System Design Document

for

KeepFit

University of Southern California
Spring 2021

Team 12

Aaron Ly
5110281976

Sajan Gutta
2725022497

Roddur Dasgupta
2659572597

Max Friedman
9342195947

Victor Udobong
9031466326

Devin Mui
2587725548

TABLE OF CONTENTS

1. Preface

This document is intended to provide a detailed design description of the architecture for the first release of the KeepFit application. This application is being developed for the benefit of the customer, Tian Xie, who has requested the implementation of the product whose architecture is contained in this document. This document is specifically oriented for readership by the eventual developers of the system. It maintains a technical focus and thus is designed to be read by people with prior software knowledge. Additionally, any other customers interested in the KeepFit application and in working with the main customer will be able to read this document to gain a greater understanding of the architecture. Please note a critical change that has been made to the application's design and is detailed in the section below, after this preface.

Following this preface, readers will be able to find an introduction to the KeepFit and why it is needed. This will be followed by high-level diagrams and descriptions of the major architectural design patterns and styles used to make KeepFit a reality. Afterwards, we will provide a thorough diagram of packages used in KeepFit and class diagrams for KeepFit's database. There will also be lower-level diagrams for the smaller systems within KeepFit such as the system powering the live streaming feature. Finally, we provide detailed sequence diagrams to depict the interactions and events that should happen within KeepFit to create the envisioned user experience.

NOTE: Redlined text indicates changes to the design, blue text indicates additions to the documents, and blue bordered diagrams indicate altered diagrams.

1.1 Change Notes

We have made one major change to implementation plans for KeepFit. We will no longer be planning to use the Youtube API for handling user video uploads. Upon deeper investigation, we anticipate that we may run into issues with Google forcing the uploaded videos to be private until we've undergone an audit of our application. We believe this audit could cause issues in meeting the timeframe required by our customer. The videos being private will undermine our application's ability to embed the uploaded videos in the UI for users to view them.

Instead, we plan to use ~~Amazon Web Services Simple Storage Service (AWS S3)~~ Firebase Firestore for video hosting. This will allow us to have minimal storage/hosting costs. Users will still be able to select videos from their native device and upload them. Essentially, the user experience is the exact same as before. The only change is on the backend where we are using a different hosting service. ~~S3~~ Firebase will also use its content delivery network (CDN) to give us unique URLs from which to embed our videos, similar to how YouTube would operate. ~~We also believe that using our own storage bucket through S3 is a much more robust and production-ready strategy which will improve application performance and leave the customer satisfied.~~

2. Introduction

KeepFit exists to solve the problem caused by the COVID-19 pandemic, mainly an inability to access facilities for maintaining fitness such as the gym. The application offers an extensive set of features allowing users to learn about working out, share their progress, track their progress and more. The end goal of our application is to provide a convenient way for users to "keep fit" during these challenging times.

Account creation on KeepFit will be implemented using Google One Tap, React Native, and Firebase. Google One Tap will provide a convenient way for users to sign-in to our app, while React Native will be used to design an account setup form where users can enter fitness information – including weight, height, and fitness level – to be stored on Firebase, where it can be added to and retrieved from.

Uploading a pre-recorded workout on KeepFit will be implemented using ~~S3 and~~ React Native. Users will create videos on their native device (iOS or Android) in .mp4 format and upload it through our app, where it will be stored on the ~~S3~~ Firebase Firestore platform. React Native will manage pushing and pulling videos from ~~S3~~ Firebase for users to access.

Starting a live workout on KeeptFit will be implemented using React Native, Firebase, and the Zoom API. We will use React Native to make API calls to Zoom in order to create a meeting link. This meeting link will be stored on Firebase and displayed to users where they can workout live. At the completion of the live workout, the link will be removed from Firebase.

Searching and viewing users and pre-recorded videos will be implemented using React Native, ~~S3,~~ and Firebase. React Native will be used to design the UI, specified in the Software Requirements Specification (SRS), while ~~S3 and~~ Firebase will be used to access pre-recorded videos and users, respectively.

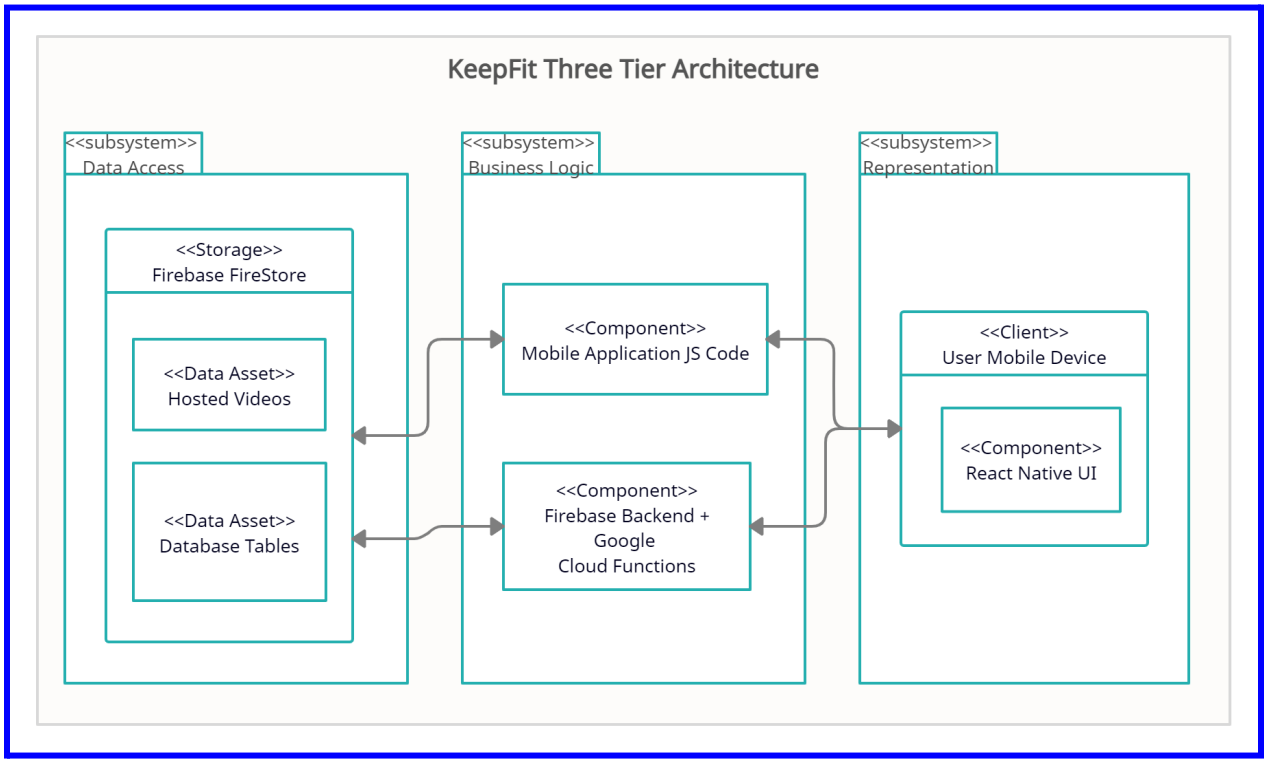
A tracking view of workout information will be implemented with React Native and allows users to select a workout type, intensity level, and start a timer. At the end of the workout, users can see the time they spent and the calories they burned. These workouts are saved on Firebase as their workout history when completed. A profile page will allow users to view and edit their personal information, view saved exercises, and view their workout history.

The complete set of features provided in KeepFit align with the customer organization's objective of helping others keep fit during the current world situation. While people are stuck home, this application will make it easy to learn exercises, track exercises, and maintain connection with others on the same journey. The architectural design that will make this happen now follows.

3. Architectural Design

In this section, we outline the major architectural **pattern** that dictates the design of the KeepFit application. We then detail a few of the major architectural **styles** that help to implement this pattern. These styles influence the practices employed while developing the entire application architecture and will be evident in many different situations. We use this section to contextualize the pattern and styles within the design of this specific application.

3.1 3-tier Architecture Pattern



KeepFit will utilize a 3 tier architectural design pattern, with layers for representation to the user, application/business logic, and a layer for data storage and access. The diagram above helps to depict this architecture. The user will see the Representation layer, which is primarily created using a React Native frontend, which should allow the application to be functional on both Android and iOS. KeepFit will be developed with best React Native practices and styled components to represent content to the user.

