1. Project Title and Authors

Team No.: 21
Project: KeepFit

Members:
1. Ayushi Gupta - ayushigu@usc.edu - 8595068906
2. Armaan Pishori - pishori@usc.edu - 4252700260
3. Samantha Tripp - stripp@usc.edu - 7855953482
4. Jason Ahn - ahnjason@usc.edu - 3832800566
5. Vedant Nevatia - vnevatia@usc.edu - 6562275237
6. Joseph Yoon - josephyy@usc.edu - 2233295549

2. Preface

KeepFit is a fitness-centric social media platform. It is intended for people to watch and/or create exercise videos or live streams to stay fit during the pandemic.

The aim of this document is to include the changes in design decisions in the actual implementation of KeepFit that are different from those in our previous documentation. We have carefully redlined these changes in the relevant diagrams from Assignment 2, and pasted them here for reference. We have also included the rationale for every departure from our initial architecture and detailed design. The document also includes hypothetical requirement changes in two given scenarios, where we explain our hypothetical methods and rationales.

Table of Contents

3. Introduction

The implementation of this system is intended to support functionality regarding authentication, the accessing and uploading of video content, streaming, and the use of a calorie counter. Decisions regarding the implementation of KeepFit were made to balance following the architecture plan with making adjustments based on learned knowledge during development. Implementation responsibilities were organized into frontend and backend teams, and within these teams, developers focused on implementation based on functional requirements. This document contains a comprehensive list of architectural, detailed design, and requirement changes that were made during the implementation phase of the process.

4. Architectural Change

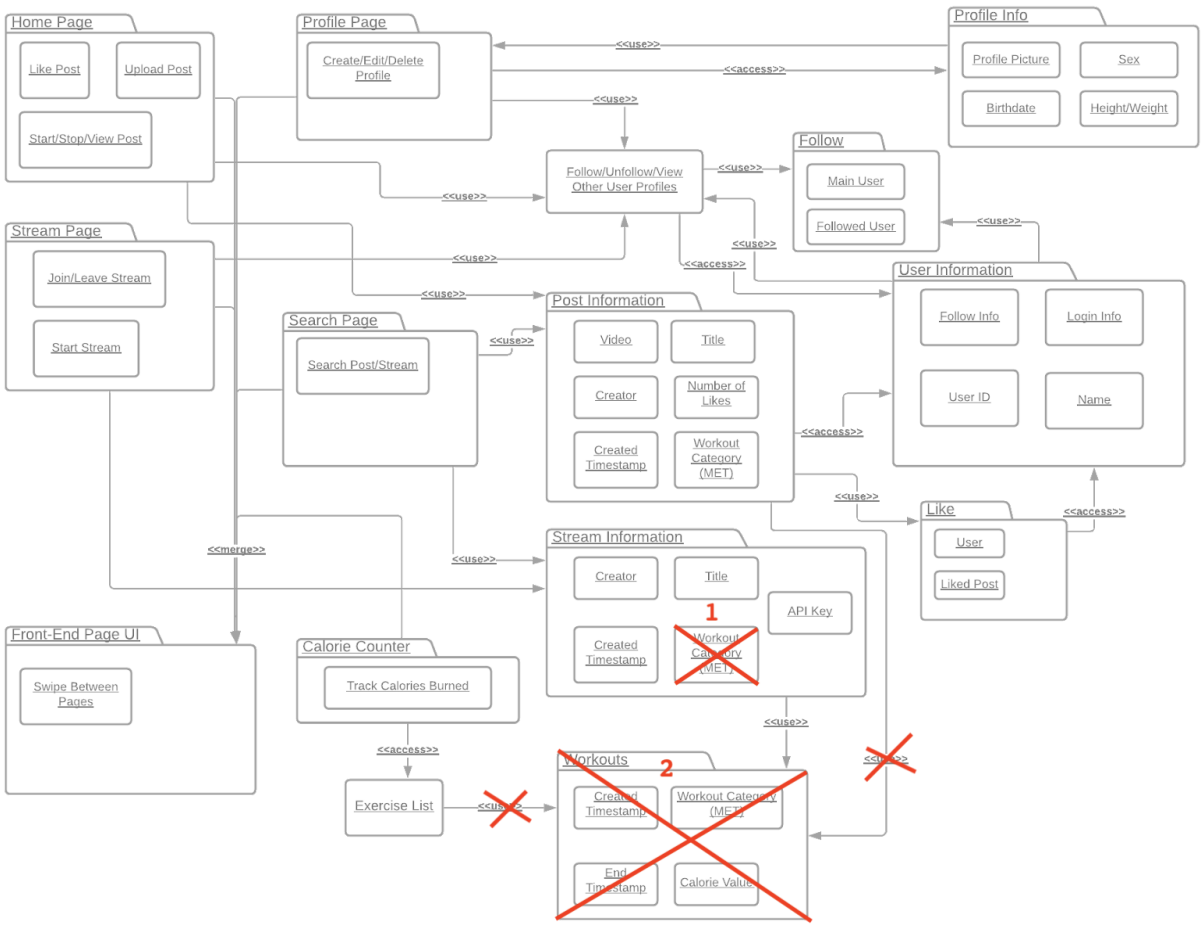
a. What architectural changes did you make during implementation?
b. What's the rationale behind the decision change?

