



Bilkent University

Department of Computer Engineering

Object Oriented Software Engineering Course Project

YACM: Yet Another Club Manager

Design Report

Ali Emre Aydoğmuş, Cemal Faruk Güney, Deniz Berkant Demirörs, Enis Özer, Yekta Seçkin Satır

Instructor: Eray Tüzün

Teaching Assistant(s): Cevat Aykan Sevinç, Elgun Jabrayilzade, Emre Sülün, Erdem Tuna, Muhammad Umair Ahmed

Progress Report
December 18, 2021

Contents

| | |
|--|-----------|
| Contents | 2 |
| 1. Introduction | 3 |
| 1.1 Contents of the System | 3 |
| 1.2 Design Goals | 3 |
| 1.2.1 Reusability | 3 |
| 1.2.2 Security | 3 |
| 1.2.3 Maintainability | 3 |
| 1.2.4 Usability | 3 |
| 1.2.5 Portability | 4 |
| 1.2.6 Definitions, Acronyms and Abbreviations | 4 |
| 2. Proposed System Architecture | 4 |
| 2.1 Subsystem Decomposition | 4 |
| 2.2 Hardware/Software Mapping | 5 |
| 2.3 Design Patterns | 7 |
| 2.4 Persistent Data Management | 9 |
| 2.5 Access Control and Security | 10 |
| 2.6 Global Software Control | 11 |
| 2.6.1 Object Design Trade-offs | 11 |
| 2.6.1.1 Functionality versus Usability | 11 |
| 2.6.1.2 Cost versus Portability | 11 |
| 2.6.1.3 Cost versus Robustness | 11 |
| 2.6.1.4 Security versus Usability | 12 |
| 2.6.1.5 Flutter | 12 |
| 2.6.1.6 Pub | 12 |
| 2.6.1.7 Firebase | 12 |
| 2.7 Object Design (Click here to open the diagram in a higher quality) | 13 |
| 2.8 Boundary Conditions | 15 |
| 2.8.1 Initialization | 15 |
| 2.8.2 Termination | 15 |
| 2.8.3 Failure | 15 |

1. Introduction

1.1 Contents of the System

This application is designed to make being involved with clubs easier in Bilkent University. Whether it be a student that wants to participate in club activities and find new clubs according to their interests or a club board member who wants to communicate with club members and improve their club or a teacher who is assigned as the faculty advisor of a club that wants to follow what their assigned club is up to.

1.2 Design Goals

1.2.1 Reusability

Reusability is a key part of software design. By building the software from reusable components we will save a lot of time and this will help us complete our project before the deadline. To achieve reusability, we will group similar parts of the software and reuse the same widgets in both parts.

1.2.2 Security

Our system has to be secure in order to prevent ill-intentioned people from harming the club management process. If a normal user can login to the system and they have the authority to post on behalf of the club there can be consequences and our application would be considered less attractive. To keep the security of our application we will have different user types with different authorizations so that some actions will only be available to certain user types.

1.2.3 Maintainability

The software must be maintainable because we need to keep the software working during the school semester. Any failure in the software during the active period of the application should be fixed quickly. To achieve this maintainability, we have built our software according to the design patterns that are discussed in the class. We will also be using object-oriented software development techniques.

1.2.4 Usability

The usability of the application is important because the purpose of building a software is for people to use that software. If no one uses this club manager then there is no point in building it. To make people use the application we have to make it usable by making the user interfaces similar to other social media applications that the users are familiar with.

1.2.5 Portability

We think that an application with functionalities that are similar to social media applications needs to be available in mobile phones. Also with mobile phones we can introduce a new functionality, QR code scanning, to keep track of participation on club events. So, as developers we are building the software with portable libraries and frameworks.

1.2.6 Definitions, Acronyms and Abbreviations

- CM: Club Manager
- Club Manager: The students that run the club.
- Advisor: Faculty Advisor of a club that is appointed by the university.