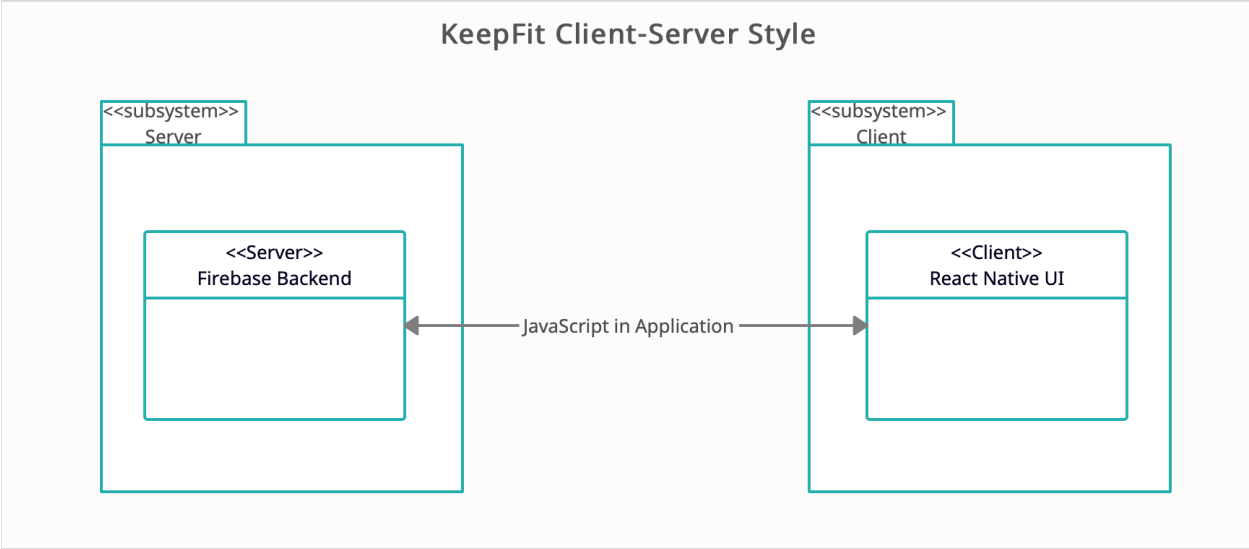
There will also be a layer containing the business logic. This will help to execute the necessary activities for KeepFit to function, and adds true functionality to the representation layer. The business logic takes user inputs and executes the necessary actions, modifying and accessing

KeepFit's stored data as necessary. The business logic layer will also deliver contact to the frontend to be represented to the user. The business logic is primarily made up of JavaScript (JS) code built into the packaged React Native application. Thus, the user's mobile device will also help to run some of the application's business logic as well. KeepFit will also have a backend built in Firebase, with cloud functions that allow modification and retrieval of critical KeepFit data such as user information, workout history, exercise options, video links, and more.

Finally, KeepFit has a data access and storage layer consisting of ~~an AWS S3 bucket and~~ a Firebase Firestore NoSQL database. The ~~Amazon S3 bucket~~ Firebase Firestore will be used to store hosted videos uploaded by KeepFit users. The unique URLs for these videos will be used to render videos in the representation layer. The Firebase database will hold all other data including user account information, workout information, and much more. Both of these components within the storage layer will be accessible by the application's business logic.

To give an example of how this architecture is robust, consider the example of a user uploading a video. A user selects and uploads a video using the UI on the representation layer. The JavaScript in the React Native application will then interact from the business logic layer with the ~~S3~~ Firestore interface. The application JS logic will upload videos directly to ~~S3~~ Firestore, retrieve a link for the uploaded content, and then utilize the Google Cloud functions to store this video link information in the Firebase database. Now, the links from the database can be fetched from the database by the business logic and used to render the videos in the representation layer for users to view them. In this way, the 3 tiers of this architectural pattern work seamlessly together and create a robust software system for the KeepFit application.

3.2 Client-Server Style



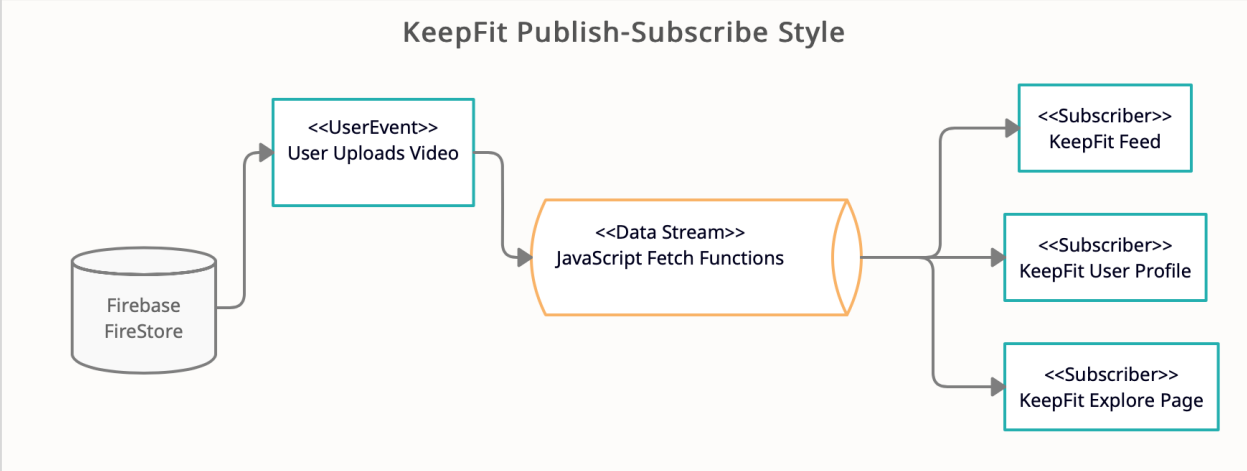
Within the overall 3-tier architectural pattern employed by KeepFit, the client-server style will be employed. This is apparent in numerous aspects of the application and one such occurrence is diagrammed above. In a client-server system, a client layer exists with the primary purpose of displaying information to users. It requests this information from a server layer, which handles requests, performs operations, and returns the requested information back to the client.

In the client-server diagram for KeepFit above, the client is the React Native application running on a user's device. This application displays relevant information to the user and must request it from a remote server. The diagram depicts our Firebase backend as the main server. This Firebase backend will contain most KeepFit data and will be the main place from which data is requested. The backend will provide various operations for manipulating data and retrieving it. The client will then show this information to the user.

KeepFit also has this client-server style when using the ~~S3~~ Firestore component. The ~~S3 storage bucket~~ Firestore is another example of a server with operations and information for KeepFit. The client application will upload and request content from the ~~S3 storage bucket~~ Firestore to be displayed on the frontend. ~~S3~~ Firestore handles storing the content, creating unique resources links for each data asset, and servicing client requests.

One final case of client-server style within KeepFit entails using external APIs. Again, the client will be the React Native application. However, in this case it will be sending external requests such as to the Zoom API. These requests will be serviced by a remote server managed by Zoom, and will yield information needed by our client. As you can see, the client-server architectural style plays a huge role in the overall KeepFit architecture and is instrumental in our overall design philosophy.

3.3 Publish-Subscribe Style



The KeepFit app architecture will use Publish-Subscribe style in conjunction with the client-server style. This style allows us to broadcast messages to different parts of the application asynchronously. This way, if a user takes an action that impacts other users or their instances of the application, we are able to update the information displayed instantly and efficiently.

Most often, this will be used when a user either creates new content (uploads a video, creates a livestream, etc.) or interacts with another user's content (follows them, saves an exercise, etc.). In these instances, what will occur is when the user performs an action, the client will make a call to the database to upload/update certain data. At this point, the database will contain the most updated version of the information. Then, the next time the user loads a page that displays any of this content, the new information will be displayed.

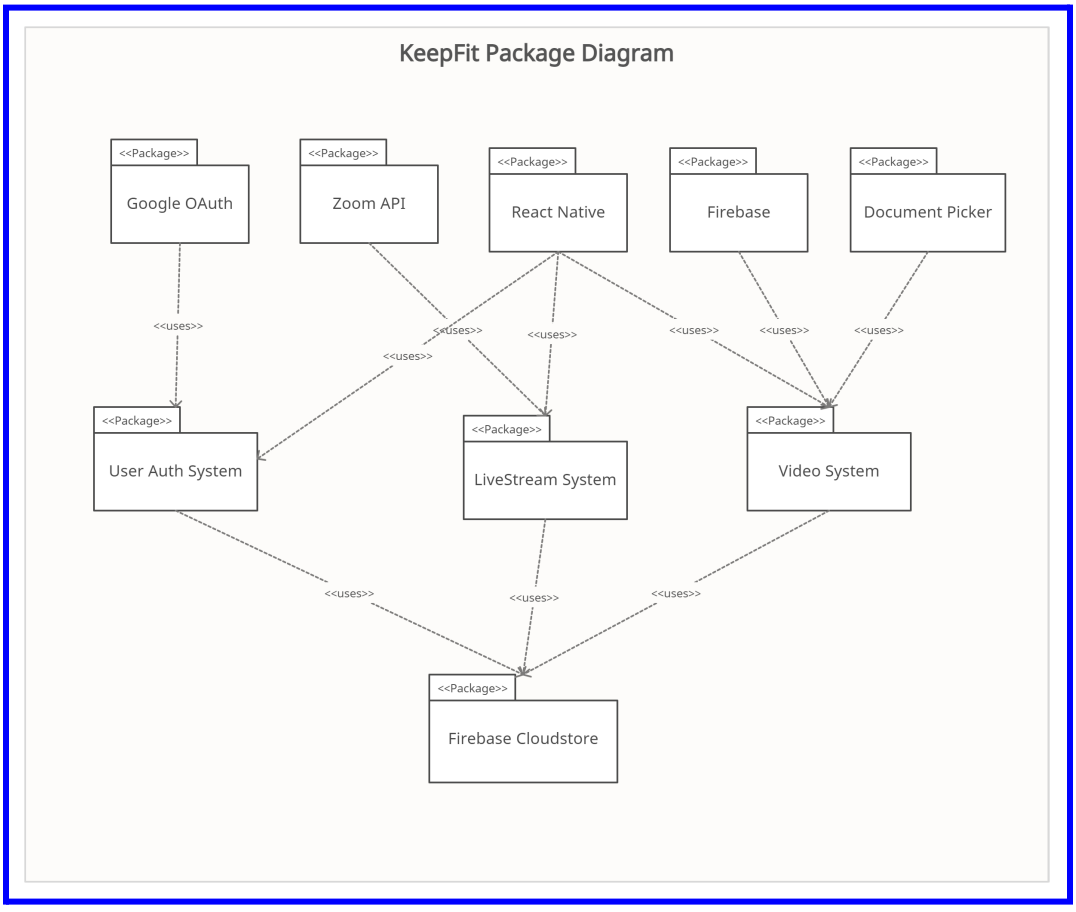
We chose to use the Publish-Subscribe model, because we do not know when or where the new content generated by a user will need to be displayed elsewhere. This way, our user serves as the publisher and certain elements of other user's UI are the subscribers. The publisher publishes data, then the subscriber that is waiting for changes is notified of the new data. In between these two, our JavaScript fetch functions will serve as a channel to route all of the correct information and data to the corresponding subscriber.

