2340 Team 6 Sprint 2

Alex Wang, Christian Chin, Eric Shao
Richard Kozyak, Brian Chen, Khoa Bui

June 2024

1 Domain Model Nouns

1. Workouts - Class
2. Users - Class
3. Community forums - Class
4. Forum names - Attribute
5. Leaderboard - Attribute / Potential Class
6. Community posts - Class
7. Community challenges - Class
8. User name - Attribute
9. Workout name - Attribute
10. Calories - Attribute
11. User weight - Attribute
12. User height - Attribute
13. User heart rate - Attribute
14. User sleep patterns - Attribute
15. Resource library - Class
16. Personal Information - Class
17. Gender - Attribute
18. User Database - Class
19. User - Attribute
20. Workout - Attribute
21. Workout Database - Class
22. Workout Tracking - Class
23. Calories - Class
24. Daily Calorie Goal - Attribute

25. Calories Burnt - Attribute
26. Reps - Attribute
27. Sets - Attribute
28. Additional Notes - Attribute

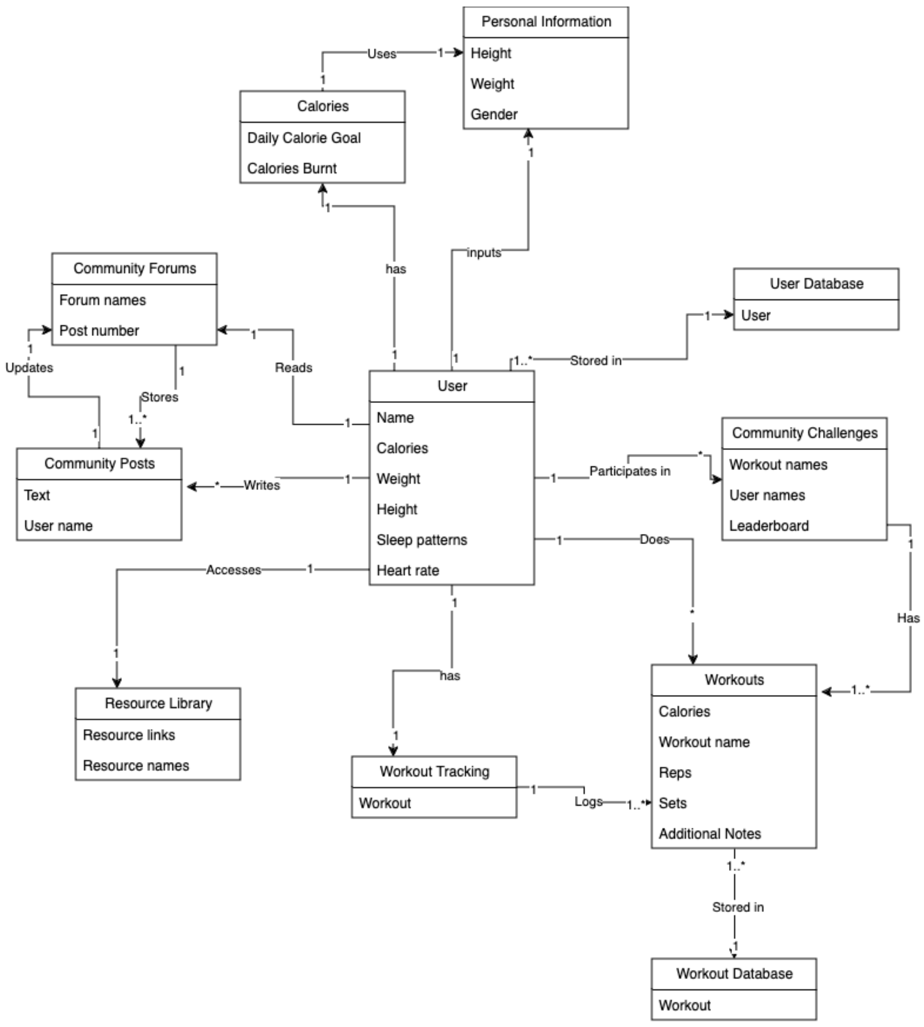


Figure 1: Domain Model

2 SOLID Principles

2.1 Single Responsibility Principle

The Single Responsibility Principle states that each class should have a single responsibility. In the case of our app, each class has a generally well-defined responsibility that can be mostly understood through just reading the name of the class and comments within each class. For example, for code regarding the viewing aspect of the app, our different screens each have their own class associated with them with code for buttons and other widgets confined to their associated class.