2. Proposed System Architecture

2.1 Subsystem Decomposition

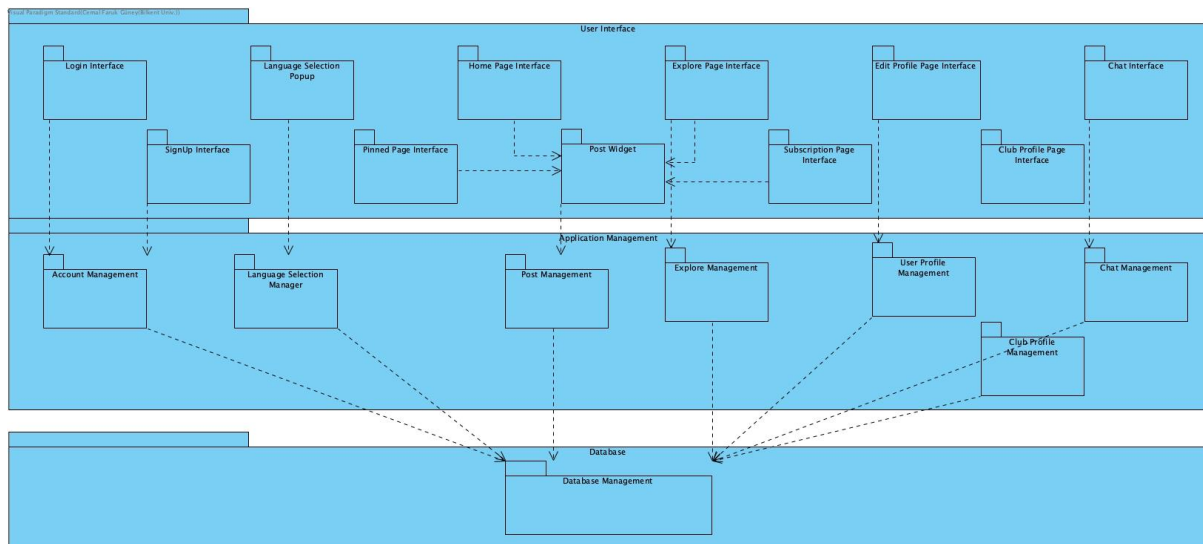


Figure 1: Subsystem Decomposition

There are three layers in our subsystem decomposition.

First layer is the user interface layer. This layer contains the parts of the application that will be seen by our user. Data in our application will be gathered by the second layer and will be presented in user interfaces. Users will also interact with these interface components. The interactions may result in change in data which will be handled by our second layer. In this layer there are some packages with dependencies. All interfaces that contain posts are dependent on the post widget because any change in the post widget will change the dependent components as well.

The subsystems in the first layer are: Login Interface, the pop-up interface where users enter their credentials and log in; Signup Interface, the pop-up interface where new users create their accounts by providing some information; Language Selection Popup, the pop-up displayed to the user that lets them select their language of preference (English or Turkish); Home Page Interface, the first

page displayed to the users after login, it has the posts from the users subscribed clubs together with the posts from suggested clubs; Pinned Page Interface, the user interface that displays the pinned posts of the user; Explore Page Interface, the page that displays the posts suggested to the user in a grid; Subscription Page Interface, the page that displays the user's subscribed clubs; Edit Profile Page Interface, the page that gives the user the opportunity to change the information on their account; Club Profile Page Interface, the page that displays information about a certain club and displays the posts from that club in a grid; Chat Interface, the interface of the club chat where the messages are present; Post Widget, the widget that is displayed in the pages where posts are present.

Our second layer is the application management layer. This layer is sometimes referred to as application logic layer. The subsystems inside this layer are mainly the controllers in the application. If there is a need for change in our data that is triggered by either user input or automatically these subsystems will manage that change. These subsystems will also be used when visualizing the data on the user interface.

The Account Management subsystem deals with the accounts it checks if the input provided by the user matches with any account in the database and it adds new accounts to the database.

The Language Selection Manager subsystem saves the user's preferred language and matches it with their profile.

The Post Management subsystem controls the posts and the actions that can be taken on posts. It can create new posts or delete a created post. It also manages the comments and participation a post gets.

The Explore Management subsystem determines the suggested posts for the user and hands them over to the Explore Interface to display.

The User Profile Management subsystem handles all the information related to a user's profile such as their profile picture and application settings.

The Club Profile Management subsystem is a subsystem that is used to get the posts of a certain club and return them to the Club Profile interface.

The Chat Management subsystem manages the chat of clubs. There are certain actions that can be taken in the Chat Interface such as muting a user and sending a message. This subsystem will control those actions and will interact with the Database Management subsystem in order to save the messages to the database.

All subsystems in this layer are connected to the subsystem called Database Management. Database Management subsystem is inside the third layer.

The third layer is the database layer. The database will contain data related to the users, clubs and events. Database is controlled by the database management subsystem.

