Team: Group 7
COS 301
Mini Project 2023
University of Pretoria

Memory Lane

Name	Student Number
Shashin Gounden	u21458686
Arno Jooste	u21457451
Reuben Jooste	u21457060
Armand Krynauw	u04868286
Luca Loubster	u20439963
Mbofho Mamatsharaga	u18045881
Bandisa Masilela	u19018182
Keabetswe Mothapo	u21543462
Andile Ngwenya	u20612894
Alistair Ross	u21489549
Kaitlyn Sookdhew	u21483974
Christof Steyn	u17021074
Tyrone Sutherland-Macleod	u21578878

Table of Contents

MEMORY LANE

●

Introduction

Memory Lane is a social media app that allows users to create and share "time capsules," which are posts that combine text and images. Users then share these time capsules with their friends.

Time is used as the social currency in the app. Time is used in two ways:
"Post-time" and "app time"

- Post-time is the period during which a time capsule can be preserved as a memory. When a user creates a post, the post will start with a fixed amount of time it will be available for other users to see and interact with. This time will begin to diminish when the post is shared. Other users can comment on the post, which increases the time given to the capsule. If the time on a post runs out, the post becomes unavailable for the user's friends to see and the time capsule is “frozen in time” (i.e. archived). Please take note that there is no time loss for the commenter, meaning there is no traditional exchange of time by interacting with time capsules.

- Users are given daily app time to utilise the application. App time increases daily, up to a predetermined point, and at that stage, the time allocated will reset back to the minimum. App time can be utilised to revive dead memories, capsules that have less than 0 seconds on them so that other users can view the memory again. A user who runs out of “app time” will not have access to the app until the next day, when they automatically allocated more time to log in to their account.

The project's vision is to allow users to create and share content within the app. Hence, having time means that a user can use the app, interact with others, and keep their posts visible for others to see.

Objectives

The high-level objectives of the project include, but are not limited to:

- Create an interactive front-end to view the content created by users and users' profiles
- Create a service to manage the connection between the data stored and the front-end
- Create a service to allow users to create time capsules
- Create a service to manage post time and app time for each user
- Store and manage data regarding users’ profiles and content shared on the app. This allows viewing of the data in the front end.
- Implement a feed/home page to view content and posts shared by other users which are organised by post time.
- Implement search functionality to search for specific users, memories, tags, dates and more.
- Implement functionality to comment on other users’ posts

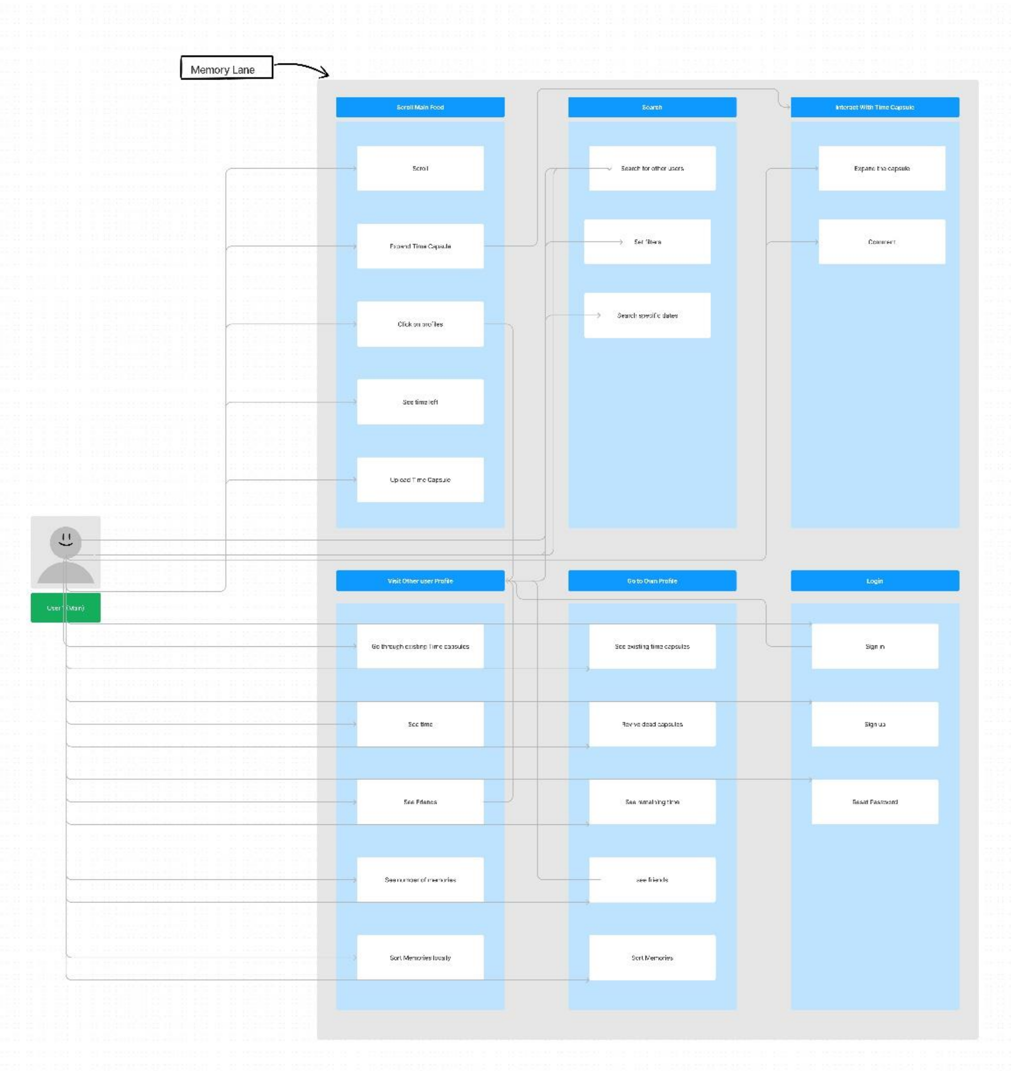
User Characteristics

A user of Memory Lane can create and share content in the form of time capsules (a combination of text and images). They will view other users’ time capsules and comment on these posts. The user interface will be in English, thus the user should have a basic knowledge of the English language.

User Stories

As a User of Memory Lane

- I can register an account
- I can log in to the application
- I can log out of the application
- I can view my feed with my friends' time capsules
- I can create a time capsule
- I can search for time capsules, dates, users, etc.
- I can view my time capsules
- I can unarchive dead time capsules
- I can view my friend count
- I can update my account details
- I can view my app time
- I can view each of my time capsules Post time
- I can view my dead time capsules
- I can view my other users' profiles
- I can comment on other users’ time capsules
- I can view comments on my time capsules
- I can view comments on other users’ time capsules


Memory Lane
Scroll Main Feed
Scroll
Expand Time Capsule
Click on profiles
See time left
Upload Time Capsule
Search
Search for other users
Set filters
Search specific dates
Interact With Time Capsule
Expand the capsule
Comment
User 1 (Main)
Visit Other user Profile
Go through existing Time capsules
See time
See Friends
See number of memories
Sort Memories locally
Go to Own Profile
See existing time capsules
Revive dead capsules
See remaining time
see friends
Sort Memories
Login
Sign in
Sign up
Reset Password

Functional Requirements

The Memory Lane app shall:

1. Provide Administrative tools for users
    - Users can login
    - Users can logout
    - Users can register an account
    - Users can update details about their account
    - Users can permanently delete account

2. Provide management of User’s friends
    - Users can add friends
    - Users can accept/ignore a friend request
    - Users can delete friends
    - Users can view another user’s friend list

3. Provide Post functionality
    - Users can create a post
    - Users can share a post
    - Users can delete a post
    - Users can revive a dead post

4. Provide Commenting functionality
    - User can add comments
    - Users can delete comments

5. Provide search functionality
    - Users can search for another user
    - Users can search for posts by using a date, place, and/or tags as parameters

6. Provide a content feed

- Users can view friends' time capsules
- Users can view comments on friends' time capsules

7. Provide a User profile page
    - Users can view the friend count
    - Users can view the number of memories
    - Users can view the alive time capsules
    - Users can view the username and profile picture

Non-Functional requirements

1. Performance Requirements

Scalability

- The system should be able to handle a large number of users.

Speed

- To create a better end-user experience, the application should be quick enough to respond to user's actions in a short period and it should not slow down with an increase in the number of users.