No architectural changes were made during implementation.

5. Detailed Design Change

a. Package diagram changes

Pasted below is a redlined version of the package diagram in Assignment 2, Question 5, Part a. Each change is numbered, and the rationale is explained below the diagram sequentially.



1. **Remove Workout Category from Stream:** We found it unnecessary to calculate calorie values for a livestream, so we removed the Workout Category and MET values from Stream Information. Given that a livestream may include elements other than consistent

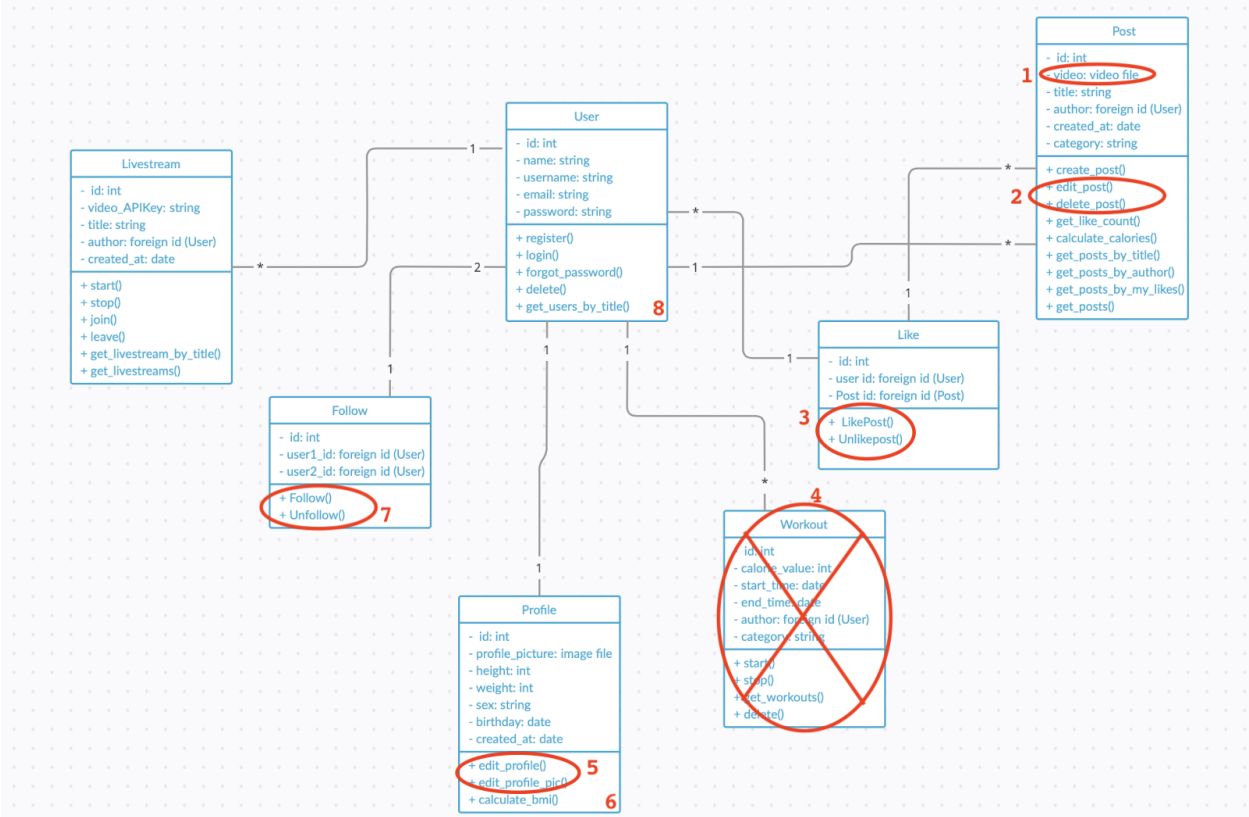
workout, the calorie value could be wildly inaccurate given that it's already a large estimation.