2.2 Hardware/Software Mapping

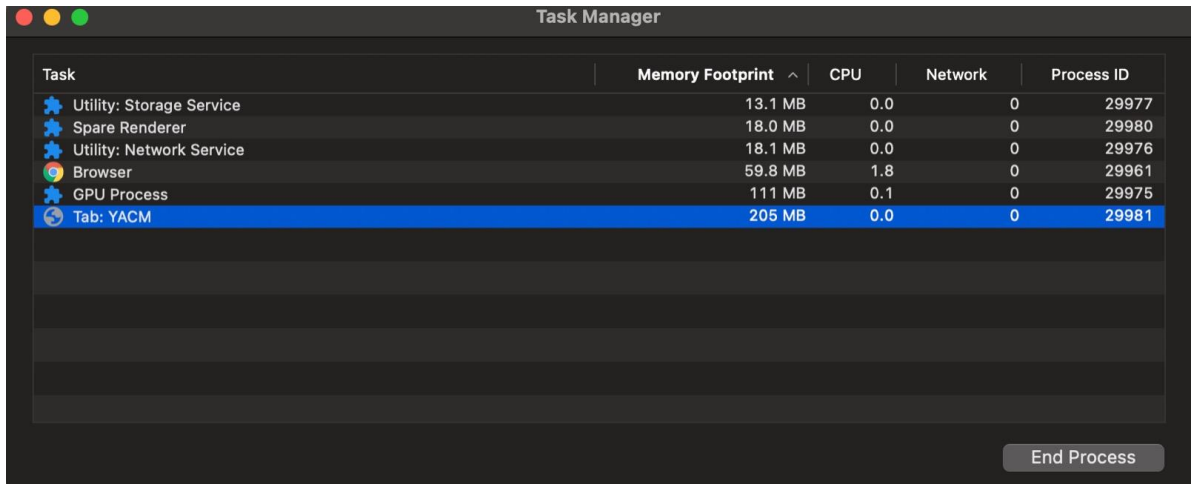
The project's interface is made using Flutter. The project is designed to work on up-to-date browsers and on devices that are connected to the internet. The interfaces will be designed for web usage on both desktop and mobile, and the ease of scalability that Flutter provides helps this work out. In addition, a mobile app version of the project will be deployed, since it is deemed more useful by the mobile users. The QR scanning feature will only be available on mobile platforms since QR scanning on desktop is not very usable.

Memory use of the database system is insignificant, since Firebase is used to read/write operations. The development team pays for the service per read/write operation.

For the RAM usage on a local browser, we have run Google Chrome's embedded Task Manager in order to check for the project's RAM usage in its current state. After testing the RAM use

in different devices of project members, it is found that the memory use ranges from 150MB to 250MB. Since the project is almost fully implemented this is not expected to change much in its deployable form. Also including that the project is run in developer mode, this RAM use will likely drop on a user's instance. There has been no signs of CPU usage in any test.

Simply put, the project is expected to use approximately 150 to 250 MBs of RAM on a local browser, if not less, and no CPU at all.



| Task | Memory Footprint | CPU | Network | Process ID |
|--------------------------|------------------|-----|---------|------------|
| Utility: Storage Service | 13.1 MB | 0.0 | 0 | 29977 |
| Spare Renderer | 18.0 MB | 0.0 | 0 | 29980 |
| Utility: Network Service | 18.1 MB | 0.0 | 0 | 29976 |
| Browser | 59.8 MB | 1.8 | 0 | 29961 |
| GPU Process | 111 MB | 0.1 | 0 | 29975 |
| Tab: YACM | 205 MB | 0.0 | 0 | 29981 |

