
Hansel

Hansel

CS3216 Assignment 3

Group 4

Cai Jialin (A0205575A)

Ian Yong Yew Chuang (A0135451M)

Sebastian Toh Shi Jian (A0196545R)

Wu Weiming (A0210627N)

Milestone 0: Describe the problem that your application solves. (Not graded)

Have you ever thought about friends and family while going about your day and wished they were there? Perhaps you ate some delicious food that you wanted to share, or you simply saw a duck which reminded you of your friend.

With our busy and fast-paced lifestyle, it is difficult for us to meet up with friends, and the pandemic has only caused these catch up sessions to become even less frequent. It is especially frustrating when you are even visiting the same places as your friends, but your paths do not cross because of conflicting schedules. For example, with most lessons being carried out online, many students who do not stay on campus find that their friends do not turn up for classes on the same day as they do.

Moreover, the immediacy of social media and messaging apps also mean that conversations are no longer something we actively anticipate or look forward to. The happiness we feel when we receive an instant message is much more diluted compared to a long awaited letter.

Milestone 1: Describe your application and explain how you intend to exploit the characteristics of mobile cloud computing to achieve your application's objectives, i.e. why does it make the most sense to implement your application as a mobile cloud application?

At its core, Hansel keeps you connected with your friends, closes the physical distance and helps to create shared memories even when you have missed each other at a place, all while maintaining a touch of novelty and surprise in your conversations. Even if you find yourself going to campus alone, finding a message left for you by your friend there could brighten up your day. The app name is a fond reference to Hansel, the eponymous character in Hansel and Gretel who leaves a trail of breadcrumbs to mark his path. Similarly, our app allows users to leave a metaphorical trail of breadcrumbs for their friends that mark their path of travel.

Users can leave media packages (‘iced gems’) for their friends at their current location, in the form of an image and an accompanying text. After it is dropped, its location will be displayed on the receiver’s map and can only be opened if the receiver is in the vicinity of the gem. After collecting and viewing the media package, the gem is then saved to be viewed again later if they wish.

Implementing it as a mobile cloud application is the most suitable due to the portability of mobile phones: our users would likely be on the go (e.g. travelling on campus) while using it as they need to be in the vicinity of a gem before it can be open and viewed. Another bonus would be that such an application would save mobile memory space, which may be an important consideration for users.

Milestone 2: Describe your target users. Explain how you plan to promote your application to attract your target users.

Target user
Our primary target users are youths and young adults who lead relatively busy lifestyles and are also interested in keeping in close contact with friends. Not only are they proficient with technology, they are also more used to communicating with friends online as compared to older generations.

For youths and young adults, friendships are especially important and indispensable. Unfortunately, they can be difficult to maintain when there are so many other things to tend to in life. Moreover, the situation is aggravated by the pandemic: in a survey of 1000 Singaporean respondents, it was found that 61% expressed that they now socialise less frequently with those outside their immediate family, and 44% even said that their social circles have shrunk.[1]

It seems evident that the typical way to keep up with friends through messaging isn't cutting it, and it is also not feasible to meet up regularly due to busy lifestyles and COVID restrictions. Hansel aims to bring value to the table for our target audience as a complement to traditional messaging apps by making a message from a friend something that they can anticipate and look forward to. Its geolocation feature also helps them to feel like they are in the same place at the same time with their friends, allowing them to create fond shared memories tied to a location.

Marketing plan
A benefit from the nature of the app is that it does not require a decent sized user base (as required by apps such as online community marketing platforms). As such, our marketing plan does not necessarily need to focus on trying to gain as many users as possible, but rather can also rely on slow and steady growth.

With this in mind, we have devised a dual pronged marketing plan to help with the expansion of our application.

---

[1] https://www.straitstimes.com/singapore/health/one-year-after-circuit-breaker-people-in-spore-socialising-less-working-more-mental

Channel 1: Online
Our first approach to publicity would be to establish a social media presence. We will set up accounts in the most common social media platforms, such as Instagram, Twitter and Facebook. We aim to keep the posts light-hearted: besides sharing about the latest features, updates and reasons to use our app, we would also like to occasionally post stories and examples of creative ways in which people use our app in real life. The design (i.e. colour palette, theme) of our posts will be aligned with our application for consistency. In addition, we can also leverage on the iced gems as characters to drive our posts. Early adopters of our app would then be able to help expand the usage of our app to their friends.

In the beginning, we can also host social media giveaways. For example, we can encourage users to post their experience of using the app on their Instagram or Facebook story for the chance to win Grab vouchers.

Channel 2: Physical
With most classes moved online, university students in particular would find themselves meeting their friends less often due to differing on-campus lesson schedules. We can therefore direct our physical marketing efforts at universities. Posters can be placed around campus that target this use case. Additionally, we can also distribute stickers to students on campus and even liaise with university/ hall/ residential student welfare unions or committees (e.g. NUSSU) to include these stickers in the biannual welfare packs if possible.

___
Besides spreading awareness, another goal is to keep user retention high. On the technical side, we hope that the usefulness and quality of the application would help in that regard. Nevertheless, we also plan to add some other features to maintain user interest.

Firstly, it would be intended for the app to notify users if a new gem was dropped for them, or if they are in the vicinity of a gem, so this naturally helps to retain users to some extent as they would be reminded as long as they have some friends who are on the app.

Additionally, we would send push notifications to users to check the application if they have not been active for a certain period of time. Users may be prompted to drop a gem, or reminded that there are unopened gems for them, if any. These notifications would be dispersed in appropriate time intervals to prevent users from getting bothered.

