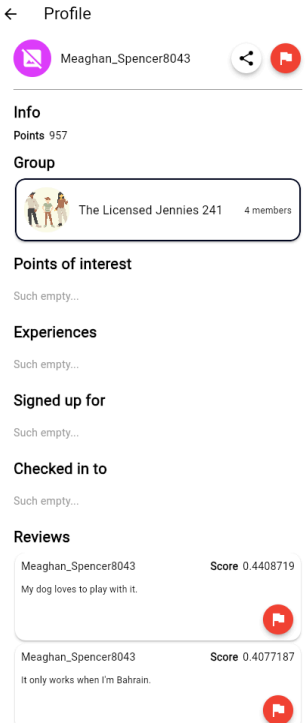
User profile (owned by current user)



User profile (owned by current user, scrolled down)



User profile (owned by other user)



New image contribution

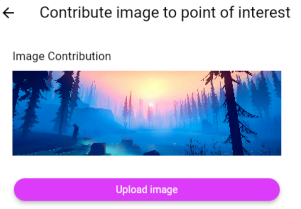
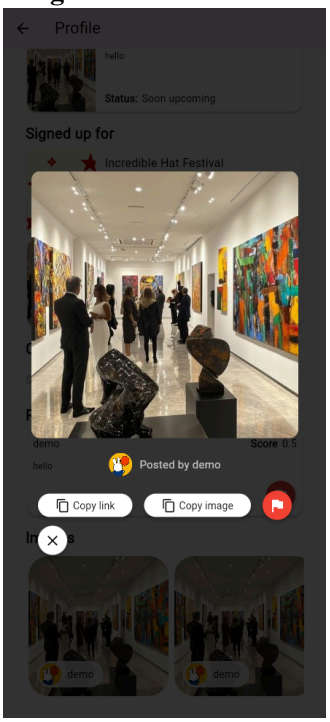
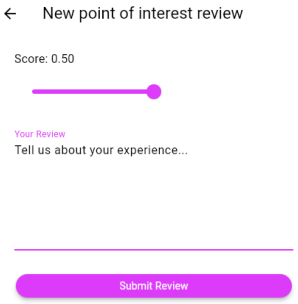


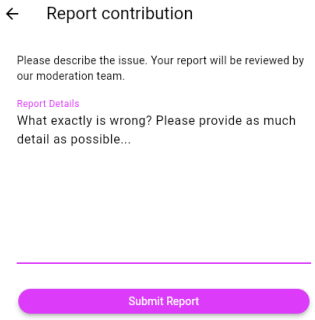
Image contribution



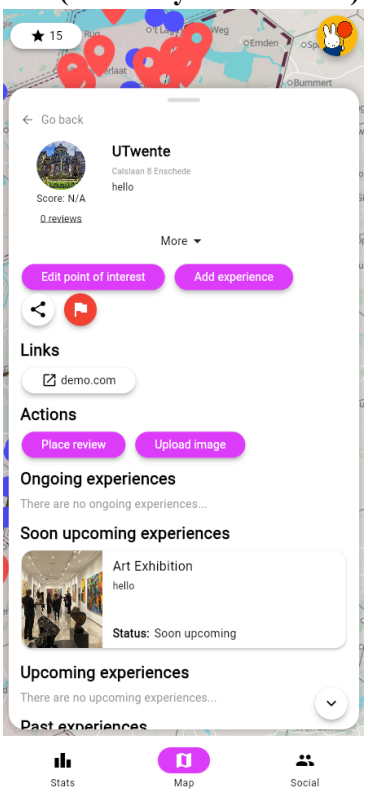
Place review



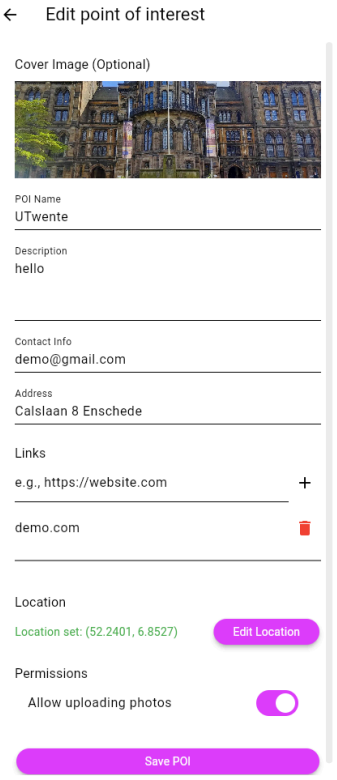
Create report



POI (owned by current user)

