# Route 42 Design Document

## Table of Contents



## Team

### Team Members and Roles

| UID | Name | Role |
|---|---|---|
| u7233149 | Kai Hirota | Full-Stack |
| u7269158 | John (Min Jae) Kim | Data Structure, Feature Testing |
| u7234659 | Honggic Oh | Search, Feature Testing |
| u7199021 | Theo Darmawan | Full-Stack |

### Meeting minutes

## Conflict Resolution Protocol

- Conflicts will be resolved through civil discussion and democratic voting process involving all parties interested in the matter.
    - For example, if someone wants to change the direction or the concept of the app, everyone must be involved in the decision-making. If someone wants to change a small class in the project, then that can be done either through voting, or by mutual agreement upon directly discussing with the person who created the class.

# Application Description

**Targets Users: Workout Enthusiasts**

Route42 is a social networking app for athletes of various levels. With Route42, users can:

1. Record workouts, including walking, running, and cycling.
2. Track performance metrics and see the recorded workouts in an interactive map.
3. Follow other users, and view and like other people's workouts.
4. Search for posts by username, hashtags, and proximity to the user's location

TODO add screenshots

## Use Case Example

1. Athletic Activity Tracking and Sharing
    1. Michael runs at ANU running club, and wants to record and share his daily runs.
    2. Once ready, he starts the `run` activity on the app
    3. The app tracks Michael's location and route, and display it on a map. Performance metrics are displayed in real time.
    4. After finishing his run, Michael ends the `run` activity on the app.
    5. The app will display a post template for sharing the completed activity.
    6. Michael may write a description and add hashtags like `#ANUrunning` before sharing it. The app will extract the hashtags and tag the post for you.
    7. The created post can immediately be viewed by other users on their feeds.
    8. Michael can also select his past posts or posts created by others and see an interactive map of the route associated with the post.
2. Social networking and searching
    1. Emily is an avid runner who recently started competing in marathons. Emily wants to connect with other aspiring athletes.
    2. Emily can search for posts on Route42 app by username and hashtags, and look at other athletes and their workouts and routes.
    3. Emily can also search for posts by geographical proximity, and the app will visualize the places where others logged their workouts on an interactive map.

### Scheduled Actions

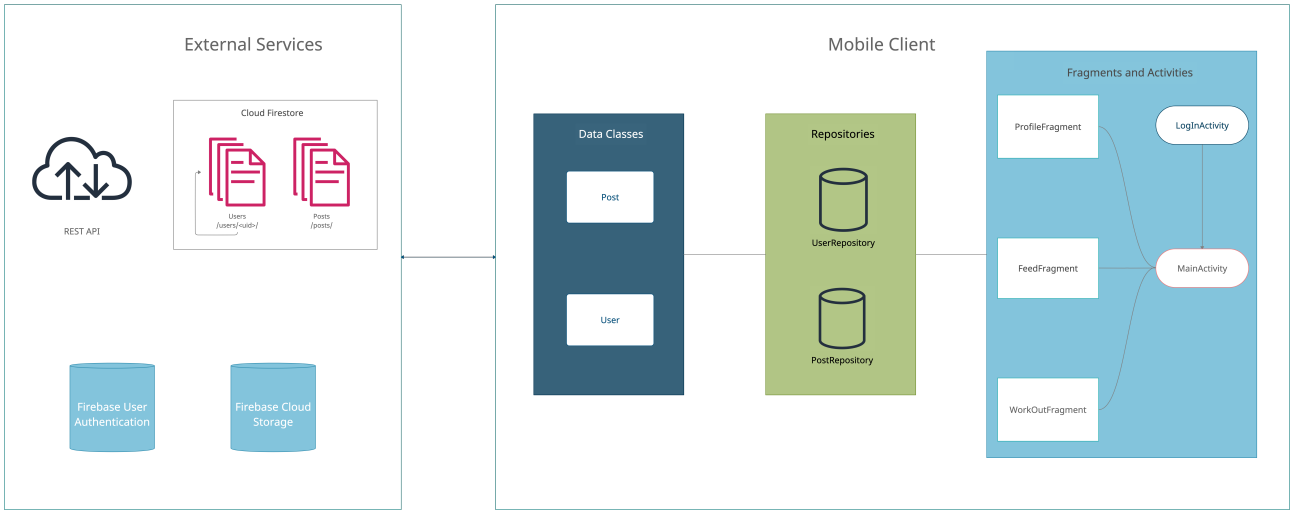
If the user does not have an active internet connection, the app allows scheduling of posts and likes.
To schedule a post, the user checks the `schedule` button and selects the time delay.
To schedule a like, the user long-clicks the like button and selects the time delay.