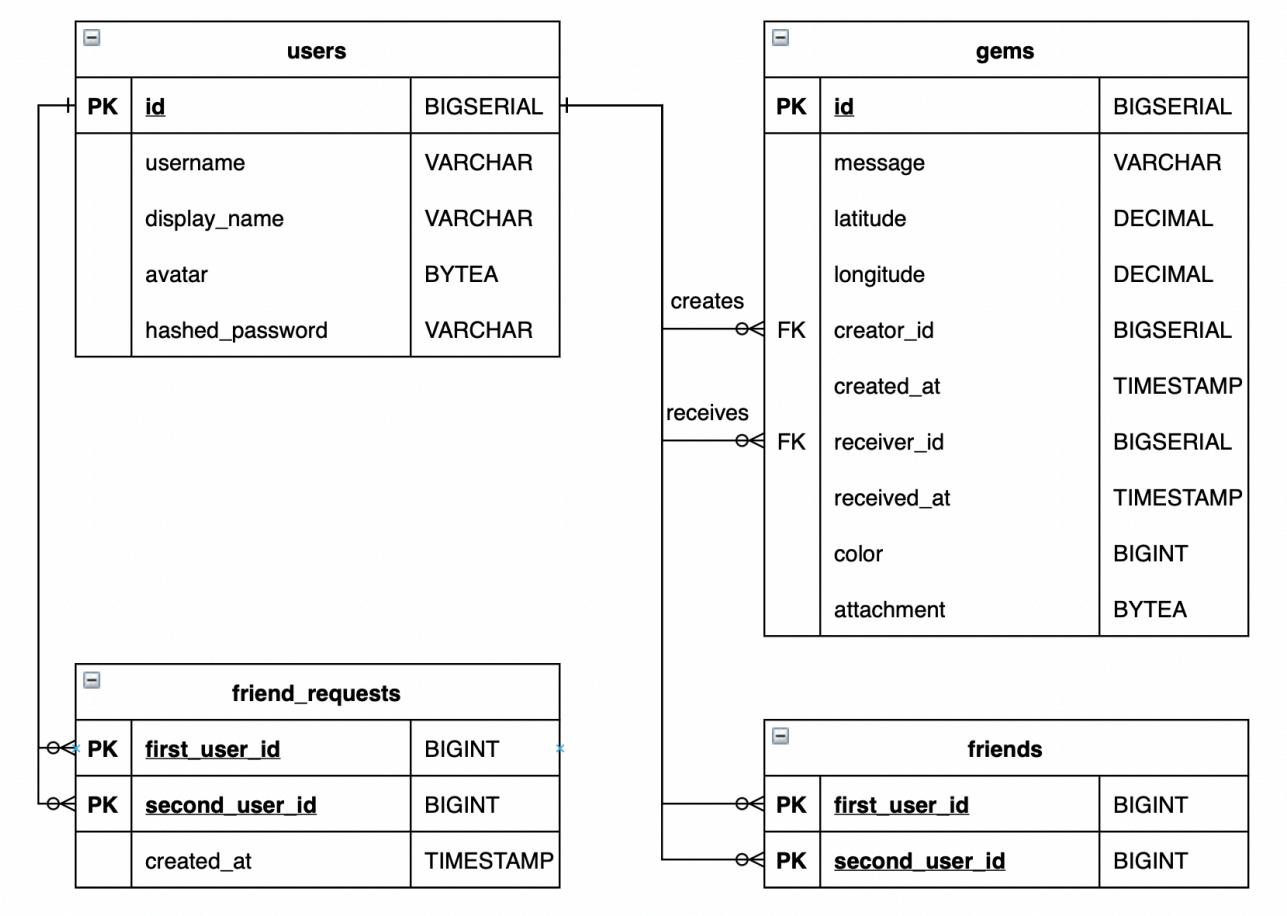
We also consider adding slight gamification to the application: we would release limited edition iced gems that users can add to their collection, which would only be available during certain periods of the year (e.g. Halloween/ Christmas themed iced gems). Special

iced gems can also be unlocked based on achievements, such as after dropping or receiving a certain number of iced gems. We would also like to implement a mini checklist of interesting achievements, such as picking up an iced gem in a different country. These features would also be marketed to drive user interest in trying out the app in addition to our marketing plans. Nevertheless, the extent to which these will be implemented would be limited as we do not wish to detract from the application's main purpose.

Figure 1: Concept poster to be put in campus. Ideally the actual poster would adhere more to the 3D design of our application.

Checkpoint: What is the primary key of the home_faculties table?
The primary key is *matric_no*. The matriculation number for every student is unique, and each student can only have one home faculty. Therefore, there would not be two instances in the *home_faculties* table of the same *matric_no* value, and each row can be uniquely identified by *matric_no*.

Milestone 3: Draw an Entity-Relationship diagram for your database schema.



Milestone 4: Explore one alternative to REST API (may or may not be from the list above). Give a comparison of the chosen alternative against REST (pros and cons, the context of use, etc.). Between REST and your chosen alternative, identify which might be more appropriate for the application you are building for this project. Explain your choice.

Our group considered the use of REST and gRPC:

Pros of gRPC against REST API

1. Smaller payloads. gRPC makes use of protobuf which uses a binary message format that allows programmers to specify a schema for the data. This is much more compact than human-readable formats such as JSON and XML.

2. Speed. As mentioned in point 1, protocol buffers are smaller than JSON. As JSON is textual, its integers and floats can also be slow to encode and decode. Furthermore, JSON is not designed for numbers, and comparing strings in JSON can be slow.

3. Strong typing. While REST APIs can have payloads of any structure, the predominant format is JSON. The process of serialising / deserialising JSON to convert it into the target programming language on both the frontend and the backend introduces the possibility of errors and also adds performance overhead.

4. The .proto file is shared between the server and client so messages and client code can be generated from end to end. gRPC's code generation feature that supports multiple languages removes the need to write boilerplate code and streamlines the development process.

5. Unlike REST API, gRPC provides an abstraction that hides the complexity of communication between the client and server, reducing the communication to just simple method calls.

6. Backward and forward compatibility. The gRPC protocol is designed to support services that change over time. Generally, additions to gRPC services and methods are non-breaking. Non-breaking changes allow existing clients to continue working without changes.

Cons of gRPC against REST API

1. Readability. gRPC is not as human-readable as REST API since data is compressed into binary format.

2. Harder to debug. Protobufs are harder to debug than JSON, as intercepting it to print will be in binary format rather than string. Extra tools will therefore be needed.

3. Less developer support and tools. As it is not as widely used, there is more limited information on best practices and workarounds.

4. Cannot be parsed without knowing the schema in advance.

5. gRPC requires more setup in some aspects: for example, it needs an Envoy proxy to forward gRPC browser requests to the backend server, as it is currently impossible to implement the HTTP/2 gRPC spec in the browser.

With some of these considerations in mind, our group decided to use gRPC as it is generally more performant. We especially considered the advantage of speed and smaller payloads that gRPC has over REST API, since our application supports uploading and receiving media packages. As we believed that we had time for learning and exploration, we were also willing to accept the challenges that we would encounter, such as having lesser developer support and having a steeper learning curve since we were less familiar with gRPC.

Milestone 5 (alternative to REST): Provide equivalent documentation of your client-server communication. For GraphQL and gRPC, you can submit your schema or protobuf with additional descriptions or use any of the tools built on top of that. Also, explain how your design leverages the features of your chosen system. You will be penalised if your design fails to utilise any key features of the chosen system.

We make use of the following terminology for greater clarity:

- *xRequest* - The shape of the payload that is being transmitted from the client to the server when the remote procedure call x is called.
- *xResponse* - The shape of the payload that is being transmitted from the server to the client when the remote procedure call x is called.

Our app has a total of 3 protofiles, which are as follows:

- *auth.proto*, containing AuthService which is the service that handles authentication RPCs
- *user.proto*, containing UserService which handles all user-related RPCs
- *gem.proto*, containing GemService which is the service that handles all RPCs related to gem activity

auth.proto

```
service AuthService {
  rpc Login (LoginRequest) returns (LoginResponse);
  rpc Register (RegisterRequest) returns (RegisterResponse);
}

message LoginRequest {
  string username = 1;
  string password = 2;
}

message LoginResponse {
  string access_token = 1;
}

message RegisterRequest {
  string display_name = 1;
  string username = 2;
  string password = 3;
}

message RegisterResponse {
  int64 user_id = 1;
}
```