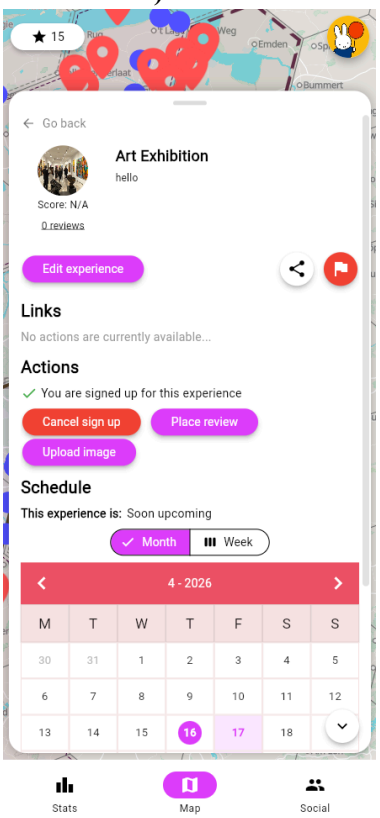


New/edit POI

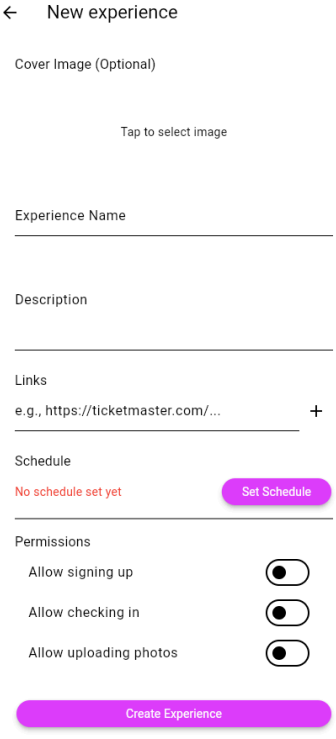


POI location picker

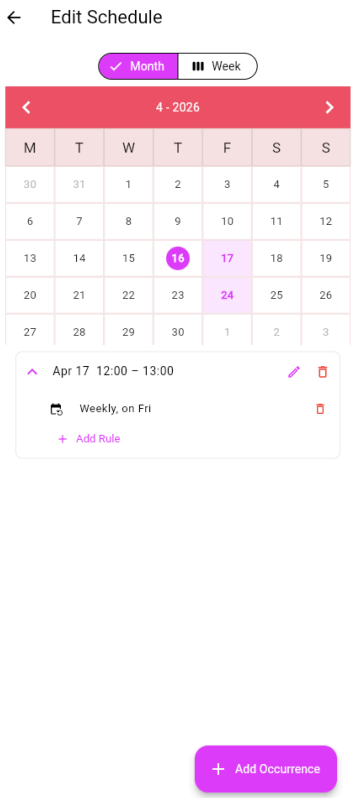
Experience (owned by current user)



