Smart Education Events System

SOEN 343 Section WW:
Software Architecture and Design

Phase 3:
Applying Design Patterns, Updating Class Diagram, and Implementation

Lab Section WL -X

Presented to:
Krupali Dobariya

Italian Stallions

Contributors:
(Team lead) Massimo Caruso - 40263285
Jammie Assenov - 40174965
Alessandro Tiseo - 40262416
Abdullah Taha - 40261146
Gabriel Rodriguez - 40208888
Vlad Tita - 40209853

Submission date: March 30, 2025

Table Of Contents

1.0 **Summary of the project.....3**
2.0 **Design pattern.....4**
2.1. Observer Pattern.....4
2.1.1 Notification Management.....4
2.1.2 Contact Form Submission.....6
2.2. Strategy Pattern.....8
2.2.1 Payment Processing.....8
2.2.2 Sign up/login.....9
2.3. Command Pattern - Event Actions.....11
2.4. Template Method Pattern - Report Generation.....12
3.0 **Class diagram.....14**
4.0 **UI bank.....15**
5.0 **Implementation.....15**

1.0 Summary of the project

The Smart Education Events System (SEES) is designed to facilitate the creation of educational event planning and management. It combines multiple aspects of event planning and management such as event scheduling, stakeholder and attendee management, networking, promotion, analytics, and secure payment processing. SEES leverages modern technologies (React.js, Node.js, Socket.IO, PostgreSQL, and NoSQL) to deliver an intuitive and scalable platform that caters to the needs of educators, learners, and event organizers. The system is built to provide real-time interactions, data management, and a seamless user experience that empowers educators and learners alike.

Here are some key features of our system:

(1) Event scheduling and management: Easy and accessible creation, editing and deletion of educational events through an intuitive interface

(2) Multi-role authentication: Authentication system that differentiates between organizers, attendees, and administrators, giving a tailored experience to each role with different permissions

(3) Report generation: Streamlined process for generating different reports such as attendance analytics, engagement reports, etc.

(4) Networking and engagement: Multiple features were implemented to facilitate the ability of users to connect through a friend request system and to allow chat system to provide the necessary engagement.

(5) Notification system: Real-time notifications from different sources (SMS, emails, etc.)

Our SEES is now built around structured design patterns, which gives it a whole new level of scalability, modularity and ease of maintenance. We have a strong foundation for our system which makes it easy to add new features and improve upon existing ones without much effort.

