Wall of Shame

CS3216 Assignment 3 Report

Low Jie Feng - A0201747J
Sieow Je Min - A0167650B
Wang Luo - A0180092L
Yifan Zhang - A0204736H

Table of contents

Milestone 0 - The problem

Procrastination haunts us all; be it in the area of studies, fitness, or even everyday tasks, we often find ourselves pushing things back later and later as we just don't have the motivation to do it now. As our deadlines approach, we find ourselves drowning in responsibilities due to prior procrastination. Stress gets to us and different people have different coping mechanisms - some healthy and others not, with the latter potentially having dire consequences.

Our team wanted to tackle the root problem of procrastination that has personally affected us greatly throughout our university life.

Milestone 1 - Our Application

Wall of Shame is an application that empowers people to curb procrastination by challenging their friends in completing tasks. It is the modern-day, responsible equivalent of betting against your friends to see who can finish a challenge, with the key difference being that instead of betting money, participants place their reputations on the line instead. With such high stakes in the form of public shaming, Wall of Shame is a fun, harmless (mostly) and effective approach to tackling procrastination with your friends.

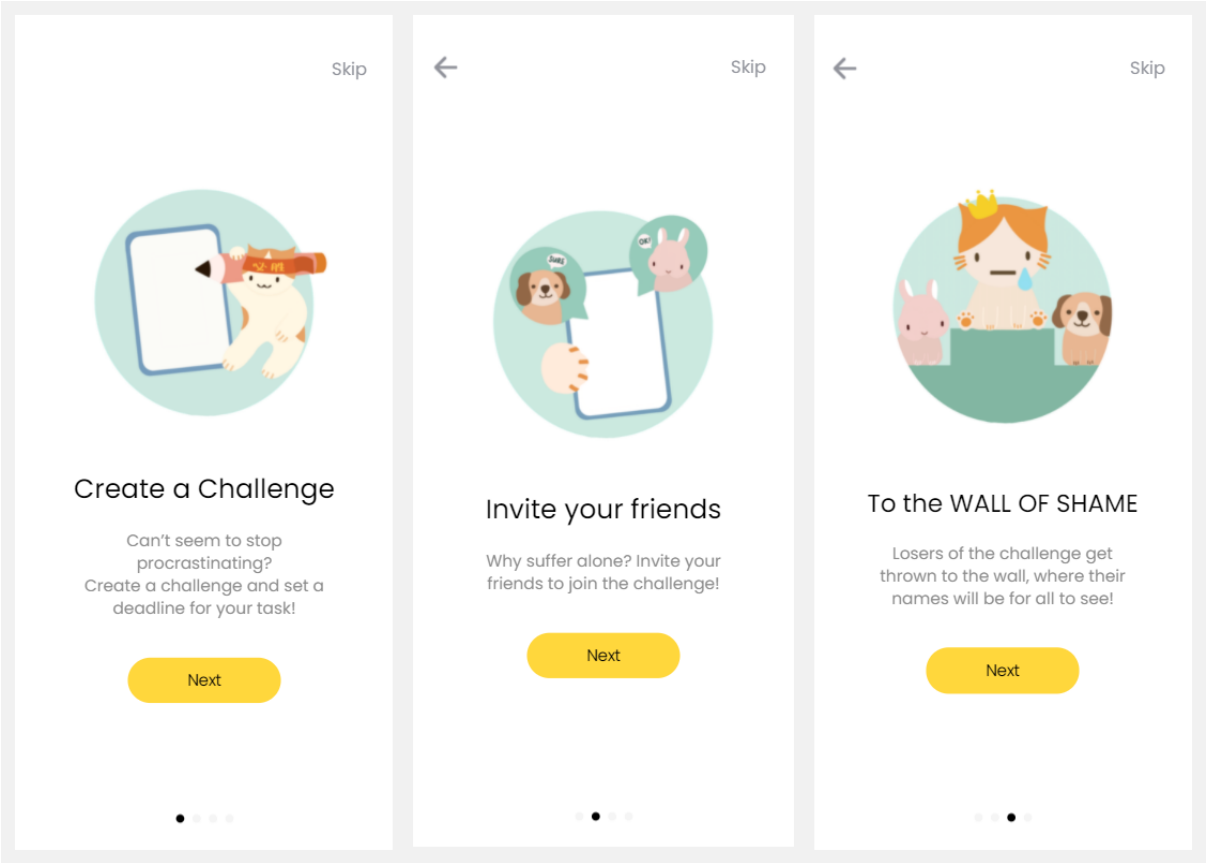


Figure 1.1. An overview of Wall of Shame

Wall of Shame operates in 4 distinct stages:
Creation -> Ongoing -> Resolution -> Conclusion

1. Creation

Users create challenges with a title, description, start time and deadline. The challenge can be anything, from “Watch 10 CS1010 lectures” to “Do the laundry”. Challenge creators can then invite other registered users to join the challenge directly on the app. Invited users will be prompted with a push notification, informing them of this new challenge.

2. Ongoing

Upon starting the challenge, participants will then be shown a countdown timer, which is essentially the time remaining for them to complete the challenge.

When participants have successfully completed the challenge before the deadline, they will be prompted to upload proof. This is so that other users will not be suspicious of them, and will not vote them out as cheaters (more to this later).

3. Resolution

Upon reaching the deadline, participants who were unable to complete the challenge in time will be sent straight to the Wall of Shame when the owner releases the results, where their names will be ‘proudly’ displayed alongside the challenge they have failed. This list of failures will be displayed publicly to all users of our app, including a leaderboard section, keeping track of users who have failed the most.

Participants of the challenge can also view each other’s proofs of completion, ensuring that they have indeed completed the task. In the case where a participant is suspicious due to the lack of or irrelevance of proof, participants can vote to banish that person to the Wall of Shame.

4. Conclusion

Once the owner decides that the challenge can be concluded, the owner will officially release the results of the challenge, including the voting results. Users will be able to view a history of all their challenges in the Profile section, in remembrance of their proud victories and shameful failures.

Why a Mobile Cloud Application?

Wall of Shame draws on some key benefits of Mobile Cloud Applications to accomplish its goals of empowering friends to tackle procrastination together.

1. Accessibility

With a mobile cloud application, getting users onboard will be a lot easier as there will be less inertia compared to having them download a native mobile application. With Wall of Shame being an app revolving around groups of friends, it makes the most sense to have a PWA, reducing the barrier of entry to a bare minimum of typing the app URL in their phone browsers.

The accessibility factor of PWAs will hence allow us to effectively utilise network effects, as the growth of our app is directly linked to the ease of adoption.

2. Availability

Mobile phones may come and go but procrastination will always be here to stay. We want users to be able to use Wall of Shame anywhere on any phone, regardless of OS or model. No room for excuses such as “Oh, I want to switch to an Android from iPhone and the app isn’t there”. Wall of Shame will follow users everywhere, shaming anyone who dares to procrastinate.

3. Low hardware requirements

Having mobile devices with low tech-specs should not stop anyone from achieving their goal to stop procrastination. With high storage data such as images (and videos in future iterations) in the application, it makes the most sense to store these data in the cloud, ensuring that mobile devices do not need large processing power or storage capacity to use Wall of Shame.

Milestone 2 - Target users and promotion

Target users

For now, our target users will be university students in Singapore who have issues with procrastination. We believe that this demographic will find the most value from Wall of Shame because of the following assumptions:

1. Highly prone to procrastination

- With the freedom and independence to plan their schedules in terms of work and play, university students can easily find themselves playing too much at the start, and studying last minute due to the lack of consistency in studying.
- With the need being more prominent in this demographic, users will be more willing to try our app.

2. Most socially active demographic

- There are many avenues to make friends and to be in a community in university, from halls and residential colleges to school CCAs and interest groups.
- University students don't have the responsibilities of adulthood such as bills and family, and have more time to spend with their friends.
- Users will have more opportunity to use the app, as they have more groups of friends.
- We'll be able to leverage network effects to reach a greater number of users.

3. Appreciates deprecating humour

- We think that university students are the group that will be most receptive to the kind of deprecating humour we are pushing for.
- Being the demographic that "suans" (verbally abuse in a friendly way) each other the most, the concept of displaying names of participants who failed could be worth sharing to others, and potentially increase the number of users due to the novel factor.

The figure below shows the persona of our target user.

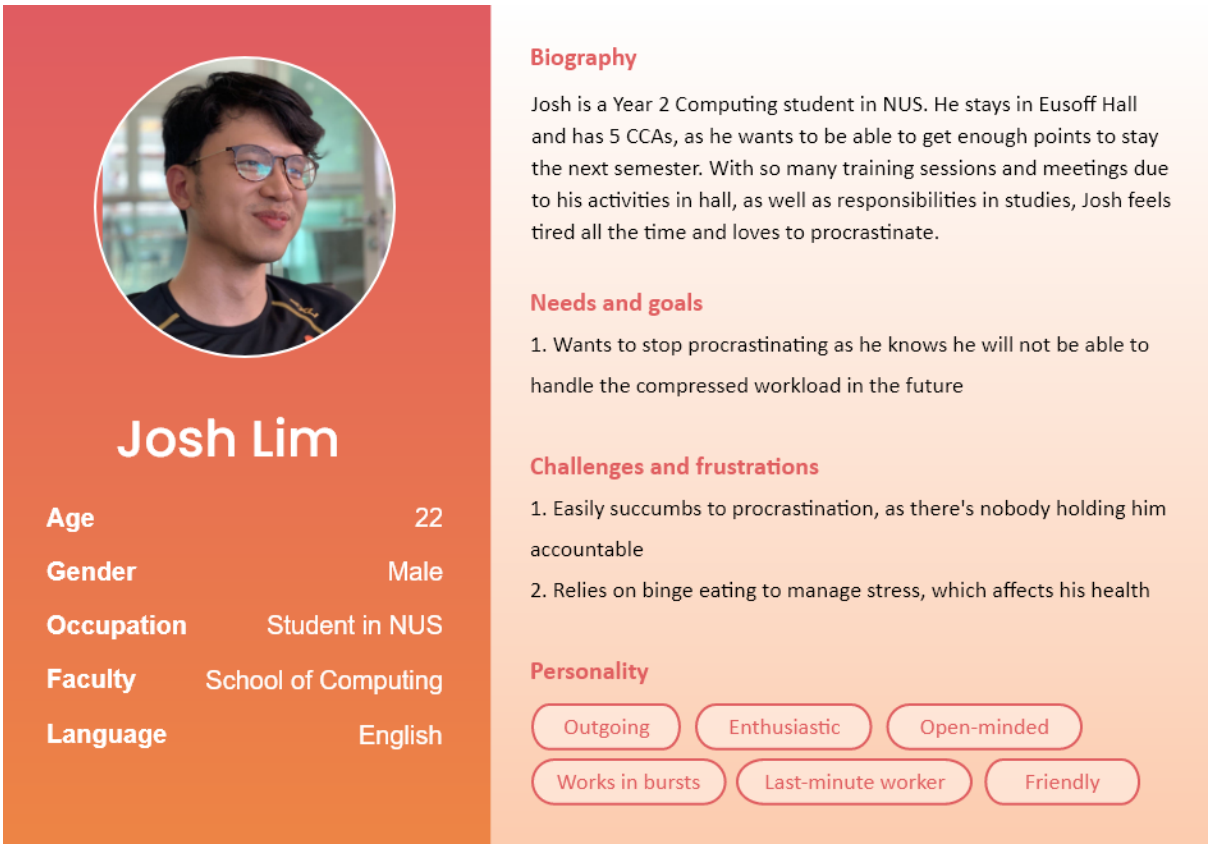


Figure 2.1. Persona card of our target user

That being said, as long as a person has a group of friends, with a common goal to stop procrastinating and get things done, they will also find value in using Wall of Shame.

Promotion plan

1. Social networks

We plan to utilise social networks such as Instagram and Facebook to generate interest in our app. As our target demographic uses these platforms extensively, we'll be able to reach them through funny posts and stories featuring the Wall of Shame.