2. Quality requirements

Usability

- The user interface should be simple and easy to use by any user

Reliability

- The system should be reliable such that it yields correct results if a user performs searches.

Availability

- The system should be available at all times. To ensure a better user experience, there should be minimal to no downtime.

Maintainability

- The system should be developed in an extensible way so it can be easy to add new features and accommodate changes to existing features.

3. Security requirements
    - During registration the email provided is validated
    - The users’ passwords should be at least 8 characters, containing at least a small character and one capital letter, a number and a special character
    - Passwords must be stored as a hash value in the database

Requirement Analysis

Viewport: Mobile

Description:

Because users may upload and share postings (referred to as "time capsules") in the form of text, photographs, or both, our application qualifies as a social media platform. These time capsules are distributed among users (i.e., friends on the app).

Time as a Currency:

Our product meets this requirement in two ways, "post time" and "app time".

User Safety and Privacy:

Users can decide whether to make a given time capsule public or private. Users can restrict who can view a time capsule to their friends; however, all profiles are visible to the general public.

CI/CD Pipeline:

Our DevOps team will continue to conduct automated tests after the product has been released and offer users software updates.

Main Features:

- User profile
- Search tool (to search for specific users, memories, tags, etc)
- Home page (the feed page where users will see other users' time capsules).

Personalisation:

Profile pictures, usernames, and private posts only friends can view

Real-Time Data:

Users' remaining time on a post is updated in real-time for all users. Real-time user comments are displayed.

Time Currency in the business sense:

Every day, the user's time is reset to the time they started with. This stimulates the user's desire to use the app, but because the user's time is limited, the user's well-being is also protected.

Exploration Feature:

There is a search feature for users, dates, and memories on Memory Lane. Users are able to view the memories of other users, including public figures whose memories have been set to be publicly accessible, in addition to those of their friends.

Agile Process Usage

Agile development methodology was used in the development of the Memory Lane App to deliver the application in iterations. The Agile approach allowed the team to quickly respond to changes in the project's requirements and allowed them to continuously deliver working software.

The development process began with the creation of a product backlog, which included all the features and functionalities that were required for the application. The backlog was prioritized based on the project's goals and objectives and was constantly updated throughout the development process.

The development process was broken down into sprints, with each sprint lasting for a week. At the beginning of each sprint, the team held a sprint planning meeting to review the backlog and identify the tasks that needed to be completed during the sprint.

During the sprint, the team used daily stand-up meetings to keep track of progress and to identify any roadblocks that needed to be addressed. At the end of the sprint, the team held a sprint review meeting to review the work that was completed and to receive feedback from stakeholders.

The Agile approach allowed the team to continuously deliver working software throughout the development process. Almost every sprint included a "demo" where the team would demonstrate the new features and functionalities that were added to the application. These demos were held with the project manager and the different sub-teams (i.e. back-end team, front-end team, documentation team)

The team also used Agile practices, such as continuous integration and continuous deployment, to automate the build and deployment processes. This allowed the team to quickly deliver new features and bug fixes.

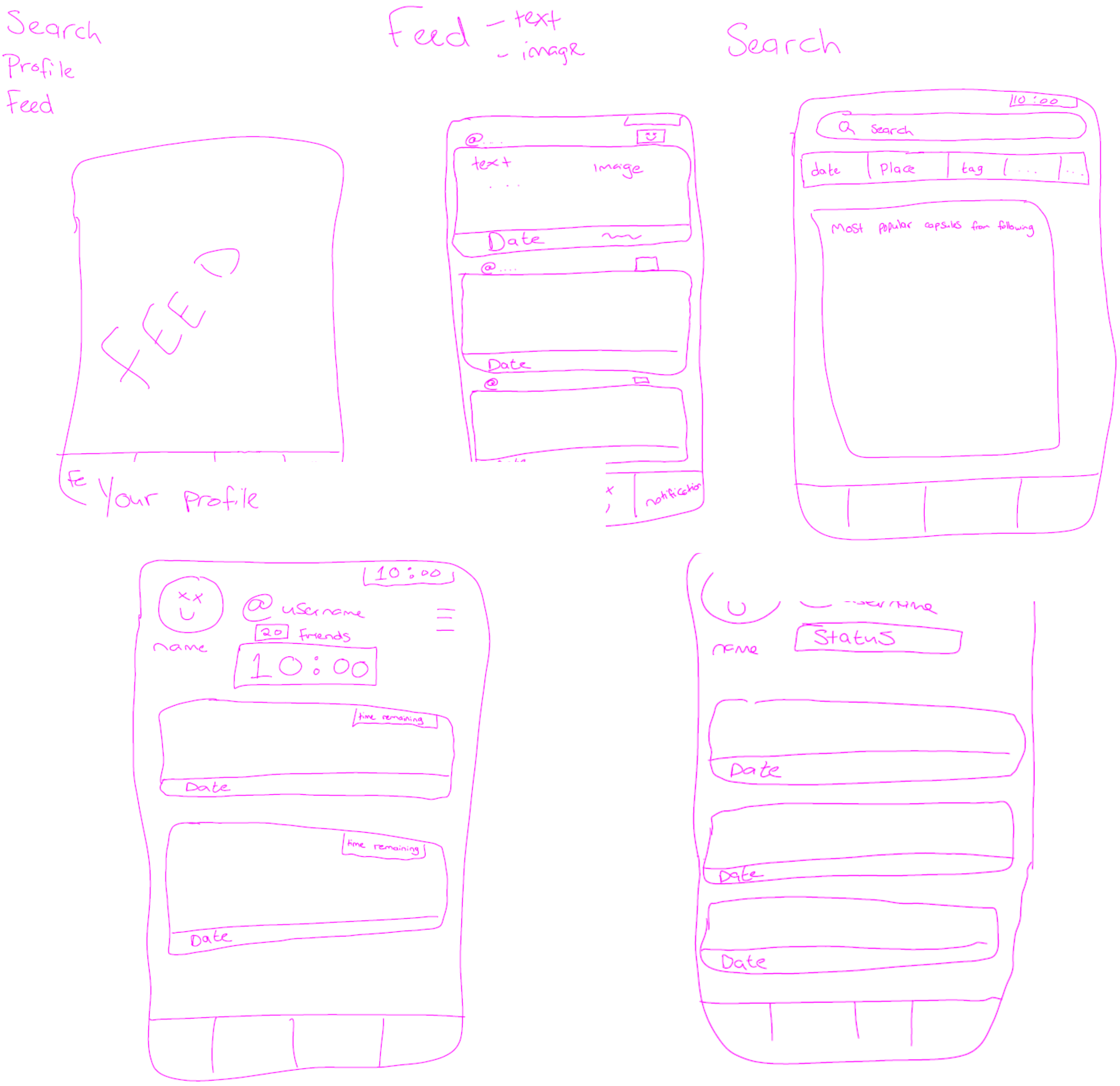
The Agile approach allowed the team to be flexible and to quickly respond to changes in the project's requirements. The team was able to prioritize the backlog based on feedback and adjust the scope of the project based on the project's goals and objectives.