2.0 Design pattern

2.1 Observer Pattern

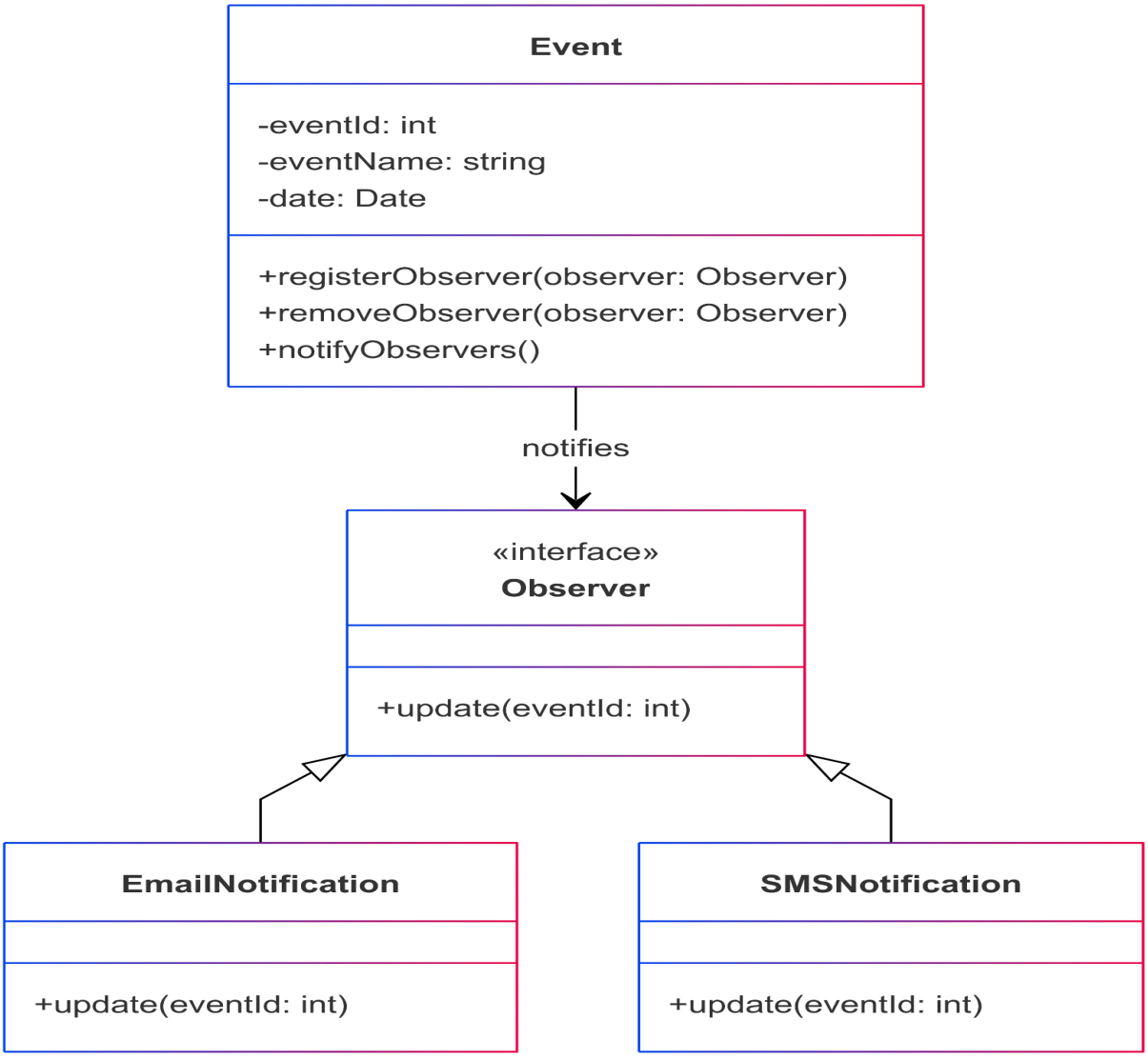
2.1.1 Notification Management

Problem: In our initial system, notifications related to event changes (such as updates to schedules or event details) were directly tied to the core event management logic. This tight coupling meant that whenever a new notification channel (such as SMS, email, or push notifications) needed to be added or modified, substantial changes had to be made to the event management code itself. This approach led to increased complexity, reduced flexibility, and higher maintenance costs.

Solution: To address this issue, we applied the Observer design pattern. In this pattern, we introduce the Event class as the subject, which maintains a list of observers such as EmailNotification and SMSNotification. Whenever an event update occurs, the Event class independently notifies each observer without being concerned about their specific implementations. Observers then handle the notification delivery.

Consequence if not used: Without implementing the Observer pattern, any addition or modification of notification channels would require direct modifications to the event management codebase. This would significantly reduce the maintainability of the system, making it challenging and error-prone to introduce or alter notification methods. Future scalability and flexibility would also suffer due to tightly interwoven dependencies.

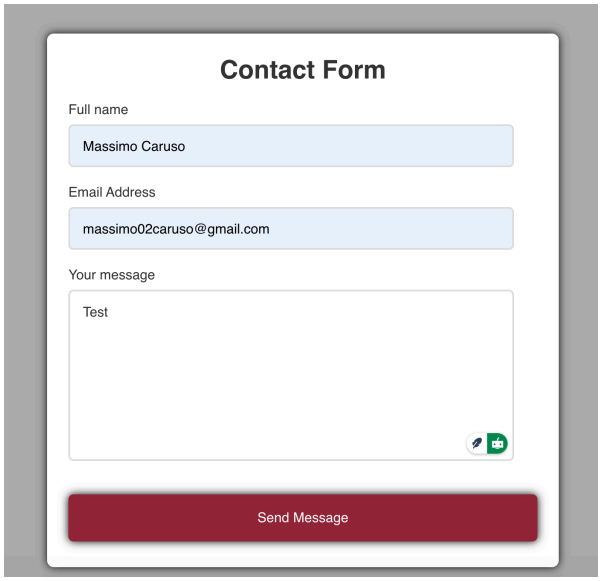
Class Diagram:

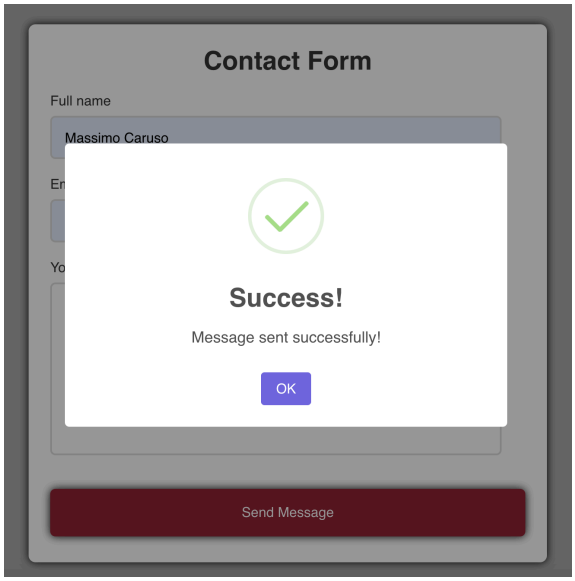


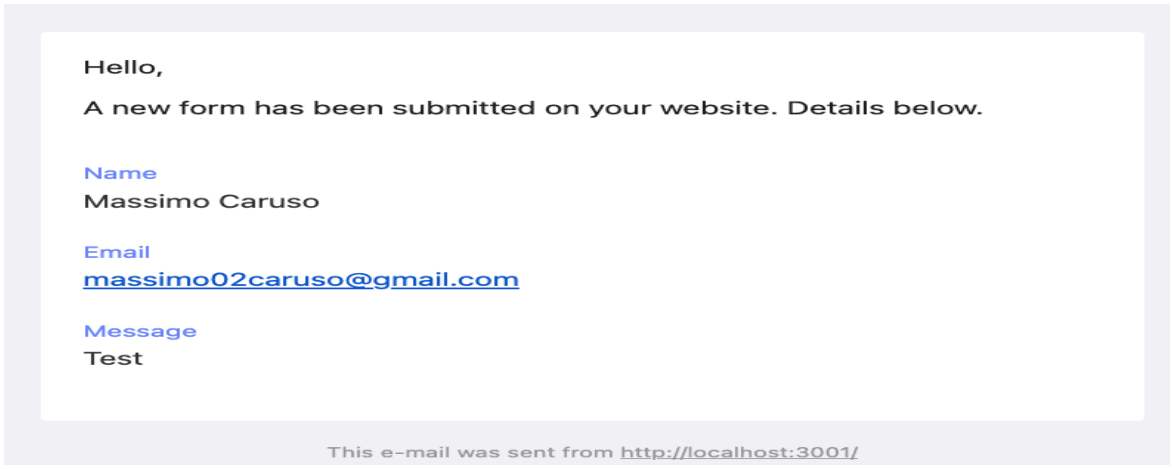
2.1.2 Contact Form Submission

Problem: When a user submits a message through the contact form, the system must process the request and send an email notification. A straightforward but rigid approach might involve directly calling an email service within the form submission handler. However, this approach makes the system tightly coupled, reducing flexibility if we later decide to extend the functionality, such as logging messages in a database, sending a copy to a CRM system, or notifying multiple recipients.

Solution: The Observer Pattern allows the system to decouple the form submission logic from the email-sending process. The contact form acts as the subject (observable), while different notification services—such as an email sender, a message logger, or an SMS notifier—act as observers that react when a new message is received. This pattern makes it easy to extend the system without modifying the core form submission logic. For example, if we want to send an SMS alert in addition to an email, we can simply add a new observer without changing the existing notification mechanism.