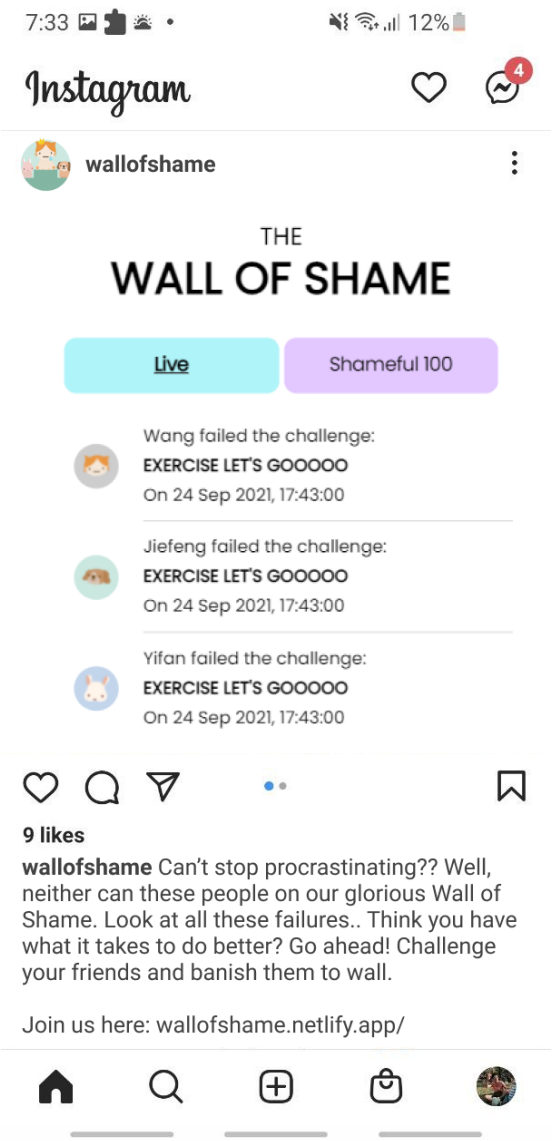


Figure 2.2. Instagram post promoting the Wall of Shame

Furthermore, due to the novel factor of the live updated wall of shame, users are likely to post Instagram stories, or share posts of their friends being displayed on the wall as well, generating further buzz and attention. Instagram stories are one of the quickest ways to achieve quick exposure, and when people see their friends on the app, there will be a high chance of them being converted into our users as well.

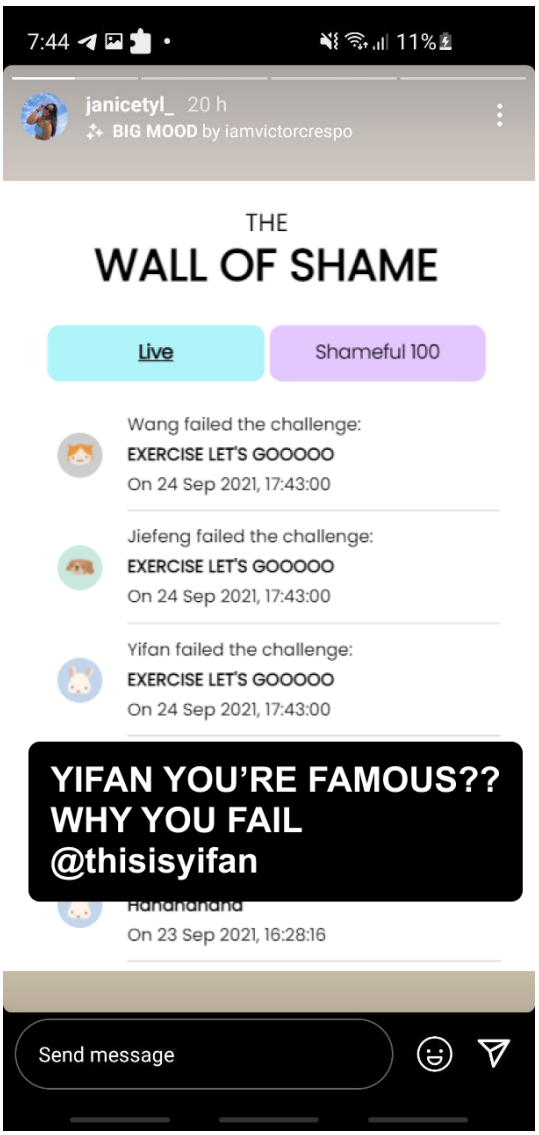


Figure 2.2. Instagram story featuring their friends on the Wall of Shame

2. Messaging apps

Messaging apps such as WhatsApp and Telegram are widely used across our target users. With these apps having capabilities for media such as GIFs and stickers, we plan to create funny sticker packs and GIFs that could be shared between friends, therefore spreading the word of our app. The advantage of using messaging apps is that groups of friends normally have their own chat groups. By leveraging network effects, we can tap into these groups, having them challenge each other as well.

Milestone 3 - ER diagram

The figure below shows the Entity-Relationship diagram for our database schema.

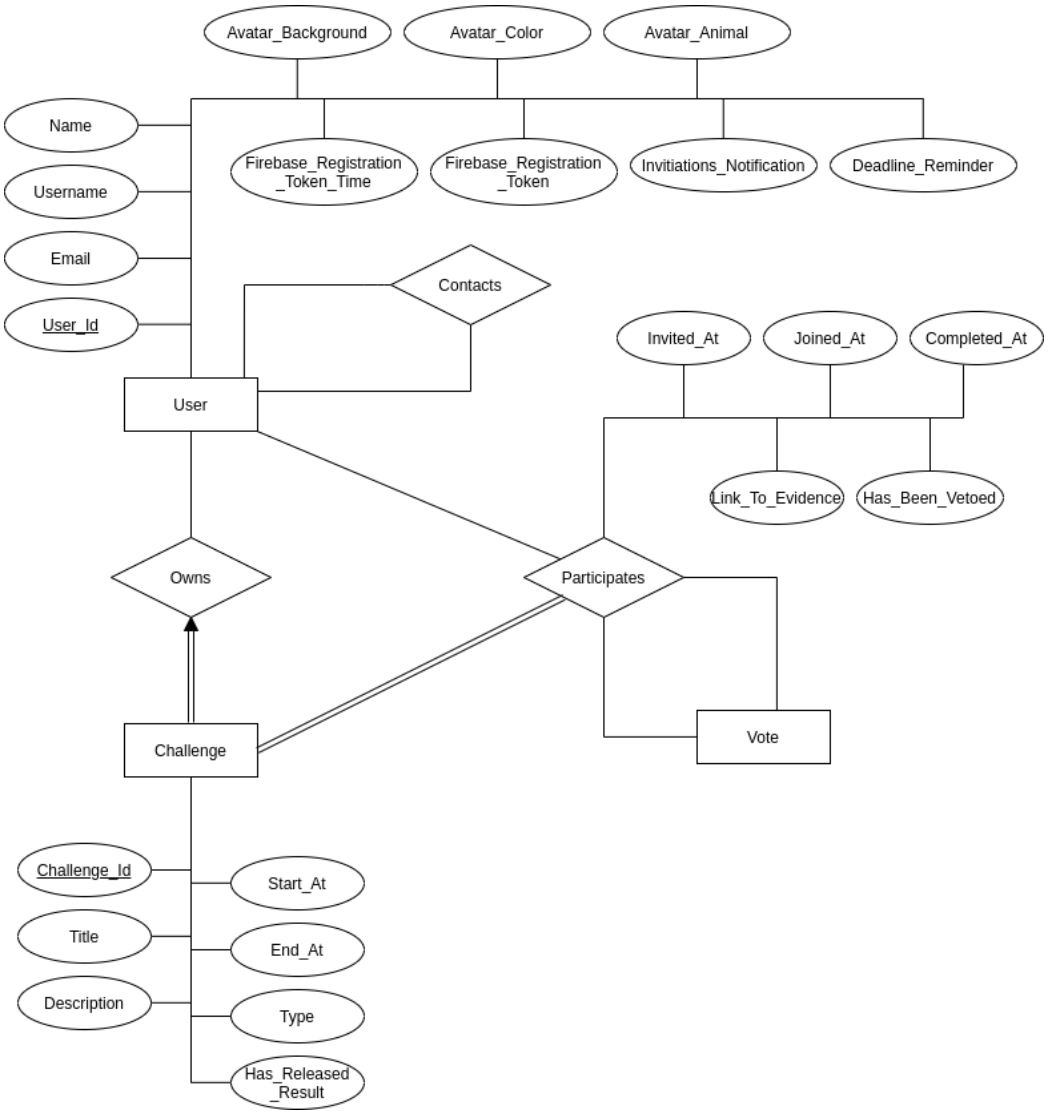


Figure 3.1: Entity-Relationship diagram

Milestone 4 - Alternative to REST API - GraphQL

We decided to explore GraphQL as an alternative to REST API.

Advantage of GraphQL - Specific data fetching

GraphQL can use its query language to create specific requests to get exactly what the developer wants. This can range from getting multiple objects to fields of each entity. With a single endpoint, GraphQL is able to fetch many different things if instructed to do so.

REST API, on the other hand, is unable to do this, and the developer will always get back the complete dataset from the endpoint. Every endpoint has its specific purpose, and developers are unable to manipulate the data to get tailored requests. This can result in multiple roundtrips with REST, where more and more information is fetched from different endpoints to accomplish the task.

GraphQL is superior in this regard, as the developer can fetch the exact data that he/she requires, and not the entire dataset as is the case of REST API (known as over-fetching). This results in savings in terms of cost, especially if the application is used by many users.

Disadvantage of GraphQL - Infancy stage

GraphQL was publicly released by Facebook in 2015. Although it's starting to get attention from the developer community, it is still relatively new compared to REST API, which has been around since 2000. As a result, GraphQL may not have as many tools available compared to REST API, an example being API analytics.

Furthermore, REST API has become an industry standard for many companies, and is used widely for different purposes such as the microservices architecture. According to the State of API 2020 Report, 82% of surveyed API practitioners and consumers use REST API, whereas only 19% use GraphQL.

Although this may not be a disadvantage of GraphQL in terms of technology, it is a disadvantage in terms of application and adoption.

Context of use

The following table shows some scenarios where GraphQL could be preferred over REST API.

Apps where nested data need to be fetched easily	• Utilising GraphQL would make fetching nested data easy with a single call. • An example of such an application would be Facebook, where posts need to be fetched alongside comments, and nested comments and information about the user commenting on the post.
Apps that use the composite pattern	• When apps retrieve data from multiple, different storage APIs, GraphQL is useful for fetching data from these different sources. • An example would be a dashboard that fetches data from logging services, third-party analytics tools, and the usual backend server.
Apps which are used on smaller devices such as smartwatches and IoT devices	• Due to the lower bandwidth usage of GraphQL requests, it is preferred when used by apps for smaller devices which may not be able to handle large amounts of data.

Why did we choose REST API?

Although GraphQL has many benefits as seen above, we believe that REST API is more appropriate for building Wall of Shame compared to GraphQL.

The data of Wall of Shame is simple in terms of structure and design, and has no requirement to get different fields from different sources quickly. Hence, there is no need for the added complexity of GraphQL to fetch specific data, or data from multiple sources from a single request. REST API is simple and straightforward, and is effective in fetching data for our app.

REST API has more capability for error handling and tooling compared to GraphQL. REST APIs follow the HTTP spec, and return various HTTP statuses for different request states. However, GraphQL returns the 200 status for every API request, including errors. Additional libraries will be needed to handle errors, which results in additional layers of complexity. Due to the short time we have to develop Wall of Shame, error handling is essential for troubleshooting and quickly fixing bugs, which REST API is reliable in providing.

Milestone 5 - REST API design and documentation

The documentation for our API can be found here on Apiary. We used Firebase as an Authentication service to make it simpler to integrate with social networks and to ensure user verification and security. We have also integrated with Firebase Realtime Database and Cloud Messaging for real-time communication. Details on our integration of the real-time database can be found here. Below is an explanation on how the API conforms to REST principles[1].

Uniform interface

All API requests for the same resource look largely the same regardless of where they are from. For example, the Challenge resource has various attributes such as title, description, type and participants. The routes exposing this resource, be it a GET, POST or PATCH request, take in and return the challenge in roughly the same shape.

Client-server decoupling

The client and server are independent of each other. The codebase for our application puts the client into one repository, and the API server into a separate repository. They communicate via the API interfaces.