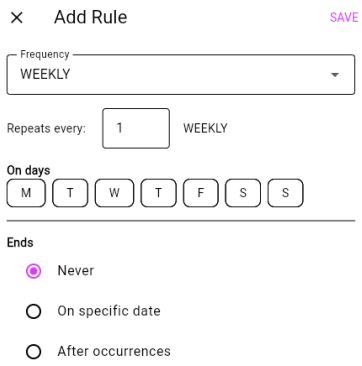
New/edit experience



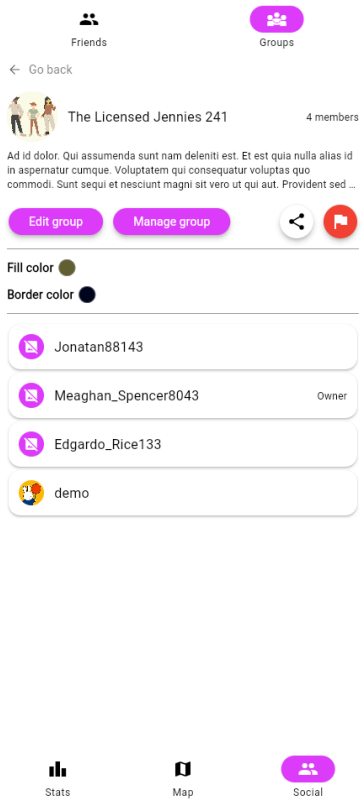
Edit schedule



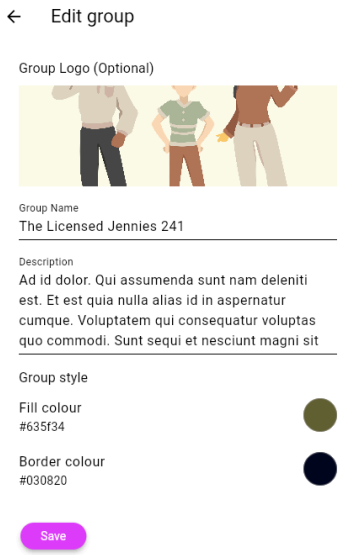
Add schedule recurrence rule



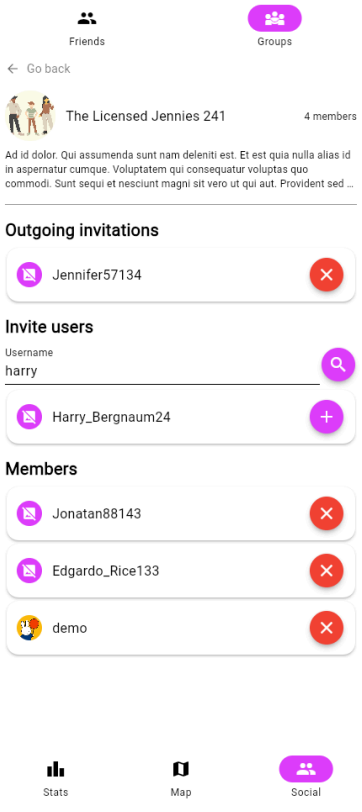
Group (owned by current user)



New/edit group



Manage group



Appendix 5. Group Contract

Team Members:
Niek Peters - s3126714
Junseo Kim - s2648687
Milan Oosterink - s3127168
Matas Aizenas - s3178757
Dio Nirwikara - s3176991

Document Purpose

The purpose of this document is to outline the norms and practices that the team will follow during this project. The guidelines outlined by this document are agreed on by all of the team members who have signed it. Any changes made to this document after the initial signing date need to be agreed and signed upon by all of the people. Failure to comply with this document will result in repercussions stated in section 4.

1 Expectations

1.1 Deadlines
It is expected that all of the deadlines, official and self assigned, are met with punctuality. If due to unforeseen circumstances a deadline can not be met all group members have to be notified of this fact as soon as possible. If a missed deadline results in unequal contribution a penalty from section 4.1 is given.

1.2 Equal contribution
All members should contribute equally to the project. It is the responsibility of each person to make sure that they are contributing an equal amount. Failure to comply with this section will result in penalty in section 4.1.

1.3 Sickness
If a team member is sick they are allowed to not work until they recover. The sick team member should notify others of their absence.

2 Communication

2.1 Communication channels
The main text communication channel is Discord. For keeping track of tasks the method of communication is Trello.

2.2 Communication period
Each team member is expected to respond within 5 business hours (Mon-Fri, 9-17:00). If the lack of communication is severe enough to cause someone to possibly receive a penalty the person who did not communicate is the one receiving it.