4. Detailed Design

Now that we've reviewed the high-level architectural pattern and styles used in KeepFit, we will take a close look at the specifics for implementing the features it has. Specifically, we will provide a package diagram to show the required dependencies. We will also provide a detailed diagram of the classes used to build the application, and UML depictions of each subsystem related to the functional requirements in the application's software requirements specification. We finish up with multiple sequence diagrams, which depict how our desired user stories will play out within the context of the application's architecture.

4.1 Package Diagram

Below you will find a diagram of the packages included in the KeepFit application. This combines specific groups of functionality or services into packages. It then shows which packages use each other, giving developers a picture of the dependencies involved in the KeepFit architecture.



4.2 Class Diagram

This class diagram is based on the database schema for KeepFit, which is included at the end of this document for reference. It was also provided in the KeepFit requirements specification. Each database table will have a corresponding class in our application for easy object-oriented manipulation. We go more in depth here about the functions each class will have to demonstrate their functionalities.

While it is not delineated in the diagram, all database table classes will inherit from a base database model and will have a name attribute that corresponds to their ref name in Firebase. Then, the base model can have some universal functions including one to return the Firebase table ref for that model. This will make query building much easier in the application logic. For example, we will be able to write something like User.where("full_name", "==", "Sajan") for our queries.

Note the relationships between each class. In many cases, for foreign keys, we store the ids of the reference items in their respective tables. For example, if trying to indicate that a user has saved a video, we create a SavedVideo object for this relationship. If the user_id is 25, that means this represents a video saved by the user with unique id 25.

The diagram also has multiple "enumerations." These are used to enforce a set of predetermined choices for specific characteristics and fields in the database. For example, fitness_level has a set of 4 choices in the FitnessLevel enumeration. Each choice has a value with all capital letters that is used to access it in the backend and also has a corresponding string or "display" value, which is typically used on the frontend.

Finally, there is an interface for operating the Firebase database, which can be used to initialize and close connections to the database. It will also provide methods for executing queries that can be reused in multiple parts of the code. This will be a key gateway between the client and server and will be utilized by multiple of the methods for the other classes.

You can find the class diagram described here on the next page.

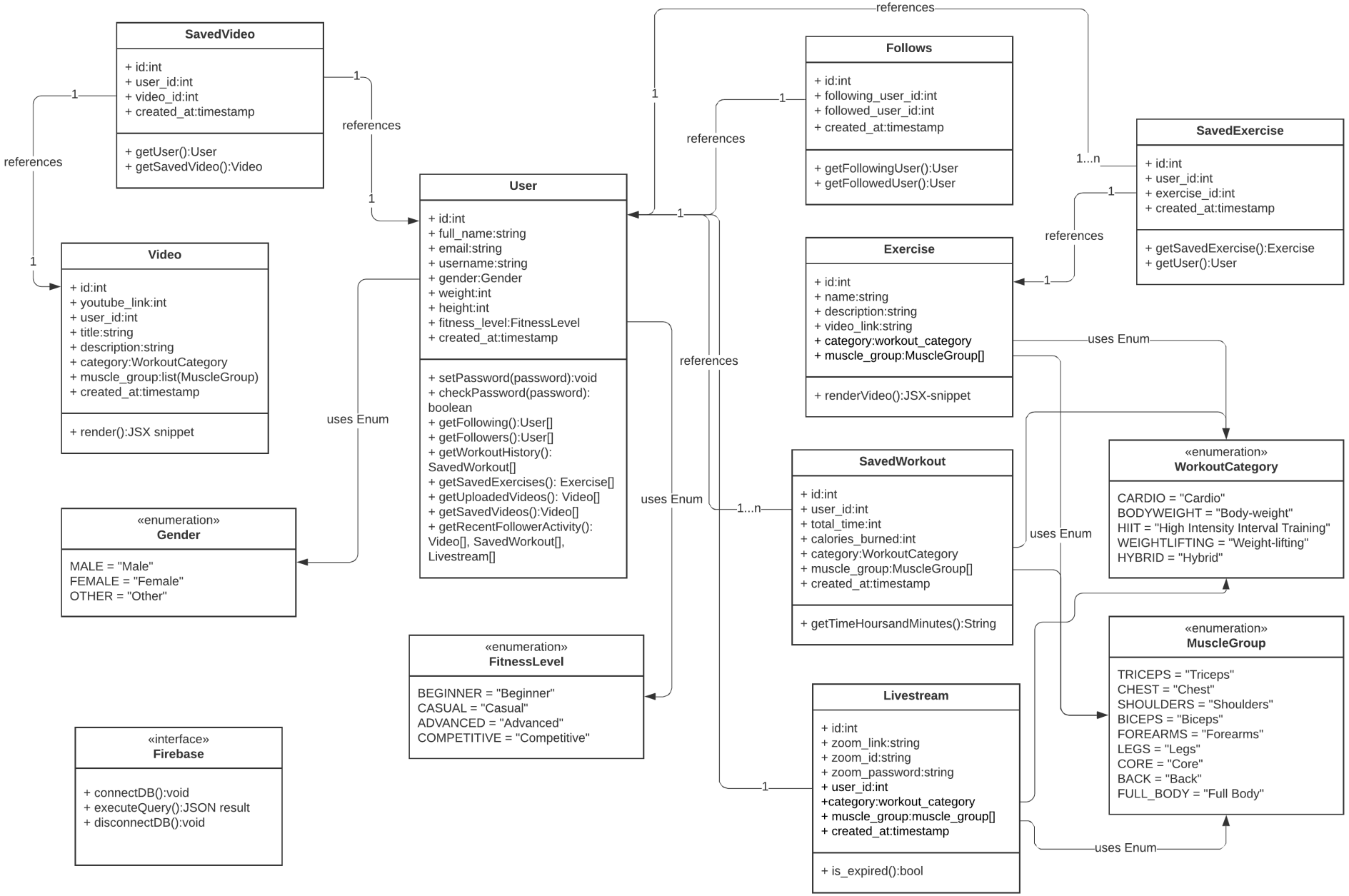
***NOTE: As noted in our implementation documentation, the member:**

```
birthday: String
```

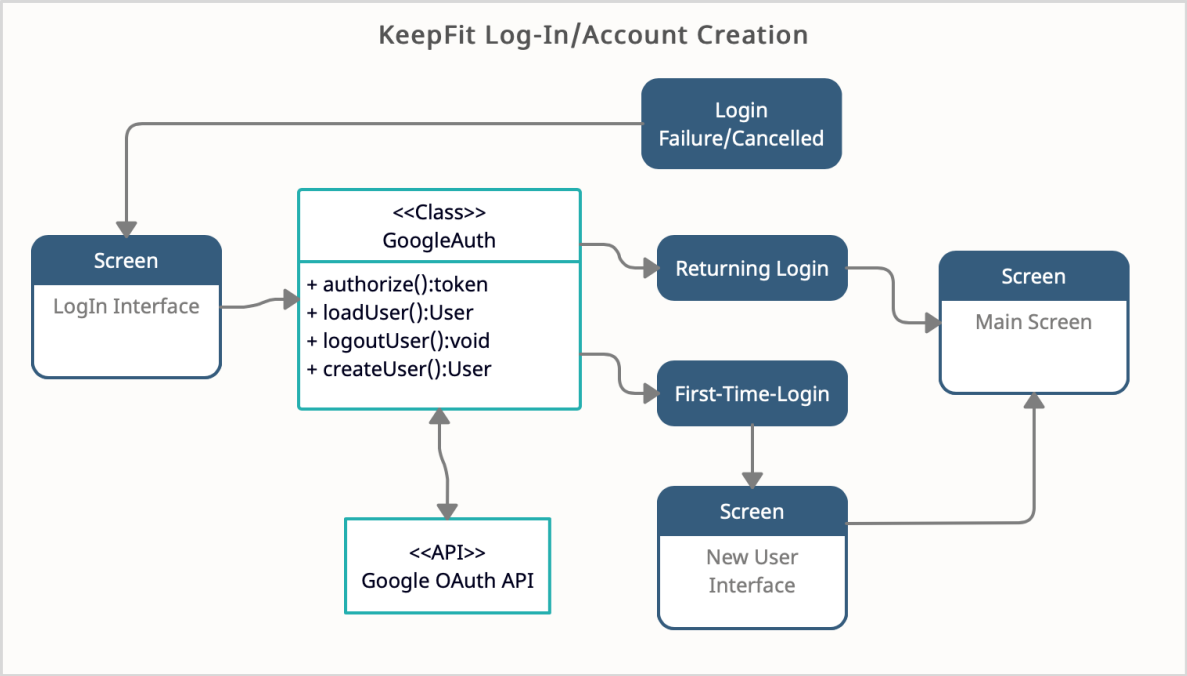
has been added as a member variable under the class:

```
Livestream
```