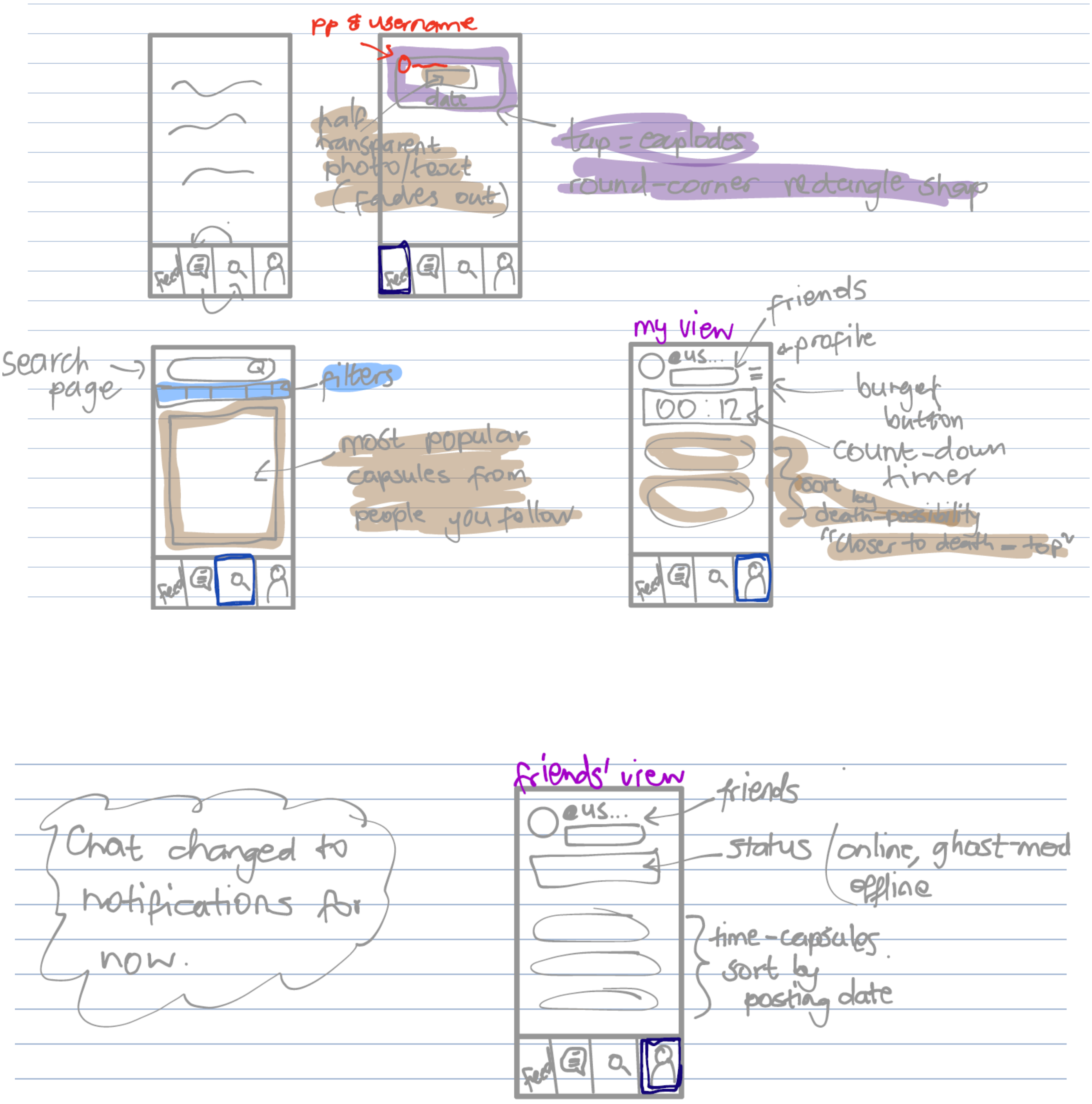
Overall, the Agile approach was a key factor in the success of the Memory Lane App development process. It allowed the team to deliver a high-quality application that met the project's goals and objectives.

Design

In one of our earlier meetings, after discussing the functional requirements of the app and all the features we wanted it to have, we started making some wireframe sketches and agreed that our app would be named “Memory Lane”. The following are the sketches that were made in the meeting

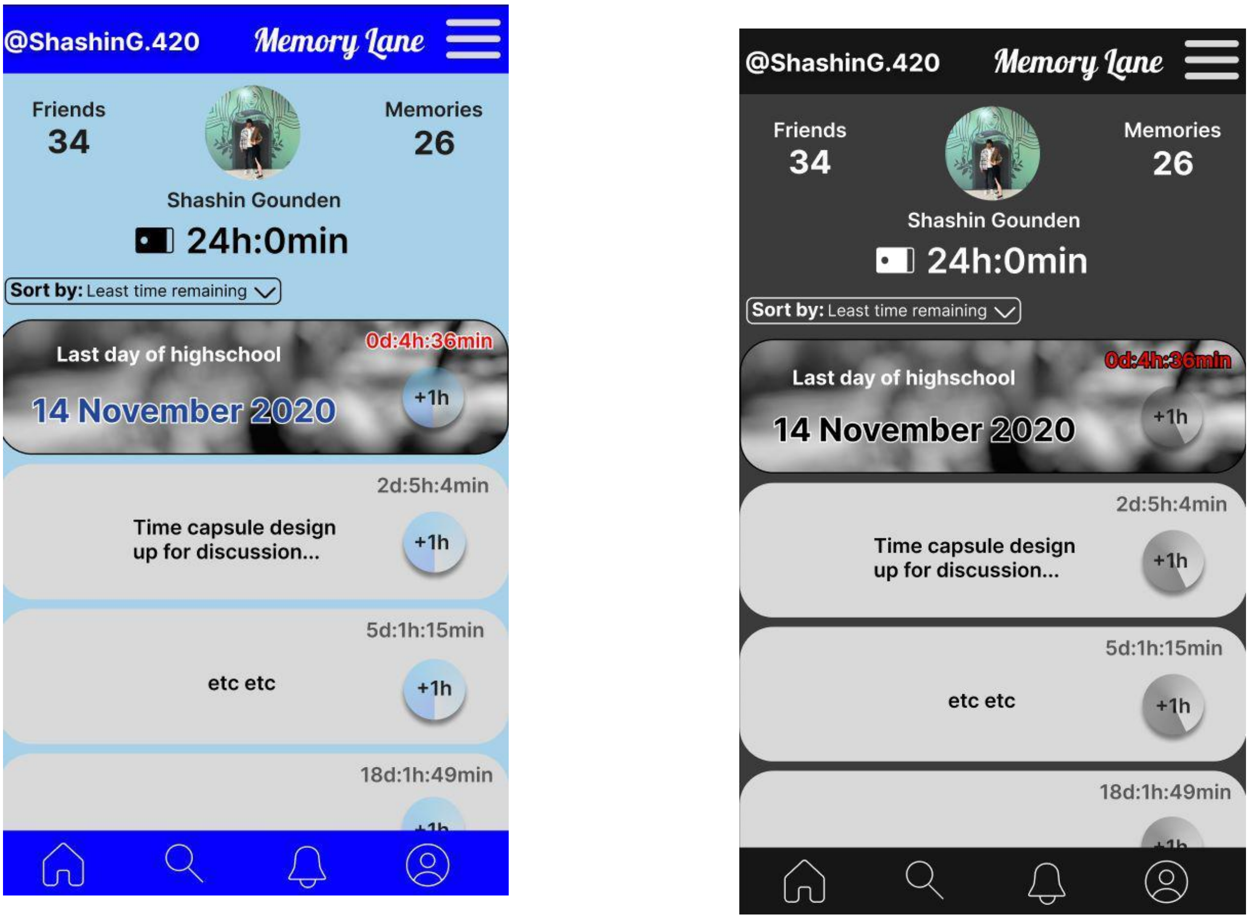
Sketches by Kaitlyn Sookdhew





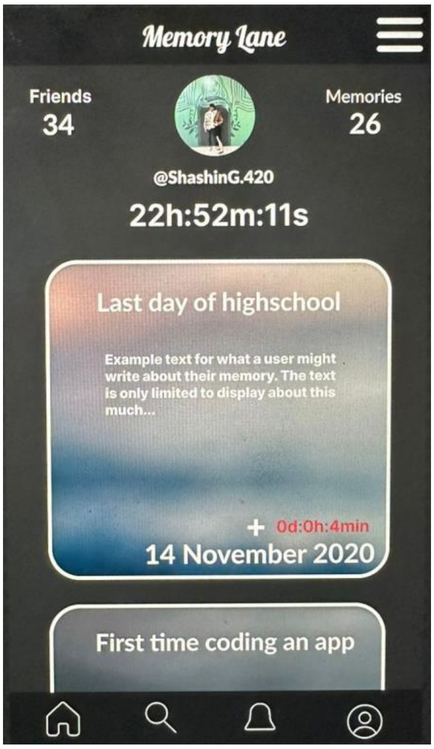
Sketches by Keabetswe Mothapo

Shortly after the meeting where the wireframe sketches were discussed and created, Tyrone (the head designer) created actual mockups with input from the rest of the team. The following are the first two mockups of the profile page, experimenting with different colours:



Original two mockups by Tyrone Sutherland-MacLeod

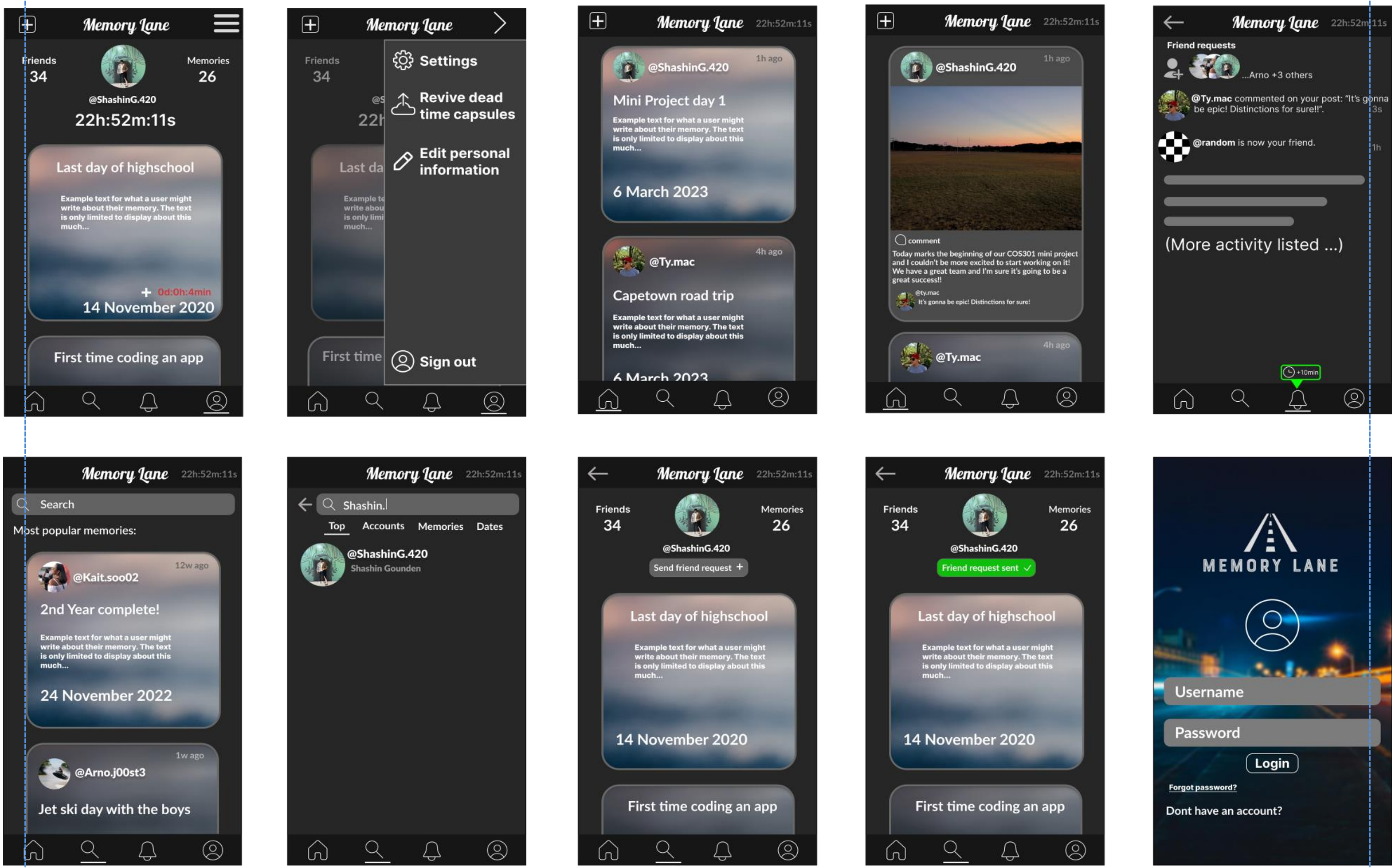
Project manager Shashin Gounden liked the idea of a neutral colour scheme and took a copy of the designs, putting his own spin on it:



Example text for what a user might write about their memory. The text is only limited to display about this much...

Profile mockup by Shashin Gounden

After the team agreed on the colour scheme and the rough layout of the first mockups, Tyrone went ahead with designing all of the app’s pages to the agreed upon scheme:



Final mockup designs by Tyrone Sutherland-MacLeod

The project file where these designs can be viewed in greater detail is found at: https://www.figma.com/file/SeZrzER521MqF792xCXK3c/Profile-design-v1?node-id=0%3A1&t=XPlvr1eOdI9zxNxP-1

Implementation

Tech Stack:

Use Case	Tech Used
Front-end	Angular with Ionic
Back-end	NestJS/CQRS
API	NestJS/Firebase Cloud functions
Data Store	FireStore/Google Cloud Storage
Version Control	GitHub
Team Organization	GitHub, Discord, Google Docs, Blackboard
Testing	Jest/GitHub Actions
Integration	Travis CI
CI/CD	GitHub
IDE	Visual Studio
Documentation	Google Docs/GitHub Wiki/HedgeDocs/Microsoft Word

Workflow Testing:

Testing in GitHub:

Every time you commit or pull any of your code, any tests you want to run on it should be executed automatically. The test results will be shown on our GitHub home page's actions tab. By clicking "Test" and looking at the errors that were generated, you can determine exactly why a test failed. There will be a red cross next to the test. Clicking the red cross will bring up a link for "Details." This link will take you to the workflow page where you can find the failed step (which will be denoted by a drop-down menu with a sizable red cross). On this page, you can see what failed and why it failed.

Linter:

Your code must pass the linter test before being pushed, pulled, or merged with either dev or main. The linter will examine the syntax and standards of your code but won't search for mistakes.
If there are no white spaces or empty lines at the end of the line, it will fail.
There are files set up that will automatically run "Prettier" on your code when you commit to "dev," eliminating any of the aforementioned problems.
However, these files won't be used when pulling into Dev, and your code might fail as a result of the aforementioned problems. Therefore, before opening the pull request, you must
run prettier on your code.

Working with GitHub

General Rules:

1. Do not commit anything to main/master unless widely discussed with others and everyone has agreed.
2. If you have a merge conflict or any issue don't just delete it, ask someone for help.

General Functionality:

This document will look at both working with GitHub Desktop and VScode (terminal and extension)

GitHub Desktop:

1. Check that you are the correct repository in the top left.
2. Check that you are in the correct branch
    - you can create a branch if you feel you need to, however, ensure that the branch doesn't already exist.
    - when creating branches name correctly
    - feature branches get feature/branch-name
    - front-end branches get front-end/branch-name
    - etc etc
3. If there are local changes to the branch pull them. (Big blue "pull" notification)
4. Fetching origin will fetch any changes to the remote repo. (Won't download just checking if there are updates)
5. Open in Vs Code (button in the middle)
6. Once you have started coding:
    - you will be able to see any changes made in GitHub desktop
    - You can stash any changes in GitHub desktop (This means that the changes you've made will be saved and you can pull them into your current branch)
    - once finished coding commit your work on GitHub desktop
    - you will need to supply a Summary heading and message when committing
    - ensure that your message isn't longer than 50 characters (the less the better), explain what and why.
7. Push commits to return

VS Code/Terminal:

1. Open the copied folder in vs code this should automatically activate git.
2. If you have never done this before sign into your git account using either in the terminal:

    - $ git config --global user.name "Your name here"
    - $ git config --global user.email "user.email"
3. Ensure that the source control in vs code(3rd icon from the top on the left) shows "message" and "commit".
4. In the terminal type "git branch", the branch you are in will have a star next to it.
5. If you are not in the branch that you should be working in then in the terminal type "git checkout 'branch-name'".
6. Ensure you're in the correct branch.
7. If a branch you are looking for isn't there try "git fetch origin" and try again
8. In the terminal type git status, if you are not up to date type "git pull"
9. Once finished coding, your changes should show in the source control tab.
10. You can stash, stage, and commit your changes in the source control tab.
11. vs code also provides an interface to view merge conflicts.

Merge Conflicts:

If you run into a merge conflict message and you don't understand the conflicts once you view them, contact the person who may be responsible for the changes (which can be viewed in the commit history) or contact the project manager.

Structuring of JSON data:

General Workflow:

To create the JSON data, the front-end and back-end workflows adhere to the idea of page-by-page completion.