Figure 2: RAM Usage from Task Manager

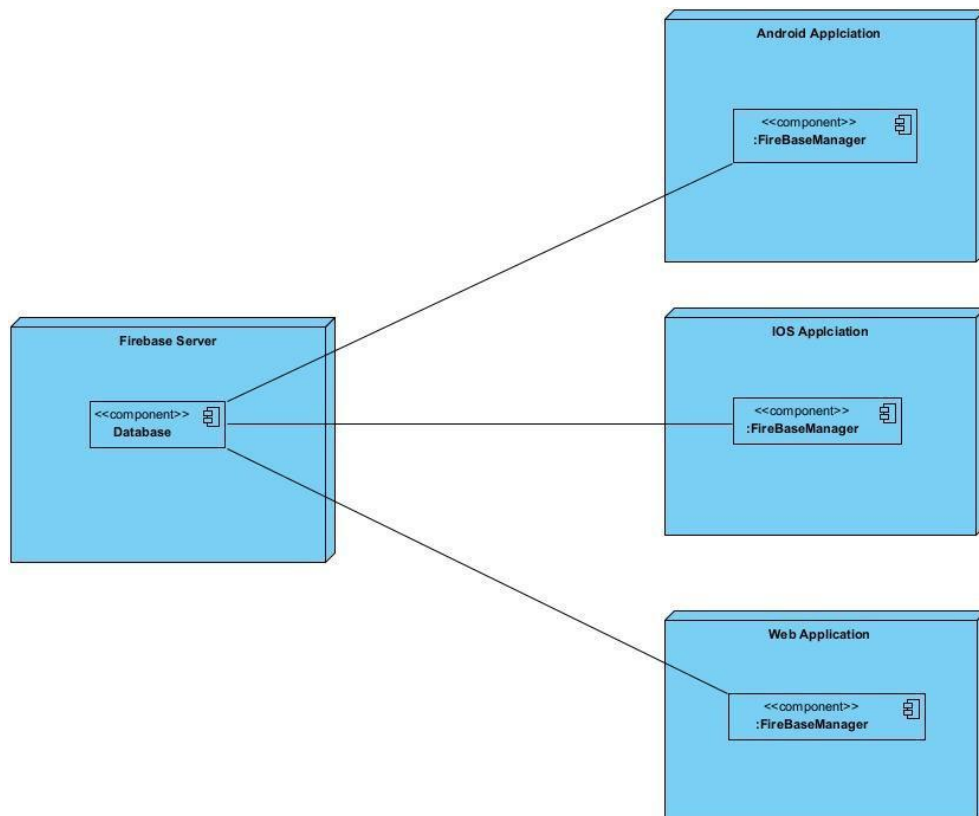


Figure 3: Deployment Diagram of the Project.

Flutter generates IOS, Android and Web applications from the same source code. Our database has been embedded inside Firebase so that we don't have to write code for the backend of our application. All queries are generated from the front-end and sent to the Firebase. The

communication between the Firebase server and our front-end applications are handled by our `FirebaseManager` class which uses Firebase libraries so that we don't have to worry about the type of communication.

2.3 Design Patterns

Singleton Design Pattern: We have used the singleton design pattern when we have to have a single object of that class that has to be maintained across the application. For example, `LanguageManager` and `ThemeManager` classes use the singleton design pattern since we have only one language object that is used to hold and change data related to language and theme on the whole application.

Since Flutter has a built-in system for singletons, it uses widget trees to provide an interface and data flow, we are not implementing our own singleton classes. Our classes are extending a class called **`ChangeNotifierProvider`** which has a method **`notifyListeners()`** and lets all the instances be aware of the update. At the top of the widget tree, a builder for **`ChangeNotifierProvider`** is instantiated with the type of the classes we use which distributes the data to the widget tree.

An example code;

```
return MultiProvider(  
  providers: [  
    ChangeNotifierProvider<ThemeManager>(  
      create: (context) => ThemeManager(),  
    ),  
    ChangeNotifierProvider<LanguageManager>(  
      create: (context) => LanguageManager(),  
    ),  
    ChangeNotifierProvider<UserManager>(  
      create: (context) => UserManager(  
        HiveManager<User>(), FirebaseManager()),  
    ),  
    ChangeNotifierProvider<ClubManager>(  
      create: (context) => ClubManager(  
        PostManager(), FirebaseManager(), MessageManager()),  
    ),  
  ],  
  child: const MyApp(),  
);
```

Chain of Responsibility Design Pattern: We have used the chain of responsibility design pattern when we needed to implement a method that could not be implemented inside one class. The method `publishPost()` is implemented in `ClubManager` and `PostManager` classes. Each of these classes do what's in their reach and pass the rest to the other class.