KeepFit Database Class Diagram

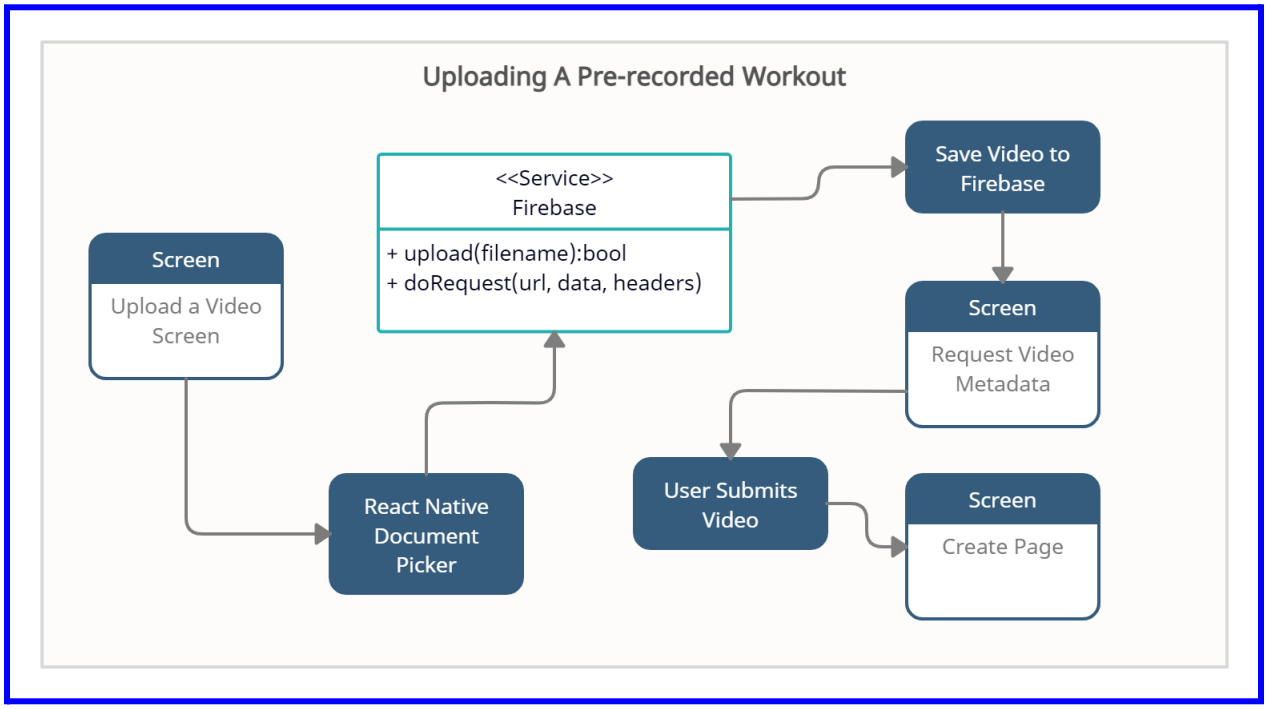


4.2.1 Log-in / Account creation



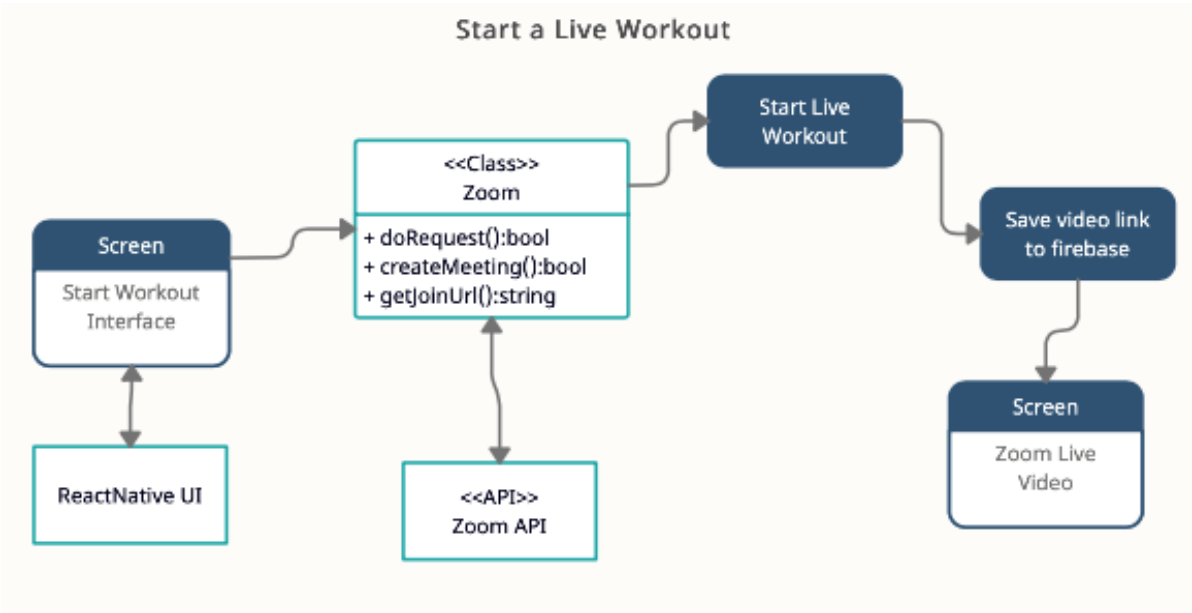
Log-in / account creation is depicted in the above UML diagram. Users will be greeted by the login interface when they enter the application. From here, they will click a button to login with Google using OAuth 2.0. Once they have proceeded through Google's designed authorization flow using functions in the GoogleAuth class, we will store user login information and load the logged in user into our application's global state. If the user is logging in to the application for the first time, they will be prompted for further information about themselves, as detailed in the software requirements. These include their gender, weight, height, and fitness level. They will then be directed to the main screen to start using KeepFit. If the user is an existing user, they are directed straight to the main screen after logging in through Google. Upon login failure, they are redirected to the login screen. User data will be stored in Firebase, and authentication will be handled using Firebase OAuth 2.0 along with Google OneTap. The creation of the form UI for new user information will be done using React Native.

4.2.2 Uploading a pre-recorded workout



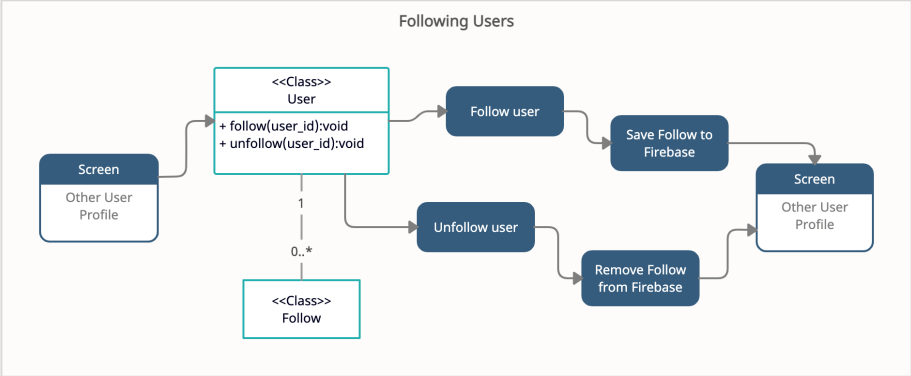
The process for users to upload a pre-recorded workout is diagrammed above. Users will navigate to the Upload a Video Screen via the create section from the middle of the KeepFit navbar (as detailed in the software requirement specification). From there, they will be able to pick a video from their phone library, using the native document picker. ~~Once they've selected a document, the S3 Service class is used to upload the file to our S3 storage bucket using its filename. S3 will return a url for the uploaded file, which is then saved to firebase with a new entry in the table for videos.~~ Once this is complete, the user is prompted to provide metadata about the video and submit it. ~~The key element here is the S3 service, which handles integration with our AWS S3 storage bucket and abstracts this functionality within itself. Other areas of the application simply have to initialize an instance of the S3 Service class and can then use its functions to interface with S3.~~ The video is then properly uploaded to the Firebase CDN hosting service as a blob object which returns a reference url. This url is then pushed to our Firestore database as a Video collection along with other pertinent identifying user data. The user is then returned to the create page.

4.2.3 Starting a live workout



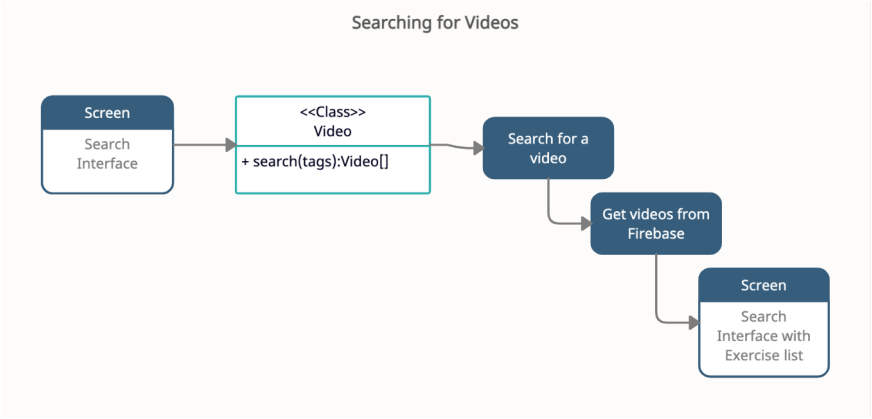
This class diagram describes the process taken by the user to start a live workout. To begin, the user will begin on the start workout interface designed with React Native. There will be a button to begin a live workout through Zoom. Using the Zoom API, a Zoom meeting link will be created and the join URL will be returned to our app. We will save the video link into firebase and then display that link into the explore interface's live streams for other users to see. We will have a garbage collector run daily to remove defunct Zoom meeting links. After clicking on a zoom live stream URL, users will be taken into Zoom, where they can then begin their live workout.