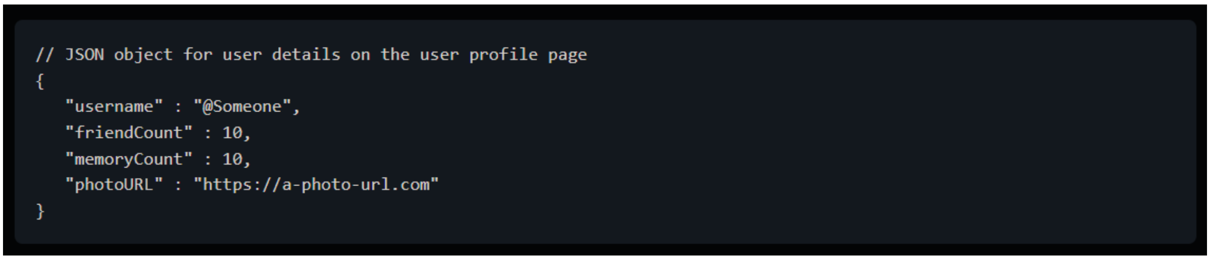
Detailed Workflow:

The data that each individual page will contain must be kept in mind as each member of the front-end team works on its design and implementation. After completing the implementation of that page, they must give a thorough explanation of the data requirements to the back-end team. Following that, the front-end team and the back-end team will discuss this description. The back-end team can then proceed to implement the data models for the API to contain the data in accordance with the information provided for them. The front-end team can then confirm whether the JSON data matches the description for the specific page after the back-end team has presented the JSON data to them.

Example:
The front-end team wants JSON data for the user's profile page. The data should contain the following about the user:

- Username
- Friend count
- Memory count
- Profile photo url

After discussing the details, the back-end team went and created the JSON data model for this data. They then present the following:



```
// JSON object for user details on the user profile page
{
   "username" : "@Someone",
   "friendCount" : 10,
   "memoryCount" : 10,
   "photoURL" : "https://a-photo-url.com"
}
```

Now both teams will discuss this JSON data to confirm if this is what the front-end team wanted. If it's not, they discuss the changes to be made, or else the front-end team can go and use this data to populate the page's contents.

JSON Data Response Object:
Pages:

- Other Users' profile
- Your Profile

Viewing a user’s profile:
The returned JSON object should contain the following:

- friends - the number of friends for the viewed user
- username - the viewed user's username
- profileImg - the link to the viewed user's profile photo
- memories - an object array of the viewed user's memories

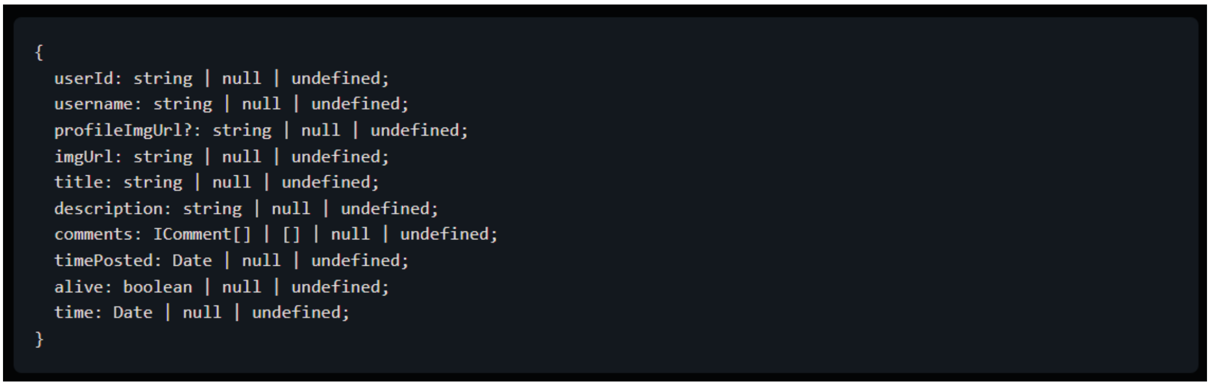
    - Title
    - Description
    - timePosted - the exact time when the user posted the memory in the following format: YYYY-MM-DD'T'hh:mm:ss.sssXXX where the XXX represents the timezone e.g. Z for GMT/UTC or it could be ±hh:mm For example: 2023-03-18T10:30:00.000-07:00 = 18 March 2023 at 10:30:00 AM (GMT -7)
    - imgUrl - a link to the memory's background image
    - comments - an object array of all the comments for a specific memory
        - username - the username of the user who commented
        - profileImageUrl - the link to the viewed user's profile photo
        - comment - the comment that the user made

Viewing Your Profile:
The returned JSON object should contain the following: (Note these properties follow the same description as above with extras to new properties)

- Friends
- Username
- profileImageUrl
- accountTime `- this will be used to display the current life time of your account.
- active memories
    - Title
    - Description
    - timePosted
    - imageUrl
    - Comments
        - Username
        - profileImageUrl
        - Comment
- dead memories `- these memories will only be used in the "Revive Memory" model
    - Title
    - Description

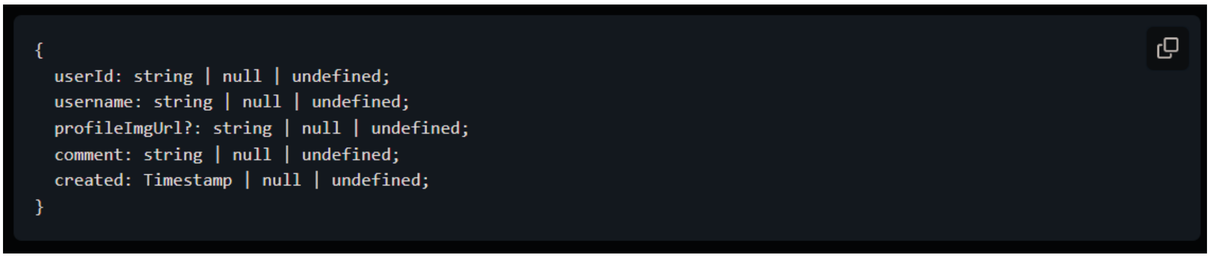
    - timePosted
    - imageUrl
    - Comments
        - Username
        - profileImageUrl
        - Comment
    - selected `- this property will be used to add color to the memory that is currently selected in the "Revive Memory" modal and to know which memory to revive once you click the "Revive" button.

IMemory:



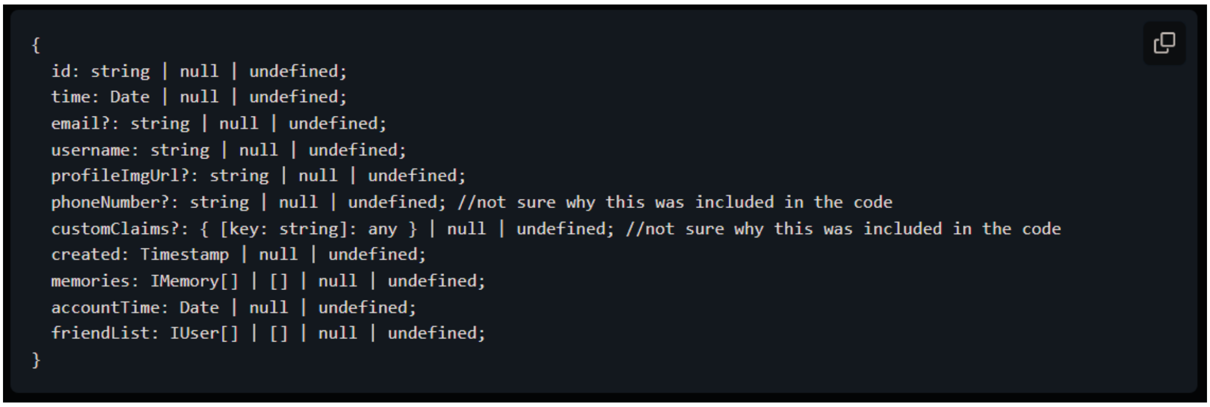
```
{
  userId: string | null | undefined;
  username: string | null | undefined;
  profileImgUrl?: string | null | undefined;
  imgUrl: string | null | undefined;
  title: string | null | undefined;
  description: string | null | undefined;
  comments: IComment[] | [] | null | undefined;
  timePosted: Date | null | undefined;
  alive: boolean | null | undefined;
  time: Date | null | undefined;
}
```

IComment:



```
{
  userId: string | null | undefined;
  username: string | null | undefined;
  profileImgUrl?: string | null | undefined;
  comment: string | null | undefined;
  created: Timestamp | null | undefined;
}
```