Statelessness

Each request contains all the necessary information to process it. Each request contains the information about the client calling it.

Cacheability

The responses are cacheable whenever the request method, status code and headers are set properly. The cacheable responses are generally the GET requests which contain the ETag header.

Layered system architecture

The API server primarily comprises an ExpressJS application that handles the routing, an ORM layer that makes it easier to handle the data, and an underlying database layer.

[1] https://www.ibm.com/cloud/learn/rest-apis

Milestone 6 – SQL queries

Query 1

We used Prisma as an ORM layer in our backend. Below is one of the Prisma ORM query we used and its underlying SQL queries.

```
const challenge = await prisma.challenge.findUnique({
     where: { challengeId },
     include: {
       votes: {
         select: {
           victimId: true,
           accuserId: true,
         },
       },
       participants: {
         include: {
           user: true,
         },
       },
     },
   });
```

```
SELECT "Challenge"."challengeId",
       "Challenge"."title",
       "Challenge"."description",
       "Challenge"."startAt",
       "Challenge"."endAt",
       "Challenge"."type",
       "Challenge"."ownerId",
       "Challenge"."has_released_result"
FROM   "Challenge"
WHERE  "Challenge"."challengeId" = $1 limit $2 offset $3;

SELECT "Vote"."challengeId",
       "Vote"."victimId",
       "Vote"."accuserId"
FROM   "Vote"
WHERE  "Vote"."challengeId" IN ($1) offset $2;

SELECT "Participant"."challengeId",
       "Participant"."userId",
       "Participant"."invited_at",
       "Participant"."joined_at",
       "Participant"."completed_at",
       "Participant"."evidence_link",
       "Participant"."has_been_vetoed"
FROM   "Participant"
WHERE  "Participant"."challengeId" IN ($1) offset $2;
```

```
SELECT "User"."userId",
       "User"."email",
       "User"."fb_reg_token",
       "User"."fb_reg_token_time",
       "User"."username",
       "User"."name",
       "User"."cfg_deadline_reminder",
       "User"."cfg_invites_notif",
       "User"."avatar_animal",
       "User"."avatar_color",
       "User"."avatar_bg"
FROM   "User"
WHERE  "User"."userId" IN ($1,$2) offset $3;
```

Explanation

This query looks for one challenge entity based on a given challengeId. It returns the challenge itself, a list of all votes given under it and a list of participants it has (a participant entity only has the userId, so it needs to further include the actual underlying user). The ORM layer converts it into 4 queries, which searches for the challenge, its associated votes and participants (and its user) entities.

Query 2

In the server, we found that to represent a user, we also need to consider the number of completed challenges, failed challenges and number of times they have been voted as a cheater. To do this easily, we added a custom view that does this computation, outside of the ORM layer. This allows us to easily find a user and get their participation scores.

```
DROP VIEW IF EXISTS "ParticipationStats";

CREATE VIEW "ParticipationStats" AS
WITH
participanttblextended AS (
   SELECT
      *
   FROM
      "Participant" p
      LEFT JOIN
         "Challenge" c
         ON p."challengeId" = c."challengeId"
   WHERE
      c."endAt" <= Now()
      AND joined_at IS NOT NULL
)
,
usercount AS
(
   SELECT
```

```
      *,
      (
         SELECT
            count(*)
         FROM
            participanttblextended t1
         WHERE
            t1."userId" = u."userId"
            AND t1."completed_at" IS NULL
      )
      AS failedcount,
      (
         SELECT
            count(*)
         FROM
            participanttblextended t1
         WHERE
            t1."userId" = u."userId"
            AND t1."completed_at" IS NOT NULL
            AND NOT t1."has_been_vetoed"
      )
      AS completecount,
      (
         SELECT
            count(*)
         FROM
            participanttblextended t1
         WHERE
            t1."userId" = u."userId"
            AND t1."completed_at" IS NOT NULL
            AND t1."has_been_vetoed"
      )
      AS vetoedcount
   FROM
      "User" u
)
SELECT
   *,
   failedcount + vetoedcount AS totalfailedcount
FROM
   usercount u
ORDER BY
   totalfield DESC ;
```

Explanation

This query creates a database view, which shows all users, and a count of the number of challenges that have failed, completed and voted as a cheater. It first considers all valid participant records for challenges that have already passed. Then, for each user, it counts the number of challenges that they have failed, completed and voted as a cheater. Lastly, it outputs the table and sorts it based on the number of times they have been shamed (i.e. number of times they failed and have been voted as a cheater) in descending order.

Query 3

The following query shows the view being queried in the server and its equivalent SQL query.

```
const raw = await prisma.$queryRaw<
    Array<
      Required<User> & {
        failedcount: number;
        completecount: number;
        vetoedcount: number;
        totalfailedcount: number;
      }
    >
  >`
    SELECT *
      FROM "ParticipationStats"
      WHERE "username" IS NOT NULL
        AND "name" IS NOT NULL
        AND "avatar_animal" IS NOT NULL
        AND "avatar_color" IS NOT NULL
        AND "avatar_bg" IS NOT NULL
        AND "totalfailedcount" > 0
      ORDER BY totalfailedcount DESC, username ASC
      LIMIT 100
  `;
```

```
SELECT *
FROM   "ParticipationStats"
WHERE  "username" IS NOT NULL
       AND "name" IS NOT NULL
       AND "avatar_animal" IS NOT NULL
       AND "avatar_color" IS NOT NULL
       AND "avatar_bg" IS NOT NULL
       AND "totalfailedcount" > 0
ORDER  BY totalfailedcount DESC,
          username ASC
LIMIT  100;
```

Explanation

This query returns the top 100 valid users (i.e. with username, name and avatar) with the highest failedcount (i.e. did not complete their challenges and/or was voted as a cheater the most), ordered by the failedcount and username. Users who have not failed any challenge will not be shown.

Milestone 7 - App icon

For the icon of our application, we used a cat, which is one of the theme animals that appears in our app. We presented it with a sad face and a crown, to illustrate the concept of our app - winning by losing (your friends). For our splash screen design, we reused the app icon with a succinct tagline attached below.

Figure 7.1: Home Screen Icon & Splash Screen Design (in Light Mode)

Milestone 8 - UI Design & CSS Methodology

For our UI design, it mainly revolves around the three principles that we have set on: (1) vibrant accent color scheme, (2) custom assets, and (3) simplistic layout and visual flow.

Since our main target users are young adults, and the fact that the app will mostly be used in a casual setting, we decided on a vibrant bright accent color scheme to add a playful touch to our application. The bright colors are mainly used as accents and a palette for our custom assets, with the primary color kept to a plain white (or black for dark mode). This provides our application with a clean look that upholds a non-distracting visual style.

We created fully customized assets and integrated them to most steps of the main workflows, so that the aesthetics can be accurately conveyed and integrated throughout the application. Each user can select an animal of their choice, and this animal will represent the user throughout his/her journey of smashing procrastination. Our assets and UI are designed such that in the different stages of a challenge, that user's exact animal will be shown if the user is in the active status during that stage. This further gamifies our application and adds a high level of personality.

We also went with a simplistic and native style of layout and visual flow, as the real-estate on mobile screens is limited and we would like to prevent the users from feeling overwhelmed. A close-to-native design also reduces the friction when users onboard onto our app.

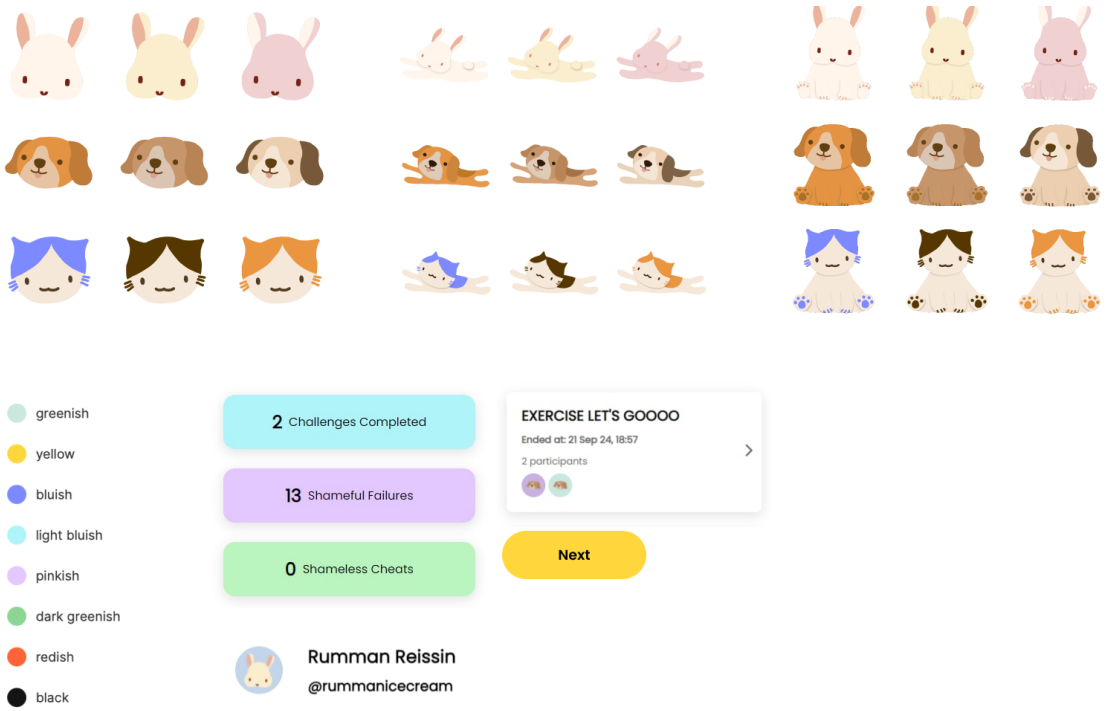


Figure 8.1 Design assets

For styling of our application, we used a combination of Component-based CSS Architecture Style and inline CSS style when necessary. Within the CSS for each styling component, we followed the principles of SMACSS.

We decided to use Component-based Architecture for CSS because of the better isolation and abstraction it can bring. One of the common problems with the traditional pure CSS architectures is the accidental overwrites of mixins. By using a component-based architecture, we allow the styles to be packed with the component with better isolation provided, thus minimizing accidental overwrites and providing better organisation. Moreover, component-based architecture avoids the possibility of redundantly-long CSS names that may occur in traditional CSS frameworks such as BEM or pure Object-Oriented CSS. Component-based CSS architecture is also beneficial for components that are atomic in nature, as it avoids unnecessary grouping of stylings that only serves a single purpose. We also chose to use inline CSS when it is necessary, because we have components that involve animations that rely on states and dynamic variables. Following the principles of SMACSS within the CSS for each styling component allows us to have better organization within the file.

Milestone 9 - HTTPS best practices

We served the frontend client on Netlify, which offers free HTTPS on all sites, including automatic certificate creation and renewal. It sends requests to our backend served on HTTPS as well, which is hosted by Heroku.

Below are the 3 best practices for adopting HTTPS for an application:

Use robust security certificates

When enabling HTTPS for a site, a security certificate is required. This certification should be issued by a valid certificate authority (CA) which actually verifies that the web address does indeed belong to the owner. When setting the certificate, ensure a high level of security by choosing a 2048-bit key.

The site is served through Netlify. The HTTPS certificate is signed and issued to *.netlify.app.

Support HTTP Strict Transport Security (HSTS)

HSTS allows for web servers to declare that web browsers should automatically interact with it using only HTTPS connection and not HTTP connection. Browsers are told to request HTTPS pages automatically, even if the user enters HTTP in the location bar.

This has been handled by Netlify as well. Trying to reach the site through HTTP (e.g. via the browser or curl) will return 301, and redirect the user to the HTTPS version instead[2].

No mixed content

The HTTPS page should not fetch content from HTTP sites. HTTPS sites that allow for HTTP content to be fetched are called mixed content pages, and are only partially encrypted, leaving unencrypted content unsecure.

Our site is served through Netlify, which serves all the assets through HTTPS. It fetches data from the backend server, which is also hosted on a HTTPS site managed by Heroku.

