Functional Properties

We completely implemented every functional property that we mentioned in Deliverable 1. Below are descriptions of how each of them took form in the finished product. The system for ClubWAT includes the front-end part of our application which supports the user interface and the interactions with the Android phone. There is also a back-end server which queries and sends information to the application front-end as well as organizes information in the database, and a PostgreSQL database to store all information such as users, clubs, events, etc.

1. **User Authentication:** As demonstrated in the demo the user sees the login screen on startup of the application and can login with the email/password. Either their account is valid and the client receives a secure JWT to start the user session, or it isn't valid and they're prompted to try again. The user can create their account with basic details and receives an email with a six digit code that they enter in the application to complete the registration and verify their identity.

2. **User Interface:** The system uses the popular four tab bottom bar layout that a user can click any tab to go to that view. The tab bar is visible at all times during the user session.

3. **Club and Event Browsing:** Clicking on the search tab in the bottom bar will go to a view that lets you search for clubs or events based on title or description. It sorts results alphabetically and each item is clickable to view more information about the club or event.

4. **Event Bookmarking and Scheduling:** From the event detail view a user can click the bookmark button in the top row of buttons, this will save the event to their home page to ensure they don't miss it. If they click the '+' button, it will mark in the system that the user is attending the event, making it visible in the home page and optionally prompting to add the event to their phone calendar (google calendar for example) automatically so they can get notified about when the event is starting.

5. **Club Registration and Management:** From the club details view, a user can register for a club by clicking the '+' button. If the club doesn't have a membership fee they will instantly get full access to the club. If it does have a membership fee, a club admin can click the settings icon in the top right of the club details view and manage members. This shows a full list of club members and allows the club admin to mark the user as 'Paid' for when they paid their membership fee. It also lets them remove a user from the club or promote them to a club admin. Note only club owners (the user who created the club) can promote users. Creating a club can be done in the profile page and clicking 'Create Club'. A super admin (for example, WUSA), would be able to approve the club creation request, making that club available to everyone. Club admins can also edit club details such as club title, description and membership fee from the club settings view.

6. **Notifications and Alerts:** The application has many options to send emails to users for notifications and alerts. There is also an inbox view by clicking the inbox icon on the top right of the home page view. The user can manage whether they want to receive email notifications through the profile page and clicking edit profile. System events such as a new event being created, a club join request being approved, and sharing an event or club will send an email to the relevant user.

7. **Feedback System:** Users can 'like' or 'unlike' clubs and events. Liking a club/event will add to the like counter of that club/event. This is how users will give feedback on clubs and events.

8. **Personalized Recommendations:** In the bottom navigation bar, the user can click the 'For You' page which will suggest clubs based on the user's interests in a list. The user configures their interests through the profile view and selecting the edit interests option, then entering their relevant interests.

9. **Event Creation and Management:** Club admins can create an event for their club through the club settings page (settings button on the top right corner of the club details view). They enter all the event details and click the submit button. They can also edit an event by going to the event detail view and clicking the edit icon on the top right of the view.

10. **Interactive Club Discussions:** Users can share clubs/events with their friends by clicking the paper airplane icon in the club/event detail view. They manage their friends in the user profile view and clicking 'Manage Friends'. Users can also send messages in the dedicated club group chat if they're an approved member of the club. Club admins can delete any message in the club discussion, but regular members can only delete their own messages.

Non-Functional Properties

We completely implemented every non-functional property that we mentioned in Deliverable 1. Below are descriptions of how each of them took form in the finished product.

1. **Security:** Passwords are hashed with 2^{11} rounds and 32bytes of salt which is significantly more than the original target of 1000 rounds and 32 bits of salt. Passwords are also required to be strong with more than 8 characters, uppercase, lowercase, and symbol characters.

2. **Privacy:** The user can go to their profile tab and either click the download data button which sends them a file with all of their stored data in the application. They can also scroll down to the 'Danger Zone' and click the 'Delete Account' button to delete all of their data from the application.

3. **Usability:** We used a bottom navigation bar layout in our application, which is visible at all times once the user has logged into the application. Therefore, it will always be possible for the user to get to the For You page within three clicks.