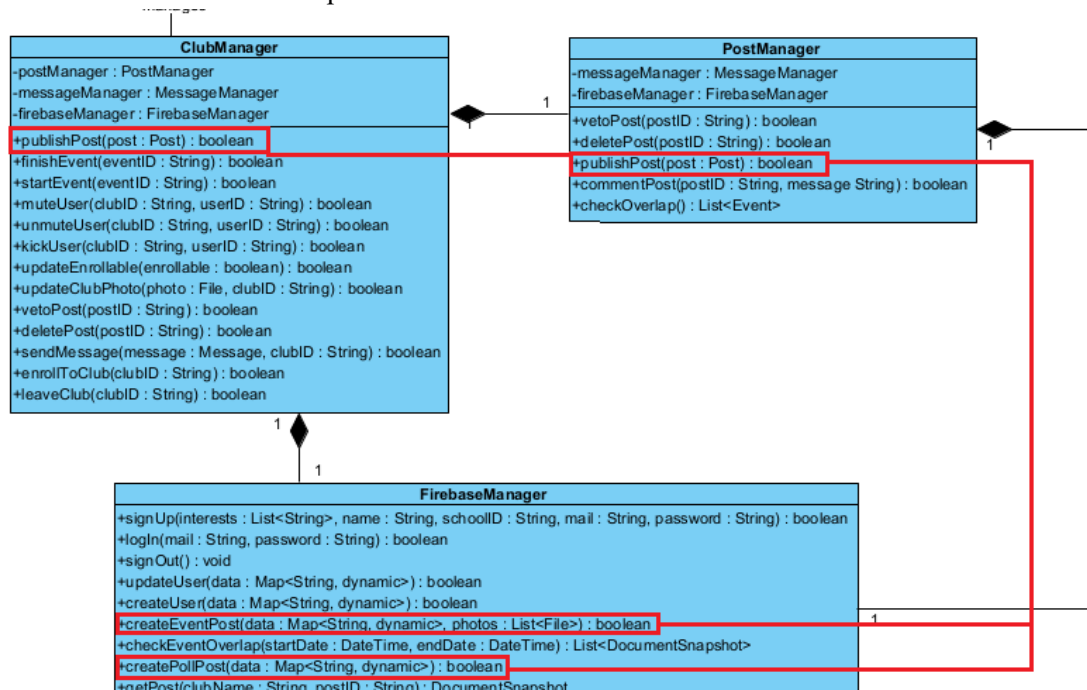


Figure 4: Chain of Responsibility Pattern

Facade Design Pattern: We have used the facade design pattern so that when we have to do some complex stuff, not all the classes need to know about the logic. For example, the ClubManager class uses the MessageManager, FirebaseManager and PostManager to do the complex stuff. In other words, the class ClubManager acts as a level of abstraction of those other classes.