gem.proto

```
service GemService {
  /* Drops a gem based on the information specified in
DropRequest. The person who drops the gem is determined by who
sent the request. If the request was successful, the response will
return the ID of the newly dropped gem. */
  rpc Drop (DropRequest) returns (DropResponse);

  /* Retrieves a list of gems that is sent for the user who called
the RPC and has yet to be picked up. The response is used to
display gems pending pickup on the map. */
  rpc GetPendingCollectionForUser (google.protobuf.Empty) returns
(GetPendingCollectionForUserResponse);

  /* Picks up a specific gem. The person who picks up the gem is
determined based on who sent the request. */
  rpc PickUp (PickUpRequest) returns (google.protobuf.Empty);

  /* Retrieves a map of user_id to GemLogsWithFriend, which
contains a list of all gems that the user who called the RPC and
their friend have sent to each other. The response is used to
display all gem logs for all their friends. */
  rpc GetGemLogs (google.protobuf.Empty) returns (GemLogs);
}

enum GemColor {
  PURPLE = 0;
  PINK = 1;
  BLUE = 2;
  BLACK = 3;
  YELLOW = 4;
  GREEN = 5;
}
```

```
message Gem {
  int64 id = 1;
  string message = 2;
  double latitude = 3;
  double longitude = 4;
  usersapi.User creator = 5;
  google.protobuf.Timestamp created_at = 6;
  usersapi.User receiver = 7;

  // Will be `null` for the gems that have not been collected.
  google.protobuf.Timestamp received_at = 8;
  GemColor color = 9;

  /* Photo that the user has attached to the gem. All gems have
one photo and one message. */
  bytes attachment = 10;
}

/* We use GemMessage when certain fields (e.g id) of Gem have not
yet been instantiated. */
message GemMessage {
  string message = 1;
  double latitude = 2;
  double longitude = 3;
  int64 receiver_id = 4;
  GemColor color = 5;
  bytes attachment = 6;
}

message DropRequest {
  GemMessage gem_message = 1;
}

message DropResponse {
  int64 dropped_gem_id = 1;
}

message GetPendingCollectionForUserResponse {
  repeated Gem gems = 1;
}

message PickUpRequest {
  int64 id = 1;
}
```

```
/* A map with keys representing the ID of a friend, and the values
containing data about that friend as well as the gems associated
with that friend. Association is defined as whether the creator
and receiver of the gem are Self and Friend (in either direction)
*/
message GemLogs {
  map<int64, GemLogsWithFriend> gem_logs = 1;
}

/* Stores details about a friend, as well as the friend's
associated gems with the user who sent the request */
message GemLogsWithFriend {
  usersapi.User friend = 1;
  repeated Gem gems = 2;
}
```

user.proto

```
/* This includes RPCs for both the user's own data (Get/Edit
OwnProfile) and relationships with other users (Add/Accept/Decline
FriendRequest) */

service UserService {
  rpc GetOwnProfile (google.protobuf.Empty) returns
(GetOwnProfileResponse);
  rpc EditOwnProfile (EditProfileRequest) returns
(google.protobuf.Empty);
  rpc SearchByUsername (SearchByUsernameRequest) returns
(SearchByUsernameResponse);

  // Retrieves all friends of the user who called the RPC
  rpc GetFriends (google.protobuf.Empty) returns
(GetFriendsResponse);

  /* Retrieves all pending friends of the user who called the RPC
(friend request sent but not accepted) */
  rpc GetPendingFriends (google.protobuf.Empty) returns
(GetPendingFriendsResponse);

  /* Retrieves all pending friend requests of the user who called
this RPC. */
  rpc GetFriendRequests (google.protobuf.Empty) returns
(GetFriendRequestsResponse);
  rpc AddFriendRequest (FriendRequest) returns
(AddFriendRequestResponse);
  rpc AcceptFriendRequest (FriendRequest) returns
(google.protobuf.Empty);
  rpc DeclineFriendRequest (FriendRequest) returns
(google.protobuf.Empty);
}

message User {
  int64 user_id = 1;
  string display_name = 2;
  string username = 3;

  /* Optional. Will be one of our default placeholders when not
set. */
  bytes avatar = 4;
}
```

```
message GetOwnProfileResponse {
  User info = 1;
  repeated User friends = 2;
}

/* Called when user is searching for other users to add as
friends. */
message SearchByUsernameRequest {
  string search_query = 1;
}

/* Returns a list of users with username matching the search query
provided in the request */
message SearchByUsernameResponse {
  repeated User users = 1;
}

message GetFriendsResponse {
  repeated User friends = 1;
}

message GetPendingFriendsResponse {
  repeated User pending_friends = 1;
}

/* A friend request with receiver set as Self (user who called the
RPC) but has not yet been declined or accepted. We include
created_at to be able to display friend requests in the same order
that they come in. */
message PendingFriendRequest {
  User requester = 1;
  google.protobuf.Timestamp created_at = 2;
}

message GetFriendRequestsResponse {
  repeated PendingFriendRequest friend_requests = 1;
}

message FriendRequest {
  int64 requester_id = 1;
  int64 receiver_id = 2;
}
```

```
/* Represents the status of a friend request. Once a friend
request is sent, it will be in the SENT_FRIEND_REQUEST
status. Once the friend request has been accepted, the friend
request will have a ADDED_AS_FRIEND status. */
enum AddFriendRequestStatus {
  SENT_FRIEND_REQUEST = 0;
  ADDED_AS_FRIEND = 1;
}

message AddFriendRequestResponse {
  AddFriendRequestStatus status = 1;
}
  /* We use individual fields instead of User as user_id and
username should not be mutable. new_display_name and new_avatar
are both optional, but cannot be both empty. */
message EditProfileRequest {
  // Optional.
  string new_display_name = 1;
  // Optional.
  bytes new_avatar = 2;
}
```

How our design leverages the features of gRPC

1. Using a byte array to represent image file attachments
Our protobuf design uses a byte array to represent image file attachments. An alternative we considered would be to use a base64-encoded string. However, this introduces unnecessary overhead in encoding and then decoding the bytes array, which is redundant given that gRPC already uses raw bytes to represent data, and only serialises / deserialises data when accessing or setting fields. Given that exchanging photo attachments is a core feature in our app and will happen very frequently, this design decision will have great payoff.

2. Using gRPC's composite type
Our design leverages on gRPC's ability for developers to specify composite types for fields, such as through the use of enumerations. In particular, we used enumerations for the field of gem colour, as the possible values are derived from a limited and predefined list of values. Similarly, we also embedded types corresponding to the domain models (e.g. Gems, Users) in the remote procedure call's response types. This helps to reduce code duplication and also ensures that any changes to the model will be correctly updated for procedure calls utilising this model.

3. Using gRPC's strong typing

Our design fully utilises the strong typing provided by gRPC by defining messages such as the GemColor enumeration. Furthermore, our frontend and backend uses typed languages (Typescript and Golang respectively). This helps to maintain strong typing even as communication is made between the frontend and the backend.

The protobufs serve as a single contract between the server and client. Because of the strong typings, we can be fully assured that when a message is sent or received, we know the structure of the data that is sent or received.

4. Using multiple services
Our design utilises multiple services. Each service is responsible for different aspects of the application (Authentication, Gem, User). This helps to maintain modularity, independence as well as separation of concerns through the creation of one service per domain entity.

5. Backwards Compatibility
Our design leverages on protobufs' guarantee of backwards compatibility by ensuring that new fields can be seamlessly added and removed.

This helps us to develop without fear of breaking existing implementations. Furthermore, we can develop features on the backend, without needing to modify the frontend immediately. This is suitable for our fast-paced development and helps in churning out quick iterations.

Milestone 6: Share with us some queries (at least 3) in your application that require database access. Provide the actual SQL queries you use (if you are using an ORM, find out the underlying query and provide both the ORM query and the underlying SQL query). Explain what the query is supposed to be doing.

GetPendingCollectionByUser

```
SELECT * FROM "gems" WHERE (("receiver_id" = <user_id>) AND
("received_at" IS NULL));
```

This query retrieves all the gems that have not been picked up by the user identified by **<user_id>**, and is used to display the uncollected gems upon loading the map.