4. **Safety:** We limited the application to only users with a valid **@uwaterloo.ca** email by using the popular code verification technique that's used in many enterprise applications. Since the user needs to enter a code that's sent to their email, and the API requires the email is an **@uwaterloo.ca** domain, the user will never be able to complete the registratiion unless they have access to that **@uwaterloo.ca** email.

Architecture Styles

Client-Server

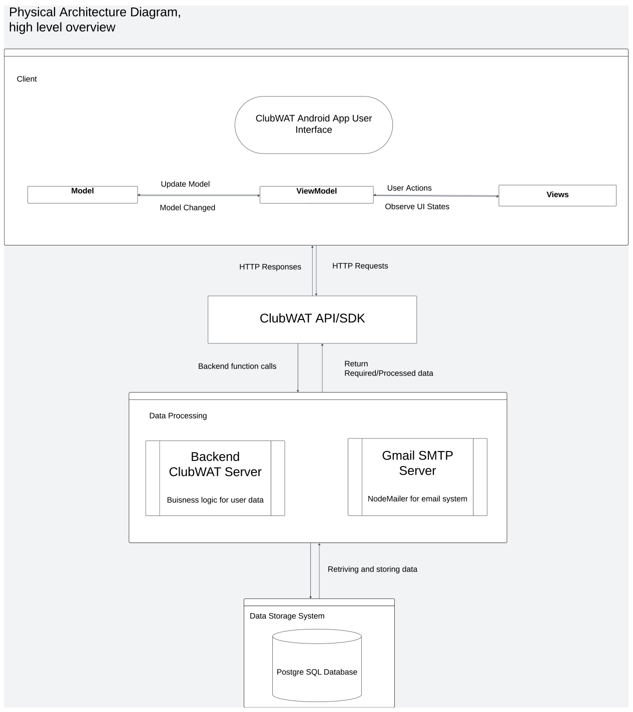
The client-server architecture style is the critical style choice we made for the application. It helps give us a way of modifying a central database (PostgreSQL in Digital Ocean) by calling defined end-points in the Server such as **/user/register**. The server is located under **cs446-project/server/**.

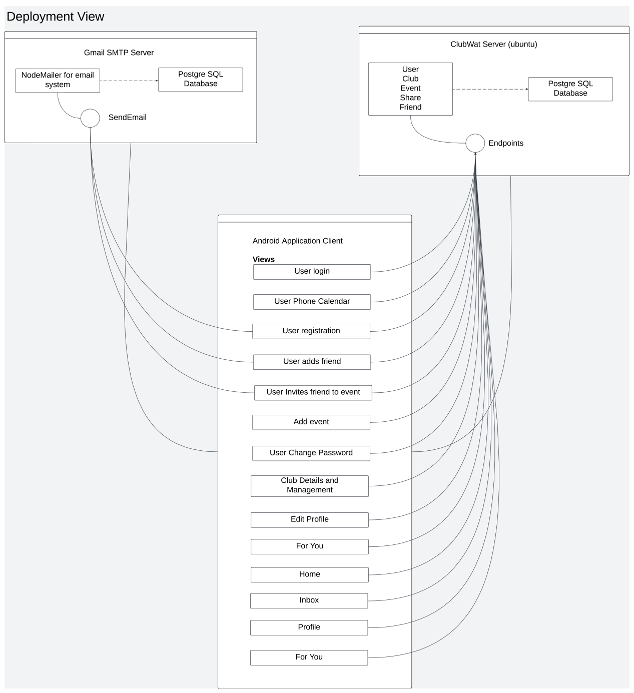
The routes and logic for each of the endpoints are located under server in **/src/api/**. The server endpoints are called from the client, which in this case is the android application which is located under **cs446-project/app/**. The ViewModels call the server endpoints, which are all located under app in **/clubwat/viewmodels/**. An example of calling an endpoint can be seen in **LoginViewModel.kt L51**.

Model-View-ViewModel (MVVM)

The MVVM architecture style is used heavily within our application. This is essential for the functioning of our application. The model layer is available in **/app/src/clubwat/model/**, which are all Kotlin data classes that define the structure of our raw data. The view layer handles the presentation and structure of the views such as where the buttons, text boxes, etc are located and it is available in **/app/src/clubwat/views/** which stores the composable views that leverage Jetpack Compose to function. Finally, the ViewModels are stored in **/app/src/clubwat/viewmodels/** and each view has a corresponding ViewModel. ViewModels define how the data is manipulated and provided to the view by operating on the Models and communicating with the server from earlier.