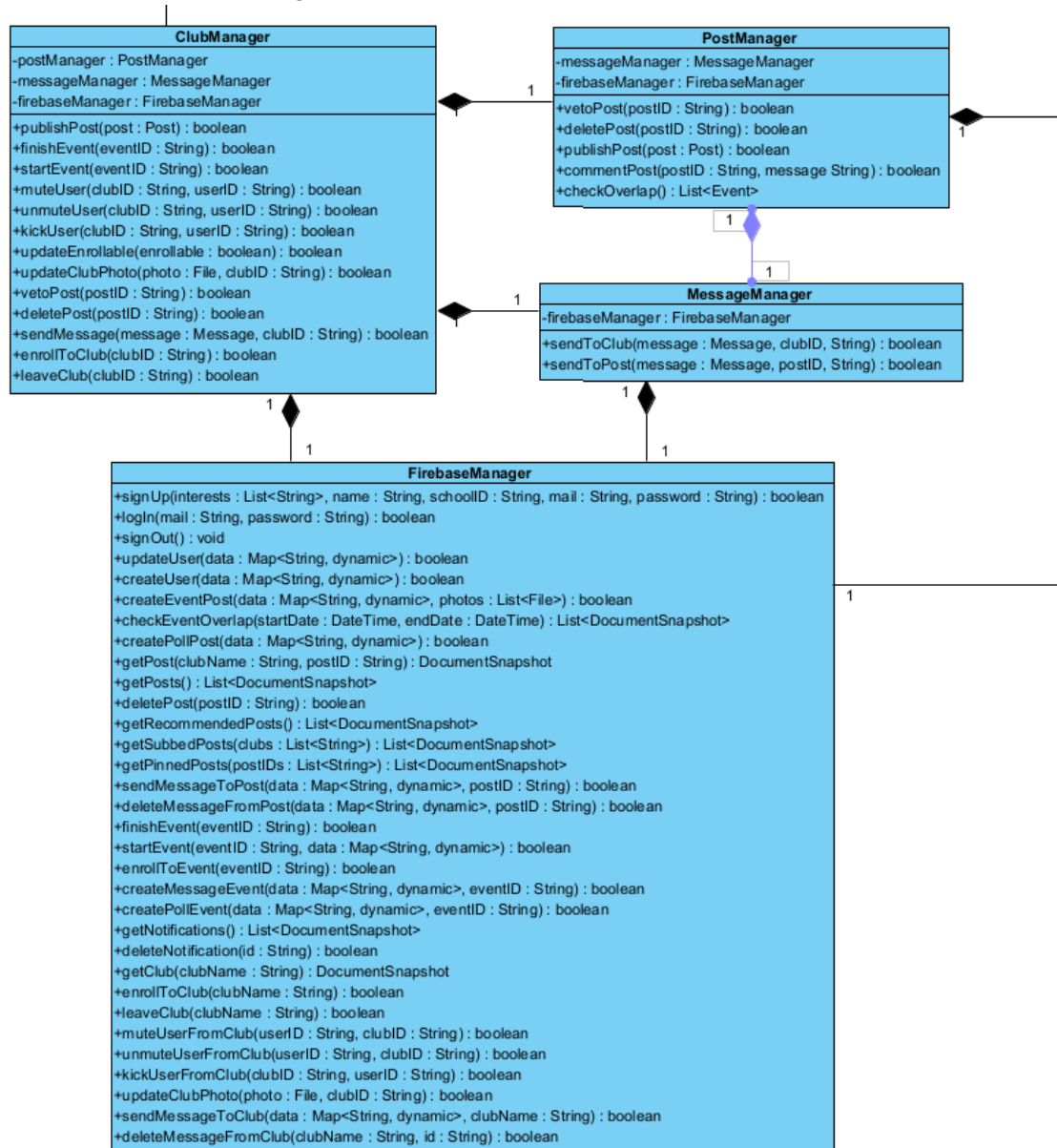


Figure 5: Façade Design Pattern

2.4 Persistent Data Management

The project will use different technologies to store different data. Data that has to be accessed by others will be stored in the Firebase database. There are many reasons why we have chosen Firebase. Firstly, we will use the database wright-heavy which is more suited for NoSQL type databases. Secondly, some of the group members already had experience. Thirdly, Firebase supports Flutter in all platforms. For storing complex local data, we will use a library called Hive. Hive enables

us to store objects locally and supports Flutter on all of the platforms. For storing simple local data like theme preferences, we will use the SharedPreferences library.

The data stored in Firebase include user authentication data (email and password), post data (post photos, event/poll start and end date, participating members/poll answers) and club data (club profile and background photo, description, member list, advisor and CMs).

2.5 Access Control and Security

The project provides access for 4 different users. Users can change the data only through the boundaries provided by the system.

The common users, Viewers, are people who have not logged on and are just visiting the site. These users can be called as ‘read-only’ users since they have no possible interactions with the posts except for viewing them and copying their link.

The next level of authorization is for logged in users who are only members of the club called Members. They can manipulate the data on simple levels, such as adding their name on an event, or uploading their photo etc.

One step further is for special Members who are also a CM of one or more clubs. These users are authenticated to change the club data by interacting with the boundaries through the application. Such interactions include changing the club description, uploading a new club background, posting an event etc.

Advisors are unique users who are Viewers but not Members. They do not interact with the posts; however, they can cancel events of the clubs that they are an advisor of without asking for authorization.

Firebase has a role in determining the authorization of different types of users, since this can be controlled by assigning security rules to accounts. This helps increase security since if a user somehow manages to manipulate data, Firebase will not allow the user to do so.

The application is specific to Bilkent Students, and as a security measure the users have to sign up by using their Bilkent Mail. Another help the system receives from Firebase is login security. Firebase can be used to force users to have login credentials that include a Bilkent Mail. The help from Firebase also includes authentication through sign-up, and resetting password through email, unique user IDs, etc.

There are also admins of the system, who are not users. They can manipulate the data manually through the database if need be. Also, admins are responsible for assigning CMs and Advisors to the clubs after they are reset each term.

Below is the access matrix for YACM.

| | Club | User |
|---------|---|---|
| Viewer | getPost() | login() signup() |
| Member | sendMessage() commentPost() getPost() | addNotification() deleteNotification() logout() |
| CM | publishPost() commentPost() startEvent() muteUser() kickUser() updateEnrollable() updateClubPhoto() deletePost() sendMessage() getPost() | addNotification() deleteNotification() scanQR() logout() |
| Advisor | vetoPost() sendMessage() commentPost() getPost() | addNotification() deleteNotification() logout() |

Figure 6: Access Control Matrix of the Project.