GetGemLogs

```
SELECT * FROM "gems" WHERE ("creator_id" = <user_id>) UNION
(SELECT * FROM "gems" WHERE ("receiver_id" = <user_id>);
```

This query retrieves all the gems that are either dropped by the user or are picked up/ pending to be picked up by the user. The user is identified by **<user_id>**.

SearchByUsername

```
SELECT * FROM "users" WHERE ("username" ILIKE '%<search_query>%')
```

This query retrieves all the users that have a username matching **<search_query>.** The matching is case-insensitive. The query is used to search for users to add as friends.

GetFriends

```
SELECT "users".* FROM "users" LEFT OUTER JOIN "friends" ON
("users"."id" = "friends"."first_user_id") WHERE
("friends"."second_user_id" = <user_id>) UNION (SELECT "users".*
FROM "users" LEFT OUTER JOIN "friends" ON ("users"."id" =
"friends"."second_user_id") WHERE ("friends"."first_user_id" =
<user_id>))
```

This query retrieves all the users that are friends of the user identified by **<user_id>.** The query is used to retrieve the lists of friends so that the user can pick one as the receiver of a gem to be dropped.

Milestone 7: Create an attractive icon and splash screen for your application. Try adding your application to the home screen to make sure that they are working properly. Include an image of the icon and a screenshot of the splash screen in your write-up.

Figure 2: our app icon

Our icon is a gem that is styled in the primary colour of our application as it could be immediately recognised and associated with our app. Our splash screen features the same image, but with the app name written below as well to facilitate better name recall in users.

Demo of adding Hansel to home screen:

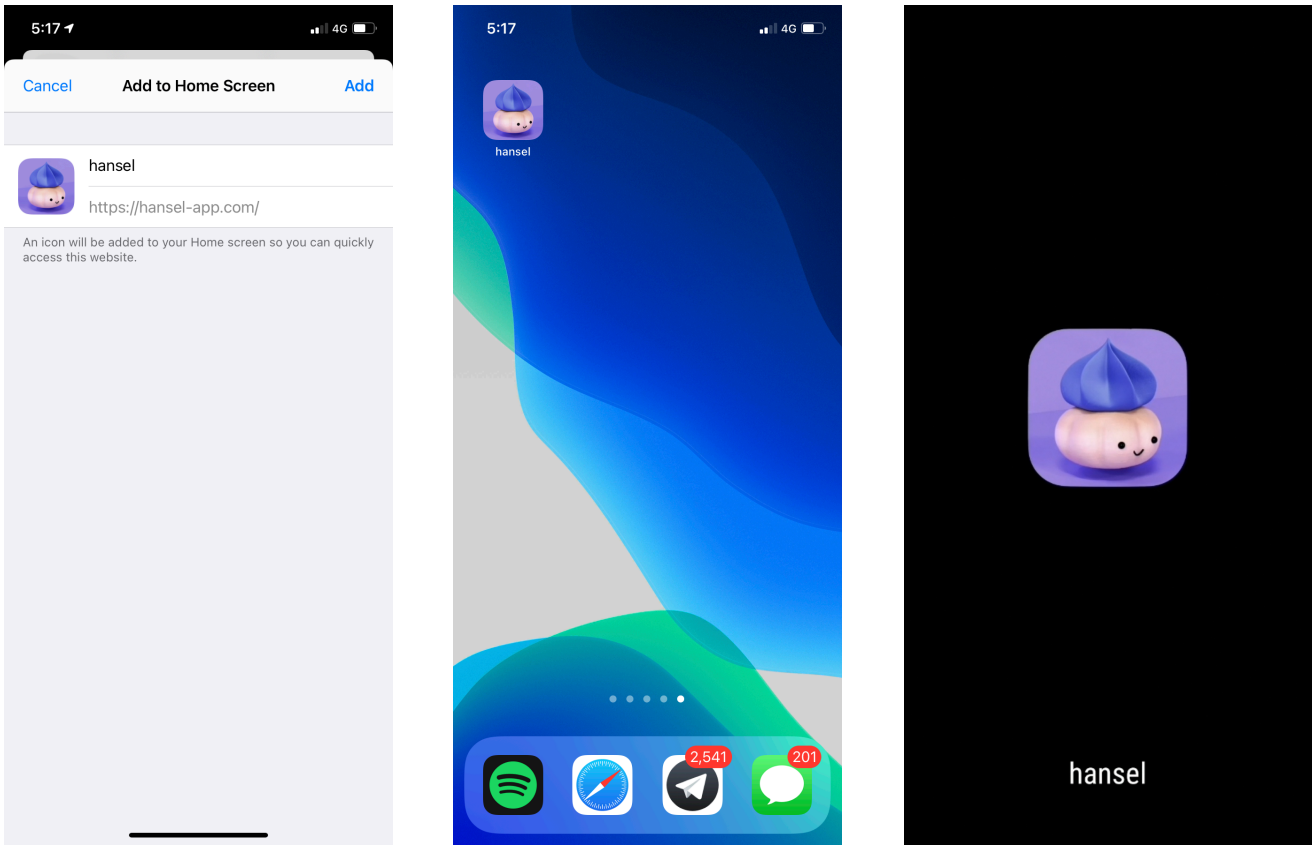


Figure 4: adding app to home screen, and splash screen

Milestone 8: Style different UI components within your application using CSS in a structured way (i.e. marks will be deducted if you submit messy code). Explain why your UI design is the best possible UI for your application. Choose one of the CSS methodologies (or others if you know of them) and implement it in your application. Justify your choice of methodology.

Our application makes use of 3D graphics and animations that are contrasted against the otherwise mostly simple and flat design of other assets. On pages where there are little graphic assets, we also added light gradients to give some depth to the design. As we decided to personify the gems, we believed that 3D design would be best in bringing out the personalities of the different gems through their expressions. The introduction of personified gems to our app helps to add a unique touch to distinguish it from other apps, and makes the user experience of leaving a message for a friend more novel and exciting.

Moreover, 3D graphics have also emerged as a popular and successful trend in recent years and have won the hearts of many UI designers. It has consistently remained as one of the most “influential” graphic design trends by many design websites. Besides providing an illusion of depth, its vibrant colours are also a key distinguishing feature, which makes it ideal for conveying a light mood to the app. Our app remains faithful to this characteristic and leverages on the cheerful pastel colours of the gems. To balance out the pastel colours and add more contrast to the app, we also used strong solid colours (black, white and purple) as the main palette for other elements.

The CSS methodology that we have implemented is object-oriented CSS (OOCSS). The two main principles of OOCSS is to separate structure and skin, and separate container and content.

Separate structure and skin

- We defined reusable CSS style for our backgrounds (.background-gradient / .reverse-background-gradient) and used them for all our pages
- We declared text-styles, font-sizes, font-colors, icon-sizes etc in one central location and used them as variables, instead of hardcoding them in each file
- We standardised text styles for all html typography (h1, h2.., p)

Separate container and content

- We defined and reused a container class to standardise the look of our pages
- We defined and reused a content class to standardise the content for our pages
- We defined classes such as ‘base’ and ‘overlay’ to reuse each time we want to stack a component on top of another

By adopting OOCSS, we benefited by having flexible, modular and swappable components. This allows us to build a consistent-looking application even with multiple developers working on the codebase.