High Level Overview and Server Deployment View





Design Description - Frontend

System Structure and Rationalization for Dependency Injection

In terms of the design of our project, we decided to use the Dependency Injection (DI) framework by Hilt. Hilt was used for DI to manage dependencies such as repositories within view models. DI allows us to create objects with their dependencies provided externally, enhancing modularity.

That is, these objects are constructed with the dependencies they need to function handed to them, rather than these objects creating instances of these components themselves. This helps with modularity through re-usability, decoupling, and boilerplate code reduction.

System Structure and Rationalization for Model-View-ViewModel

Additionally, we also used the Model-View-ViewModel (MVVM) architecture. The model layer in this application consists of data classes such as 'Club', 'User', 'Event', etc. These are Kotlin classes that are intended to define the structure of the data we are working with. By defining them here, it makes it easier to fetch and store data from both the front-end and the back-end. The view layer includes '@Composable' functions like 'LoginView', 'HomeView', 'SignUpView', etc. These views define the user interface (UI) of the application. The '@Composable' tag renders the UI based on the state it receives from the ViewModel. The view models are used to provide connections between the model and view layers. Each view has a corresponding view model class in our application. It's purpose is to retrieve data from the model, apply logic to the view, and then show the processed data to the end user. More specifically, the view model exposes state and event handling methods to the view layers. An example within our application is the 'allClubs' variable which is exposed as a'MutableStateFlow' property, which the UI components observe for changes. Functions like 'getAllClubs()' are used to trigger data fetching from the network or database. The MVVM pattern was used for this application because of its support for reactive programming. That is, MVVM allows our UI to react to changes in the view model's state. This helps with creating a responsive user experience. By using MVVM, we were also able to decouple the UI logic from the model logic.

Alternatives

Comparing our decision to alternatives, Model-View-Controller (MVC) and Model-View-Presenter (MVP) patterns did not seem suitable for our application as they did not provide seperation of concerns like MVVM. MVVM is also better for reactive programming, which we utilized a lot within our application. When it came to deciding what to use for DI, we also considered Dagger. However, Hilt is tailored for Android, making it a clear choice for our application. Finally, we chose Jetpack Compose instead of the XML-based way of creating views as it lets us outline what our application looks like in a more adaptable way. More specifically, with Jetpack Compose, it's more efficient to make the application's UI change whenever the data changes due to its design.

Design Description - Backend

System Structure and Rationalization for RESTful Architecture

In the back-end of our system, we used an approach that revolves around the REST architectural style to ensure a scalable and efficient server-side application. The back-end is created with many RESTful routes, each corresponding to a specific domain within our application. A few instances of domains include clubs, events, discussions, and user profiles.

System Structure and Rationalization for PostgreSQL Database

We have chosen to use a PostgreSQL database to store data for this application. This is because PostgreSQL provides ACID (atomicity, consistency, isolation, and durability) compliance. Not only this, but it has strong support for maintaining data integrity and performance when multiple users access the database simultaneously. Additonally, we have chosen Prisma as our ORM (Object-Relational Mapping) tool to interact with our PostgreSQL database. Prisma provides a strong typing system that works with TypeScript's type safety. Additionally, Prisma's migrations and data modeling give us a robust framework to define our application's data schema.

System Structure and Rationalization for Middleware

All of our routes in the back-end also use middle-ware for authentication and serves as access control. This was added to ensure that the user is logged in and has the permissions to perform certain actions. An instance of this is making sure a only club admins can see options to edit clubs or events. This is important for defining our endpoints and protecting sensitive data. That is, we do not want club users to be able to tamper with club information.

Alternatives

In terms of alternatives, a non-RESTful architecture, such as SOAP, could have been used for server-client communication. This is because these protocols are capable of handling requests and responses but with different conventions. However, RESTful architecture was chosen due to its large use and familiarity, along with its compatibility with the HTTP protocol. In terms of databases, other database systems like NoSQL databases such as MongoDB could have been used. Our group preferred PostgreSQL for its ACID compliance, which is critical for data integrity. Finally, instead of middle-ware, we did look into client-side validation. However, this would lead to authentication and authorization being tightly coupled with application logic, which is not ideal. Hence, we opted to go this with middle-ware authentication.