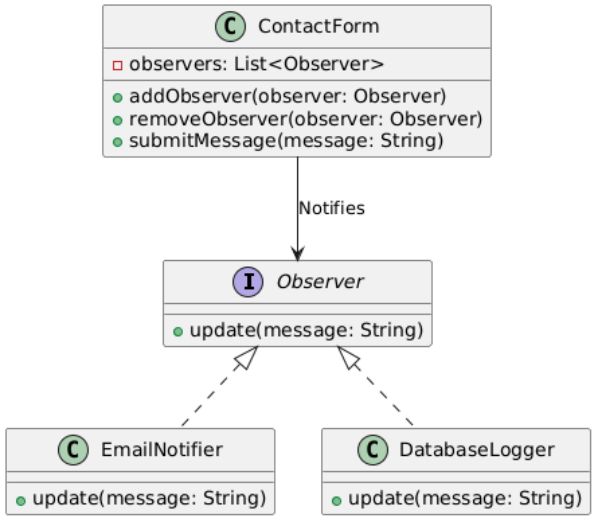




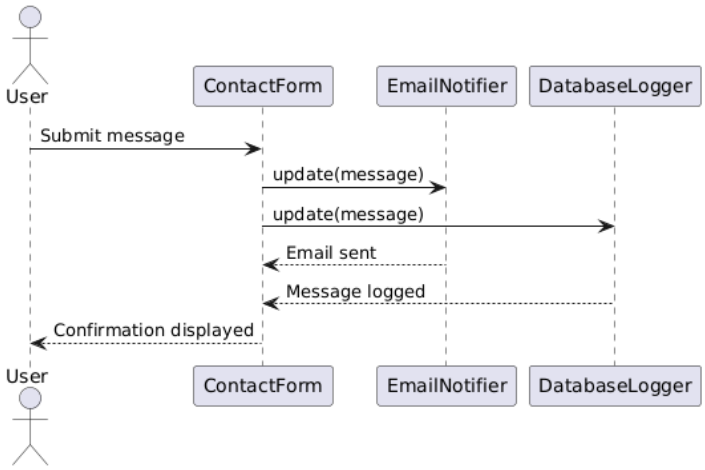


Consequences if not used: Without the Observer Pattern, the contact form would be tightly coupled to the email service. If additional notification methods were needed, the form submission logic would require modification, making the system harder to maintain and extend. The Observer Pattern, on the other hand, ensures flexibility, allowing new observers to be added without altering the existing form submission process.

Class Diagram:



Sequence Diagram:



2.2 Strategy Pattern

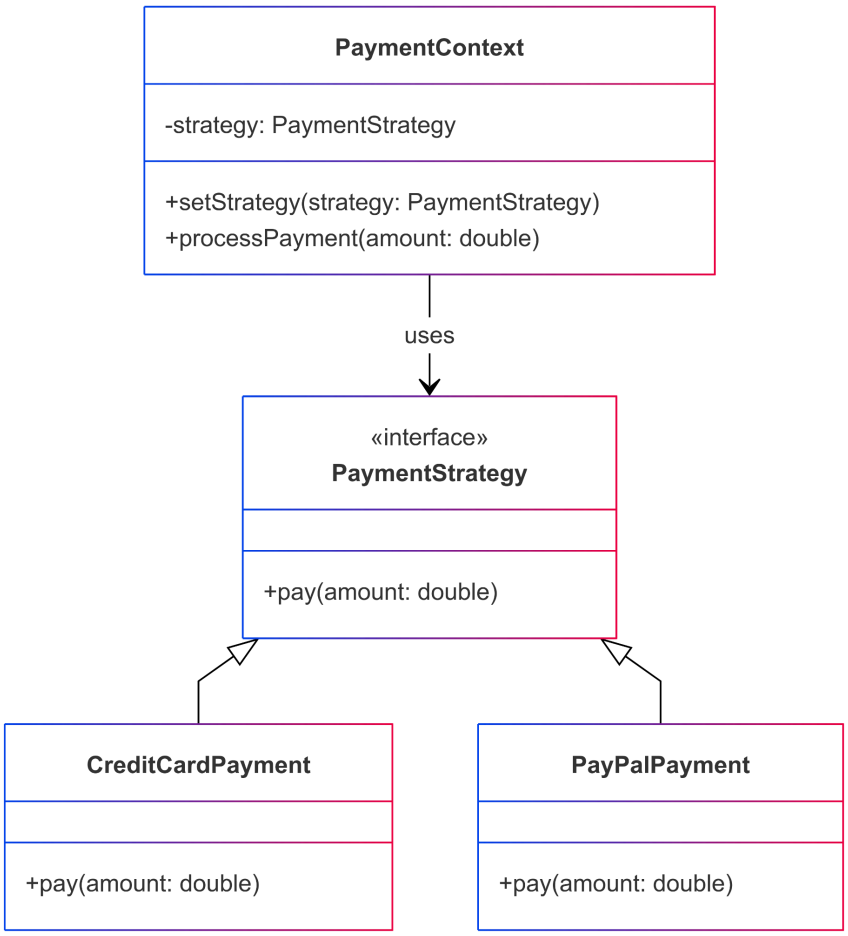
2.2.1 Payment Processing

Problem: Initially, our system directly embedded various payment methods within the registration logic. This meant each new payment method required invasive code changes, reducing flexibility and complicating future adaptations.

Solution: We implemented the Strategy pattern, creating a PaymentContext class that dynamically selects payment strategies (e.g., CreditCardPayment, PayPalPayment) at runtime. This encapsulation allows for the easy addition of new payment methods without impacting the existing logic.

Consequences if not used: Without the Strategy pattern, adding new payment methods would lead to code redundancy and complexity, significantly complicating maintenance and reducing the system's adaptability to emerging payment technologies

Class Diagram:



2.2.2 Signup/Login Page

Problem: We have a system with four main user types:

1. **Organizers** (event organizers and planners, sponsors and exhibitors)
2. **Attendees** (speakers and learners)
3. **Administrative Users** (technical maintenance personnel, executive personnel)
4. **Stakeholders** (educational institutions, event management companies, and technology providers)

Each user type has distinct behaviors and permissions:

- **Organizers** might create events, manage schedules, and handle sponsors.
- **Attendees** may only view or register for events, among other actions.
- **Administrative Users** manage the system (add, edit, remove users; oversee finances).
- **Stakeholders** (institutions, companies, tech providers) could have read/write permissions to certain event data, and sponsor events.

The login and sign-up system must accommodate several user roles (Organizers, Attendees, Administrative Users, and Stakeholders) that each have distinct behaviors and requirements. For instance, while an Administrative user might immediately gain access to event creation features, an Organizer may require additional checks (e.g., two-factor authentication or manual approval) before logging in.

Without a structured approach, the implementation may rely on numerous conditional statements (if/else or switch blocks) scattered across the code. This leads to reduced flexibility, making it hard to introduce new roles or modify existing behaviors, low maintainability, making it so that changes for one role might inadvertently affect others, and scalability issues, making the login logic more complex and error-prone as the number of roles increases.

Solution: To address these issues, the Strategy Pattern provides an elegant solution by encapsulating role-specific login behaviors into separate classes that implement a common interface. With this pattern, the login context delegates authentication to the appropriate strategy based on the user type. For example, an Organizer's login process, which might immediately provide access to event creation features, is isolated from an Administrative User's login, which could require additional security checks such as two-factor authentication. This separation of concerns not only improves maintainability by isolating role-specific logic but also enhances flexibility since adding a new role becomes as simple as implementing a new strategy class without the need to alter the core login process.