[2] https://www.globalsign.com/en/blog/what-is-hsts-and-how-do-i-use-it

Milestone 10 - Offline functionality

In offline mode, the application shows the user a list of challenges that are currently ongoing or pending, as long as the data of challenges has been fetched before. They can still see what tasks they need to complete as well as time remaining till the deadline. When the user returns online and navigates around the app, it will fetch the latest challenge list and challenge details, synchronizing the client with the server. Below are some cases that we have considered:

1. Challenge has started or is waiting to start

The user can see the challenge details such as the title and description, along with the animation and countdown timer. They will also be able to see the last updated list of participants who have joined and are still pending. This fits the user's expectations as they can still see what the challenge is about, what and when they need to complete and also the list of participants even if they are not connected to the internet.

2. Challenge has ended

When the challenge has ended and the user remains offline, the details page will show that the challenge has ended. As a participant, the user will not be able to vote if they are offline since they cannot get the latest proofs from their fellow participants. This ensures that the voting is fair and a vote can only be cast if the voter has access to the latest list of proofs.

Our verdict

We believe that the current level of offline functionality is sufficient as it covers the most important aspect of our application, which is to view the details of an accepted challenge even if the user is offline. Although it is currently not possible to mark a challenge as completed or upload proof for a challenge, considering that our target audience are university students in Singapore who are likely tech-savvy, it is reasonable to assume that they will be able to establish an Internet connection before the challenge ends.

That being said, there can definitely be future extensions to our offline functionality, such as allowing the user to mark a challenge as completed and attach a proof even when offline. The requests will be cached locally and sent to our server whenever the user is back online.

Milestone 11 - Authentication

Both token-based and session-based authentication are tested and proven to be effective. That being said, they both have their pros and cons.

	Token-based authentication	**Session-based authentication**
Scalability (Token > Session)	Relatively easy to maintain and scale as the tokens are stored on the client side.	After the user logs in, the session data will need to be stored in a database or in memory on the server. As a result, scaling can become an issue especially when there is a large number of people using the app at the same time.
Multiple domains (Token > Session)	Does not rely on cookies as the token is included in the request header. With CORS enabled, it is easy to expose the APIs to different services and domains. As a result, the API can serve both web and mobile platforms, making them mobile ready.	Cookies normally only work on a single domain. Therefore, there can be issues when the API service is from different domains for mobile and web platforms.
System performance (Token > Session)	Additional information such as the role and user id can be stored in the token. Once the token is verified, only a single call to the database is needed to retrieve the data.	Additional look-ups of the database could be needed, not only to verify that the session id is valid, but also to check if the user has access to the data. An example would be the case where a user needs to be of the role *admin* to access a particular API endpoint. A call would be made to the server to first check if the session id is valid, another call to get data about the user to verify the

		admin status, and finally a call to get the data.
Mobile applications (Token > Session)	Able to implement in mobile applications.	Not suitable for mobile applications as they are not able to store cookies.
Size (Session >Token)	Tokens are large as they contain a lot more user information. This could potentially lead to slower requests.	Session id stored in cookies is very small.

Token-based authentication is the choice for our application, due to its many advantages over session-based authentication as shown in the table above. Furthermore, as the size of our requests are relatively small, the large token sizes were not as big of a concern to us.

Milestone 12 - Frameworks/libraries and mobile site design principles

Authentication Service - Firebase

We have chosen to use Firebase for our authentication workflow over building our own authentication system. This is because of Firebase's strong integration with social login features like Facebook and Google. It also provides common authentication services like email and password sign-up, as well as email verification and password resets. In addition, Firebase helps to store the user's passwords, which aids in ensuring that they are stored securely. Firebase also provides additional services like a real time database and sending push notifications. Once a user is successfully authenticated by Firebase, our frontend will communicate the Firebase token of the user to our backend to log in the user to our services.

The alternative approach would be building the authentication workflow on our own. In this case, we would have to set up routes and services for user sign-up, our own email servers for email verification and password resets. Also, we would need to salt and hash the passwords accordingly. This meant additional complexity in the backend in terms of proper user verification, password storage and integration with social networks.

Frontend - Ionic

We have decided to use the React-Redux-TypeScript tech stack for our frontend development, therefore, upon researching on some of the common frameworks UI frameworks, we shortlisted Ionic, Material UI and Ant Design.

The Ionic framework offers a large collection of native-looking UI components out of the box. More importantly, due to the fact that Ionic aims to be a cross-platform framework, it offers native-looking designs for both iOS and material design (commonly used on Android applications). As such, it allows us to deliver a user interface that feels familiar to the user regardless of the device he or she is using, including the look, layout and interactions of the UI components. This is a great advantage as it reduces the time and effort the user has to spend to adapt to a non-familiar interface. On the other hand, as the majority of the components are pre-built, it is slightly more restrictive when it comes to specifically customising one component.

Material UI is another popular UI framework for building beautiful user interfaces for the web. It is highly customisable where we could easily create highly styled components using the `withStyles` method. The downside would be that Material UI is not specifically targeting native-looking designs, meaning that more time and effort are required to ensure that the layout and design of the components are suitable on mobile devices which typically have smaller screens. Since we are on a tight delivery schedule, we have decided to rule out Material UI.

Last but not least, we have also researched into Ant Design, which offers a large collection of mobile-optimised UI components, similar to Ionic. However, we decided to not go with Ant Design as the overall design of this framework is targeted more towards applications used at the workplace. As a result, it does not fit our theme of a platform with gamified elements and cute assets to engage the younger generations.

Mobile Site Design Principles

1. Dark Mode

We understand that users have different preferences in terms of theme. By having the option to switch to light/dark mode, users can enjoy using Wall of Shame while having it fit the global theme of their mobile device. We ensured that all pages on Wall of Shame support dark mode, empowering users to counter procrastination even at night!

2. Optimize the site for mobile

We used a responsive layout for different screen sizes on mobile devices. Figure 12.1 shows the mobile view of our application on an iPhone X and Figure 12.2 shows the same page on an iPad. In addition, no horizontal scrolling is required from the user on any page to reveal more information.

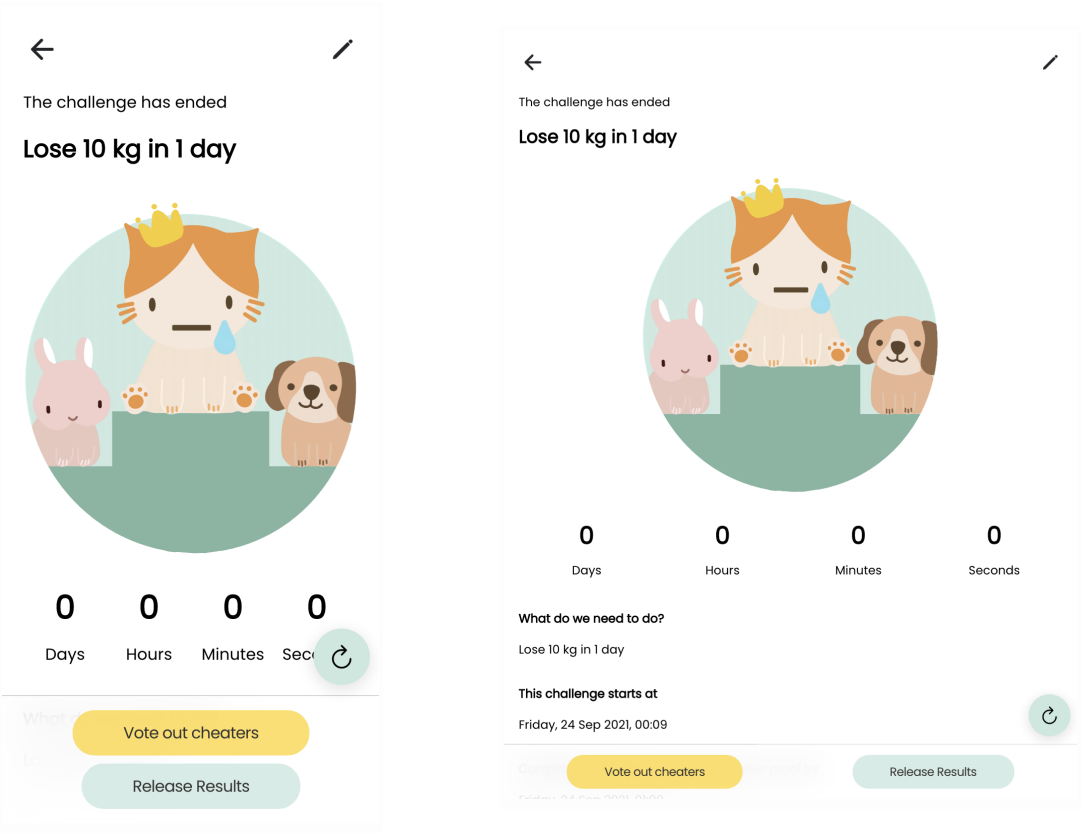


Figure 12.1 (iPhone X, Left) and 12.2 (iPad, Right): Responsive design

We have also made special optimisations for different devices based on the operating systems. Figure 12.3 and 12.4 shows the same page on the Google Pixel XL2, an Android device using material design, versus an iPhone X running on iOS. These subtle differences would make our application feel more familiar to use for the user while maintaining a common design language across different devices.

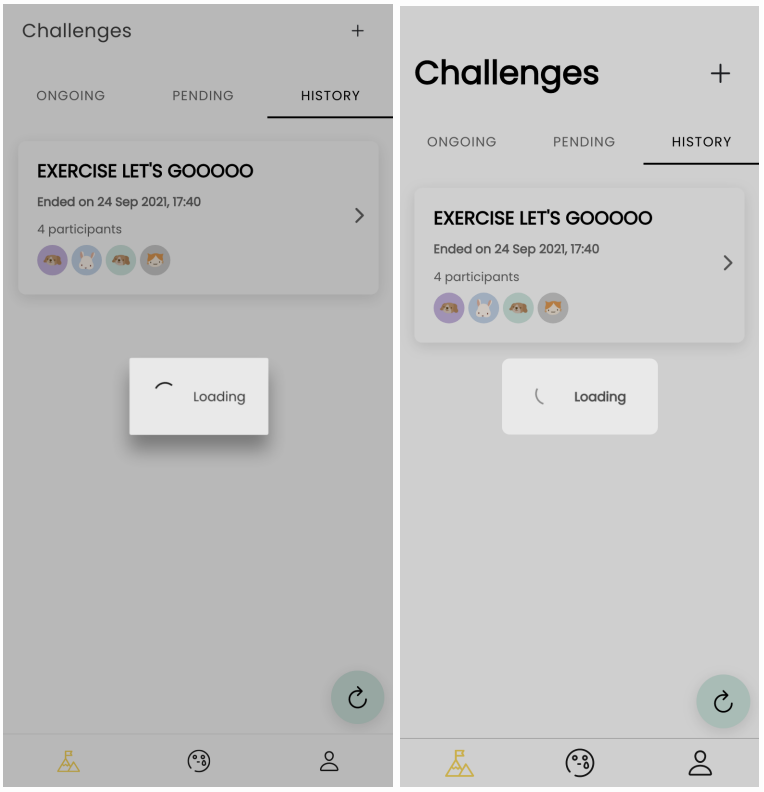


Figure 12.3 (Left) and 12.4 (Right). Challenges page Android and iOS

3. Avoid information overload by utilising visual buffers

We ensured that pages don't overload users with information by having proper spacing and sections for the different pieces of information.

An example is shown below in the created challenge page, where a cute animation of the current users waiting for the challenge to start breaks the mundane information dump. Furthermore, participants joining the challenge are placed into a new section, clearly divided by a grey bar. This ensures that users are able to differentiate between details of the challenge and who exactly is joining the challenge.