### Pausing a workout

If the user needs to pause the workout, they can manually do so. Otherwise, navigating away from the `Activity` screen will automatically pause it for them.

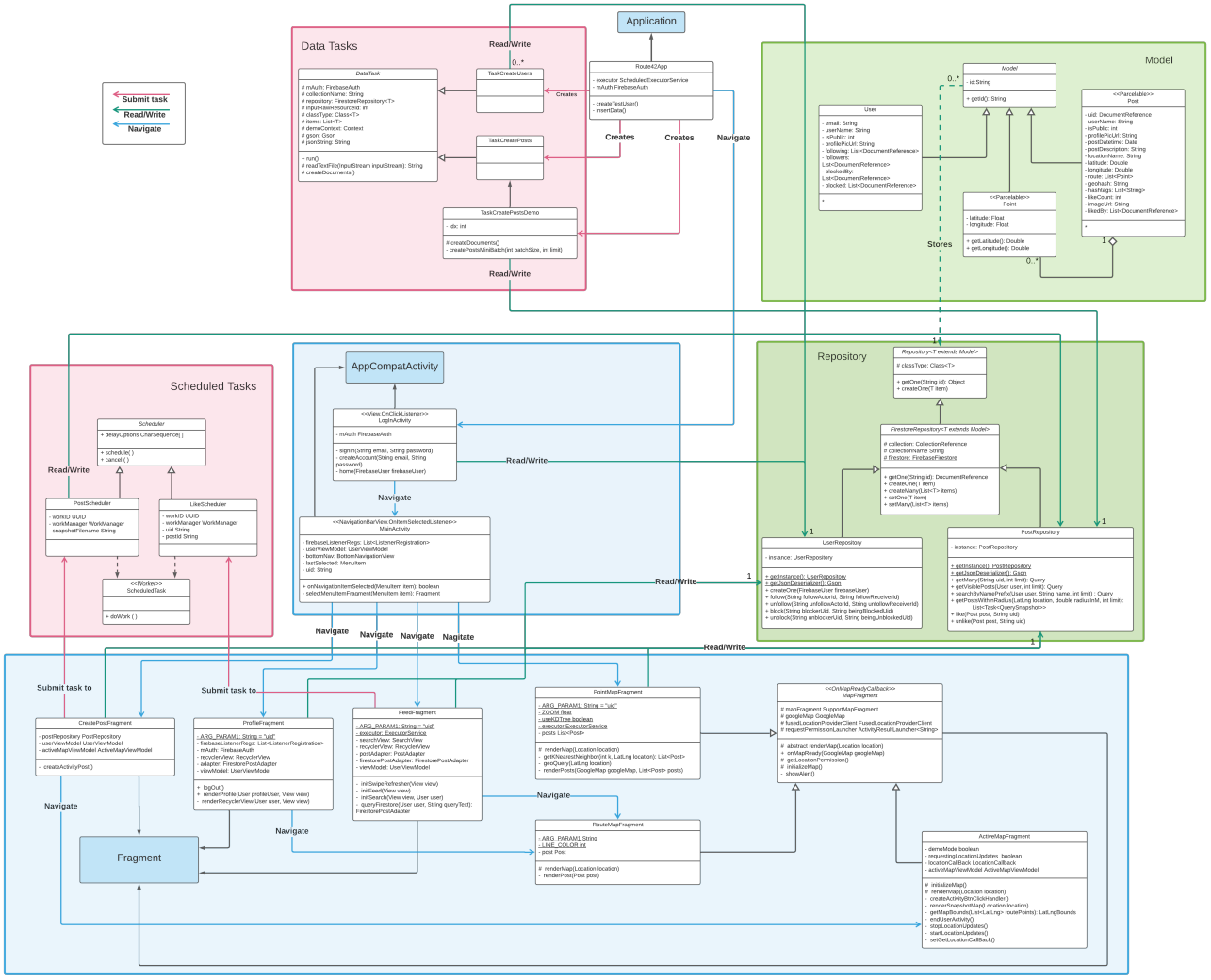
# Diagrams

## Architecture



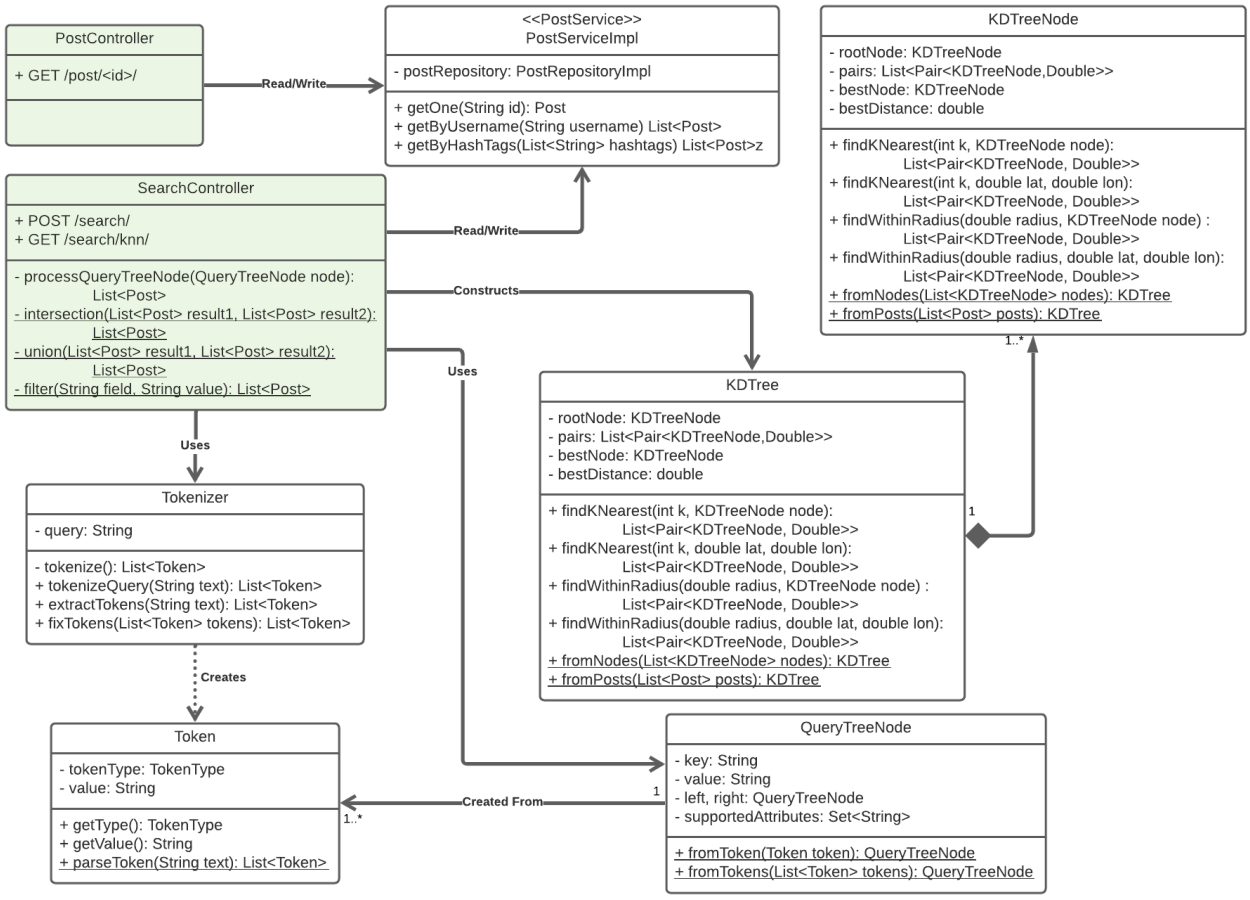

Link

## Mobile App




[Link]