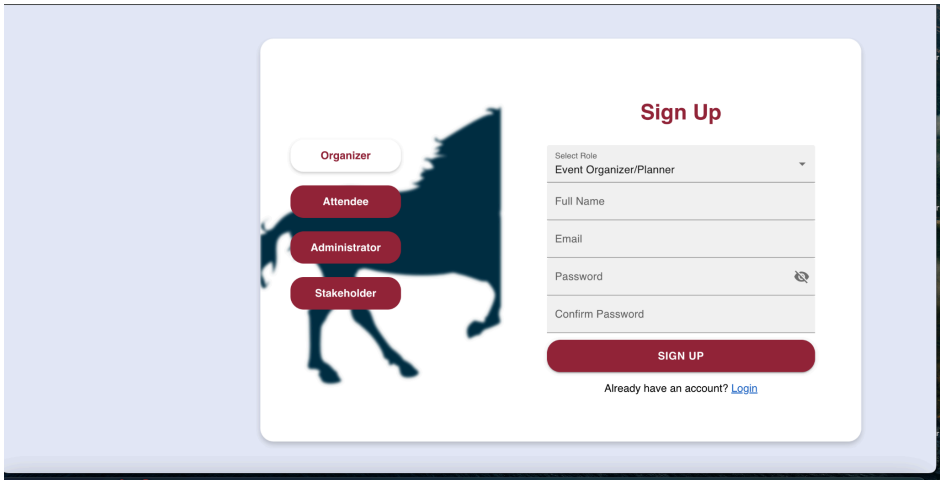
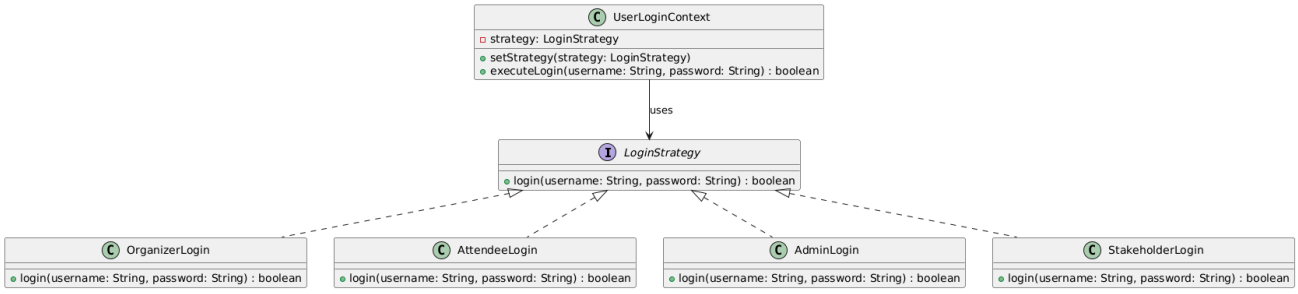


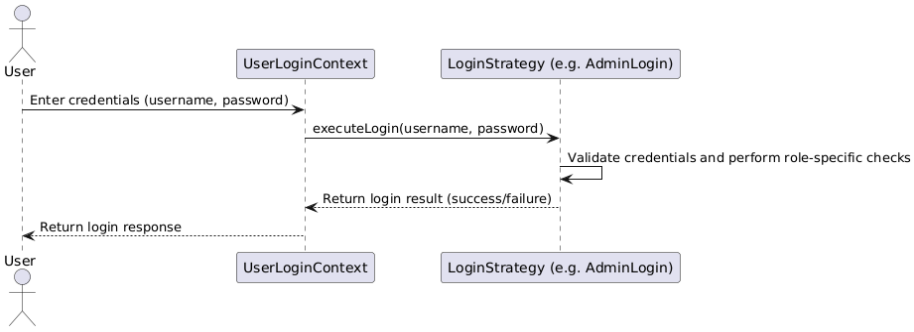
Figure 1.1: Sign up page.

Consequences if not used: If the Strategy Pattern was not used, the system would suffer from a monolithic login function littered with if/else statements, resulting in tightly coupled logic that is error-prone and difficult to modify. The absence of this pattern would make the code less readable, complicate maintenance efforts, and significantly increase the risk of bugs when updating or extending the system. In contrast, the Strategy Pattern's clear separation of role-specific logic ensures a modular design, enabling easier debugging, testing, and scalability while maintaining a clean and cohesive codebase.

Class Diagram:



Sequence Diagram:



2.3 Managing software content- Decorator Pattern

Problem: In our React application, multiple components need access to shared functionalities like user authentication state or routing context. If we were to embed the needed logic directly into each component, it would result in a lot of coupling and duplicated logic. Thus, we needed a way to dynamically extend a component's capabilities without altering its core logic.

Solution: In our project, we used the Decorator Pattern to add extra features to components without changing how they work inside.For example, we have a component called MainLayout. By itself, it just shows the layout of the page and doesn't know anything about the user. But when we wrap it with something called UserProvider, the component now has access to information about the logged-in user — like their name or ID. This lets us show messages like "Welcome back, Abdullah" without adding that logic directly into MainLayout.So instead of putting the same code in every component, we added it in one place (UserProvider) and used it to decorate the components that need it.