IUser:



```
{
  id: string | null | undefined;
  time: Date | null | undefined;
  email?: string | null | undefined;
  username: string | null | undefined;
  profileImgUrl?: string | null | undefined;
  phoneNumber?: string | null | undefined; //not sure why this was included in the code
  customClaims?: { [key: string]: any } | null | undefined; //not sure why this was included in the code
  created: Timestamp | null | undefined;
  memories: IMemory[] | [] | null | undefined;
  accountTime: Date | null | undefined;
  friendList: IUser[] | [] | null | undefined;
}
```

Running Firebase Emulators

Introduction:

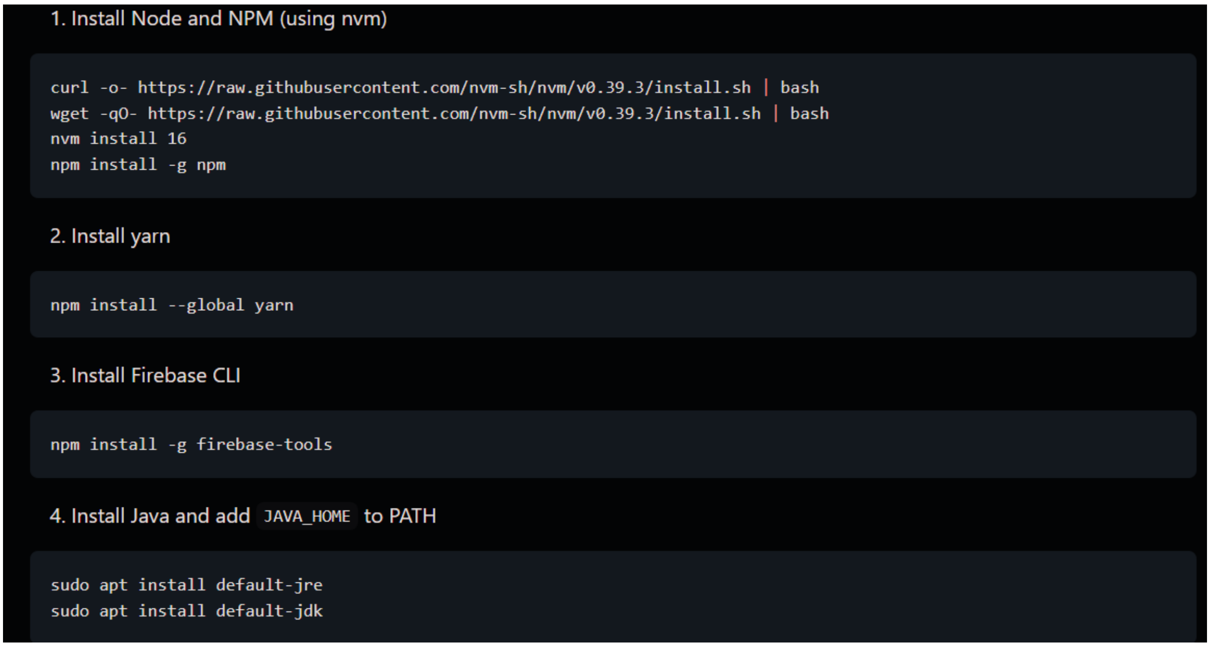
This manual is a work in progress, and revisions will be made as necessary. Given the different work environments, it can be challenging to develop an extensive manual that is appropriate for every team member. Please do not hesitate to contact the author for help if you run into any difficulties while completing the setup instructions.

Dependencies:

- Node 16
- Firebase CLI
- Java (JDK Version 11 or higher)

Installing Dependencies:

Choose the necessary steps from the following installation instructions, which were applied to a freshly installed version of WSL.



1. Install Node and NPM (using nvm)

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash
nvm install 16
npm install -g npm
```

2. Install yarn

```
npm install --global yarn
```

3. Install Firebase CLI

```
npm install -g firebase-tools
```

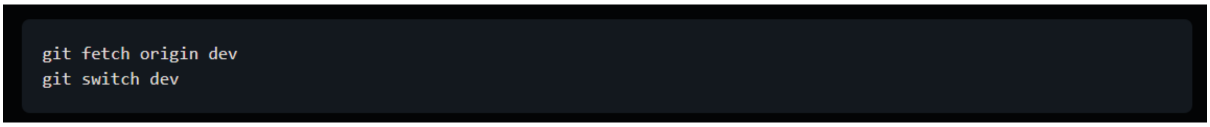
4. Install Java and add `JAVA_HOME` to PATH

```
sudo apt install default-jre
sudo apt install default-jdk
```

Running The App:

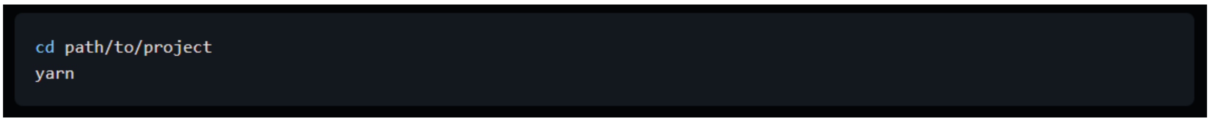
1. Clone the Repo.
   [Repo]

2. Make Sure You’re on the correct branch
   The default branch is the dev branch. Instead of pulling, which retrieves the remote branch and automatically merges it with your current branch, do a fetch and switch.



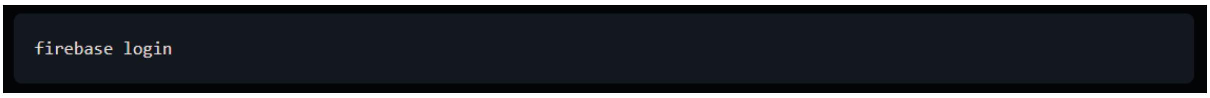
```
git fetch origin dev
git switch dev
```

3. Install dependencies



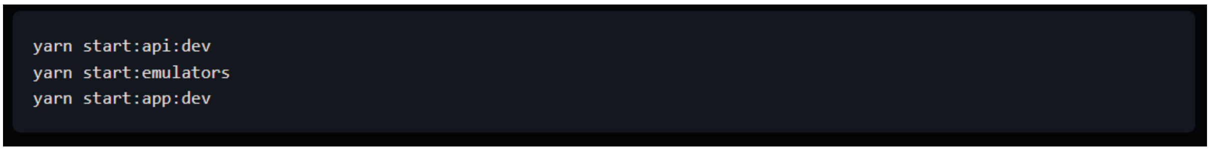
```
cd path/to/project
yarn
```

4. Login with Firebase
   Login with whatever Google account you like.



```
firebase login
```

5. Run the app stack
   Run the following commands in separate terminals:



```
yarn start:api:dev
yarn start:emulators
yarn start:app:dev
```

The commands will launch the Firebase emulators in demo mode without using any live Firebase resources. If the application attempts to interact with a Firebase resource, the local emulators will be employed instead.

We chose to use a local demo project for the following reasons:

- Team members don't need to be added and authenticated on the project portal.

- If something goes wrong during development, it won't affect live Firebase resources.
- Project credentials don't need to be stored in version control.

Various ways the Emulators can be used:

The information listed below will most likely be useful in the coming weeks. However, you can safely ignore this section for now.

Exporting the Emulator States:
While the emulators are running, you can export their states before closing them to save data, such as the current state of Firestore, for later use when restarting the emulators. To do this, run the command "yarn export:emulators in a separate terminal while the emulators are still running. This will create a ".emulators" folder in the project. Remember not to commit this folder; it is already added to the.gitignore file for your convenience.

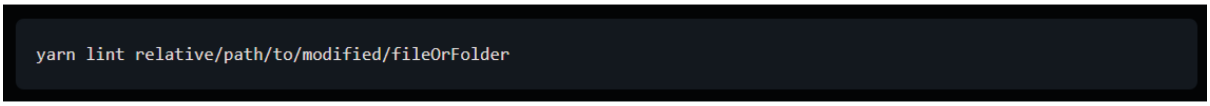
Importing the Emulator States:
If you want to restore the previous state of the emulators you were working on, run yarn start:emulators:imported instead of yarn start:emulators. If you prefer to start from a clean slate, use yarn start:emulators.

Seeding the Firestore database:
Run the emulators and seed the Firestore database with "yarn seed:emulators”. This adds dummy data to the database, which the app can use.

Linting using Eslint

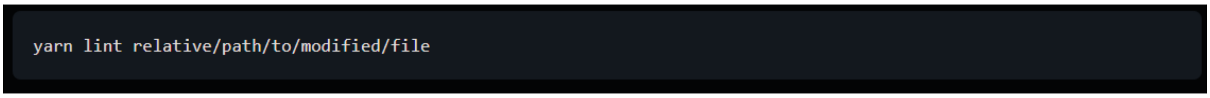
Guide for local linting:

1. Go to the folder/file you have modified
2. Copy the relative Path
3. Go to the root of the project and run