2. **Remove Workouts:** We decided against storing workout data unnecessarily, and we hardcoded the calorie calculator into the exercise list using MET values. As a result, the calorie counter allows you to pick a workout, and will calculate your calories as you do it. Storing the data beyond this seemed redundant. The corresponding links (<<use>>) have also been removed using smaller Xs.

All other aspects of the Package Diagram remain unchanged in our implementation.

b. Class diagram changes

Pasted below is a redlined version of the class diagram in Assignment 2 (Question 5, Part b) Each change is numbered, and the rationale is explained below the diagram sequentially.



1. **Change video type to string:** We initially planned to store the videos as a video file, later transitioned to using a string containing a youtube unique identifier for the video. This simplified the backend implementation, while reducing the amount of storage space needed - while maintaining the functionality of video playback for users.

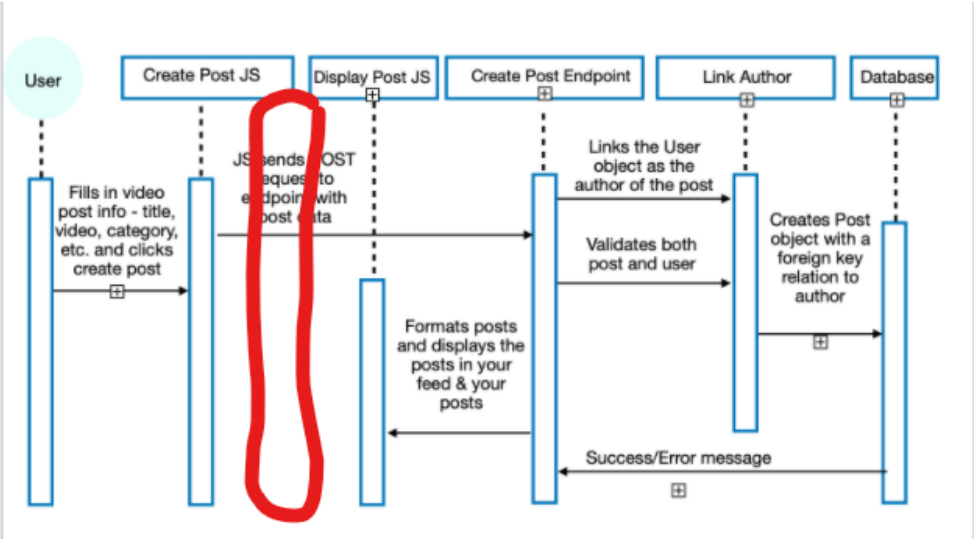
2. **Combine edit_post() and delete_post():** Initially planned as two separate functions, we combined them into one view with two possible requests (put or delete). This simplified our implementation while maintaining functionality. It reduces the no. of endpoints by 1 too. This function is called PostView().
3. **Combine Likepost() and Unlikepost():** Initially we planned two separate functions to like and unlike a post. During implementation, we found that it made more sense to combine them into a single function with a condition, allowing users to toggle between like and unlike with ease. It reduces the no. of endpoints by 1 too. This function is called ToggleLikeView().
4. **Delete Workout class:** While implementing the workout calorie counter, we decided against storing the workout data since it began to seem redundant. We decided to implement a live calorie counter for workouts, without the need to store extra information. So, we deleted the workout class entirely from the backend.
5. **Combine edit_profile and edit_profile_pic():** We decided to allow editing of both at the same time, thereby reducing the no. of endpoints by 1. This made the implementation more elegant than we had planned.
6. **Add function get_profile():** This added functionality that our app required. We did not predict that at an earlier stage, but found the need for it as we implemented it.
7. **Combine Follow() and Unfollow():** Initially we planned two separate functions to follow and unfollow a user. During implementation, we found that it made more sense to combine them into a single function with a condition, allowing users to toggle between follow and unfollow with ease. It reduces the no. of endpoints by 1 too. This function is called ToggleFollowView().
8. **Add function logout() to user:** This added functionality that our app required. We did not predict that at an earlier stage, but found the need for it as we implemented it.