3 Meetings

3.1 Meeting attendance

Meeting attendance is mandatory although if needed and agreed upon, it can be done online. For failing to attend a minimum penalty is given from section 4.1. If two meetings are missed in a row the penalty from section 4.2 is issued with no warning.

3.2 Meeting schedule

There are weekly meetings every Thursday at 12:00. These meetings will be used to facilitate SCRUM and for any other project related discussion. The SCRUM master each week is assigned to book a room for the meeting. Other project meetings will be made on demand if necessary.

4 Penalties

Before any penalty is issued the offending team member must be given a warning and some time to fix their mistake. Except section 4.2.

4.1 Unequal contributions

The other team members decide if the lack of contribution is a minimal infraction or a large infraction

Minimal Infraction

The offending team member is assigned to do extra work which they are expected to complete.

Large Infraction

The offending team member is given a red card at the end of the project.

4.2 Missed meetings

The offending team member has to provide snacks for the next meeting.

5 SCRUM Master Assignment

Sprint Start Date	Sprint Number	SCRUM Master
5-2-2026	1	Dio
12-2-2026	2	Niek
19-2-2026	3	Junseo
5-3-2026	4	Milan
12-3-2026	5	Matas
19-3-2026	6	Dio
26-3-2026	7	Niek
2-4-2026	8	Junseo
9-4-2026	9	Milan

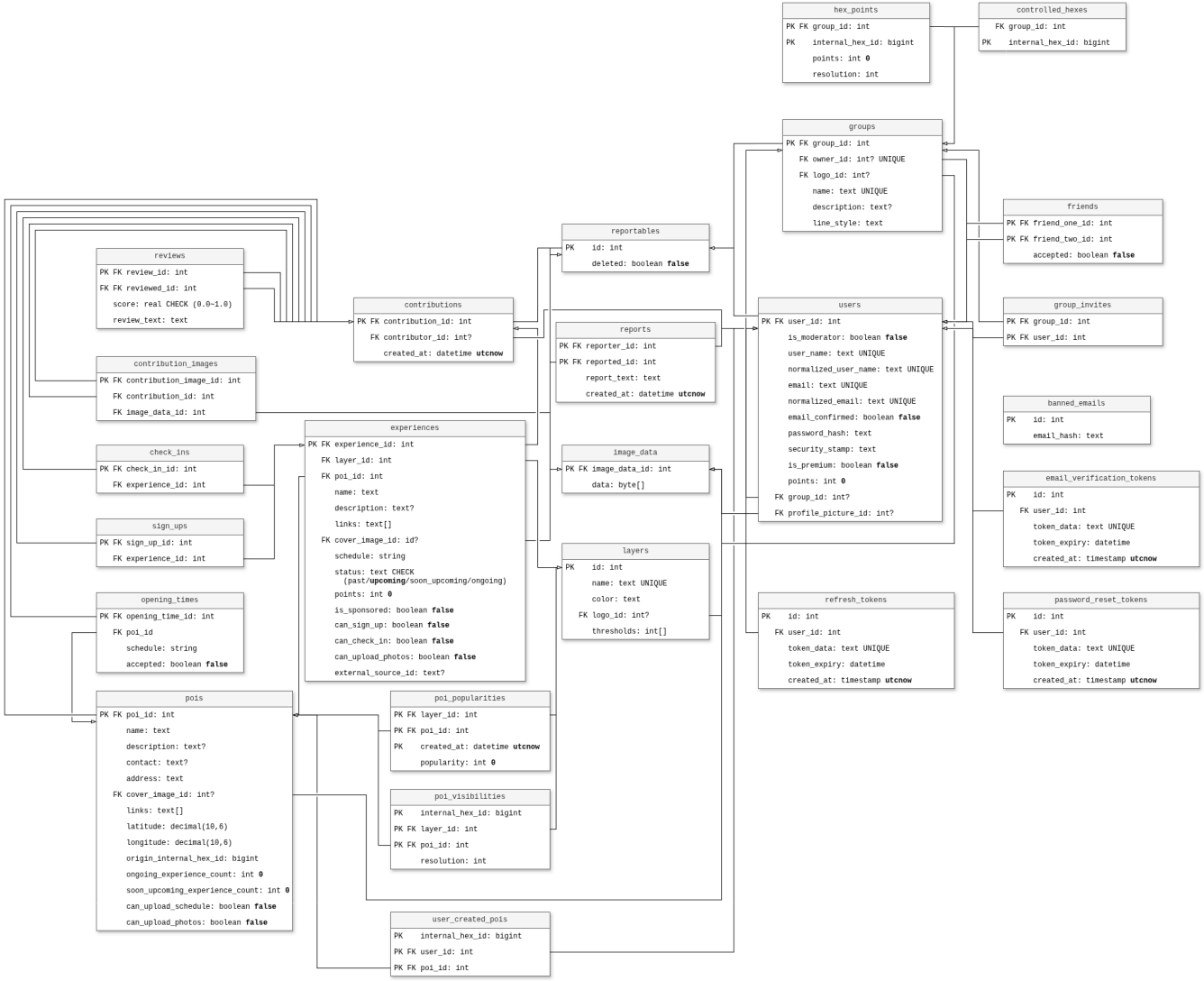
Team Member	Date	Signature
Niek Peters - s3126714	09-02-2026	
Junseo Kim - s2648687	09-02-2026	
Milan Oosterink - s3127168	09-02-2026	
Matas Aizenas - s3178757	09-02-2026	
Dio Nirwikara - s3176991	09-02-2026	