Key Patterns

Some key classes, abstractions, and algorithms in our application include password hashing, email services, and client state management in UI repositories. These components are important in ensuring the project's security, reliability, and efficiency.

Password Hashing

Our project utilizes password hashing, facilitated with bcrypt, to encapsulate the complexity of password encryption and validation. Bcrypt is a secure password hashing service that provides a simple interface for hashing passwords by abstracting the details of cryptographic algorithms. We use it in combination with salt generation and comparison mechanisms, to guarantee that user passwords are securely stored and validated. This significantly reduces the risk of security breaches and exploitation of the data that is held. Additionally, the abstraction of the original algorithms used behind our hashPassword and compareHash functions allows the security measures of our project to be maintained and updated regularly without impacting wider application logic.

Email Services

Our application facilitates communication with users by email through our EmailService class which abstracts the difficult details of interacting with the Gmail SMTP server. Essentially, it uses node-mailer to securely send emails using Gmail's infrastructure which encapsulates the details of email transmission behind several user-friendly methods. We abstract this service so that it can be used to send many different types of emails such as verification, event and club sharing, and club joining requests. The centralized design and implementation of this service allows for efficient management of user notifications and consistent communication patterns throughout the application.

Our project design promotes code reusability and separation of concerns by modularizing our services into a library. Then, by integrating these services into the API layer, it provides a close-knit interface that handles the application's more complicated interactions. The approach we used for integration helps significantly boost the reliability of our application by ensuring the various components work well together.

User Repository

The UserRepository class in our application is responsible for managing the client state within the UI. It uses a reactive state management approach to abstract the modification and storage of user data to make sure that the UI always contains the latest user information and continuously remains responsive. Encapsulating user data within a dedicated repository is pivotal in our project design as it allows the application to efficiently manage user sessions, authenticate users, and update user information throughout. This class helps simplify state management within the UI and enhances the scalability of the system.

Design Patterns

Dependency Injection

Dependency Injection (DI) is a design pattern that allows you to remove the hard-coded dependencies between objects in your application and, instead, provide them with their dependencies through a central location. This makes it easier to change the dependencies of an object without affecting the rest of the application, and also makes it easier to test your application by providing mock dependencies in a test environment.

We used a type of DI, called Constructor Injection, to primarily improve the entire performance of ClubWAT. Typically in an MVVM architecture, whenever the app opens a new View, it instantiates a new ViewModel with the Factory Design pattern, which affects performance - we implemented this pattern initially. We resolved this issue entirely by injecting a ViewModel instance through the Hilt framework and replaced the code for the Factory design pattern. An example of this can be found in **ApproveClubView.kt**, where the ***ApproveClubViewModel*** is being injected into ***ApproveClubView***.

Chain of Responsibility

Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain. This design pattern was used in our backend APIs in the form of middleware to handle authorization, segmented into a chain of handlers - this acts like a security layer between the API and the requester. When any one of the handlers fails its verification process, then the entire process stops and the API is never executed. You can find the list of handlers in the file **cs446-project/server/src/middlewares.ts**. Specifically, you can take a look at the middleware function ***authenticateToken***, which is used to verify that the requester is a real user of the ClubWAT app rather than a hacker that's trying to access or corrupt data in the database.

If we want only club admins to call an API, then we attach the ***verifyIsClubAdmin*** function to the chain of middleware. Alternatively, if we would like only the super admin to access an API, then we can attach ***verifyIsSuperAdmin*** to the chain. We add them to the chain because at the end, we still want to check if the user is really from ClubWAT with ***authenticateToken***.

An example of this can be found in the file **ClubAdminRoutes.ts** within the first GET API */:id/members* that uses two handlers, ***authenticateToken*** and ***verifyIsClubAdmin***, in the form of a chain. Notice that we have transformed particular behaviors into stand-alone objects called handlers rather than intertwining code for different responsibilities - this is the beauty and benefit of the Chain of Responsibility design pattern.