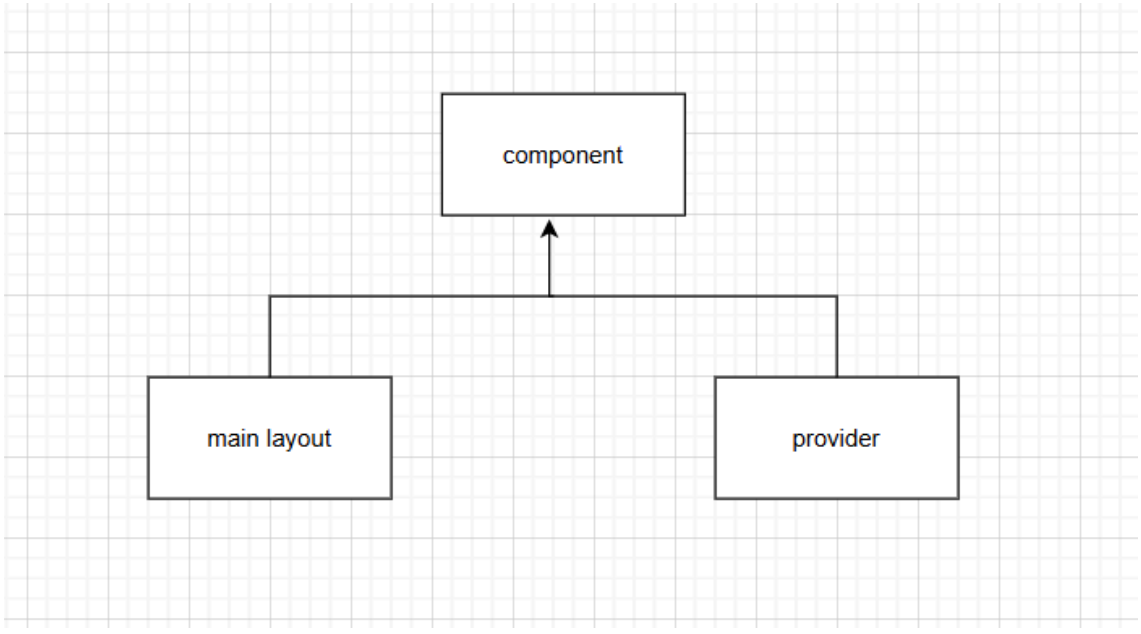
Consequence if not used:

Without the Decorator Pattern, every component that needs user information would have to:

- Manually fetch user data
- Re-implement state logic
- Lose reusability and increase code duplication

This will lead to inconsistent logic and difficulty in maintaining or scaling the app.

ClassDiagram:



2.4 Database Design - Singleton Pattern

Problem: Managing database connections in a Node.js server can lead to inefficiencies and system instability if new connections are created repeatedly across different files. Originally, each controller or module could create its own instance of the database connection pool or Neo4j driver which results in many redundant connections, increased memory usage and an increased possibility of limits being reached eventually.

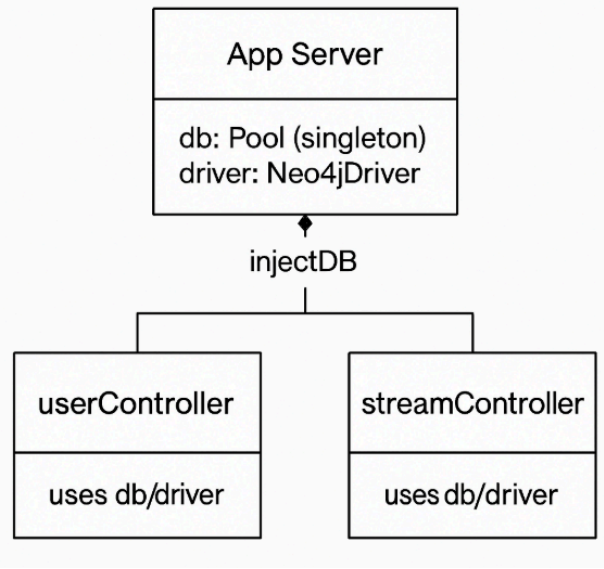
Solution: We applied the Singleton Pattern by basically creating a single shared instance of both the PostgreSQL connection pool (db) and the Neo4j driver (driver)in our main backend file (server.js). These instances are injected into other modules such as userController and streamController using dependency injection (injectDB(db)). This ensures that all the parts of our system are using the same exact database instance.