2.6 Global Software Control

2.6.1 Object Design Trade-offs

2.6.1.1 Functionality versus Usability

The system is designed for quick and practical use most of the time. That is the reason usability is favored over functionality. However, there still is some complex functionality that has to be implemented for less frequent use. For those functionalities, usability is less significant of a concern. Therefore, the system will be presented in a user interface that favors usability unless the user decides to use the said functionalities.

2.6.1.2 Cost versus Portability

Even if the time to deliver the system is limited, portability is more important. For the usability goal of the project, in addition to the web application, will contain deployments to mobile platforms, IOS and Android. Because, the system is designed for quick and practical use as well.

2.6.1.3 Cost versus Robustness

To be able to afford the cost, robustness will have to be sacrificed. Except for significant problems like crashes, decreasing the time it takes to deliver will be the main focus.

2.6.1.4 Security versus Usability

The system does not take security measures that may decrease usability of the application. Initially, Bilkent mail address is used to authorize accounts. But there is not an enforced authentication process repeated each time a user enters the application. It is left up to the user's choice to stay logged in.

2.6.1.5 Flutter

In order to provide our service by both web and mobile devices, we needed a framework that would satisfy our needs therefore we chose Flutter.

2.6.1.6 Pub

In order to use some functionalities in our app and to not waste time reinventing the wheel, we use Flutter's Pub package manager to manage third party libraries.

2.6.1.7 Firebase

Since we are developing both for web and mobile, we needed a backend that would work for both of them. Also, since we do not have enough resources to support our own database with a machine of our own, a cloud-based database was a better choice.