Minimizing Coupling

Dependency Injection

The Dependency Injection design pattern reduces coupling as it removes the responsibility of finding the dependency from a class locally and places that responsibility elsewhere - this is known as Inversion of Control (IoC). Dependencies are provided to classes externally through injection, such as Constructor Injection and Property Injection. This allows classes to work with interfaces rather than concrete implementations - different implementations of the interface can be called, thereby reducing coupling.

Chain of Responsibility

The Chain of Responsibility design pattern reduces coupling as it separates authorization logic for different ClubWAT roles, like ClubMember and ClubAdmin, into separate handlers - making it easier to read, modify, and test these handlers. Handlers can now be easily added onto or removed from the middleware chain of an API without affecting other handlers, thereby reducing coupling.

Future Requirement Changes

Dependency Injection

The Dependency Injection design pattern enables us to build new Views to the frontend of Club-WAT in the future without affecting performance substantially as we can inject dependencies into a ViewModel instead of creating new instances. In the future, as we expand the capabilities of the software, we can expect the introduction of entirely new features, which means more Views, ViewModels, and Models will be required for ClubWAT.

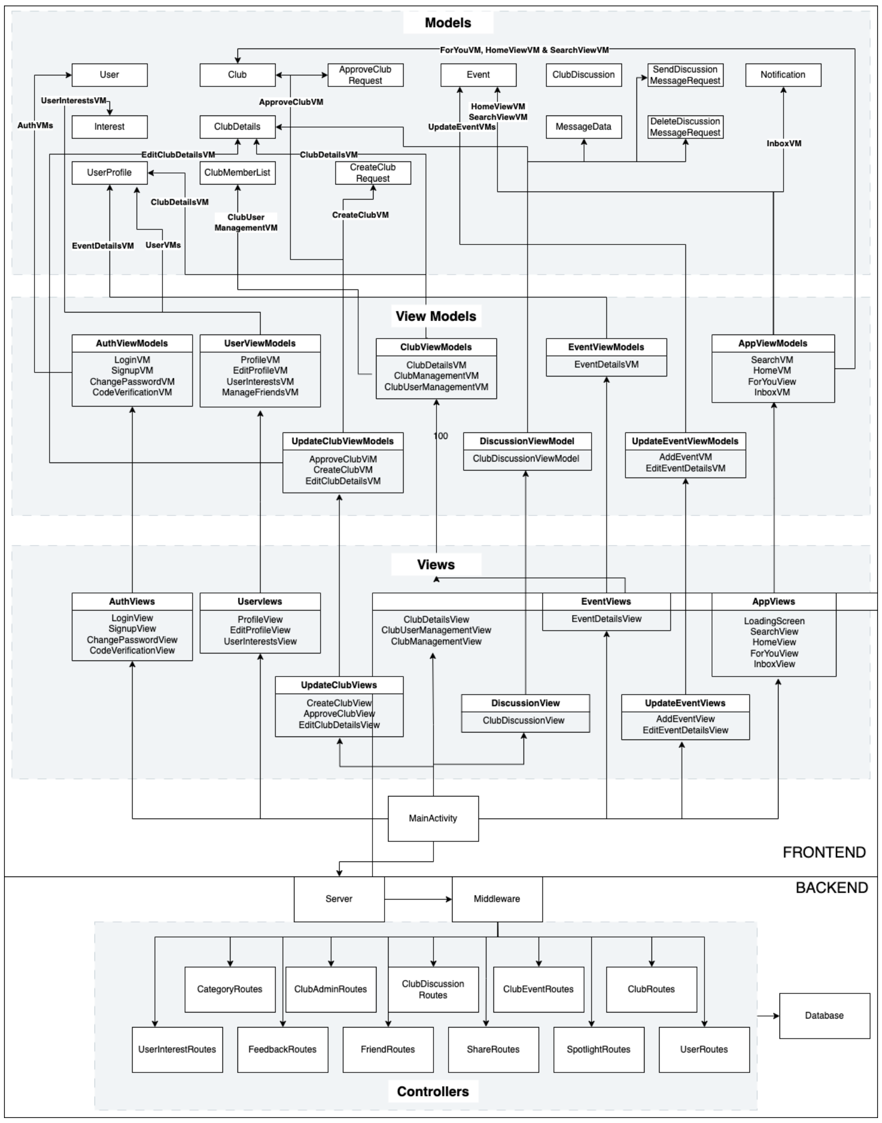
An example of this would be implementing a feature to directly pay on the app through a payment system, like PayPal or Stripe, instead of paying externally to make ClubWAT even more of a central hub for all things related to clubs at Waterloo. We can call the payment screen as ***PaymentView***. The ***PaymentViewModel*** would surely need the details of the user so that it can be passed onto the third-party payment system. Instead of creating a new instance of ***UserRepository***, we can have it injected into ***PaymentViewModel*** directly. The ***PaymentViewModel*** can then also be injected into ***PaymentView***. These injections would improve performance substantially when compared to creating their instances.