2.2 Open/Closed Principle

The Open/Closed Principle states that software entities should be open for extension but closed for modification. In the case of our app, methods that have already been implemented are mostly closed for modification and do not need any additional changes. They are created in such a way that we are able to make new methods and add new functionalities, but old functionalities are already in place and do not need any adjustments. For example, the UserDatabase and UserDatabaseRepository classes are designed to be immutable and do not need any changes within their internals workings with further design features being added in the form of new classes and methods.

2.3 Liskov Substitution Principle

The Liskov Substitution Principle states that superclasses must be replaceable by the objects of subclasses without breaking the application. Our app currently only uses subclasses extending abstract classes of packages from Android Studio. As the parents of these subclasses are abstract, the superclasses are able to replaced by their subclasses without breaking the application. Therefore, our program follows the Liskov Substitution Principle.

2.4 Interface Segregation Principle

The Interface Segregation Principle states that classes should never be forced to implement interfaces that are not necessary or implement methods that are not needed. Our code follows the interface segregation principle as we do not use any unnecessary or confusing interfaces in our code.

2.5 Dependency Inversion Principle

The Dependency Inversion Principle states that high-level classes should be easily reusable and rely on abstractions to do so. Our code follows the dependency inversion principle as we do not have any high-level classes that need to reused as most of our implementations do not need higher-level abstractions and only require relatively simple code.

3 Use Cases

3.1 Add Workout

3.1.1 Main Success Scenario

The user enters the application home screen and signs into their account. They click on the workout tracker section and new workout. They fill out the form consisting of fields such as workout name, sets, repetitions, and calories per set. The user submits the form and is able to view their completed workout in a history of workouts.

3.1.2 Alternative Scenarios

If the database connection encounters an error and the workout is unable to be logged into Firebase, the system will send an error message and ask the user to try again.
If the user enters invalid values into certain fields such as sets or repetitions (such as negative numbers), they will be asked to enter accepted values.

3.2 Create Account

3.2.1 Main Success Scenario

The user enters the application home screen and clicks the register button. They fill out the form with their email address and password (twice to confirm the password and make sure no typos are present). They press submit and the system confirms whether or not an account exists with the given email and if no account was found, the account is successfully created.

3.2.2 Alternative Scenario

If the system encounters an error while trying to create the account such as not being able to properly access Firebase or finding another account associated with the given email, the user is prompted to try again or use a different email.

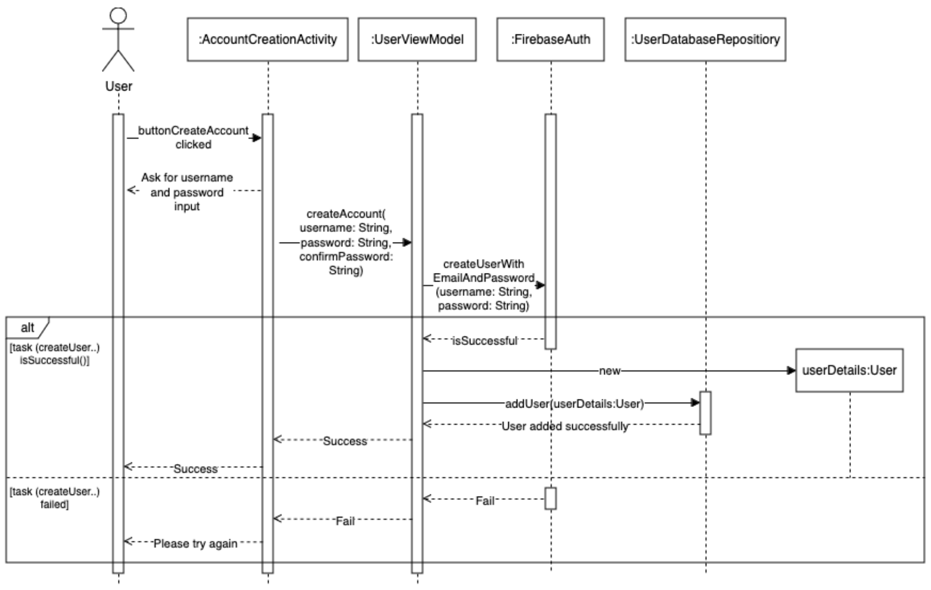
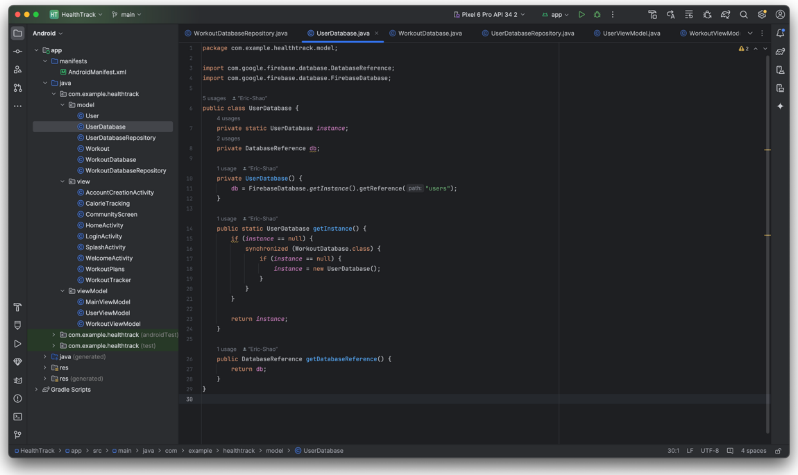