To support the occasional need to override CSS styles, we also utilised Vue’s Single File Components, which allows us to encapsulate specific styles that are scoped to the current component.

Milestone 9: Set up HTTPS for your application, and also redirect users to the https:// version if the user tries to access your site via http://. HTTPS doesn't automatically make your end-to-end communication secure. List 3 best practices for adopting HTTPS for your application.

1. **Use Transport Layer Security (TLS) for all pages.**
   TLS should be used for all pages, and not just those which are considered sensitive such as the login and registration pages. This is because if any page does not enforce the use of TLS, an attacker would be able to sniff sensitive information, such as the access tokens used to authenticate protected remote procedure calls.

   For Hansel, all application data is served over HTTPS. In order to provide users with the best experience possible, we also set up our NGINX reverse proxy to redirect unencrypted HTTP connections to the equivalent HTTPS domain via a permanent redirect (HTTP 301).

2. **Use the "Secure" cookie flag**
   All cookies used should be marked with the "Secure" flag so that the web browser only sends them over encrypted HTTPS connections. This prevents the contents of the cookie from being sniffed from an unencrypted HTTP connection, such as when an attacker mounts an active man in the middle attack by presenting a spoofed web server on port 80.

   For Hansel, we use cookies that are marked with the "Secure" flag to persist the JWT access token across sessions.

3. **Use HTTP Strict Transport Security (HSTS)**
   HSTS instructs the web browser to always request the site over HTTPS and prevents the user from bypassing certificate warnings. The result of this is that when the user types in a HTTP domain in the web browser, the browser automatically connects over HTTPS instead of first connecting over HTTP, then getting redirected by the reverse proxy. This eliminates a lot of opportunities for an attacker to perform a man in the middle attack to intercept the initial HTTP request which is not secure.

   For Hansel, we make use of HSTS by setting the "Strict-Transport-Security" response header in the response from the reverse proxy.

Milestone 10: Implement and briefly describe the offline functionality of your application. Explain why the offline functionality of your application fits users' expectations. Implement and explain how you will keep your client synchronised with the server if your application is being used offline. Elaborate on the cases you have taken into consideration and how they will be handled.

When users are offline, they can view their gem logs, which keep a history of all the gems exchanged between them and their friends. They will not be able to perform any other actions in the application. We feel that such behaviour fits into the expectations of our users since Hansel is primarily a social networking application. As such, it is reasonable that all of its social features require an active Internet connection to work. This is similar to the behaviour of prevalent messaging apps like Telegram, where you can view past messages, but not send or receive any new ones.

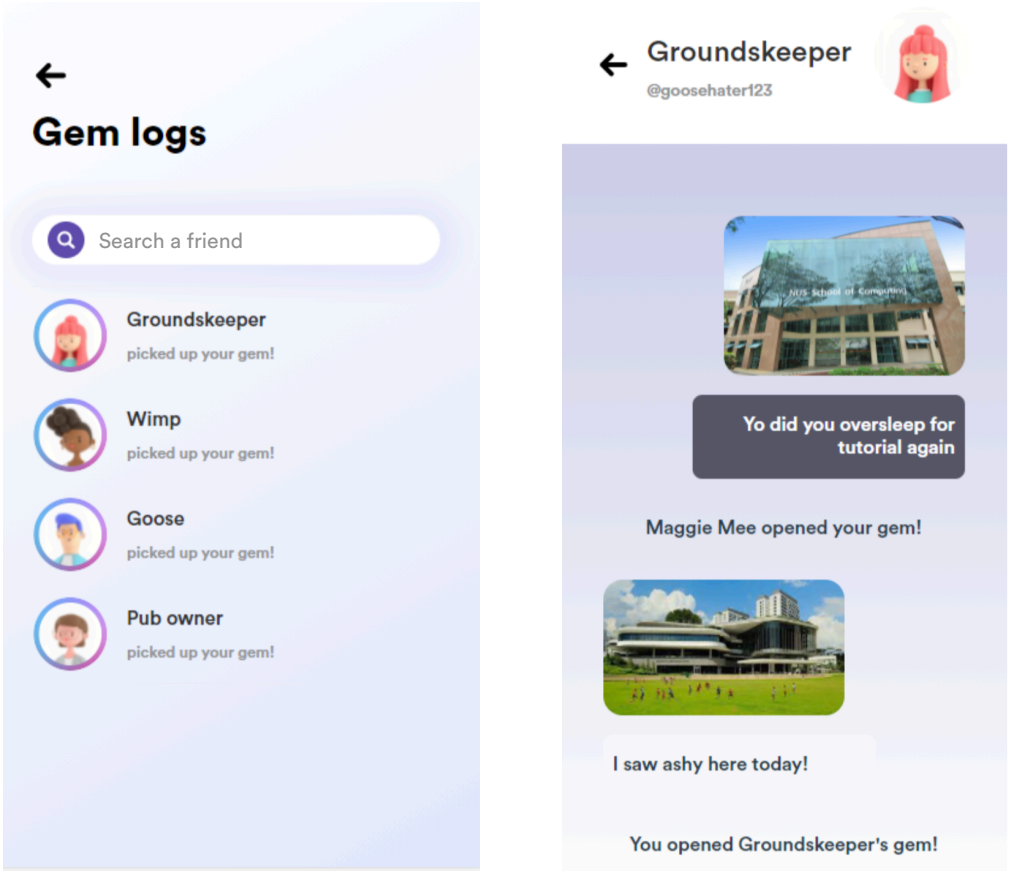


Figure 5: screenshot of gem logs

Milestone 11: Compare the advantages and disadvantages of token-based authentication against session-based authentication. Justify why your choice of authentication scheme is the best for your application.

Token-based authentication is stateless and stores the user state on the client. Upon login, user data is encrypted into a JSON Web Token (JWT) which is then sent to and stored on the client side and sent as a header for every subsequent request. The server validates the JWT through its signature before sending a response back to the client.

On the other hand, in session-based authentication, a session is created for the user upon login by the server, and a cookie containing the session id would be sent with every subsequent request while the user is logged in. This id is compared with the session state stored on the server to verify the identity of the user before the server proceeds with processing the request.

There are many points of consideration when deciding which authentication method to use, and it may often come down to the specific use case or even the developer themselves.

One benefit of token-based authentication over session-based authentication is scalability, as sessions make use of server memory whereas tokens are simply stored on the client side. With session-based authentication, the server would have greater problems handling application load and would require greater expenditure of resources.

With token-based authentication, although we remove the dependency on a server-side session, it also has its own challenges. In terms of security, session-based authentication tends to be safer as one can easily leverage on existing implementations that are secure, well tested and widely used. On the other hand, using a cryptographic token-based authentication, the token would need to be stored and transported securely, which would often rely on the developer to design, implement and deploy well.

We favoured the use of token-based authentication due to its scalability. Since Hansel is built upon the interactions of our users, the application must be able to scale well to support many users. With session-based authentication, it is much harder to scale up the infrastructure horizontally (e.g., adding more machines to the pool of servers) as the session state would need to be synchronised across all servers. Otherwise, connections from the clients cannot be load balanced as the session information would only be present on a single server.

Milestone 12: Justify your choice of framework/ library by comparing it against others. Explain why the one you have chosen best fulfils your needs. Lastly, list down some (at least 5) of the mobile site design principles and which pages/ screens demonstrate them.