Appendix 6. Meeting Overview

Meeting Purpose	Date	Meeting Leader	Notes
Initial Project Meeting	04/02/2026	N/A	Contacted client, started to find a supervisor, discussed the project and group strength
Initial Client Meeting	05/02/2026	N/A	Requirements elicitation, defined contact with client, set up recurring meetings
Team Contract Meeting	06/02/2026	N/A	Created team contract, finished requirements list, created initial system design, decided on technology to use
Supervisor Meeting	09/02/2026	Niek Peters	Introduced project to supervisor, defined supervisor's role, set up recurring meetings
Group Meeting	10/02/2026	N/A	Responded to client feedback on project proposal, continued design on database and API routes
Group Meeting	11/02/2026	N/A	Finished initial database design, started wireframing, set up github, updated project proposal
Scrum Meeting and Client Meeting	12/02/2026	Niek Peters	Completed scrum meeting, discussed project proposal with client in-person
Group Meeting	13/02/2026	N/A	Updated project proposal
Supervisor Meeting	16/02/2026	Niek Peters	Discussed supervisor's comments on project proposal, discussed licensing, discussed use of AI
Scrum Meeting and Client Meeting	19/02/2026	Junseo Kim	Showed initial UI wireframes and demo map, discussed user testing, discussed licensing
Supervisor Meeting	02/03/2026	Junseo Kim	Showed UI progress and Trello board, discussed possible meeting between supervisor and client
Scrum Meeting and Supervisor Meeting	05/03/2026	Milan Oosterink	Initially meeting with supervisor and client, client could not come
Supervisor Meeting	09/03/2026	Milan Oosterink	Showed project progress
Scrum Meeting and Client Meeting	12/03/2026	Matas Aizenas	Midway point meeting with client, discussed plans for final prototype

Supervisor Meeting	16/03/2026	Matas Aizenas	Showed project progress, discussed design report
Scrum Meeting and Client Meeting	19/03/2026	Dio Nirwikara	Discussed lowering priority of some requirements, discussed must-have requirements for final prototype, discussed future plans
Supervisor Meeting	23/03/2026	Dio Nirwikara	Showed project progress
Scrum Meeting and Client Meeting	26/03/2026	Niek Peters	Showed project progress, discussed final todos for the final prototype meeting with client
Supervisor Meeting	30/03/2026	Niek Peters	Showed project progress
Supervisor Meeting	07/04/2026	Junseo Kim	Showed final progress, discussed deadlines for final deliverables and chair meeting
Final Prototype Client Meeting	07/04/2025	Junseo Kim	Final prototype meeting with the client, showed the final prototype of the application, discussed refinements that will take place in the coming week
Final Meeting with Client, Supervisor, and HMI Chair	13/04/2025	*Everyone*	Final presentation of project in front of client, supervisor, and chair of HMI group