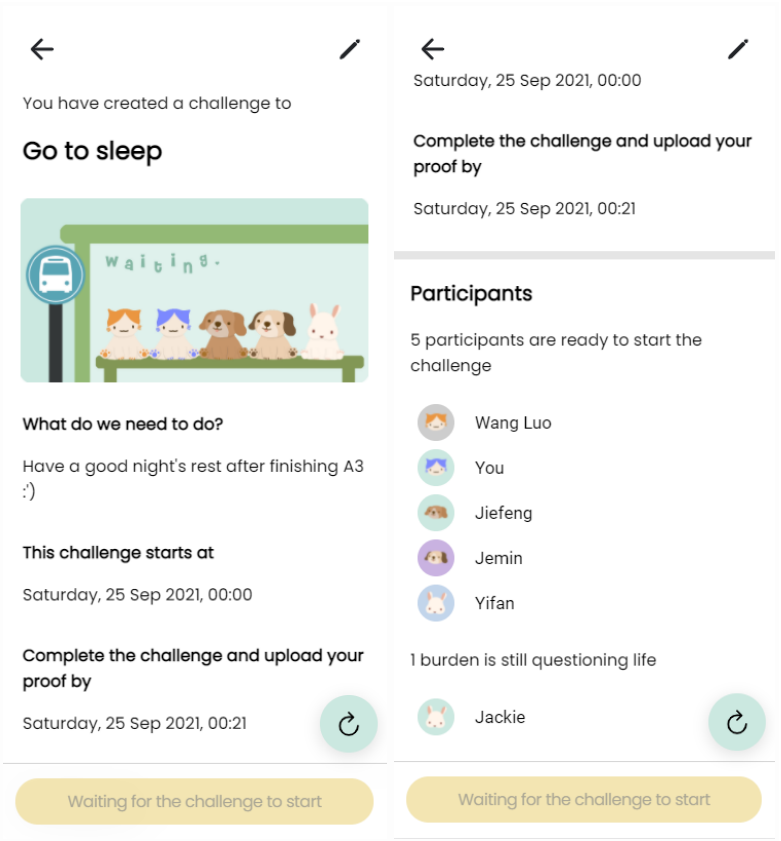


Figure 12.4. Created Challenge page (2 images showing scrollable page)

4. Clear hierarchy of information

We ensured that information is shown to the user in terms of importance. As users will look at the page from top to bottom, we placed high priority items at the top.

An example would be the Ongoing Challenge page, where information right at the top shows what the user challenge is. This is then followed up by a key visual displaying the number of participants who have yet to complete the challenge, where the avatar animals of the participants who have yet to complete the challenge will appear running on the wheel. As users see fewer animals over time, they will know that it is time to step up their game. Below the visual would be the timer, which is important as users will then know how much time they have left to complete the challenge. The information below will then be a reiteration of what they already know - the challenge details as well as the entire list of participants.

By providing a clear hierarchy, users won't need to peel their eyes to find what they are looking for.

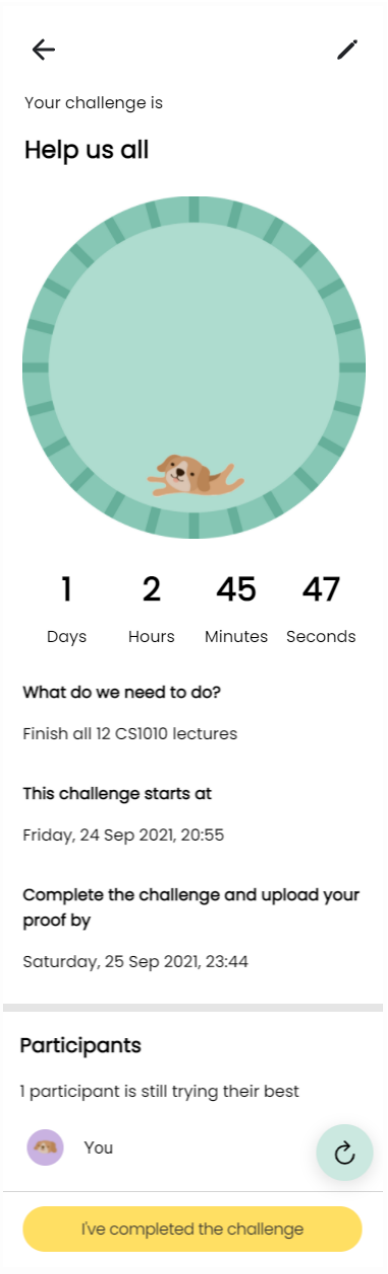


Figure 12.5. Ongoing Challenge page

5. Make call-to-action buttons easy-to-click and prominent

We placed call-to-action buttons such as “Continue with Google” and “Create New Account” in the prominent positions on the page, as shown in Figure 12.6. On pages with more content, such as the create challenge page and challenge details page, we have placed the call-to-action button(s) on a footer at the bottom of the screen that will always stay at that position, as shown in Figure 12.7. This ensures that regardless of scrolling, the call-to-action button(s) are always easily reachable.

Figure 12.6. Login page

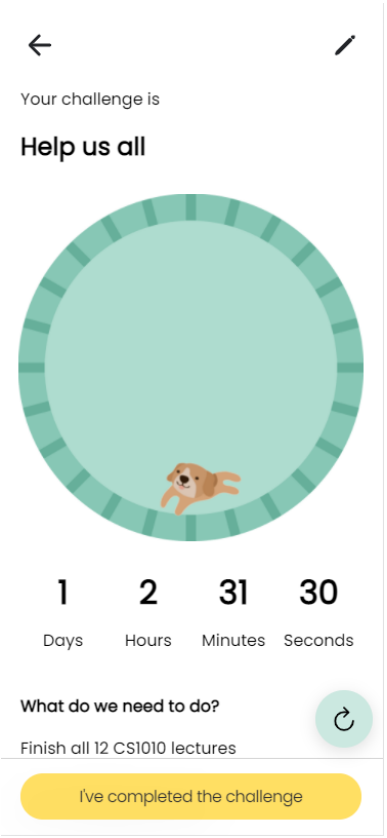


Figure 12.7. Challenges page

6. Minimise form errors with labelling and real-time validation

To make the process of validating form responses more convenient for the user, we have added clear labels for each field in the form. In addition, we have added real-time validation for fields so that they will be highlighted in red whenever an invalid input is entered. This allows the users to be notified of invalid input before they submit the form.

This principle is most noticeable in the sign-up/log-in pages and the create/edit challenge pages, as shown in Figure 12.8 and 12.9 respectively. If an email input is not valid, it will be labelled as red. In addition, the action button will be disabled to prevent sending of invalid requests. For the title and description of the challenge, we indicated that they are compulsory fields by using the asterisk. If the user has selected an end time that is before the start time of the challenge, the field label will turn red.

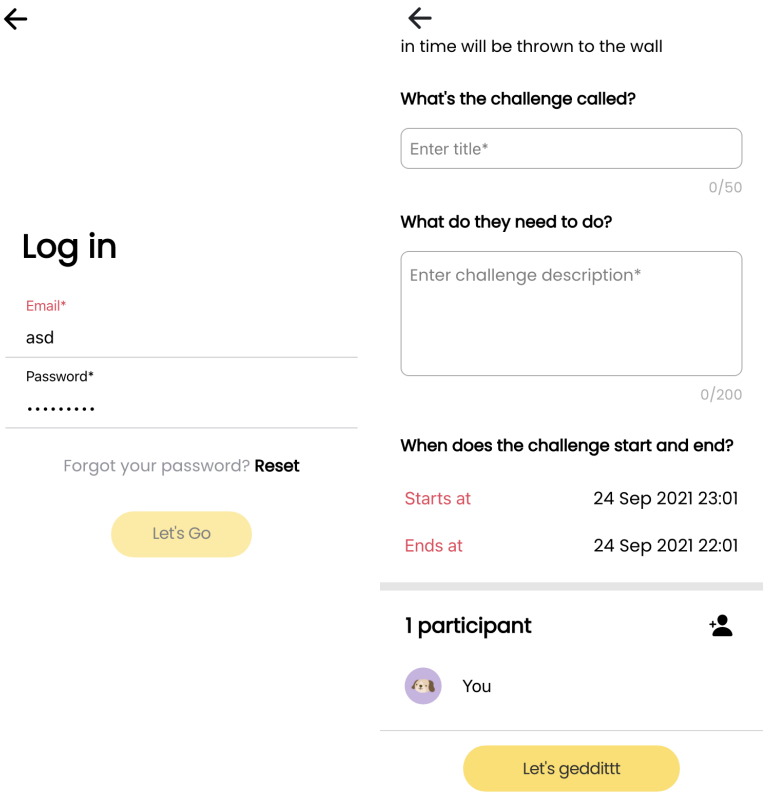


Figure 12.8 (Left) and 12.9 (Right). Create Challenge

7. Don't make users pinch-to-zoom

All the UI components in our application are optimised for mobile viewing. There is no need for the user to zoom in or out to view any content. In fact, pinch-to-zoom has been disabled on our application to simulate a more realistic native experience.

8. Keep your user in a single browser window

Other than Google and Facebook login options, our application remains as a single-page application, there is no need for the user to open up another browser tab to complete an action.

Milestone 13 - Workflows

Workflow 1: Picking an avatar

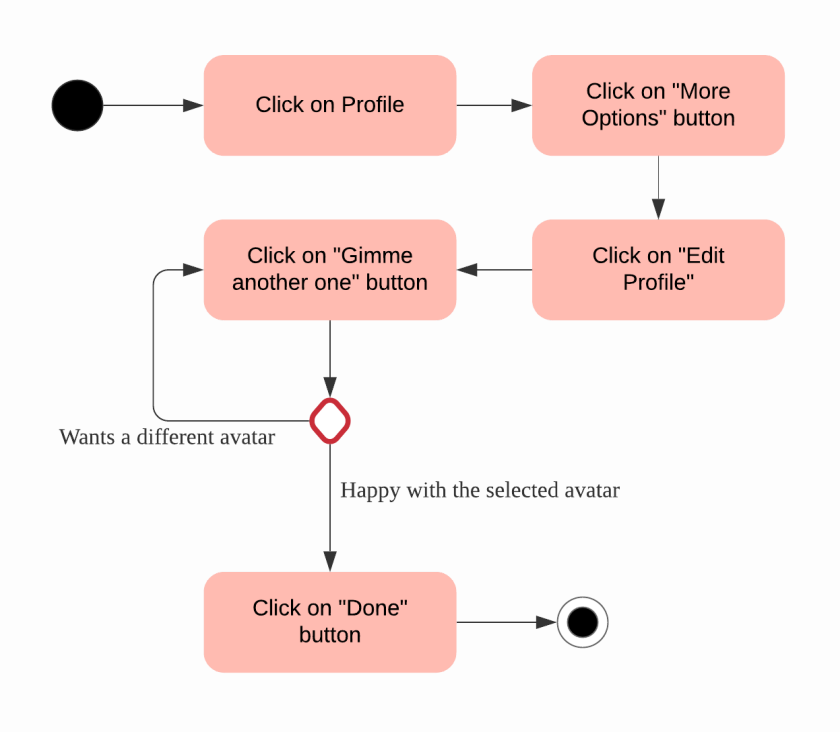


Figure 13.1: Activity diagram for picking an avatar

"Customisation is the soul of a great website."

- Sameer Jain, CEO, Net Solutions

Customisation lets users make their own selections about what they want to see, which is the reason that we have chosen to allow the user to customise his or her profile, especially avatar. The "picking an avatar" workflow we have implemented involves a random avatar generator, which randomly mixes the avatar animals with various colour schemes and backgrounds, allowing the user to roll for an avatar of his or her preference.

A more conventional workflow would be to let users upload images as their avatars, similar to most existing social media applications. However, we decided against this workflow because it requires extra effort from the users to open up their phone galleries and search for a suitable image. In contrast, our random avatar generator would simply require the user to click on one dice button to re-roll his or her avatar. Although it is not as freely customisable as uploading one's own avatars, it still

maintains a sense of personal touch. The inclusion of RNG also gives a more game-like feeling to our application, which aids to reinforce our overall theme.