Our application uses the library Vant which provides mobile UI components for Vue. Although Vuetify is the most widely used component library, it does not support Vue 3. Looking at the Vant library, the components are flat and minimalistic which contrast well with the louder 3D designs in our application. The components were also largely customisable to fit our overall design, and sufficient documentation was provided on their website for our use.

Rule: Keep Your User In a Single Browser Window
This rule arises because it is much more troublesome to switch between windows on mobile as compared to on desktop, and may risk users not navigating back to the app. Our application keeps its functionality within a single window and does not have any actions that would redirect users to another site or open other browser windows.

Rule: Design your site so users don't have to pinch-to-zoom
We designed our app taking into account that mobile devices have narrower margins, and ensured that all assets are appropriately sized and clearly visible with fonts adhering to a minimum recommended of 16px. It is also for the same reason that we steered away from having displays with multiple columns, as this makes content less readable and users would often have to zoom to see the content of each column properly.

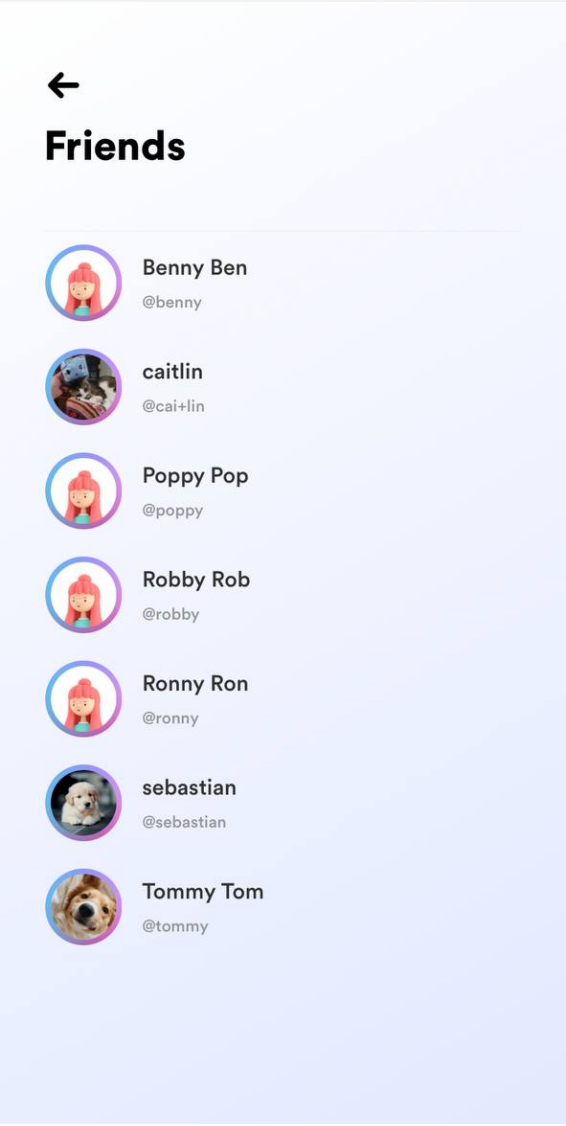


Figure 6: Single column interface that is easily viewable

Rule: Design finger-friendly touch targets
Any site actions, particularly buttons, are sized sufficiently large such that users would be able to click and interact with them easily with their fingers.

Rule: Make it easy to get back to the home page
Although we chose not to implement a bottom navbar with a link to go back to the home page, we purposefully included back buttons or a close button (for drop gem workflow) for all pages besides the home page, which is in line with the UI of most native mobile apps. Additionally, we also implemented swipe gestures for actions such as closing a side menu or opening a bottom drawer. We ensured that the number of times in which users have to navigate back to return to the home page is kept minimal: most pages only require a single back navigation, while a few pages that were slightly more deeply nested would require 2.

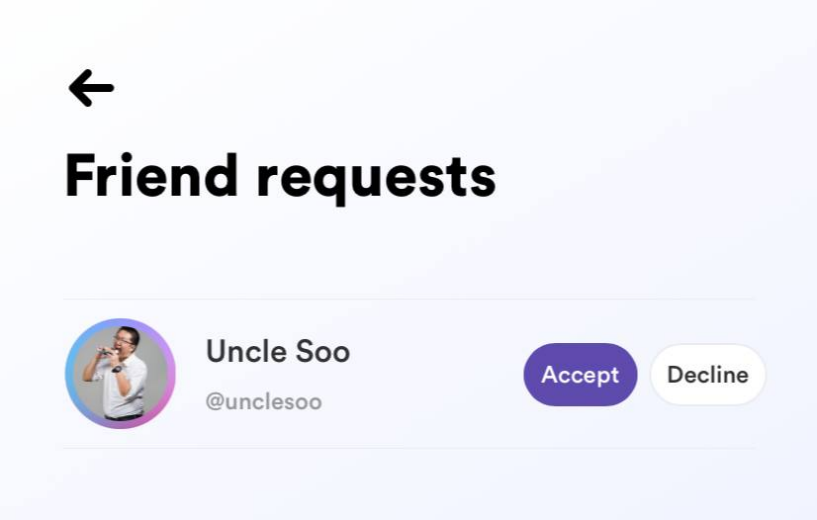


Figure 7: depiction of back button, and sufficiently large buttons

Rule: Keep menus short and sweet
Our app has a hamburger menu in the home page which opens up to a side navigation bar that contains only the 4 essential paths and an option to log out, which makes the app easier and more intuitive to navigate around.

Rule: Calls-to-action front and center
The main feature of our app is the home page: upon entering, the user immediately sees the map and is alerted of how many gems there are, and of the closest gem. The user is therefore immediately informed of any gems left for them, and can begin to retrieve the gems if they wish. The drop gem feature is also in the form of a large button in the home page that is easily accessible to the user.

Milestone 13: Describe 3 common workflows within your application. Explain why those workflows were chosen over alternatives with regards to improving the user's overall experience with your application.

Workflow 1: dropping a gem

Rather than compress the entire process of choosing a user, selecting a gem, choosing an attachment and writing a message into a single page, we split up the process into more manageable components spanning 3 pages. This allows form entry to be easier for users, saving the need to scroll and also reducing the amount of clutter.