4.2.4 Following users/viewing their information



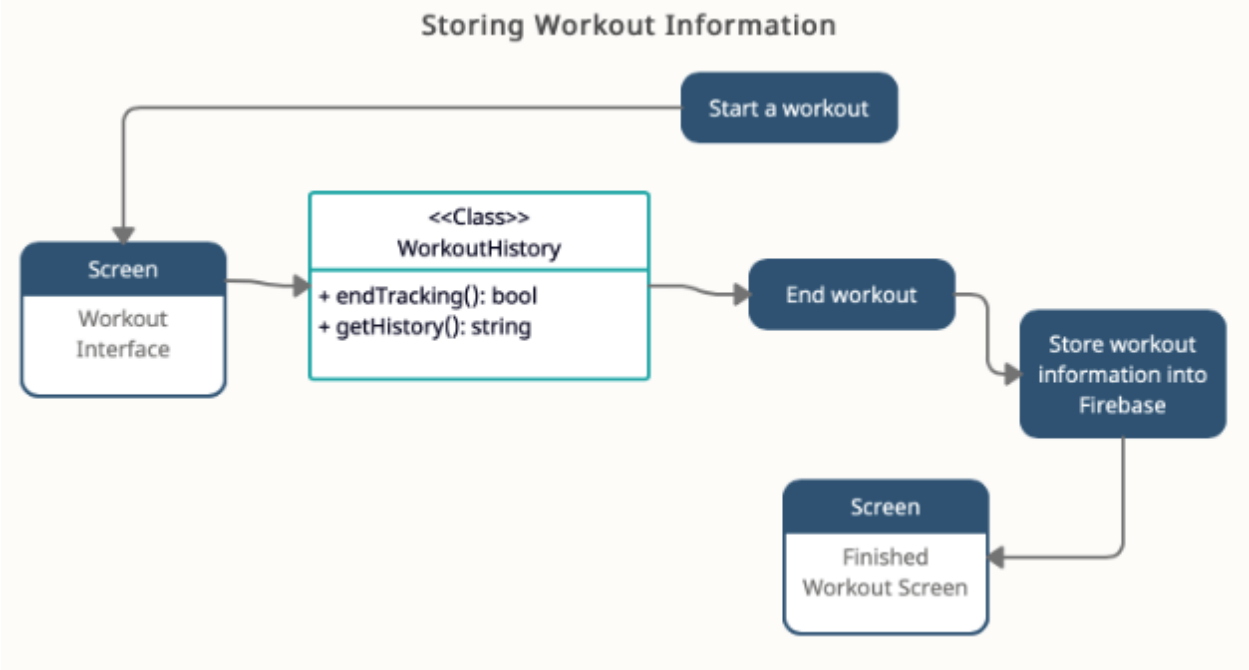
This class diagram describes the process taken by the user to follow another user. To begin, the user will begin on the target user profile interface. If the user is not following the target user, the target user profile interface will display a follow button. Otherwise, the interface will display an unfollow button. When the user clicks on the follow button, the follow will be saved into Firebase, and the interface will update to display that the user has followed the target user. When the unfollow button is clicked, the existing follow will be removed from Firebase, and the interface will update to display that the user has unfollowed the target user.

4.2.5 Searching/filtering for videos



This class diagram describes the process taken by the user to search for videos. To begin, the user will begin on the search interface. The user selects tags relevant to filter out videos from Firebase. When the user executes the search, Firebase will retrieve videos with the selected tags and update the search interface with the list of returned videos.

4.2.6 Tracking and Storing workout information

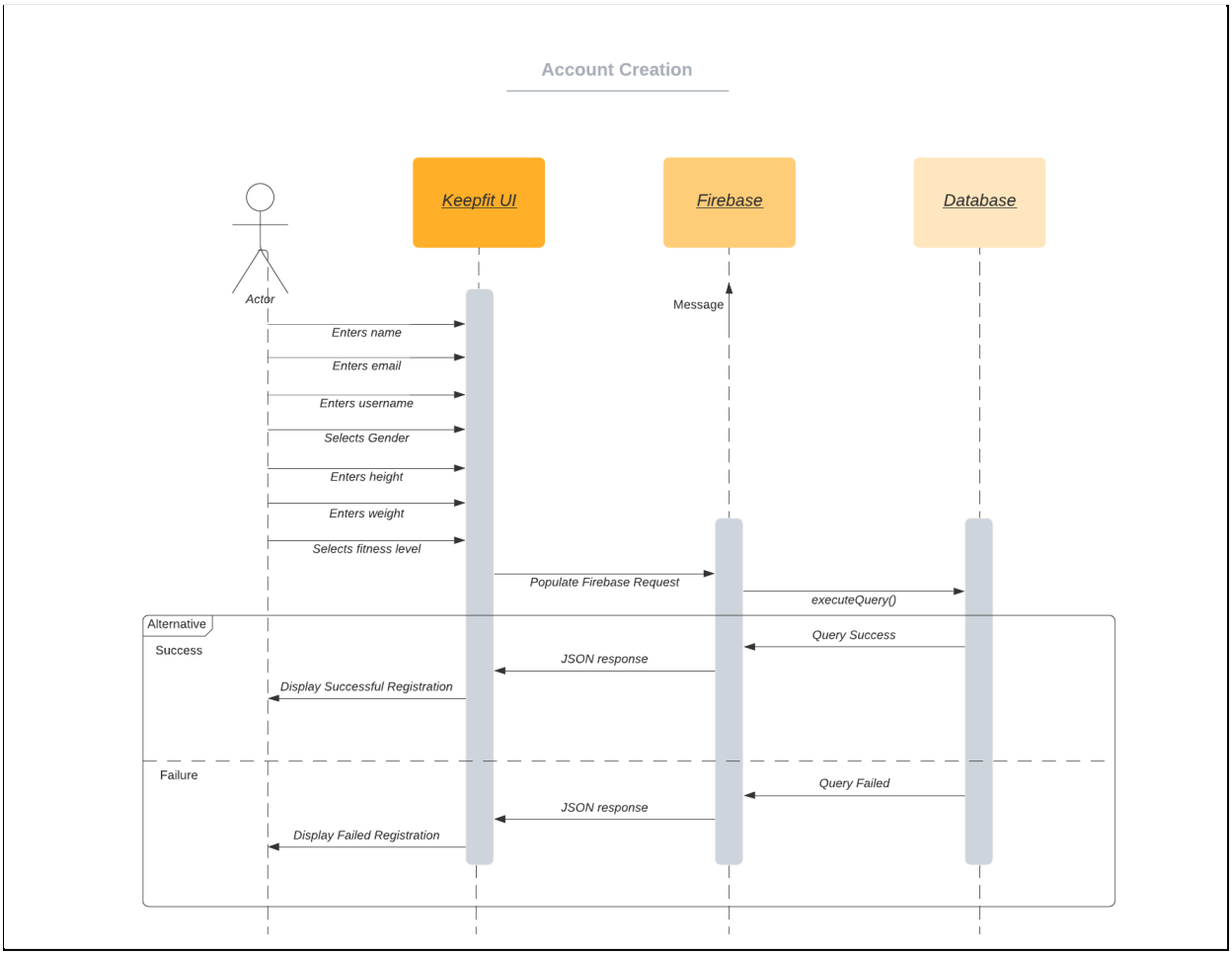


This class diagram describes the process taken by the user to track and store workout information. To begin, the user will begin on the workout interface. The user begins a workout session by creating a WorkoutHistory instance. When the user ends their workout by pressing the finish button, Firebase will update the WorkoutHistory instance to indicate that the session has ended and calculate the number of calories burned and time spent exercising. The user will be redirected to the finished workout screen which will display the number of calories burned and time spent exercising.