The following images show the workflow of picking an avatar:

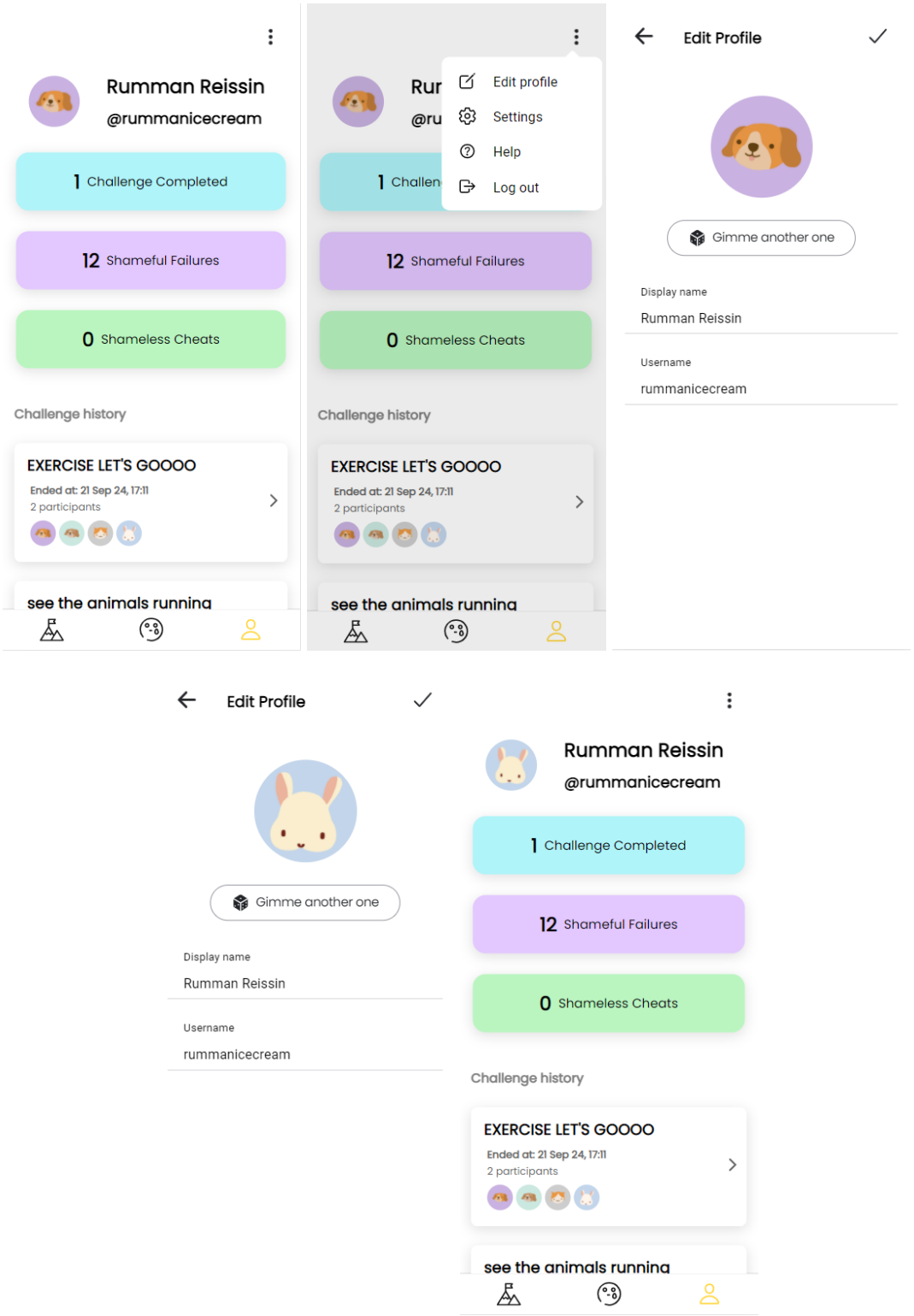


Figure 13.2: Workflow of picking an avatar

Workflow 2: Create a challenge

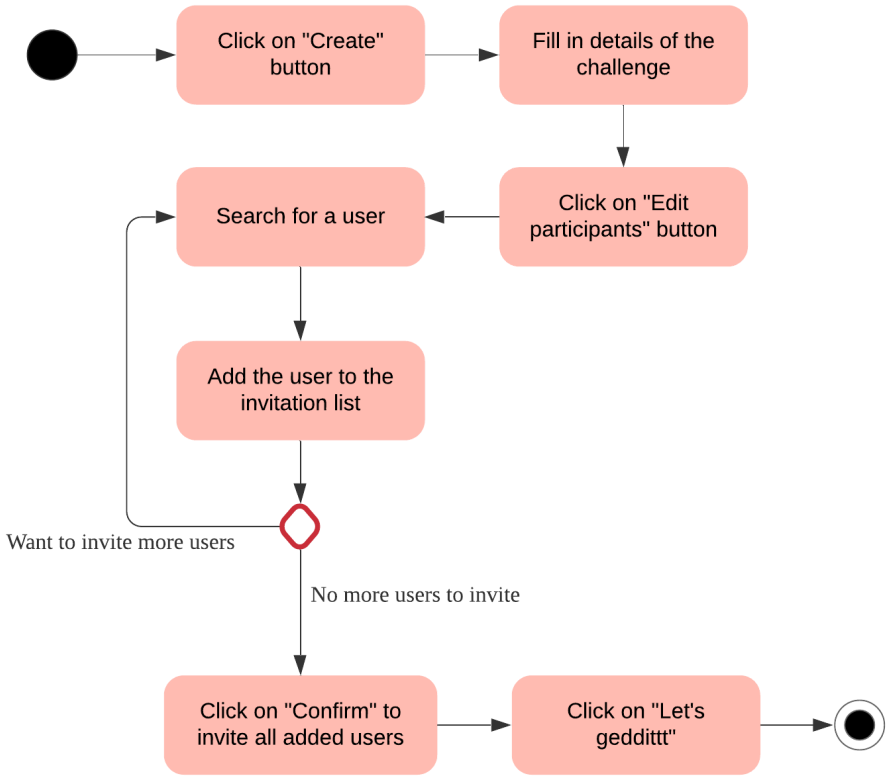


Figure 13.3: Activity diagram for creating a challenge

Challenging friends is one of the key features of Wall of Shame, as the users work with their friends towards a common goal via challenges. As such, we have designed the challenge related workflows extremely simple and intuitive. One such workflow is the "create a challenge" workflow (as shown in Fig 13.4 below), which we expect the majority of the users to perform on a regular basis.

When conceiving this workflow, we had two candidate designs:

1. The owner of the challenge only specifies the duration of the challenge. The challenge begins at the moment that the owner decides to start the challenge, and the participants will be notified via email or notifications about the start of the challenge.
2. The owner specifies both the "Starts at" and "Ends at" timestamps of the challenge, the challenge will be scheduled to automatically start at the "Starts at" timestamp (current choice).

Although design (1) may reduce the complexity of the owner's workflow by offloading the computation of the start-at and end-at timestamps, we decided to implement design (2) because design (2) grants more certainty to the participants. From the perspective of a participant, it would be more assuring for him or her when the exact start time of the challenge is made clear. In addition, the invited users could make a more informed decision regarding whether to accept the challenge with the exact start time in mind, as they could check their availability based on their calendars.

The following figures show the workflow of creating a challenge and inviting friends:

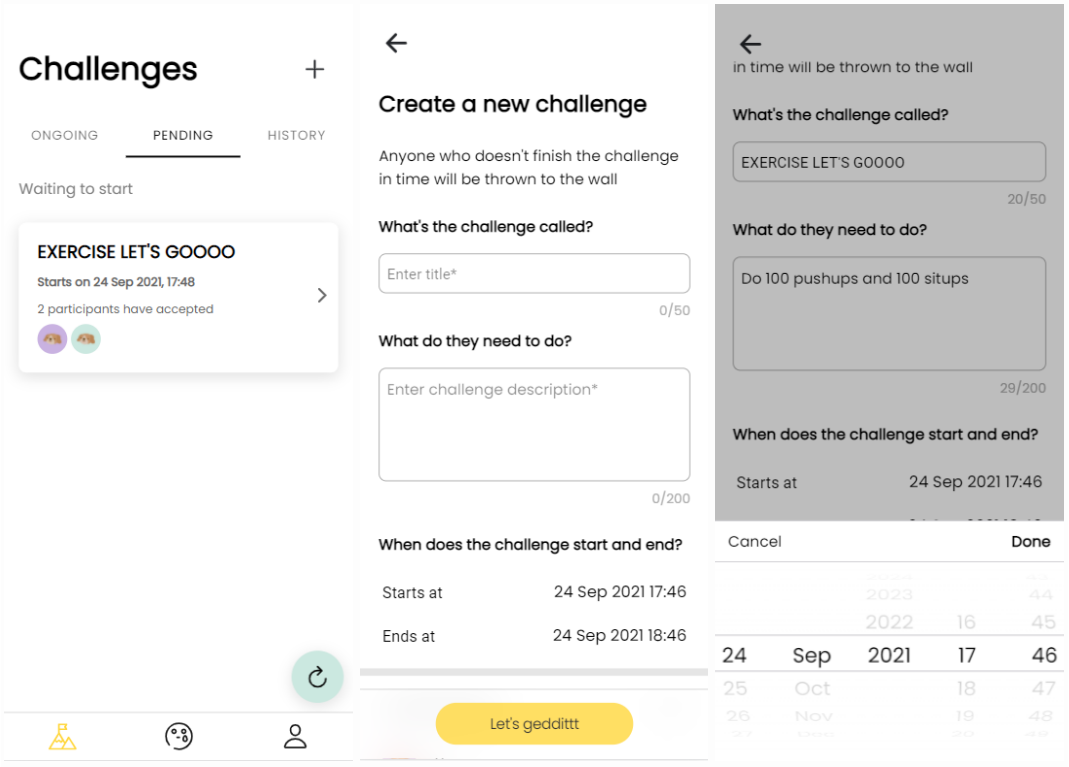


Figure 13.4: Workflow of creating a challenge

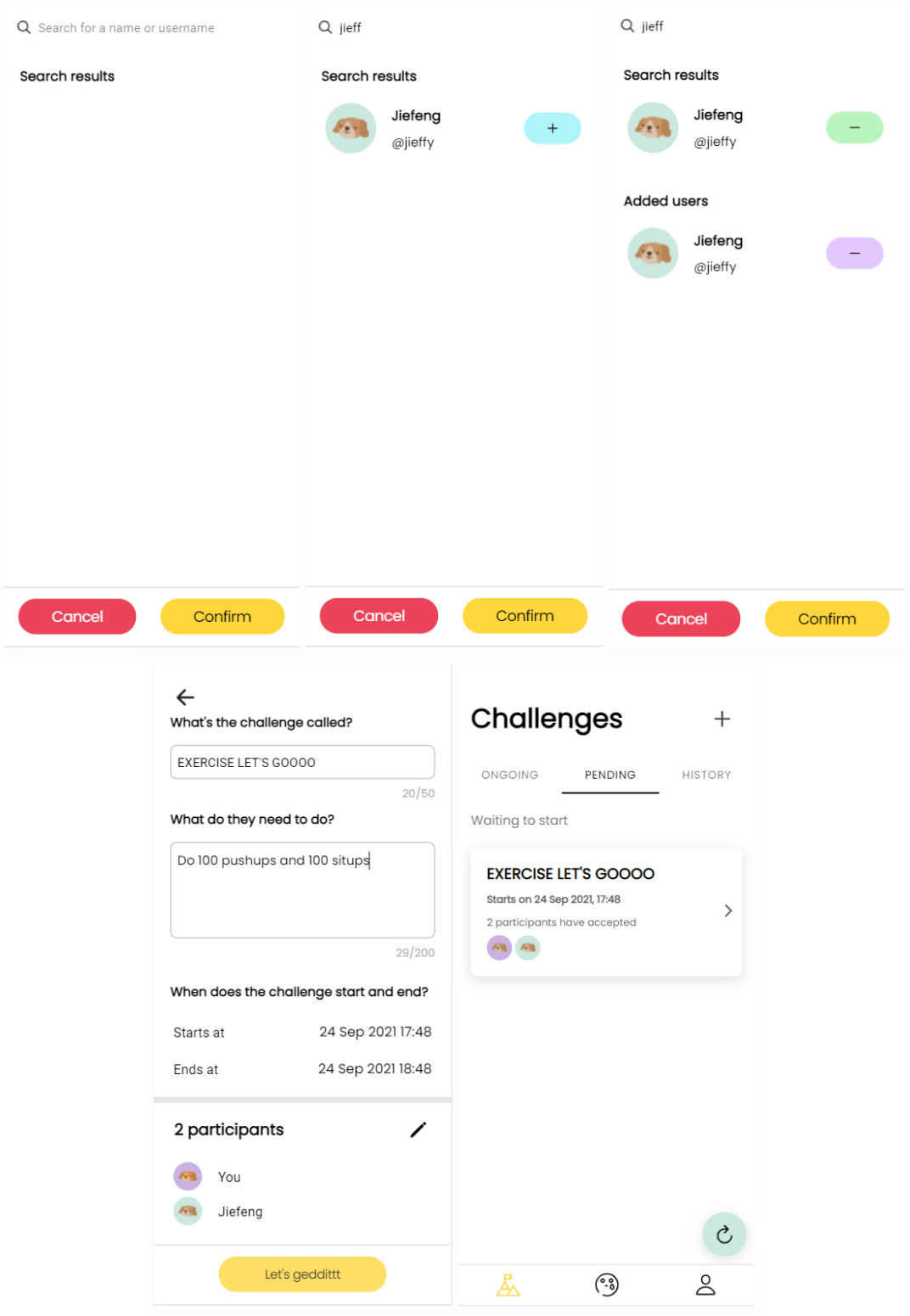


Figure 13.5: Workflow of inviting friends

Workflow 3: Completing a challenge

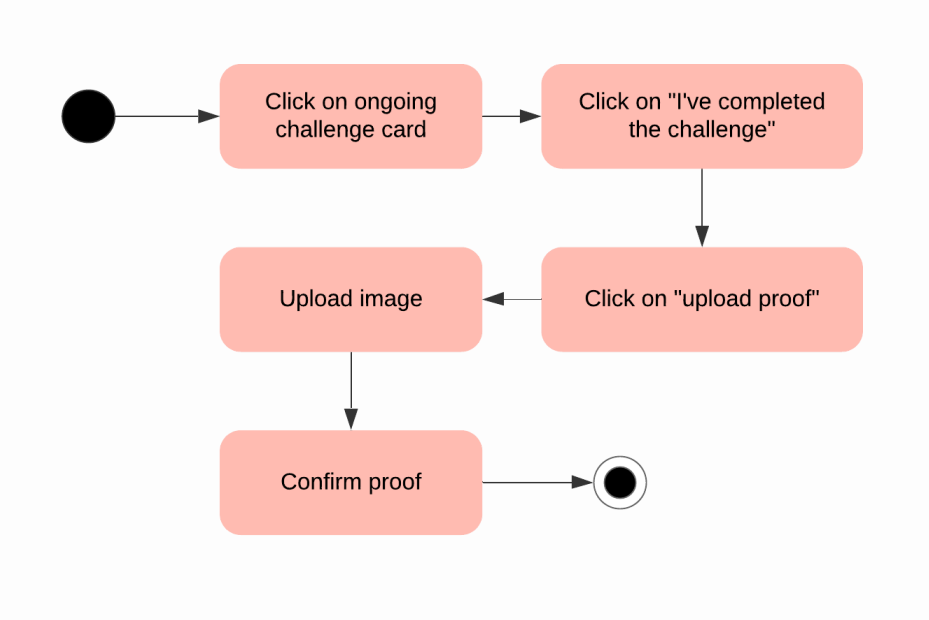


Figure 13.6: Activity diagram for completing a challenge

Completing a challenge is a key workflow for users, and we wanted to make this process as smooth as possible. Users will be able to upload their proof only after confirming that they have completed the challenge, but will be able to reupload anytime only until the challenge is completed. By having this clear step by step flow of user actions, the process will be as clear as possible.

An alternative workflow we thought about would be to allow users to reupload proof after the challenge has ended. This would provide users the flexibility and freedom to change the images. However, we decided against this as we felt that users would be able to easily cheat. By having a proper end time where users can reupload, we ensure that the voting process will be done with consistent proof.

The following images show the workflow of successfully completing a challenge:

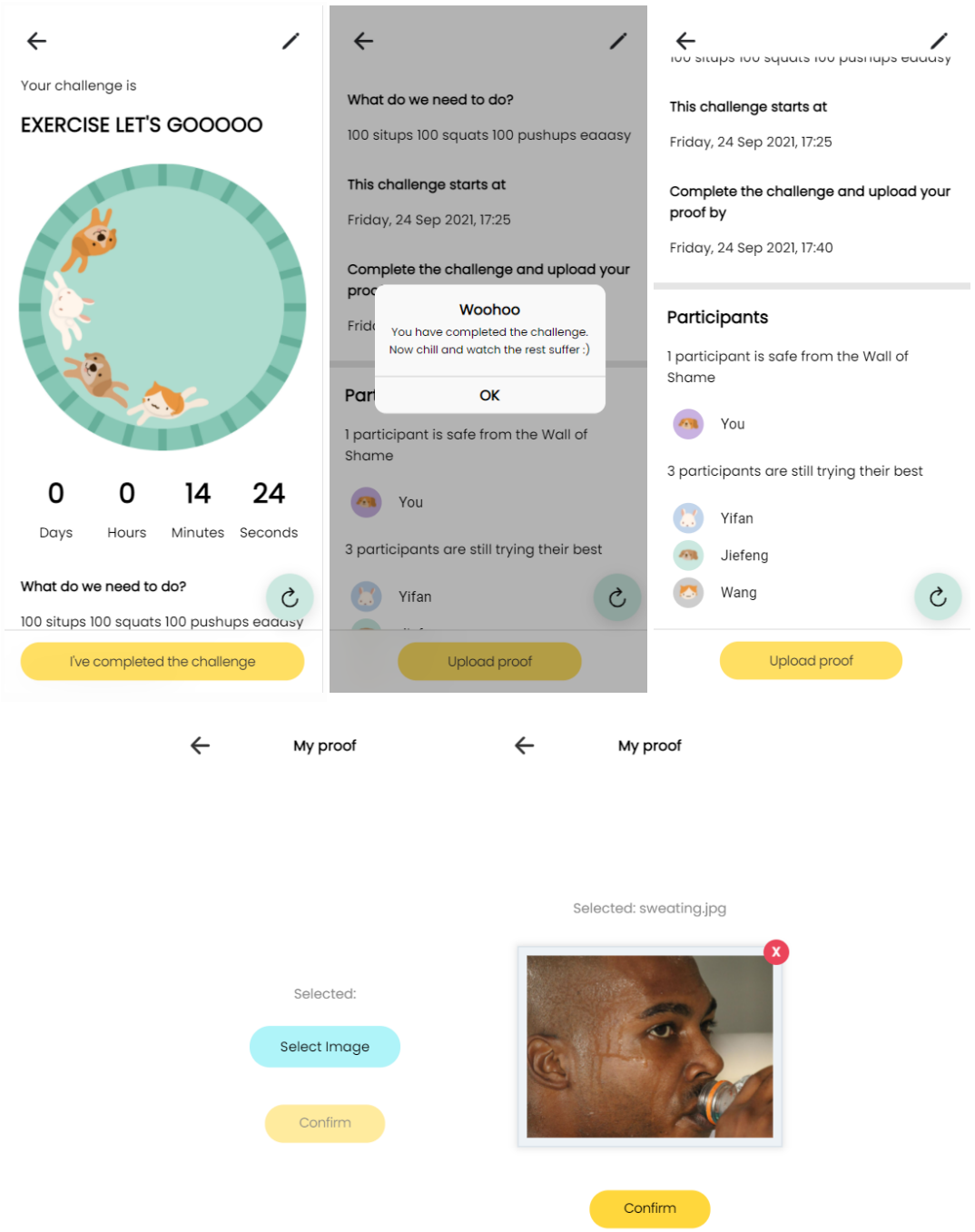


Figure 13.7: Workflow for completing a challenge

Workflow 4: Concluding a challenge as the challenge creator

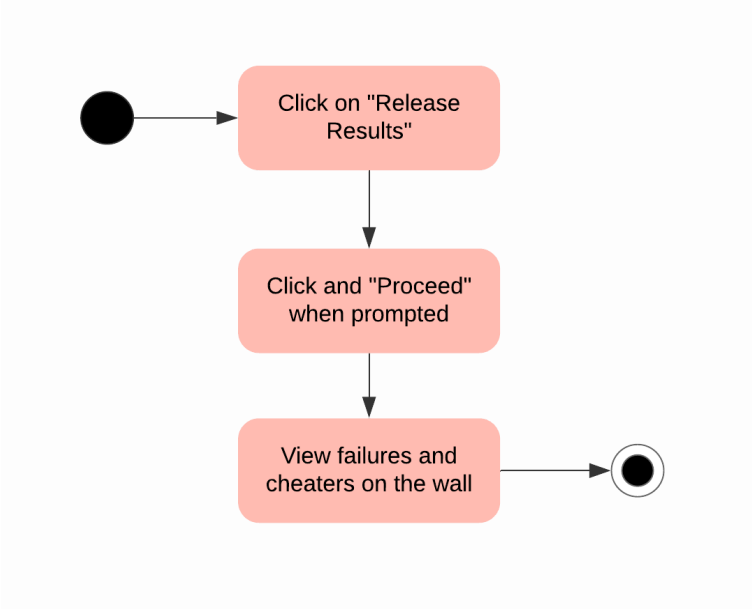


Figure 13.8: Activity diagram for creating a challenge

Another important workflow of our application is the process of concluding a challenge, which helps to uphold the fairness of the final outcome. The final outcome is determined by two types of results, the challenge results (whether a participant completed the challenge) and the voting results (whether a participant is deemed to be honest by other participants). We believe that cheating (by uploading a fake proof) in a challenge is fundamentally different from failing to complete a challenge. For this rather complex workflow, we have considered the following two possible implementations:

1. Both the challenge results and the voting results will be automatically released according to the following scheme:
    a. The challenge results will be released (throw participants who failed the challenge to the Wall of Shame) immediately when the challenge ends.
    b. The voting results will be released on a per-participant basis, meaning that whenever a participant receives votes from at least half of the participants, he or she will be banished to the Wall of Shame for cheating.
2. Both the challenge results and voting results are staged, the owner has control over whether to release the results:
    a. Once a challenge has ended, all participants will be able to view the results on the challenge details page. However, those who have failed to

complete the challenge are not immediately published to the Wall of Shame.

b. Similarly, as participants vote for cheaters, the results are accumulated and stored, instead of banishing a participant immediately when he or she has enough votes.

We chose implementation 1 with the intention of providing a buffer period for any unforeseeable circumstances during the challenge, i.e. valid reasons to not finish the challenge and malicious votings. With the challenge results and voting results staged before releasing, we are essentially granting the owner an opportunity to verify the results before making a final decision to publish.

While it is clear that implementation 1 reduces the responsibility on the owner to manage the challenge, as well as increases the overall sense of competing with one another, we believe that the additional verification process is more crucial in protecting users against unfair shaming. Hence, we have chosen implementation 2.