```
yarn lint relative/path/to/modified/fileOrFolder
```

4. The syntax will be fixed where possible, and detailed errors will be produced, which should be fixed
5. Fix remaining errors, if any
6. Test by running the command again



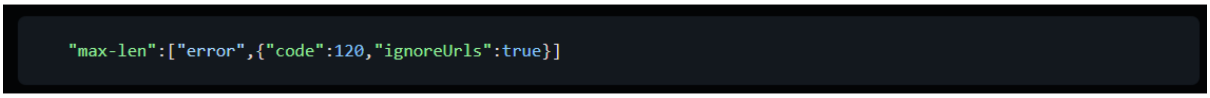
```
yarn lint relative/path/to/modified/file
```

Additional Info:

1. Eslint supports linting an entire folder, meaning you can lint an entire folder, if you paste its relative path
2. Use Git status to check all the files you have modified and ensure that they have been linted

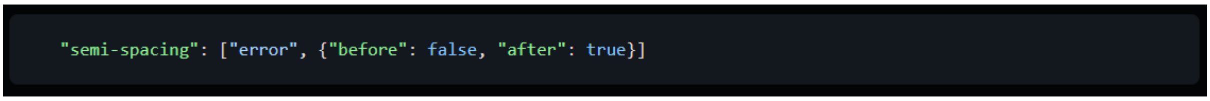
Currently Set-up rules

- Maximum length per line of code is set at 120 chars, excluding urls



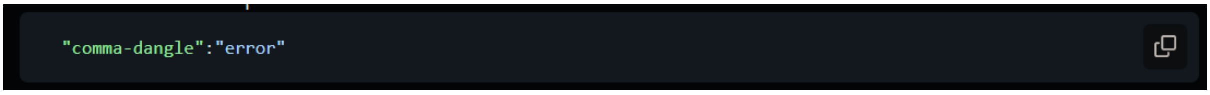
```
"max-len":["error",{"code":120,"ignoreUrls":true}]
```

- No spacing before a semicolon, spacing after it



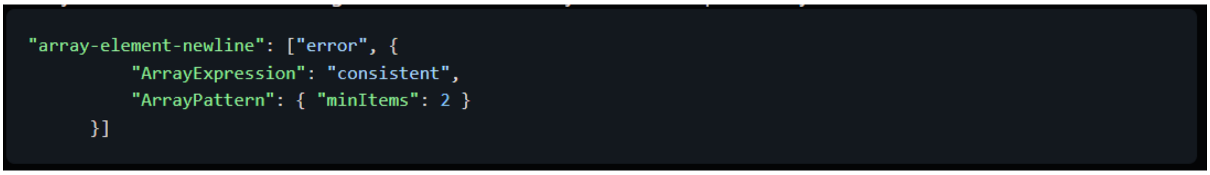
```
"semi-spacing": ["error", {"before": false, "after": true}]
```

- Commas that do not separate elements aren't allowed



```
"comma-dangle":"error"
```

- 2 array elements in one line during initialization, after, they should be separated by a new-line



```
"array-element-newline": ["error", {
          "ArrayExpression": "consistent",
          "ArrayPattern": { "minItems": 2 }
      }]
```

Important

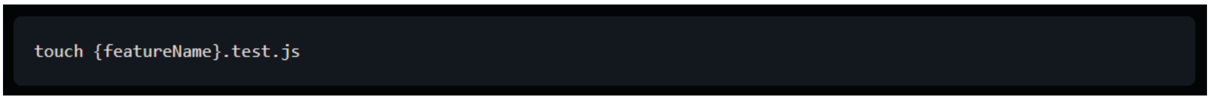
- There are many more rules applied than the ones listed above since we are extending from other standard rules

Testing Endpoints:

In the same folder as the src folder for cloud functions: libs/API/core/feature/test

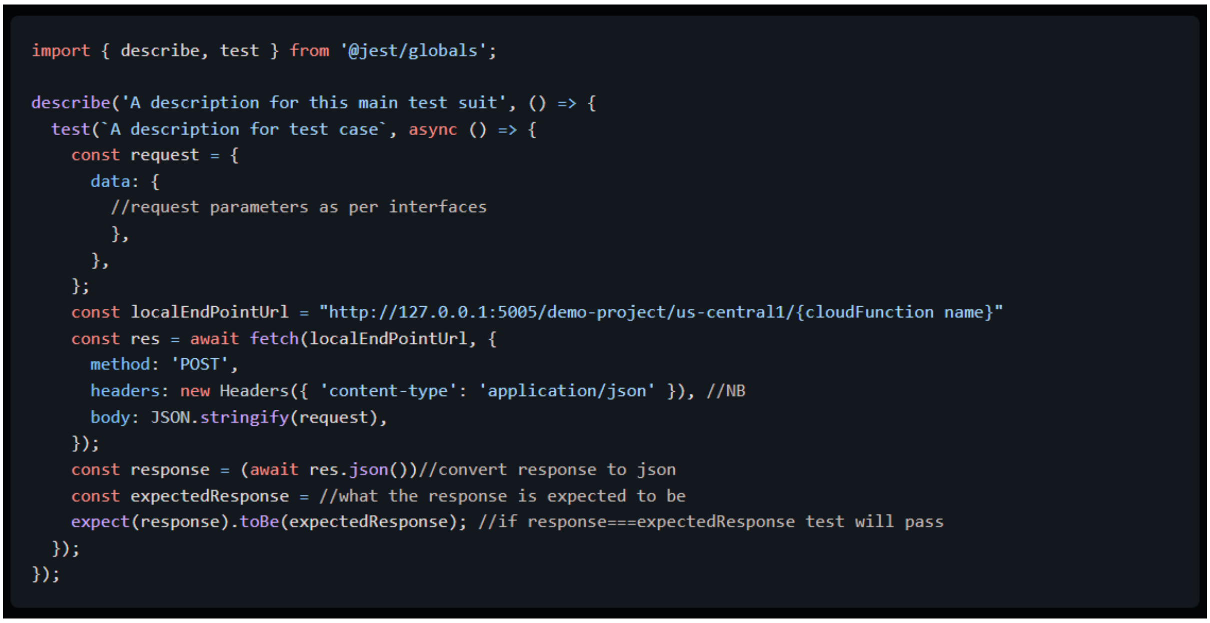
Create a test file

- Create a test file:



```
touch {featureName}.test.js
```

- Open the file and create a test case using the following structure:



```
import { describe, test } from '@jest/globals';