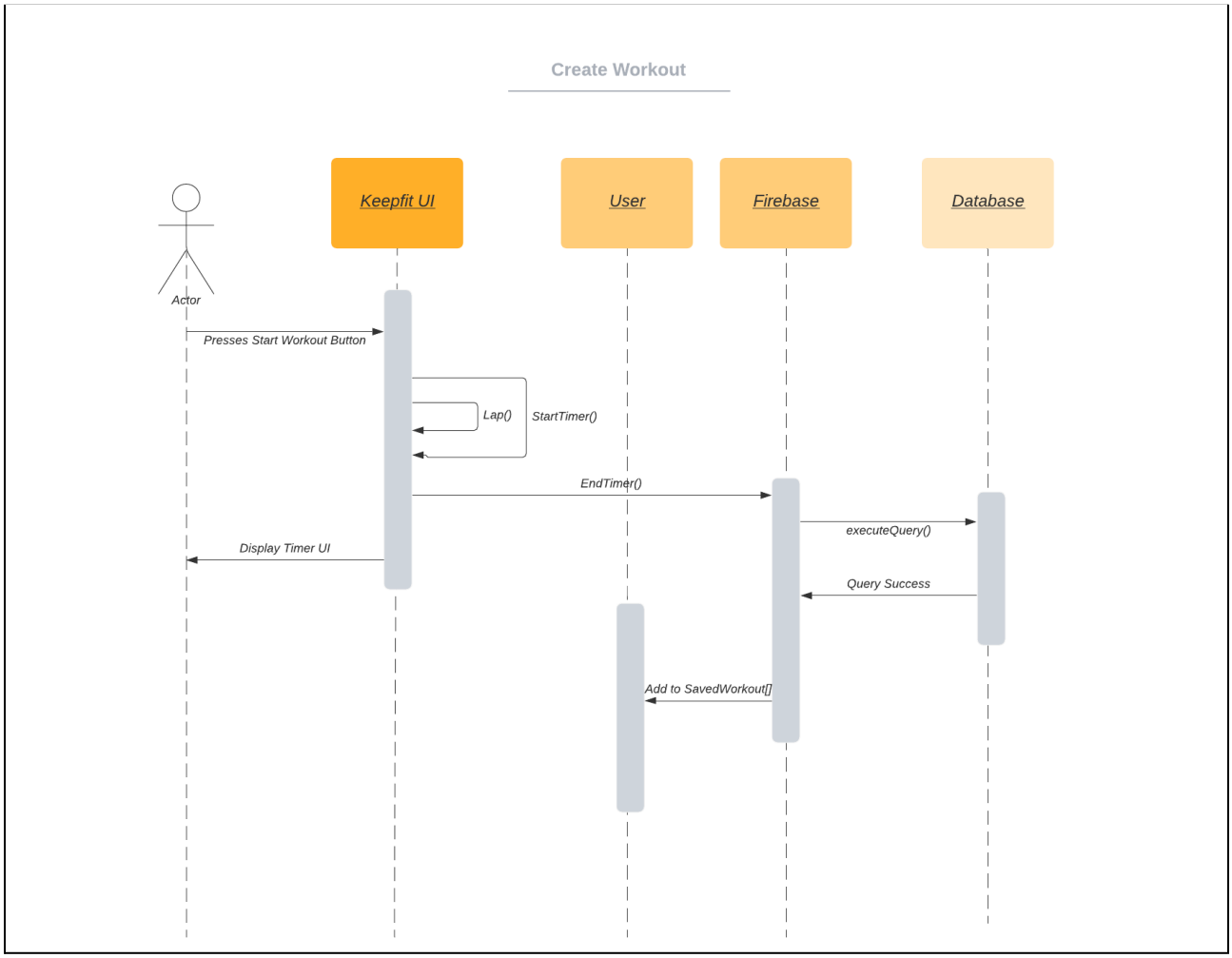
4.3 Set of Sequence Diagrams

The diagrams below depict user stories and how the flow of activities should be conducted within the KeepFit application. These are meant to help developers envision the user experience provided by the application and develop accordingly. The complex interactions necessary to make this application work are detailed below.