The following images show the workflow of concluding a challenge:

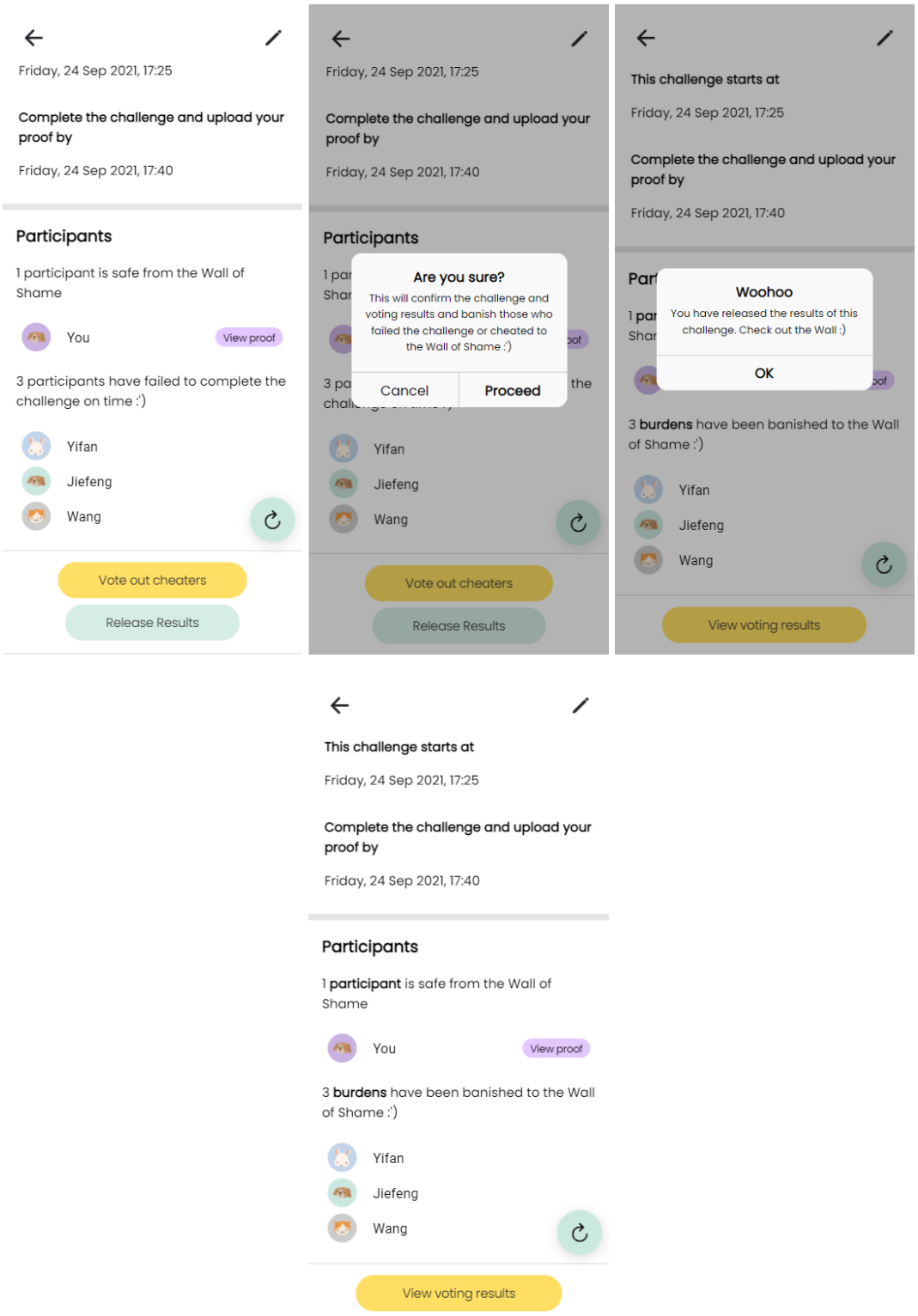
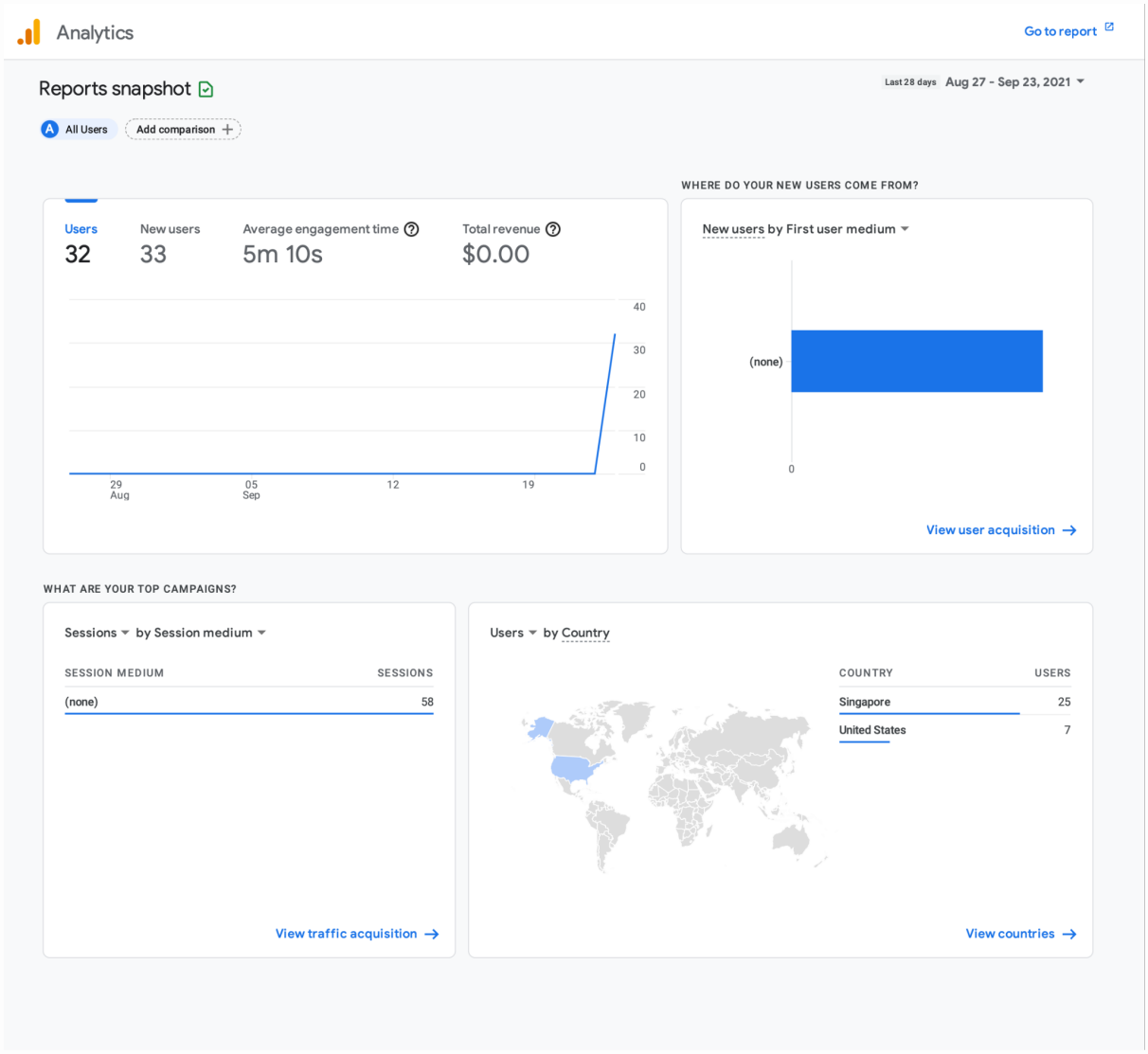


Figure 13.9: Workflow for releasing the results

Milestone 14 - Google Analytics

Below is the screenshot of our Google Analytics report. The complete report can be viewed here.



Milestone 15 - Lighthouse

Our Lighthouse report can be found in the GitHub submission repository.

Milestone 16 - Integrating social networks

As our target audience are university students who are likely to be active users of social media platforms such as Facebook, we have integrated with Facebook to enable the users to directly use our application with a Facebook account. In addition, as the majority of university students in Singapore would likely have a Google account (for collaboration with school mates or personal entertainment), we have also integrated Google Sign-In into our application so that users could simply sign in to our application using their Google accounts.

As a future extension, we could take advantage of Facebook sharing or even the friends list APIs to allow the users to share our application on Facebook or invite friends more easily. We believe that the target users would be willing to share our application on their social media networks as our application is collaborative in nature and part of the fun in using our application is to throw their friends onto the Wall of Shame.

Bells and Whistles

Cache Busting

As a PWA caches data heavily on the client site, we have added cache busting logic to our application to ensure that the user is using the latest version of our application. We have followed the guide here for our implementation of cache busting logic. Essentially, every deployment of our application will result in a newer version being written to a meta file. When the application running on the client's device detects a mismatch between the client's local version of the application and the received version from the server, the application will trigger a refresh of cached data in order to serve the latest application.

Live Leaderboard with Firebase Realtime Database

In order to make the app feel more gamified, we wanted to make the leaderboard on the Wall of Shame page update in real-time. We implemented this using the Firebase Realtime Database.

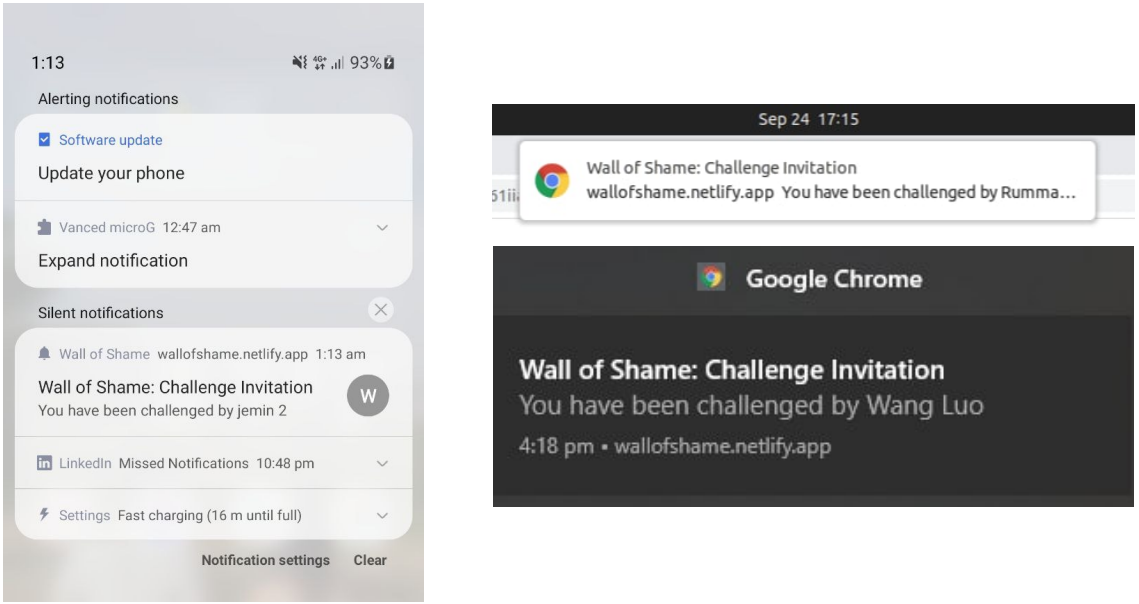
The Wall of Shame subscribes to the Firebase Realtime Database and listens for events. When the owner of some challenge decides to release the results to the wall, the frontend sends the user's name, avatar, the challenge's name and end time to the Firebase Realtime Database. The Wall of Shame page for all users, who have the page open, will then be updated in real-time to show those who have failed the challenge.

This approach was chosen over building our own sockets as setting up our own sockets is time-consuming and involves careful software engineering. On the other hand, Firebase provides a relatively straightforward service to update the page in realtime.

Notifications with Firebase Cloud Messaging

As users may not be on the app all the time, we wanted to ensure users get notified when they are invited to a challenge. To do this, we utilised Firebase's Cloud Messaging feature to send messages, and the service workers to receive and display them. When a challenge gets created, the server sends a notification to users who have allowed notifications from our application. This notification informs them that

they have been invited to a challenge, so they can go into the application to view the challenge.



Sample Notifications

Unfortunately, this functionality is not available on iOS devices due to the lack of support for the Firebase Cloud Messaging SDK.

Other Design Choices

Backend - ExpressJS and Prisma

We chose to use ExpressJS with Typescript to handle the routing. This was chosen as it provides type checking and is minimal. Since we have a limited amount of time, we do not need complex libraries like Django, Ruby on Rails and NestJS that provide a lot of features which we will not be using. Another minimal framework is Flask, but it does not have an easy way to check for types. Using Typescript also allows for API interfaces to be easily copied over to the frontend for smoother integration.

We decided to use Prisma as an ORM library. An ORM was chosen as it allows for easy management of the database schema migrations, as compared to raw SQL. The use of an ORM layer adds type checking based on the schema, which makes it easier to develop and debug code. In addition, it makes querying for data more intuitive.

Prisma was chosen as the ORM library as it makes construction of the schema easier, since it helps to validate the schema and checks that relations between entities are properly set up. When querying related entities, it is also easy to tell what related entities are being queried for and included through the arguments and types, as shown in Milestone 6. TypeORM, an alternative ORM library, does not provide an easy way to type check for related entities at compilation.

An area where Prisma does not handle well is the use of database views, which meant it was difficult to utilise views to abstract over the underlying entities and relationships. However, it allows developers to write raw SQL and pass in a generic, so we can still use views and have type safety. In our case, since the ParticipationStats view is simple in nature and is almost completely identical to the User (and thus has the auto-generated typings for User), we can still have type safety while using views.