# REST API




Link

# Design Decisions

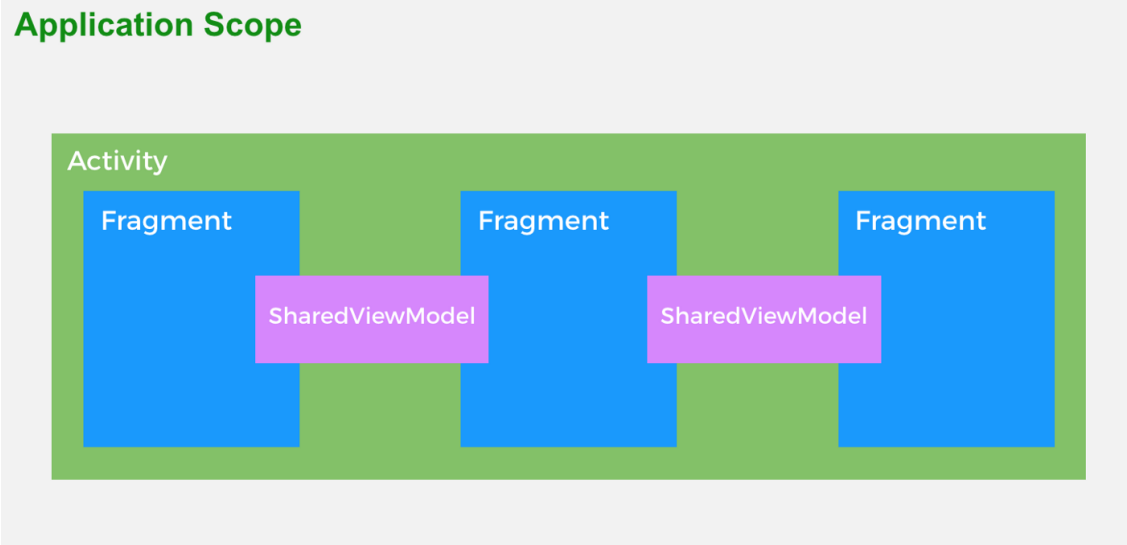
## Data Structures

- KD Tree
    - Where: REST API `GET /search/knn` with `k`, `lon`, `lat` parameters.
    - Why: KD Tree (K-dimensional tree) is used to store and search 2-D data of location (longitude, latitude). KD tree is useful for finding nearest neighbors and performing range search based on multiple dimensions of data - such as longitude and latitude.
- HashMap
    - Where: Used by the REST API for union and intersection operations between lists of `Posts`.
    - Why: It's the most efficient way of finding set union and intersections. This is used to chain the left and right results when executing commands in `QuerySyntaxTree`.
- Binary Tree
    - Where:
        - REST API `POST /search/` endpoint uses `QuerySyntaxTree` to process the query text sent by a client.
        - `QueryTreeNode` is used to extract the hierarchical structure of nodes, each representing a binary operator and two expressions.

    - Why:
        - It allows efficient parsing of tokens, and allows us to express various operations in a recursive manner, making the code easy to read and maintain.