describe('A description for this main test suit', () => {
  test(`A description for test case`, async () => {
    const request = {
      data: {
        //request parameters as per interfaces
        },
      },
    };
    const localEndPointUrl = "http://127.0.0.1:5005/demo-project/us-central1/{cloudFunction name}"
    const res = await fetch(localEndPointUrl, {
      method: 'POST',
      headers: new Headers({ 'content-type': 'application/json' }), //NB
      body: JSON.stringify(request),
    });
    const response = (await res.json())//convert response to json
    const expectedResponse = //what the response is expected to be
    expect(response).toBe(expectedResponse); //if response===expectedResponse test will pass
  });
});
```

Important:

You can compare any values that are expected to be returned instead of the response as a whole. A working case can be seen on commit #80

App Interfaces

Overview:

This document will serve as the single source of truth for all established contracts. It is an ongoing collaboration between frontend, backend, and integration teams and must be regularly reviewed to guarantee that the contract specifications remain up-to-date and meet the requirements of both teams.

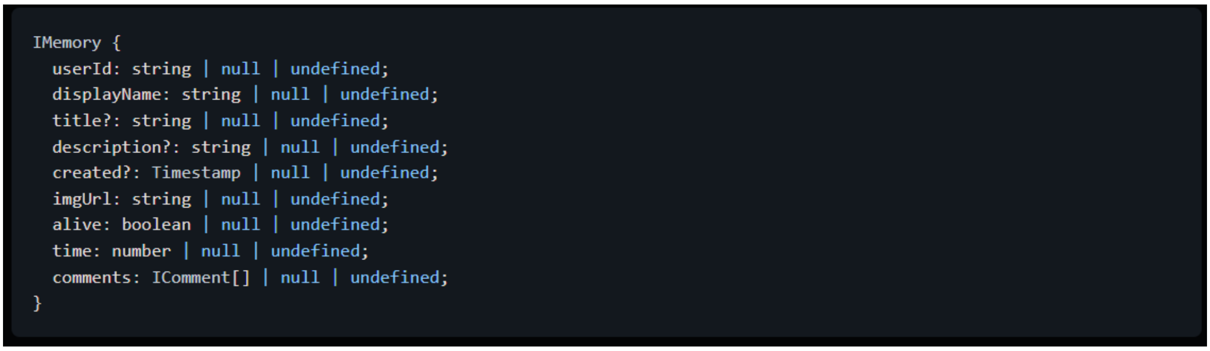
The interfaces to be used by both the backend and frontend will be defined in the libs/api directory.

- Interfaces are defined under libs/api/feature/util/interfaces
- Requests are defined under libs/api/feature/util/requests
- Responses are defined under libs/api/feature/util/responses

App Interfaces:

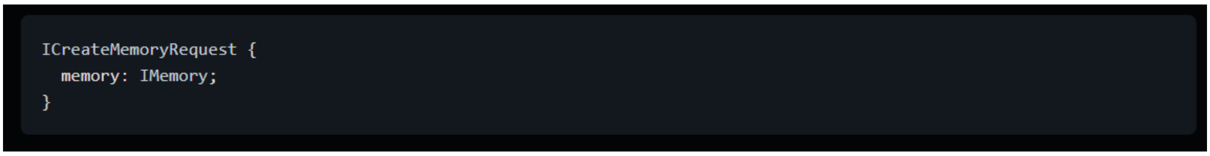
Memories:

Interfaces:



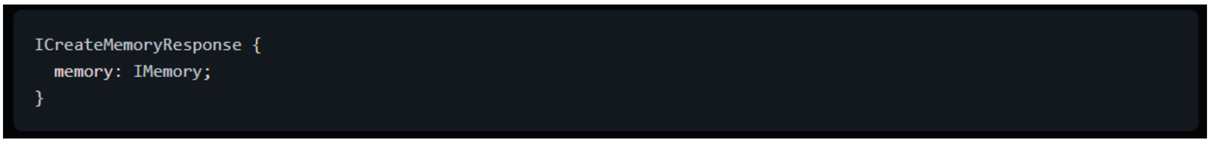
```
IMemory {
  userId: string | null | undefined;
  displayName: string | null | undefined;
  title?: string | null | undefined;
  description?: string | null | undefined;
  created?: Timestamp | null | undefined;
  imgUrl: string | null | undefined;
  alive: boolean | null | undefined;
  time: number | null | undefined;
  comments: IComment[] | null | undefined;
}
```

Requests:



```
ICreateMemoryRequest {
  memory: IMemory;
}
```

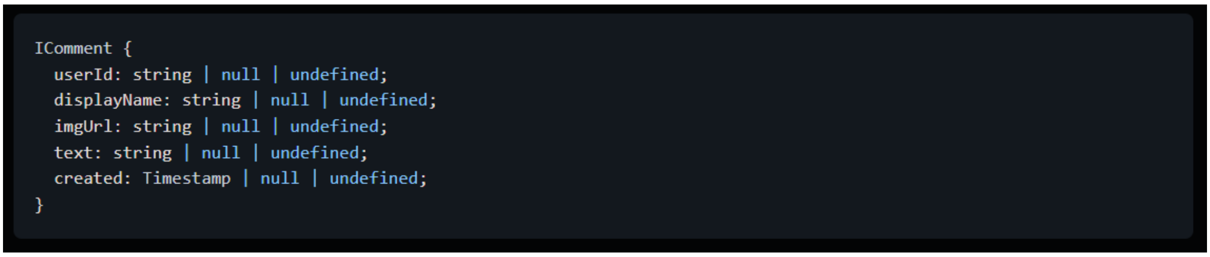
Responses:



```
ICreateMemoryResponse {
  memory: IMemory;
}
```

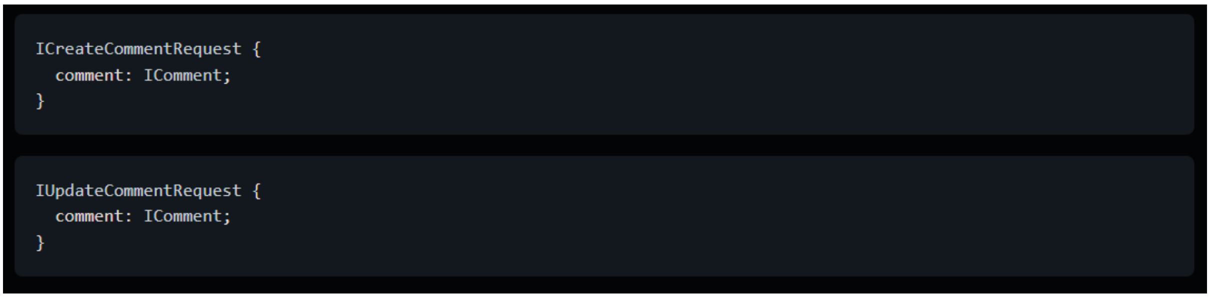
Comments:

Interfaces:



```
IComment {
  userId: string | null | undefined;
  displayName: string | null | undefined;
  imgUrl: string | null | undefined;
  text: string | null | undefined;
  created: Timestamp | null | undefined;
}
```

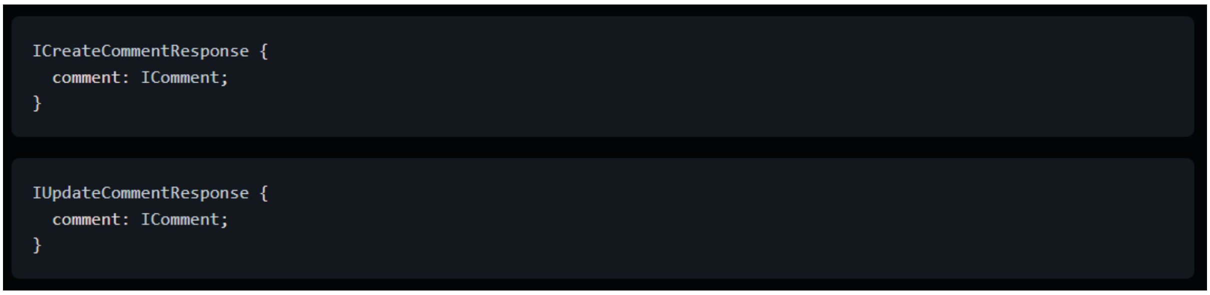
Requests:



```
ICreateCommentRequest {
  comment: IComment;
}
```

```
IUpdateCommentRequest {
  comment: IComment;
}
```

Responses:



```
ICreateCommentResponse {
  comment: IComment;
}
```

```
IUpdateCommentResponse {
  comment: IComment;
}
```

Testing:

Testing strategies:

1. Manual Testing: This involves testing the app manually, step by step, to identify any functional or usability issues.
2. Automated Testing: This involves using automated testing tools to test the app's performance, functionality, and security.
3. Exploratory Testing: This involves testing the app by exploring and navigating it as a user would, to identify any issues that may not be immediately obvious.
4. Regression Testing: This involves testing previously fixed bugs to ensure they have not recurred after new updates or changes have been made.

Tests to be run:

1. Functionality Testing: This involves testing the basic functionalities of the app such as logging in, signing up, creating a profile, posting content, commenting, and liking content, and more.
2. Usability Testing: This involves testing the app's ease of use, intuitiveness, navigation, and overall user experience.
3. Performance Testing: This involves testing the app's speed, responsiveness, and stability under normal and heavy loads.
4. Security Testing: This involves testing the app's security measures to ensure that user data is safe and secure.
5. Regression Testing: This involves testing previously fixed bugs to ensure they have not recurred after new updates or changes have been made.

Deployment:

Deployment Strategy: GitHub Actions

The deployment strategy works with GitHub Actions as follows.
When someone merges into the dev branch, the deployment is set up on Github, and GitHub Actions deploys the project live. It deploys the following parts:

- Hosting
- Cloud Functions
- Firebase Rules
- Cloud Storage Rules