All other aspects of the Class Diagram remain unchanged in our implementation.

Sequence Diagram Changes

Sequence Diagram #2

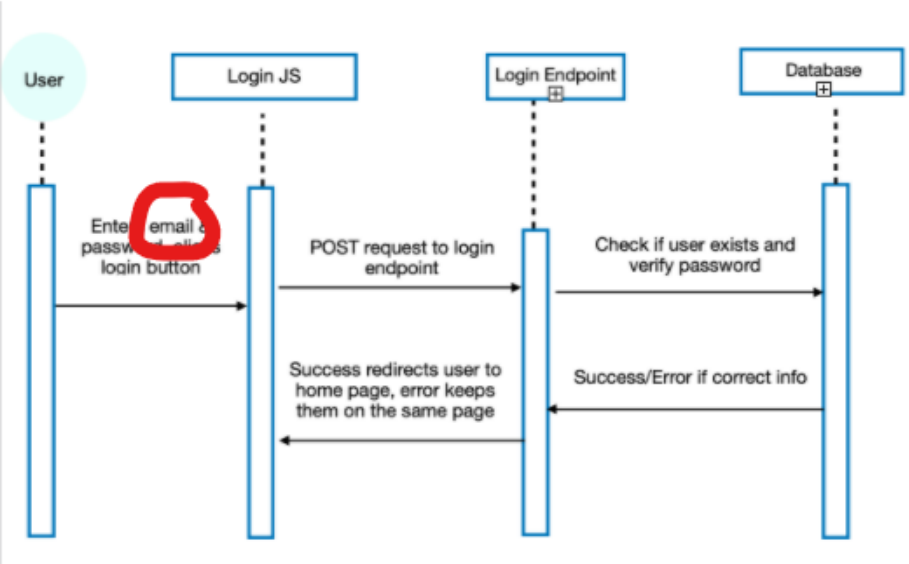
2. Create video post: The user fills in the video post info including the video's title, the video itself, the category of the video, etc. and clicks the create post button. The Create Post JS sends a POST request to the endpoint with the post data. The Create Post Endpoint links the current User object as the author of the post, also validating both the post and the user. The Link Author creates a post object with a foreign key relation to the author of the post. The database sends a success/error message to the Create Post Endpoint based on the validity. The Create Post Endpoint formats the post and displays the post in the current user's feed.



Rather than sending the video file directly to the backend, we now access Youtube's API and save the video file there and only save the returned url (as a string) in our database. This allows for the frontend to easily integrate the video file on the various pages. We would have to add a layer (marked in red) to incorporate this change.