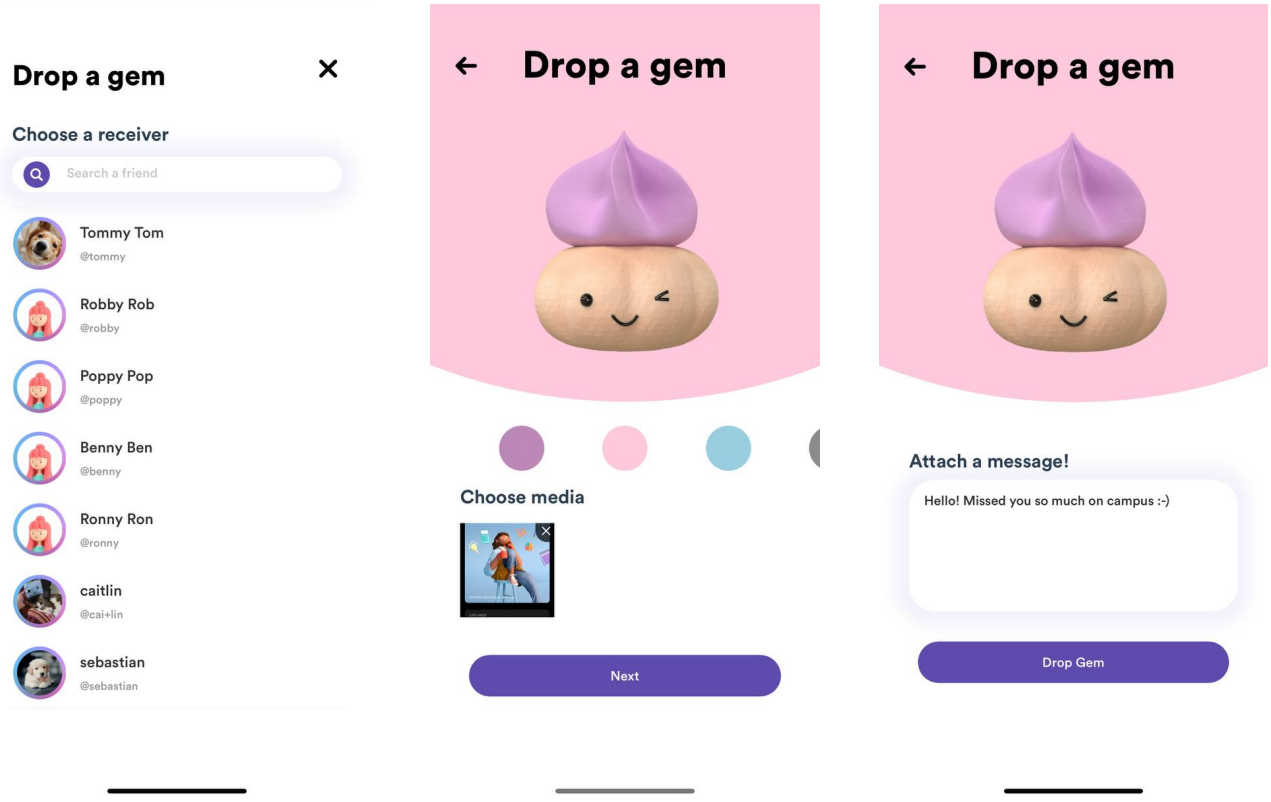


Figure 8: drop gem flow

Workflow 2: picking up a gem

To pick up a gem, users will need to be in the range (50m) of the gem. We marked the radius around the user's current position for their convenience to know when a gem is in range and can be retrieved.

Once within range, users will click on the gem and press the pick up button to confirm their action. This ties in with the UX principle of asking for confirmation for users before

any irreversible action. Adding this button prevents users from accidentally clicking and opening a gem if they do not wish to do so. For example, they might wish not to open the gem at the moment, or even open the gem from a particular friend at all. Implementing a confirmation button would therefore prevent such scenarios from occurring.

Subsequently, they will be greeted with a happy success message and prompted to open the gem to view its contents.

The user can then view the attached image and text. Finally, clicking or swiping the bottom drawer would bring up more details about the gem, such as the sender, location and date of message, as well as a button which would lead the user back to the map. Although the message does obscure the lower portion of the image, we felt that it was an acceptable design choice as we wanted the text and picture to be viewable together as a combined media package so as to prevent the message from feeling disjointed. Additionally, the size of the text box in the “drop gem” workflow is such that users are prompted to add short form text which should not obscure a significant portion of the image. Moreover, users can also view the full image in the gem logs if they wish to do so.

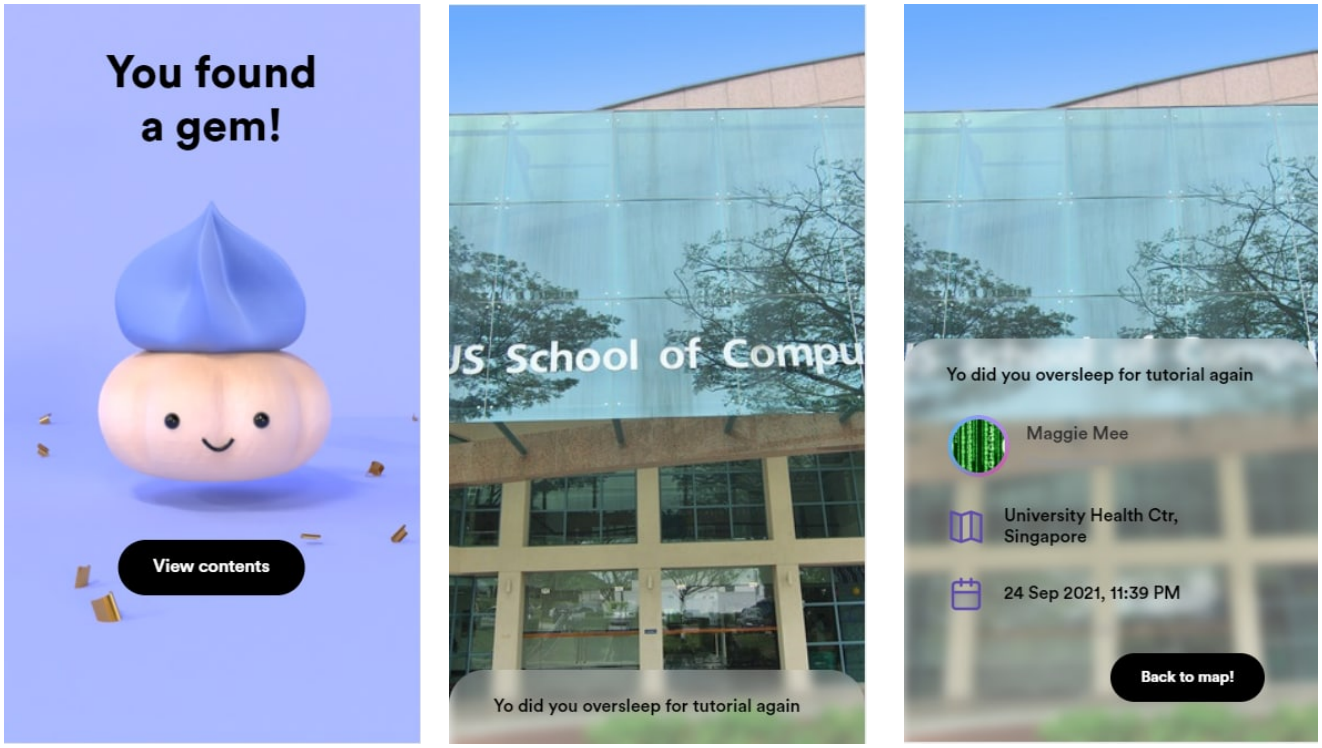


Figure 9: pickup gem flow

Workflow 3: add friends

Friendships in Hansel are two-way relationships. To add a friend, users will proceed to the “Add friends” page where they can search for other users by their username and send them friend requests. The other user is then given the option to either accept or decline the friend request. The add friends page is easily accessible via the side menu as opposed to a hidden away feature to allow user convenience. The other user will then receive a friend request, which can also be accessed via the side menu. A success message upon successful acceptance of a friend helps to provide user feedback.

We decided to go with two-way relationships as opposed to one-way relationships as seen in applications like Instagram or Twitter as the core activity loop of our application is symmetrical between both parties.