User
:AccountCreationActivity
:UserViewModel
:FirebaseAuth
:UserDatabaseRepositiory
buttonCreateAccount clicked
Ask for username and password input
createAccount(username: String, password: String, confirmPassword: String)
createUserWith EmailAndPassword (username: String, password: String)
alt
[task (createUser..) isSuccessful()]
isSuccessful
new
userDetails:User
addUser(userDetails:User)
User added successfully
Success
Success
[task (createUser..) failed]
Fail
Fail
Please try again

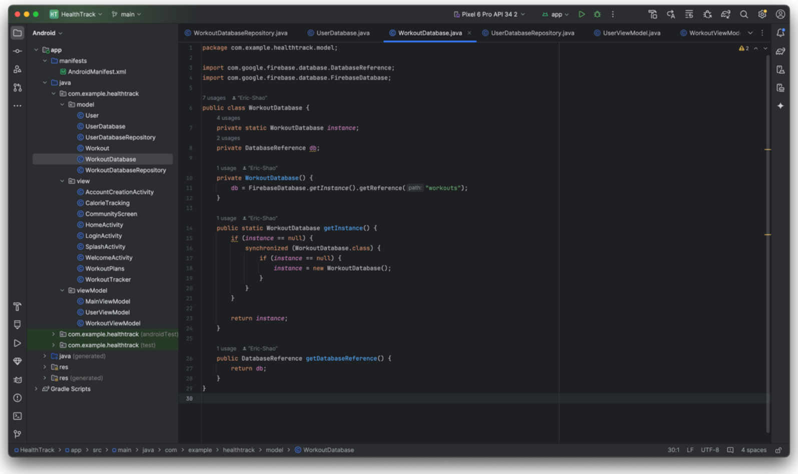
Figure 2: Sequence Diagram for Account Creation Use Case

4 Design Pattern

In our project, the Singleton design pattern is implemented in the UserDatabase and WorkoutDatabase classes to ensure that only one instance of each database exists throughout the application. This design pattern ensures that only one instance of these database classes is created throughout the application's lifecycle. By doing so, it prevents the issues that could arise from having multiple instances, such as data inconsistency, race conditions, and unnecessary memory usage. The getInstance() method in both classes is responsible for checking if the instance is null, and if so, it initializes the instance while synchronizing on the respective class to ensure thread safety. This approach guarantees that only one instance of the UserDatabase and WorkoutDatabase is created, thus maintaining data integrity and improving the efficiency of database operations across the application.



```
package com.example.healthtrack.model;

import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;

5 usages  "Eric-Shao"
public class UserDatabase {
    4 usages
    private static UserDatabase instance;
    2 usages
    private DatabaseReference db;

    1 usage  "Eric-Shao"
    private UserDatabase() {
        db = FirebaseDatabase.getInstance().getReference( path: "users");
    }

    1 usage  "Eric-Shao"
    public static UserDatabase getInstance() {
        if (instance == null) {
            synchronized (WorkoutDatabase.class) {
                if (instance == null) {
                    instance = new UserDatabase();
                }
            }
        }

        return instance;
    }

    1 usage  "Eric-Shao"
    public DatabaseReference getDatabaseReference() {
        return db;
    }
}
```



```
package com.example.healthtrack.model;

import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;

7 usages  "Eric-Shao"
public class WorkoutDatabase {
    4 usages
    private static WorkoutDatabase instance;
    2 usages
    private DatabaseReference db;

    1 usage  "Eric-Shao"
    private WorkoutDatabase() {
        db = FirebaseDatabase.getInstance().getReference( path: "workouts");
    }

    1 usage  "Eric-Shao"
    public static WorkoutDatabase getInstance() {
        if (instance == null) {
            synchronized (WorkoutDatabase.class) {
                if (instance == null) {
                    instance = new WorkoutDatabase();
                }
            }
        }

        return instance;
    }

    1 usage  "Eric-Shao"
    public DatabaseReference getDatabaseReference() {
        return db;
    }
}
```

Figure 3: Screenshots