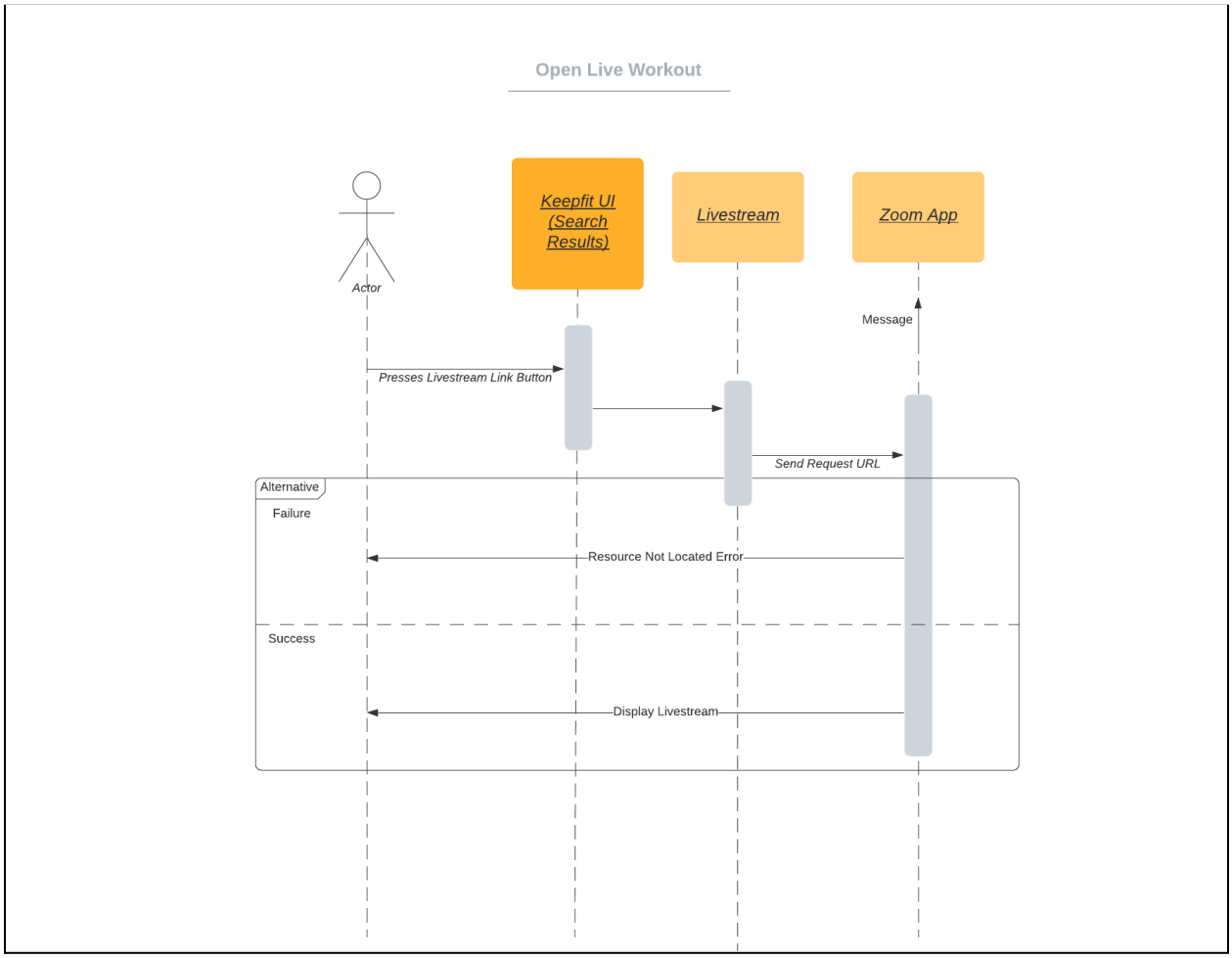
4.3.1 Creating an Account



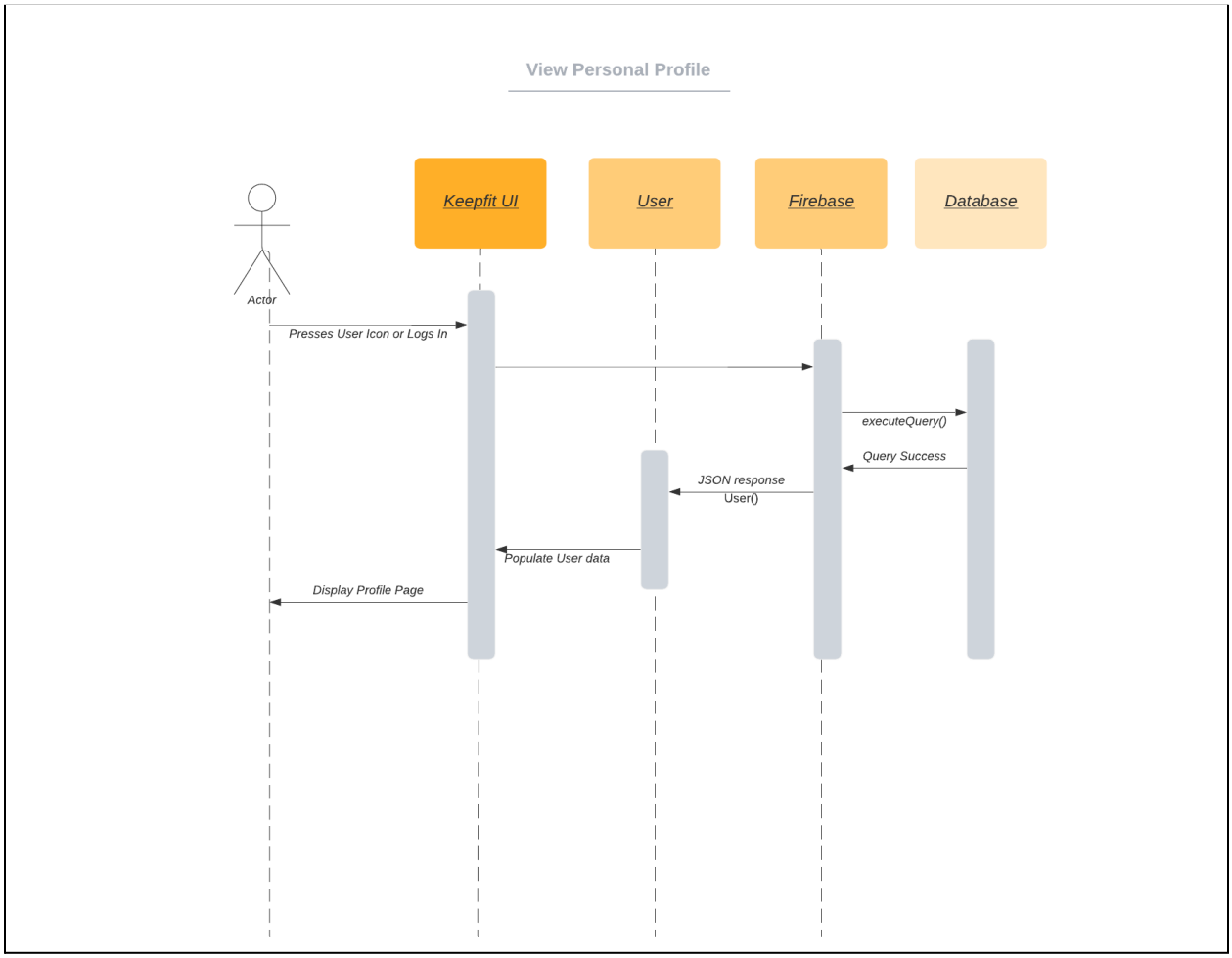
4.3.2 Starting a Workout



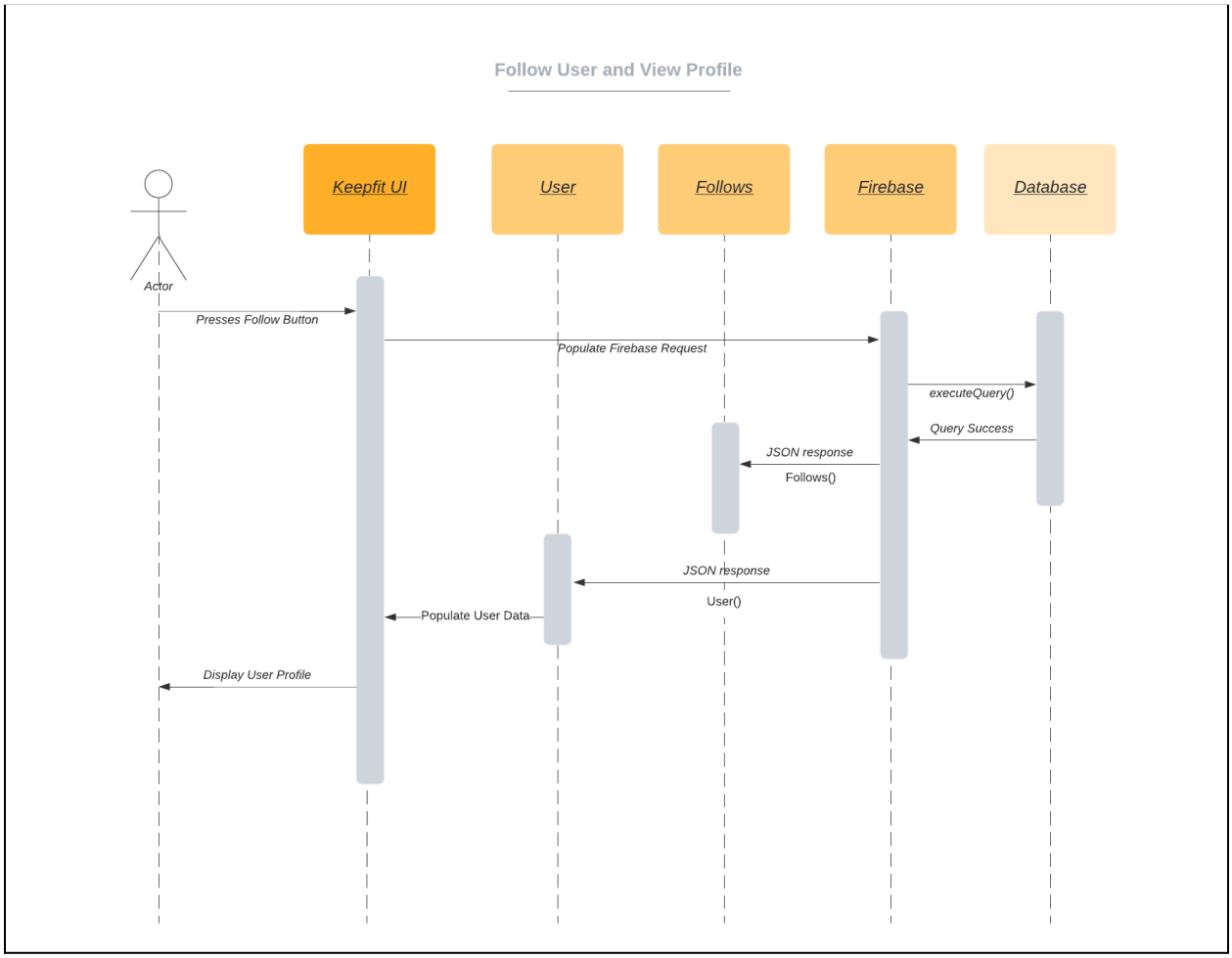
4.3.3 Viewing Live Workouts



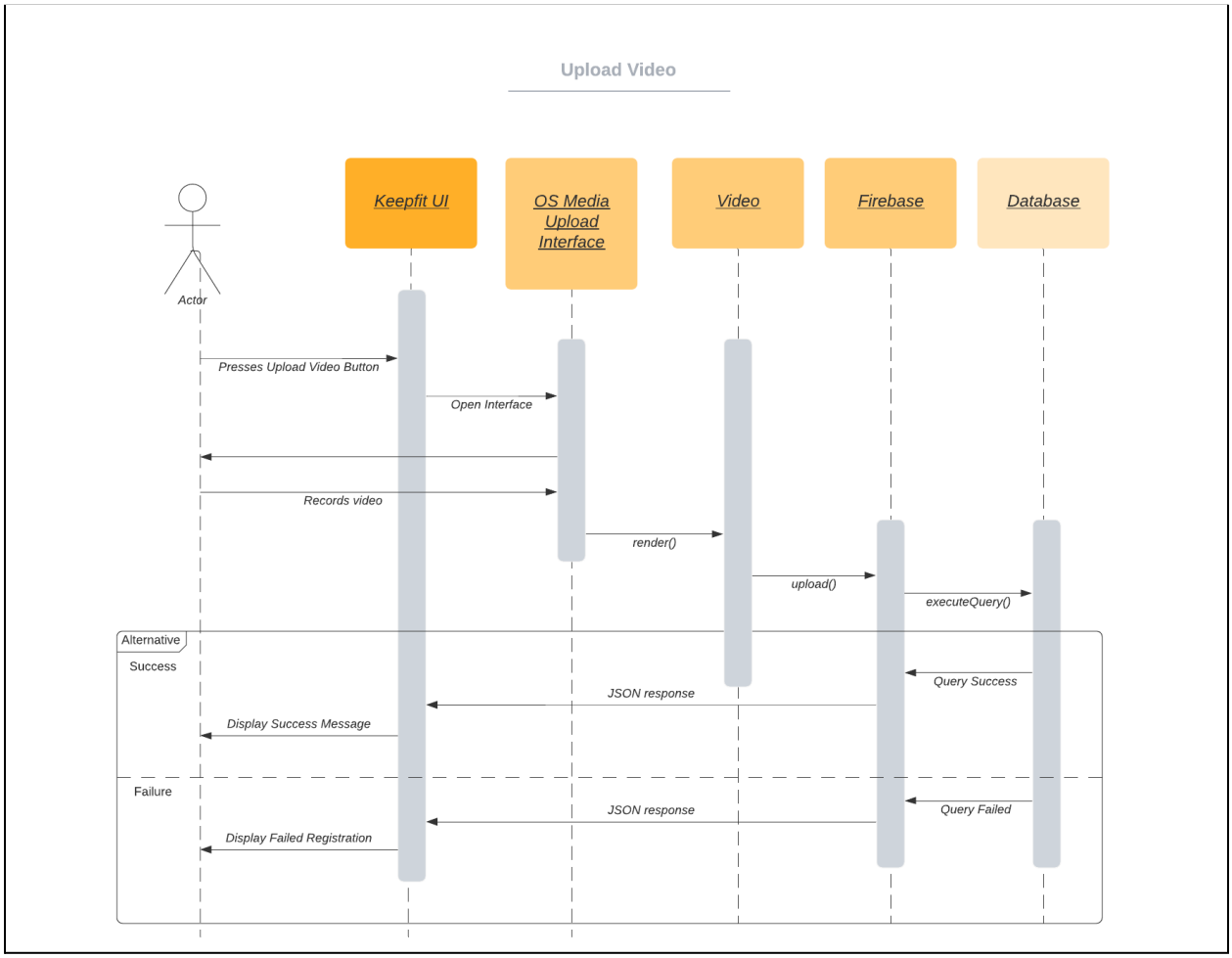
4.3.4 Viewing Personal Profile



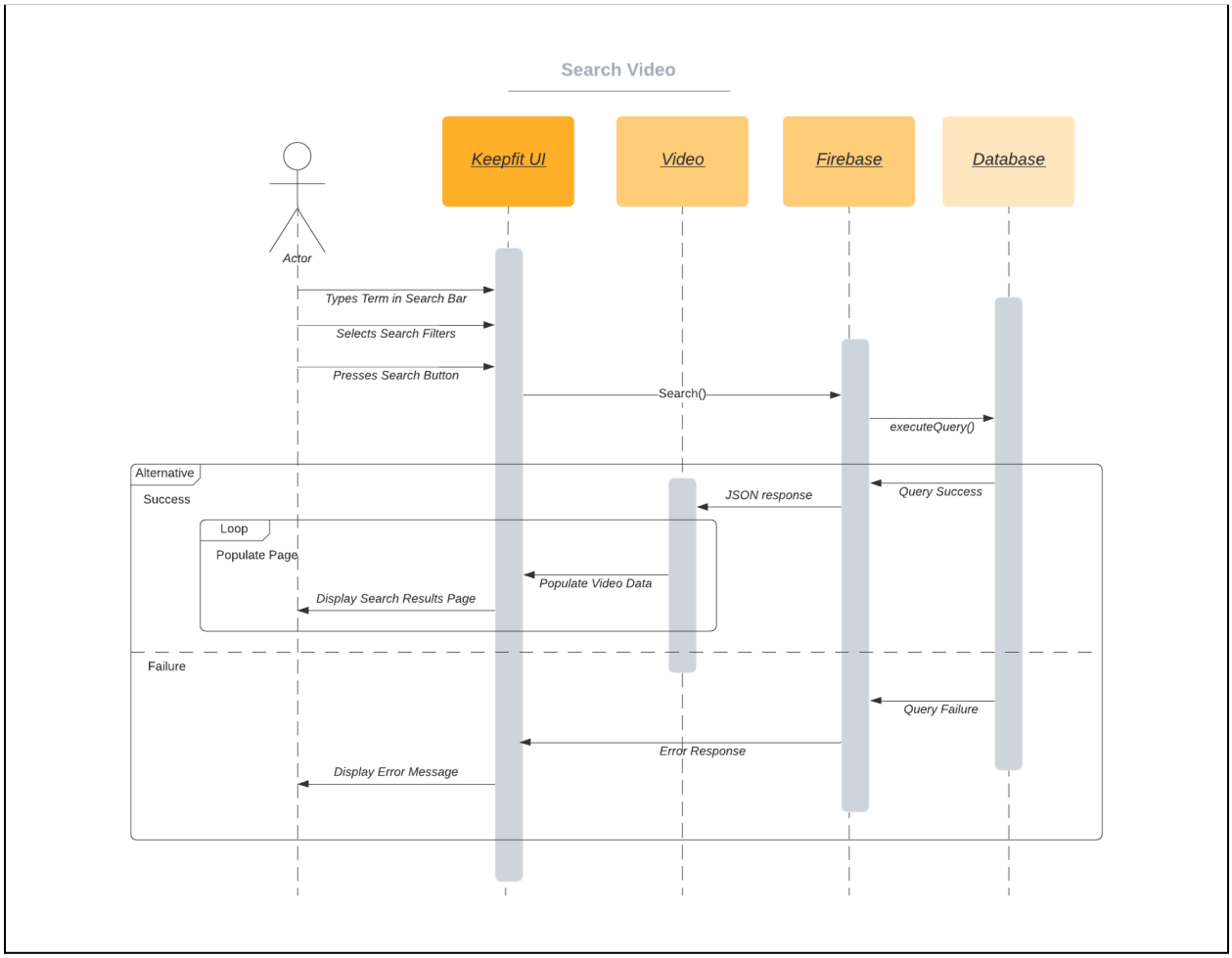
4.3.5 Following Another User and Viewing Their Information