## Design Patterns

- Singleton & Repository
    - Where: `UserRepository` and `PostRepository` classes under `repository` submodule.
    - Why:
        - Singleton pattern prevents unnecessary creation of multiple instances of connections with the database. By using singleton, the program uses less memory, and managing the connection with the database is easier.
        - Repository pattern abstracts the database operations and allows decoupling of the database access logic from the application logic.
- Single-activity architecture




    - Source
    - What: Composition of Android application based on single or a couple activities, each managing one or more fragments.
    - Where: Entire application.
    - Why:
        - Route42 primarily has one activity called `MainActivity` which contains a fragment container view, which swaps between fragments based on user interactions.
        - Usage of this architecture reduces lines of code required for the whole app, while making it easier to prototype new features.
- REST API
    - Where: In the cloud (AWS EC2 instance)
    - Why:
        - When using Cloud Firestore Android SDK, we have some limitations.
            - Cannot perform partial text search - for example, we cannot query on substring of a text

field.
        - Cannot use more than one `arrayContains` in a single query.
        - No support for boolean OR operation between multiple filters.
        - Every CRUD operation must be asynchronous in order to not freeze up the UI thread.
    - Using the REST API allows us to use the Firebase-admin SDK, which gives the REST API higher privilege and more capability than the mobile client.
    - Using REST API allows us to simplify database reads and writes. The downside is that we sacrifice Firestore's document listener feature, where we can listen to updates on documents of interest.

- ViewModel
    - What: An intermediate observable class which enables data to persist independent of fragment lifecycle and attaching listeners to data changes.
    - Where: `ActiveMapViewModel` and `UserViewModel`
    - Why: By storing data in a view model class, data is not deleted when views are destroyed (e.g. when the user navigates to another page, or when the phone is rotated). Also, by listening to changes to `LiveData` members of the view model, views can update directly to changes in persistent data stored in Firebase, through listening to the `LiveData` class. This improves separation of UI layer from the data layer, as the UI is not dependent on any repository classes.
- Multi-threading / background execution
    - Where: `PhotoMapFragment`, `ScheduleablePost` , `SchedulableLike`
    - Why: When making the REST API call to `search/knn`, the communication is handled by a background worker thread. This ensures the UI thread (the main thread) does not freeze and remains responsive.
      Scheduled actions involving IO operations and network calls are also handled in the background to minimize load on the UI thread.

## Grammars

*Production Rules*

<Term> ::=  <Factor> | <Term> + <Term> | <Term> + <Connector>

<Factor> ::= <Keyword> | <bracket>

<Connector> ::= <and> | <or>

*[How do you design the grammar? What are the advantages of your designs?]*

- Our grammar is simple and obvious. Dividing input data into factors and connectors to understand it easily.
- This is using for search data so mostly input data is keyword and need to classify it to bracket and connectors

*If there are several grammars, list them all under this section and what they relate to.*

### Tokenizer and Parsers

*[Where do you use tokenisers and parsers? How are they built? What are the advantages of the designs?]*

Every token either contains an operator and two expressions, or a key and value.
Tokens are extracted by prioritizing parenthesis, and then extracting from left to right.
Example 1 below is an example of a token that has the key `hashtags` and value `["test"]`.
For example, if a query consists of 10 hashtags chained by OR, then the resulting `QuerySyntaxTree` will be equivalent to a linked list, where each node only has a right child.

```
EXAMPLES
1. "test" → {hashtags: ["test"]}

2. "username: xxx hashtags: #hashtag #android #app" →
{OR: [
    {userName: "xxx"},
    {hashtags: ["#hashtag", "#android", "#app"]}
  ]
}

3. "(username: xxxx or hashtags: #hashtag #android #app) and username: yyy" →
 {AND: [
     {OR: [
         {userName: "xxx"},
         {hashtags: ["#hashtag", "#android", "#app"]}
     ]},
     {userName: "yyy"}
 ]}

```

### Other

*[What other design decisions have you made which you feel are relevant? Feel free to separate these into their own subheadings.]*

# Summary of Known Errors and Bugs

*[Where are the known errors and bugs? What consequences might they lead to?]*

1. Search functionality can handle partially valid and invalid search queries. (medium)
2. *Bug 1:*

- *A space bar (' ') in the sign in email will crash the application.*
- ...