Sequence Diagram #3

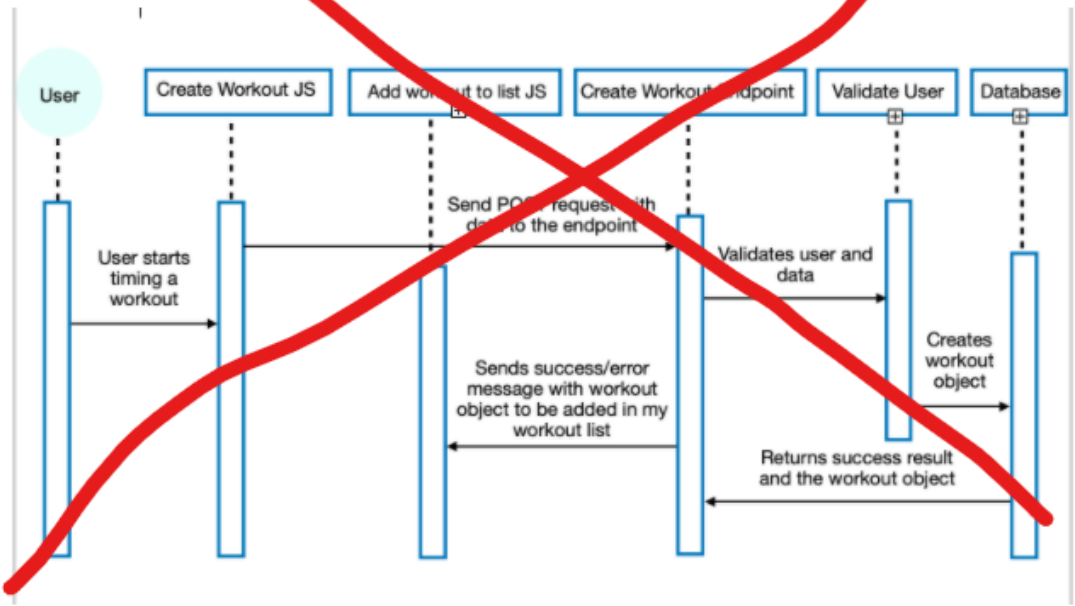
3. Login User: (POST request): The User enters their email and password and clicks the login button. The Login JS sends a POST request to the login endpoint, which checks if the user exists and verifies the password entered. The database reports a success/error for correct/incorrect information respectively. A successful input will redirect the user to the homepage, while an erroneous input will keep them on the same page.



For the login implementation, we decided to change the login identifier from the user's email to their username.

Sequence Diagram #4

4. Create Workout (POST request): The user starts timing a workout on the calorie calculator, and when they stop, they can choose to save their workout by clicking the save button. The Create Workout JS sends a POST req with the workout data to the Create Workout endpoint. The user and data is then validated, after which a new workout object is created in the database. The database returns a success/error result and the workout object. The Create Workout Endpoint then sends a success/error message with the workout object to be added and displayed in ther user's workout list.



As mentioned in the previous sections of the Detail Design changes, we decided to move away from saving Workouts in the database as it seemed redundant.

6. Requirement Change

Scenario 1
Assume while the user is looking through several live streams. some live streams expire due to out of time or the hosts canceling the streams. The user should see a message that this stream has ended, which should include an option to resume watching if the streams go live again.

1. Does this change your design?
2. If not, why not
3. If so, what should be changed and how?

This does change our design as we would need to change the livestream class to include a timer attribute and include functions to allow users to pause/cancel and resume their live streams. On the front-end, we would need to include a function to display a message on the livestreams page or in the stream itself to notify users who click on a stream when it is paused. And we would need to add a view for users who are in a cancelled/paused stream to have options to leave the stream and be able to wait until the stream is resumed.

Scenario 2
Assume the user clicks the Yoga button in the exercise interface to record his exercise. However, he does not remember how to do the Yoga moves accurately and decides to watch a video first. We need to allow the user to directly jump to the Yoga videos without searching in the exercise demo interface. Also, if the user exits the training to watch videos, he should be accounted for one exercise period and be recorded by the app.

1. Does this change your design?
2. If not, why not
3. If so, what should be changed and how?

This changes our design as we did not design an exercise demo interface or demo videos, so we would need to record these videos as official demonstration videos and save them into a database that would store them as .mp4 files. Also, we would need to re-implement workouts as a class to save to the database, so that we can count training videos as exercise periods.