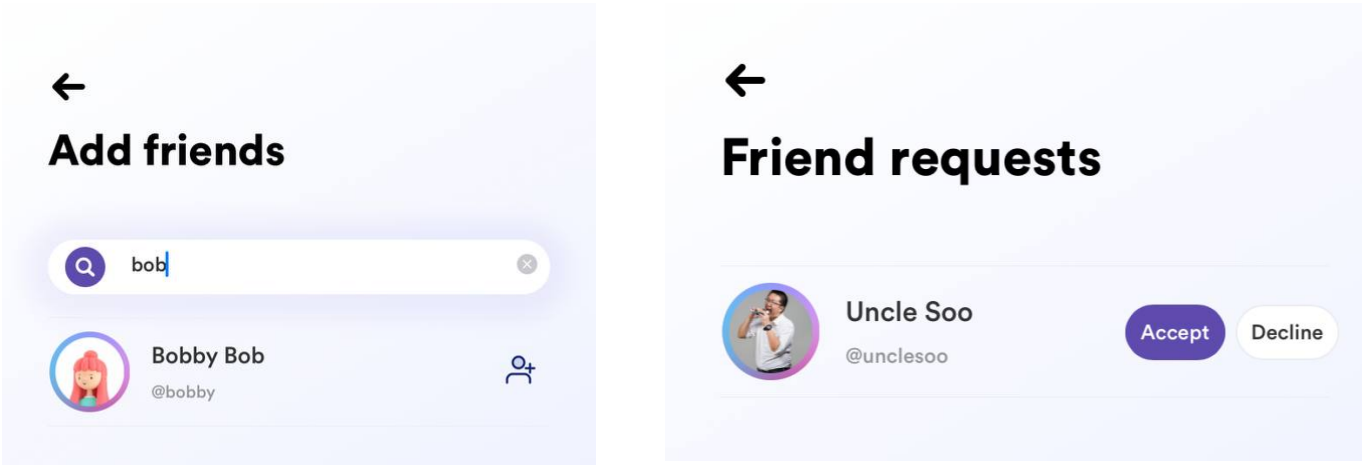
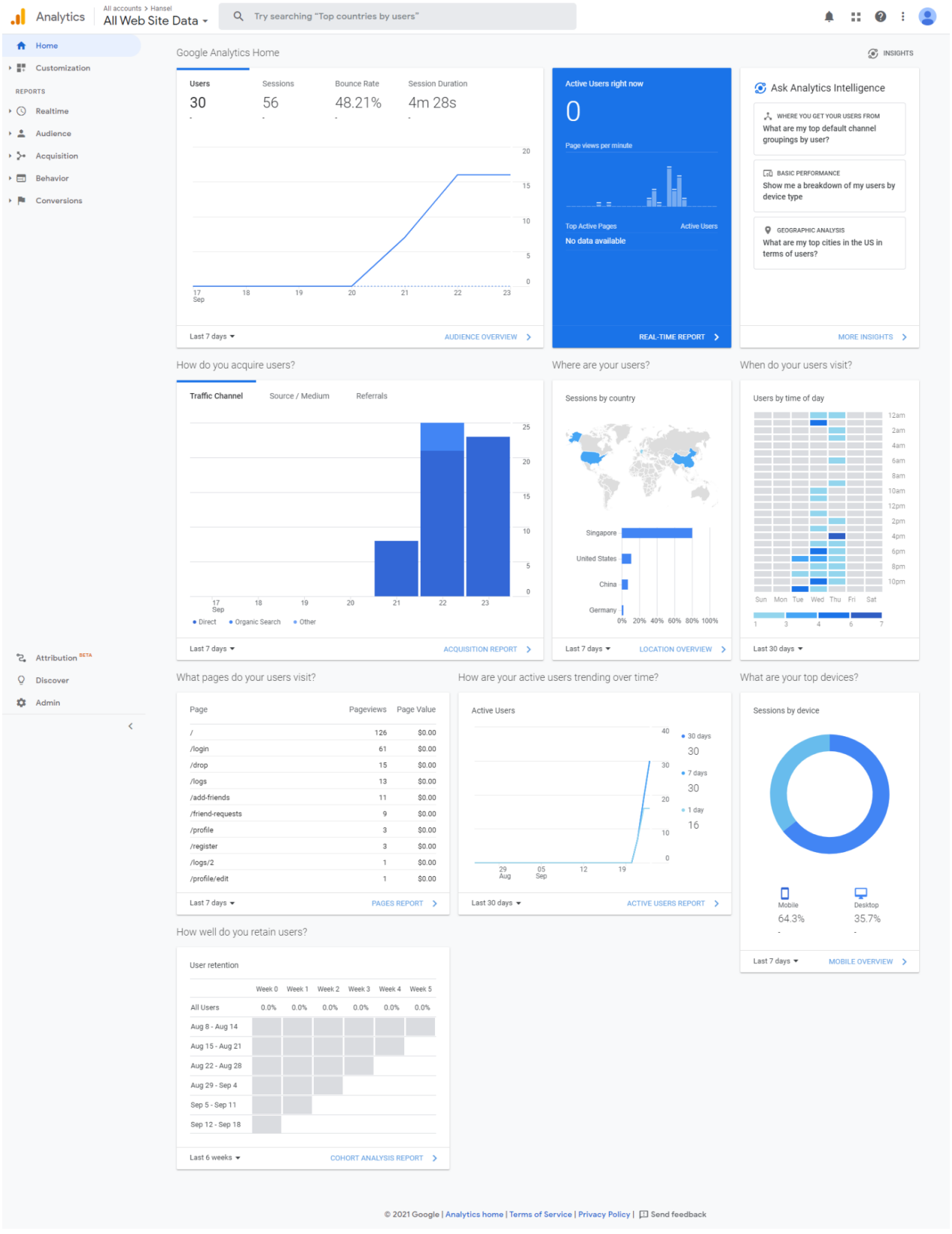


Figure 10: add friends feature, and view requests feature.

Milestone 14: Embed Google Analytics or equivalent alternatives in your application and give us a screenshot of the report. Make sure you embed the tracker at least 48 hours before the submission deadline as updates for Google Analytics are reported once per day.



Milestone 15: Achieve a score of at least 8/9 for the Progressive Web App category on mobile (automated checks only) and include the Lighthouse HTML report in your repository.

We achieved a score of 9/9 for the Progressive Web App category! The Lighthouse HTML report can be found in our Github repository.

Milestone 16: Identify and integrate with social network(s) containing users in your target audience. State the social plugins you have used. Explain your choice of social network(s) and plugins.

We currently did not integrate our app with a social network, however if given more time, we would like to integrate Facebook login into our app. With the integration of Facebook, we can implement a feature where we suggest people that users may know so that they may add them as friends on our app more easily. Additionally, our app can also show which friends of the user might not be in the app yet, and have an option for the user to send an invitation to their friend to join the app.

We think that Facebook integration is the most reasonable as the relationships on Facebook between users are two-way, as compared to one way relationships on other apps such as Instagram (which separates into followers and people you follow).

Milestone 17: Make use of the Geolocation API in your application.

Our application uses the Geolocation API to determine where the user is currently located. This is needed in the application to determine where a gem should be placed, as well as whether the user is in the vicinity of any other dropped gems.

References

Google's Principles of Mobile Site Design:
https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/principles-mobile-site-design-delight-users-drive-conversions/
gRPC information:
https://devblogs.microsoft.com/aspnet/grpc-vs-http-apis/

Resources used:

Avatar profile pictures from VAGO