2.7 Object Design (Click [here](#) to open the diagram in a higher quality)

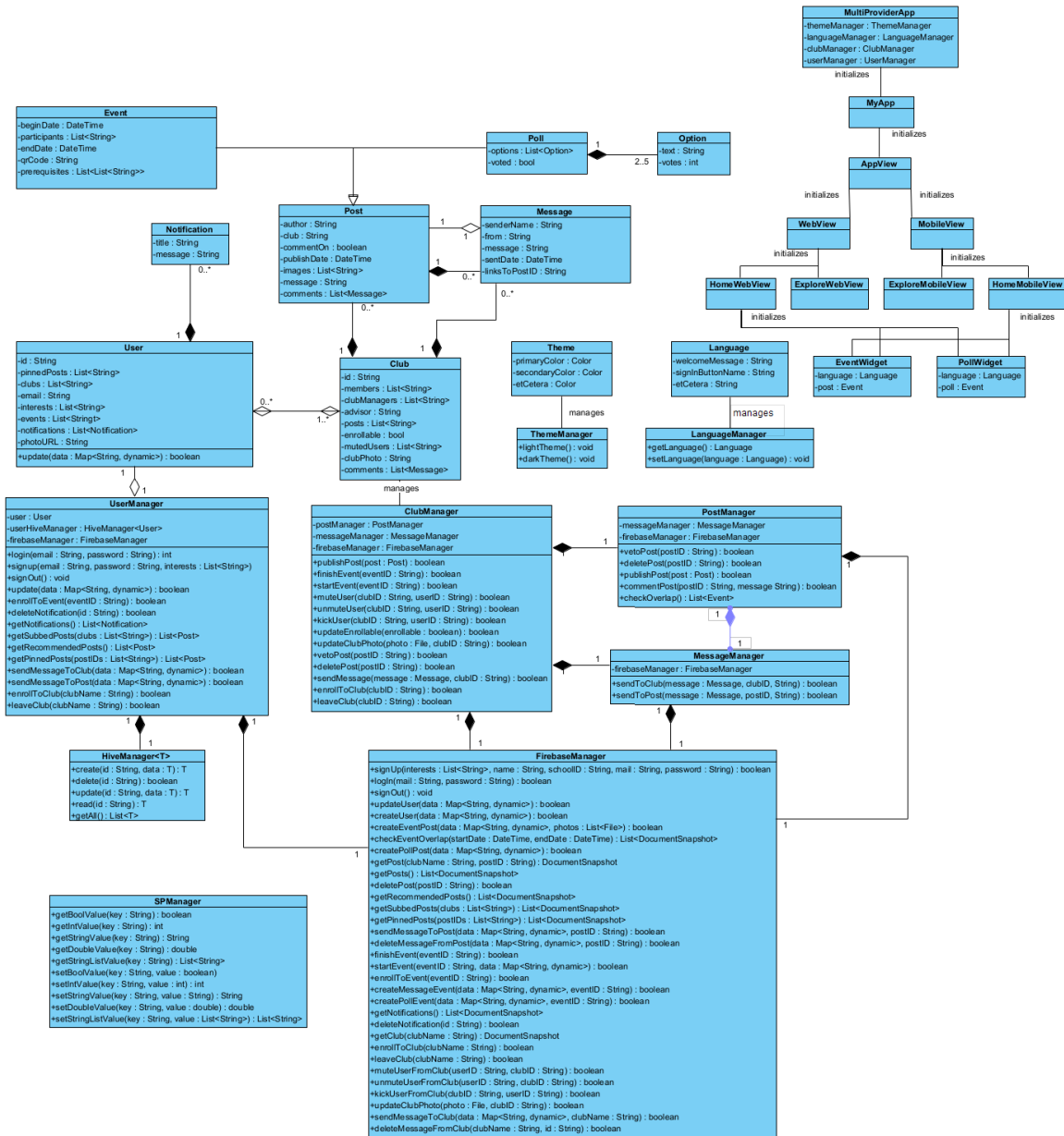


Figure 7: Class Diagram

Above is the Class Diagrams for Dart code.

In the class diagram we have a mini class tree that represents the widget tree of Flutter. The widgets in Flutter are the UI aspects of the framework which are classes that have build methods and extends StatefulWidget or StatelessWidget class. Since all the UI aspects of our application are widget, we just put a representative small class diagram that explains the widget tree.

Because the number of properties in Language and Theme classes were too much to fit in this diagram, only some representational properties were used.

Post: Post is a class that has some core attributes for a post

Event: Event is a class that extends Post class and is specified for just Event Posts.

Poll: Poll is a class that extends Post class and is specified for just Poll Posts. It has Options for the Poll.

Option: Option is a class to implement the options for a better interface that is used with Poll class.

Notification: Notification is a class so that the upcoming notifications have a better format to present the data to interface.

Message: Message is a class that holds the message data that is sent to either posts as a comment or a message to club chat.

User: User is a class to hold the currently logged in user's data.

Club: Club is a class so that the clubs have a better format to present in the user interface.

Theme: Theme is an abstract class that has attributes about the colors of the widgets.

Language: Language is an abstract class that has attributes about the strings of the widgets.

SPManager: SPManager is a class that helps to store primitive data such as boolean and integer in the local machine. It has read and write methods. It has no connections to other classes since it is not needed.

HiveManager: HiveManager is an abstract class that helps to store data of non-primitive objects of type T. It has create, read, update and delete operations.

ThemeManager: Manages the theme operations of the app.

LanguageManager: Manages the language operations of the app.

MessageManager: Manages the operations such as sending messages to chats.

PostManager: Manages the operations of a post such as deleting or publishing a post or to comment on it.

ClubManager: ClubManager is a controller class that has methods of what a club manager can do such as updating club photo, starting events.

UserManager: UserManager is a controller class that manages the events of the current user or a user signing or logging in to the application.

FirebaseManager: FirebaseManager is a controller class that handles the communications between the app and the database. It is used in some other controllers such as MessageManager and PostManager. The user does not have direct access to this class.

2.8 Boundary Conditions

2.8.1 Initialization

The project has three parts: web application, mobile application and Firebase backend. Both mobile and web applications will have similar and relatively simple initialization processes. To use the app there is no need for internet connection. User cannot fetch or post data but he/she will have working software. So there are no checks during the initialization. The Firebase backend will be online 24/7 and does not have a startup sequence other than the very first creation of the database. All details during the creation of the database have been handled by Firebase.

2.8.2 Termination

During the termination of the front-end applications there will not be any loss in local data and any connection with the database will be terminated immediately. The database will not be terminated for any reason other than completely shutting down the software.

2.8.3 Failure

Any temporary loss of connection will be handled by the Firebase library working in the frontend and the backend. When the connection is lost the user will be notified.

Since we are using Google's cloud systems, we are not expecting any bugs/errors related to hardware, connection or power on the server side. Only problems that may occur will be related to our code working in the cloud which will be tested before deployment. So, we are not expecting much problems on the backend.