Chain of Responsibility

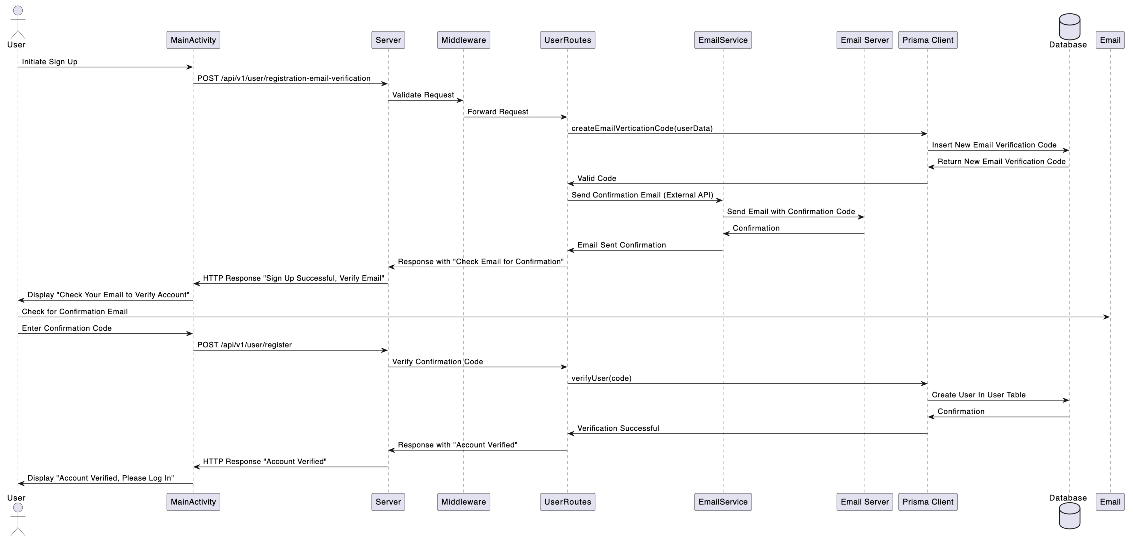
The Chain of Responsibility design pattern allows for the support of as many role authorizations (handlers) as we need by adding them onto the middleware chain. In the future, as we expand the software, we can expect the software requirements to introduce brand new user roles to ClubWAT.

An example of this can be a ClubFinanceAdmin, who is the only one allowed to approve membership payments being made by incoming and recurring members of a club on ClubWAT. Adding authorization checks for APIs is very easy with this design pattern. We can create a new handler, ***verifyClubFinanceAdmin***, and attach it to the middleware of all the GET, POST, PUT, and DELETE APIs for club membership payments of a specific club.