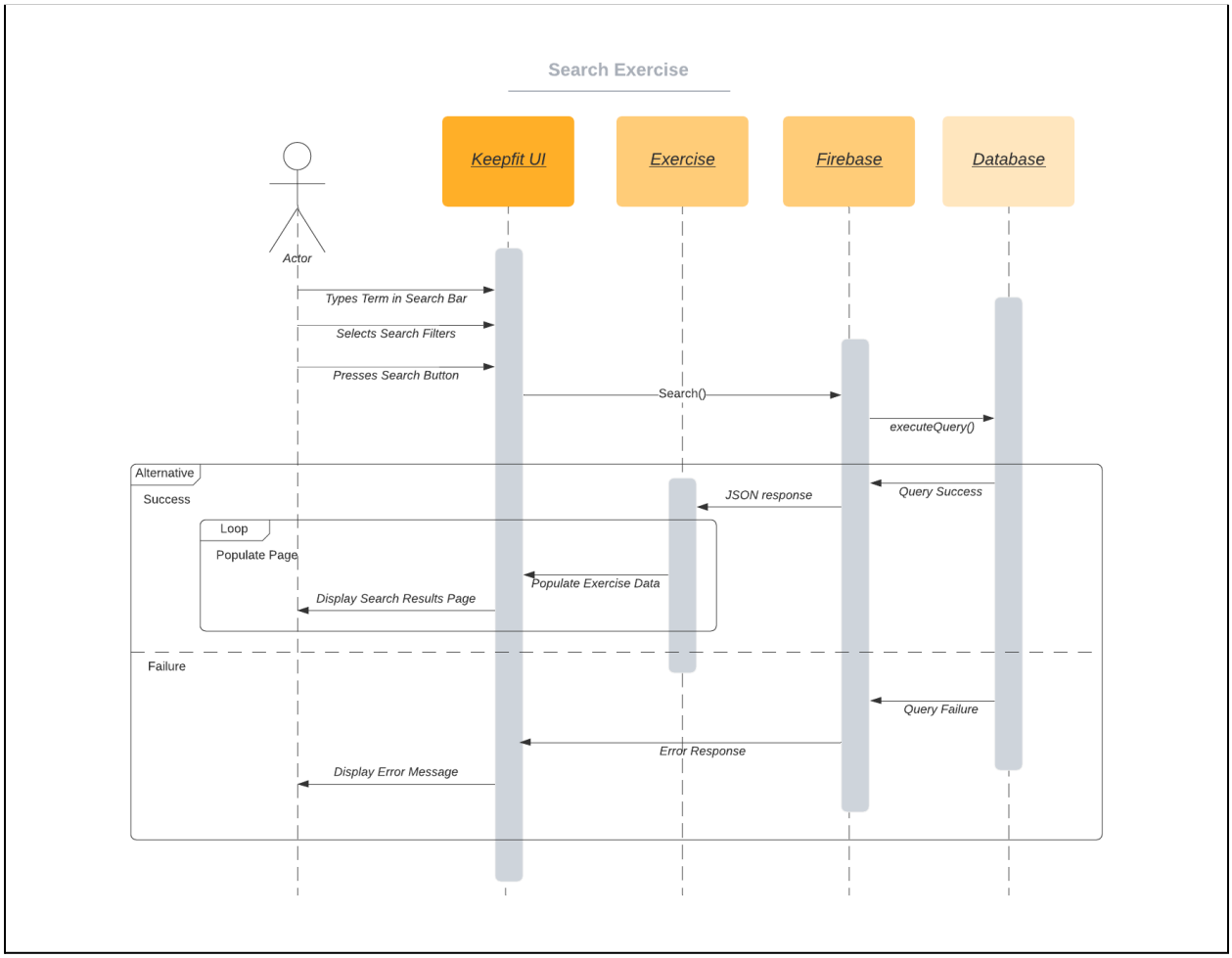
4.3.6 Uploading a Video



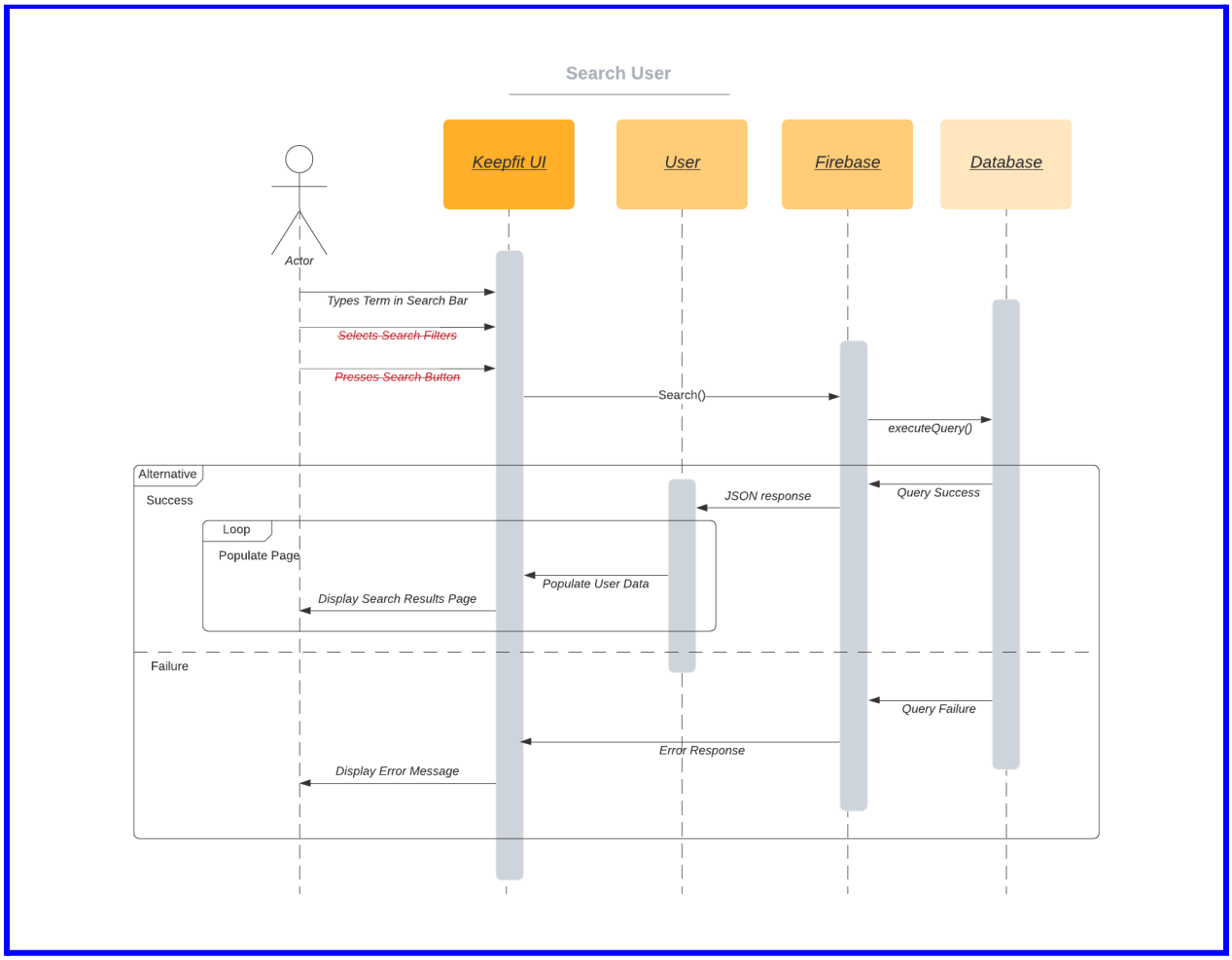
4.3.7 Searching for Videos



4.3.8 Searching for Exercises



4.3.9 Searching for Users



4.3.10 Searching for Livestreams