2. *Bug 2:*
3. ...

*List all the known errors and bugs here. If we find bugs/errors that your team do not know of, it shows that your testing is not through.*

# Testing Summary

TODO: Why not just link to test coverage report?

- Number of test cases: 13
    - UI Tests : 15
    - Unit Tests : ??(update later)
- Types of tests created: ...

| UI/Unit Tests | Class Name | Test Description | Code Coverage | Numbers of Tests |
|---|---|---|---|---|
| UI | LoginTest | • Check login with correct and wrong id and password | N/A | 2 |
| UI | MainTest | • Switch the page throughout navigation bar<br>• create posts and check the post is properly made<br>• cancel making a post<br>• making a schduled post<br>• like and unlike the post<br>• block and unblock user<br>• follow and unfollow user<br>• block following user<br>• follow blocked user | N/A | #of tests |
| Unit | KNearestNeighbourServiceTest | • Test call method of rest-api service post<br>• Test printing Strings | 70% | 2 |
| Unit | QueryStringTest | • Test query is properly made | 100% | 2 |
| Unit | SearchServiceTest | • Test search input data is properly made to the query<br>• Calling query | 100% | 1 |
| Unit | CryptoTest | • Check Encryption | 100% | 3 |
| Unit | UserListAdapterTest | • ? | ? | ? |
| Unit | BaseActivityTest | • Check factors | 100% | 1 |
| Unit | PointTest | • Check latitude and longitude | 75% | 4 |
| Unit | UserTest | • Check factors | 92% | 7 |
| Unit | SchedulableTest | • ? | ? | ? |
| Unit | UserViewModelTest | • Tests method of getUser and setUser | 75% | 2 |
| Unit | PostTest | • Check factors<br>• Check extracting hashtags from input text | 86% | 17 |
| Unit | ActiveMapViewModelTest | • Check activity data and types<br>• Check elapsed time<br>• Check last update time<br>• Check reset function<br>• Check pastlocation<br>• Check snapshot file name | 73% | 7 |

*Please provide some screenshots of your testing summary, showing the achieved testing coverage. Feel free to provide further details on your tests.*

# Implemented Features

- Easy: 6
- Medium: 6
- Hard: 1
- Very Hard: 1

Improved Search

1. Search functionality can handle partially valid and invalid search queries. (medium)

UI Design and Testing

1. UI tests using espresso or similar. Please note that your tests must be of reasonable quality. (For UI testing, you may use something such as espresso) (hard)

Greater Data Usage, Handling and Sophistication

1. Read data instances from multiple local files in different formats (JSON, XML or Bespoken). (easy)
2. User profile activity containing a media file (image, animation (e.g. gif), video). (easy)
3. Use GPS information. (easy)
4. User statistics. Provide users with the ability to see a report of total views, total followers, total posts, total likes, in a graphical manner. (medium)

User Interactivity

1. The ability to micro-interact with 'posts' (e.g. like, report, etc.) [stored in-memory]. (easy)
2. The ability for users to ‘follow’ other users. There must be an adjustment to either the user’s timeline in relation to their following users or a section specifically dedicated to posts by followed users. [stored in-memory] (medium)
3. Scheduled actions. At least two different types of actions must be schedulable. For example, a user can schedule a post, a like, a follow, a comment, etc. (medium)

User Privacy

1. Privacy II: A user can only see a profile that is Public (consider that there are at least two types of profiles: public and private). (easy)

Peer to Peer Messaging

1. Privacy I: provide users with the ability to ‘block’ users. Preventing them from directly messaging them. (medium)

Firebase Integration

1. Use Firebase to implement user Authentication/Authorisation. (easy)

2. Use Firebase to persist all data used in your app (this item replace the requirement to retrieve data from a local file) (medium)
3. Using Firebase or another remote database to store user posts and having a user's timeline update as the remote database is updated without restarting the application. E.g. User A makes a post, user B on a separate instance of the application sees user A's post appear on their timeline without restarting their application. (very hard)