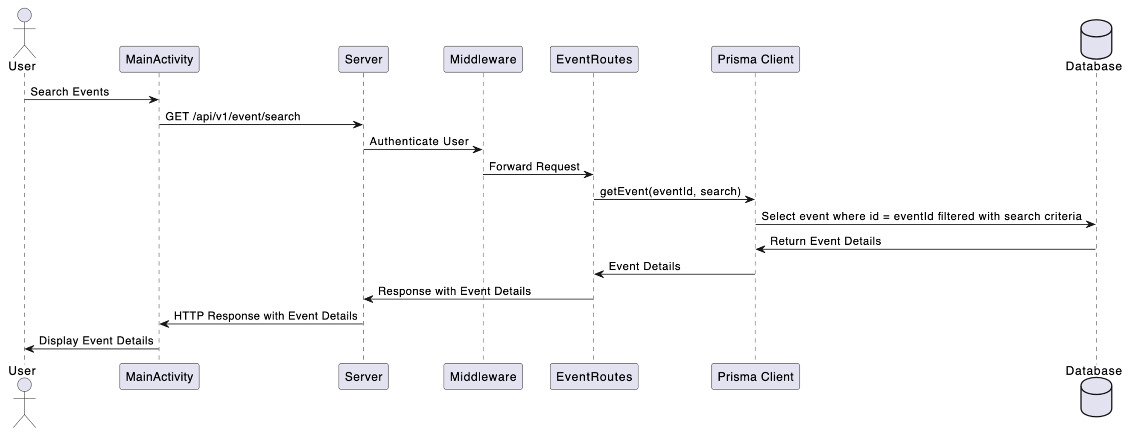
High Level Class Diagram



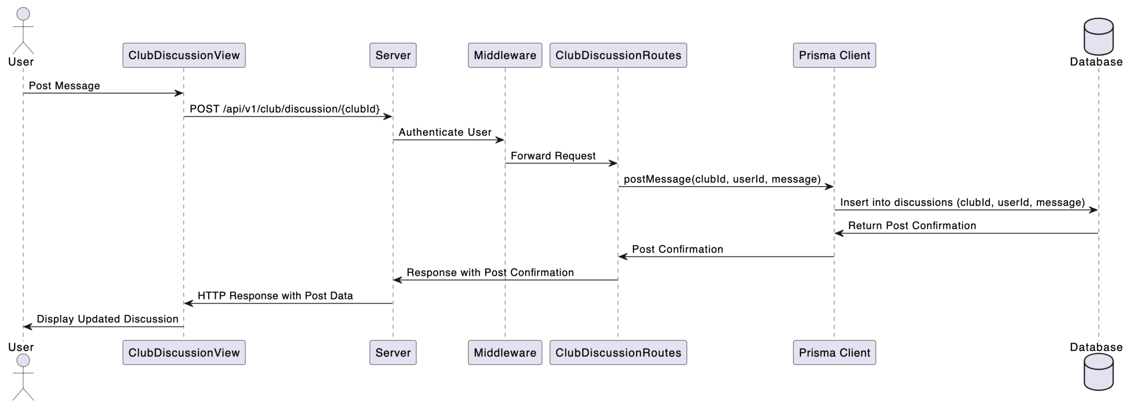
Signup Scenario


User
MainActivity
Server
Middleware
UserRoutes
EmailService
Email Server
Prisma Client
Database
Email
Initiate Sign Up
POST /api/v1/user/registration-email-verification
Validate Request
Forward Request
createEmailVerticationCode(userData)
Insert New Email Verification Code
Return New Email Verification Code
Valid Code
Send Confirmation Email (External API)
Send Email with Confirmation Code
Confirmation
Email Sent Confirmation
Response with "Check Email for Confirmation"
HTTP Response "Sign Up Successful, Verify Email"
Display "Check Your Email to Verify Account"
Check for Confirmation Email
Enter Confirmation Code
POST /api/v1/user/register
Verify Confirmation Code
verifyUser(code)
Create User In User Table
Confirmation
Verification Successful
Response with "Account Verified"
HTTP Response "Account Verified"
Display "Account Verified, Please Log In"