Appendix 7. Database Design



Users: represents all users in the system, both regular users and moderators
- user_id: FOREIGN KEY from **Reportables** table
- group_id: FOREIGN KEY from **Groups** table
- profile_picture_id: FOREIGN KEY from **Image Data** table

Banned Emails: represents all banned emails in the system, banned emails cannot be used to create an account in the application

Friends: represents users that are friends and friend requests
- friend_one_id & friend_two_id: FOREIGN KEY from **Users** table

Groups: represents all groups in the system
- group_id: FOREIGN KEY from **Reportables** table
- owner_id: FOREIGN KEY from **Users** table and is UNIQUE
- logo_id: FOREIGN KEY from **Image Data** table

Group Invites: represents all group invites, incoming and outgoing, in the system
- group_id: FOREIGN KEY from **Reportables** table
- user_id: FOREIGN KEY from **Reportables** table

Hex Points: represents the points a group has in the hex
- group_id: FOREIGN KEY from **Reportables** table

Controlled Hexes: represents the group that controls the hex by having the most points in the hex
- group_id: FOREIGN KEY from **Reportables** table

Refresh Tokens: represents the JWT refresh tokens used for authorization
- user_id: FOREIGN KEY from **Reportables** table

Email Verification Tokens: represents the JWT tokens used for email verification
- user_id: FOREIGN KEY from **Reportables** table

Password Reset Tokens: represents the JWT tokens used for resetting a password
- user_id: FOREIGN KEY from **Reportables** table

Layers: represents all implemented layers in the system
- logo_id: FOREIGN KEY from **Image Data** table

Reports: represents all reports made by users on a reportable item
- reporter_id: FOREIGN KEY from **Users**table
- reported_id: FOREIGN KEY from **Reportables** table

Reportables: represents all reportable entities in the system

Contributions: represents all contributions entities in the system
- contribution_id: FOREIGN KEY from **Reportables** table
- contributor_id: FOREIGN KEY from **Users** table

POIs: represents all POIs in the system
- poi_id: FOREIGN KEY from **Contributions** table
- cover_image_id: FOREIGN KEY from **Image Data** table

POI Popularities: represents the popularity of POIs in the system
- poi_id: FOREIGN KEY from **POIs** table
- layer_id: FOREIGN KEY from **Layers** table

POI Visibilities: represents the visibility of POIs in the system
- poi_id: FOREIGN KEY from **POIs** table
- layer_id: FOREIGN KEY from **Layers** table

Experiences: represents all experiences in the system
- experience_id: FOREIGN KEY from **Contributions** table
- cover_image_id: FOREIGN KEY from **Image Data** table
- poi_id: FOREIGN KEY from **POIs** table
- layer_id: FOREIGN KEY from **Layers** table

Reviews: represents all reviews made by users
- review_id: FOREIGN KEY from **Contributions** table
- reviewed_id: FOREIGN KEY from **Contributions** table

Check Ins: represents users “checking in” to an experience
- check_in_id: FOREIGN KEY from **Contributions** table
- experience_id: FOREIGN KEY from **Experience** table

Sign Ups: represents users "signing up” for an experience
- sign_up_id: FOREIGN KEY from **Contributions** table
- experience_id: FOREIGN KEY from **Experience** table

Opening Times: represents the opening times of a POI
- opening_time_id: FOREIGN KEY from **Contributions** table
- poi_id: FOREIGN KEY from **POIs** table

Contribution Images: represents images made for contributions
- contribution_image_id: FOREIGN KEY from **Contributions** table
- contribution_id: FOREIGN KEY from **Contributions** table
- image_data_id: FOREIGN KEY from **Image Data** table

Image Data: represents all images in the system
- image_data_id: FOREIGN KEY from **Reportables** table