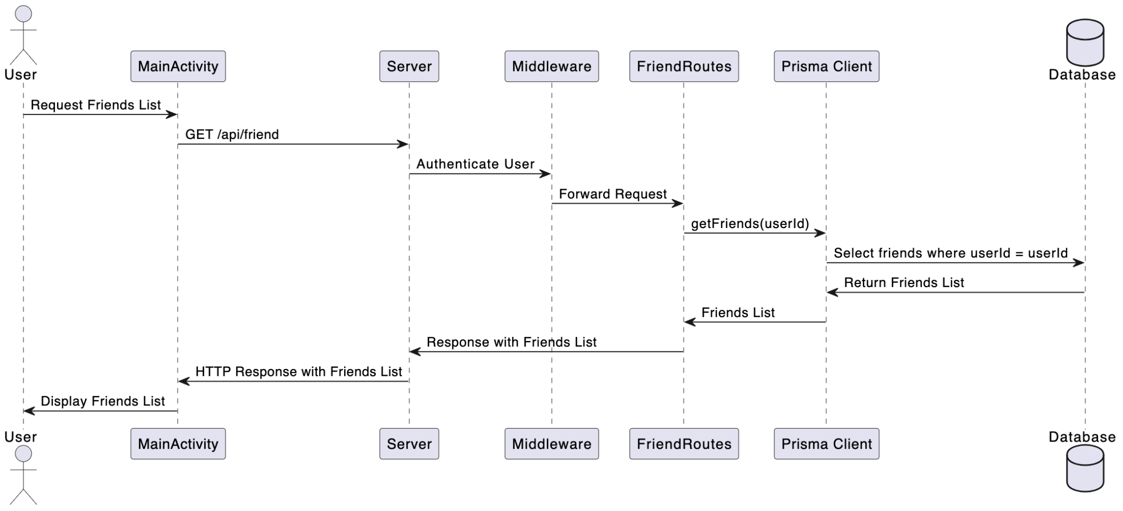
Search Event Scenario



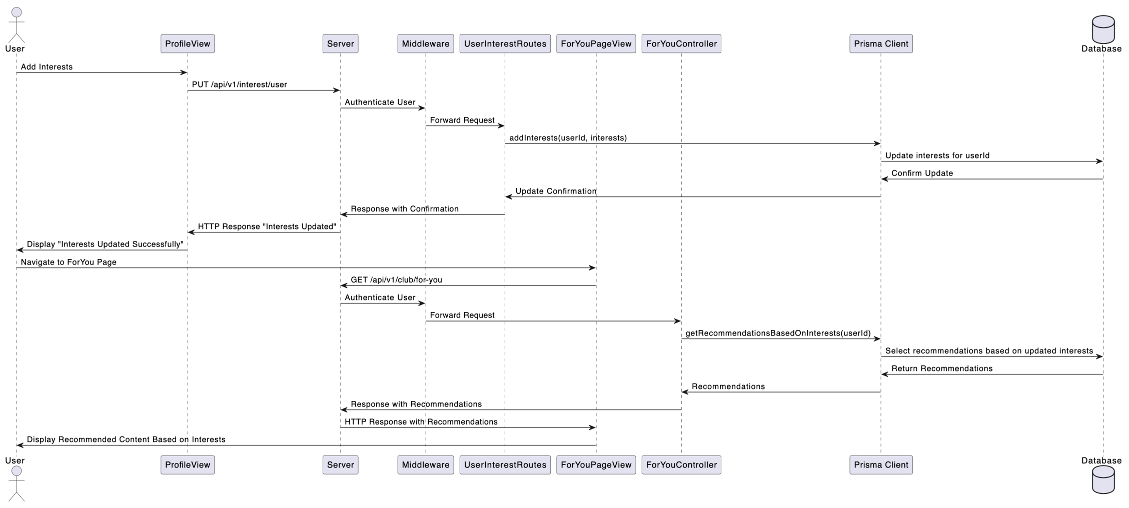
Send Discussion Message Scenario


User
ClubDiscussionView
Server
Middleware
ClubDiscussionRoutes
Prisma Client
Database
Post Message
POST /api/v1/club/discussion/{clubId}
Authenticate User
Forward Request
postMessage(clubId, userId, message)
Insert into discussions (clubId, userId, message)
Return Post Confirmation
Post Confirmation
Response with Post Confirmation
HTTP Response with Post Data
Display Updated Discussion

View Friends Scenario



Update Interests Scenario


User
ProfileView
Server
Middleware
UserInterestRoutes
ForYouPageView
ForYouController
Prisma Client
Database
Add Interests
PUT /api/v1/interest/user
Authenticate User
Forward Request
addInterests(userId, interests)
Update interests for userId
Confirm Update
Update Confirmation
Response with Confirmation
HTTP Response "Interests Updated"
Display "Interests Updated Successfully"
Navigate to ForYou Page
GET /api/v1/club/for-you
Authenticate User
Forward Request
getRecommendationsBasedOnInterests(userId)
Select recommendations based on updated interests
Return Recommendations
Recommendations
Response with Recommendations
HTTP Response with Recommendations
Display Recommended Content Based on Interests