Consequence if not used: Without the Singleton pattern, every request handler or module would create its own database connection. This could have lead to:

- Connection flooding
- Resource exhaustion
- Inconsistent data states
- Increased difficulty in debugging and managing DB-related logic

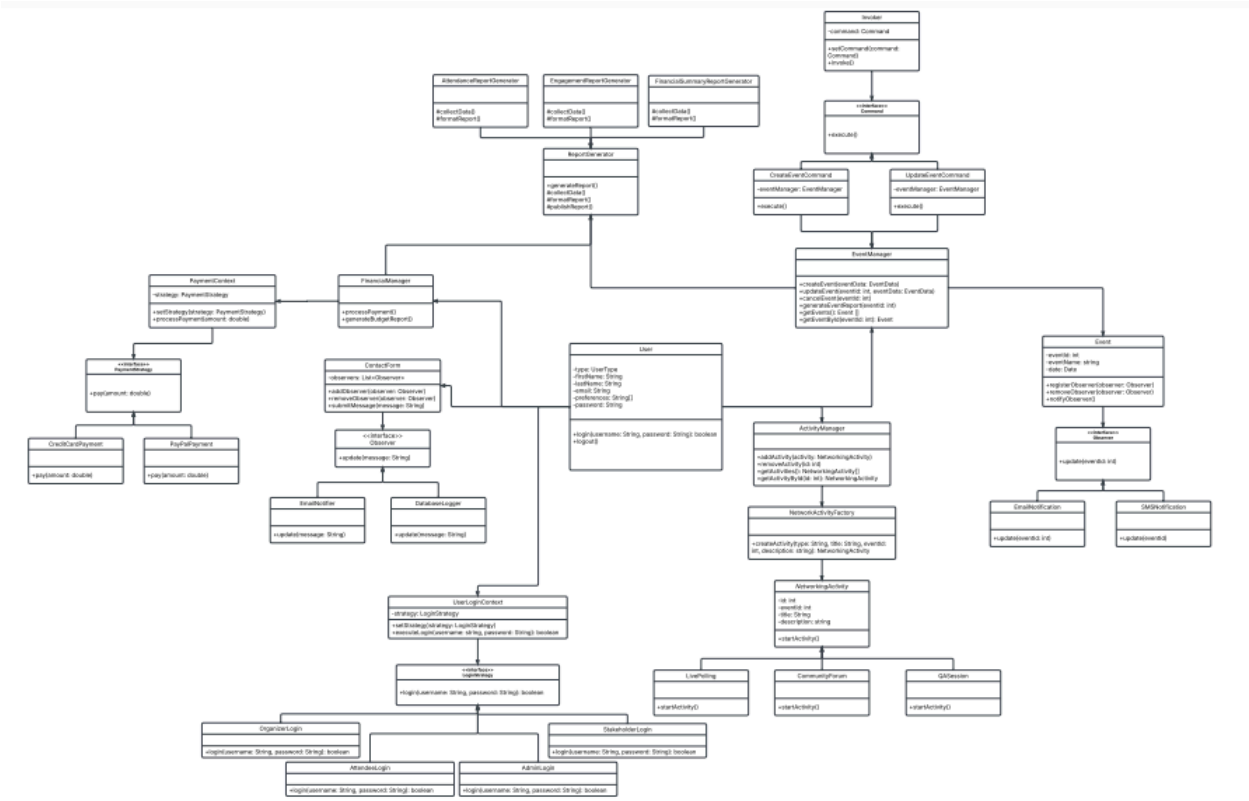
It would also make the application less scalable and more error-prone under heavier and heavier load.

Class Diagram:



3.0 Class diagram

A pdf will be given to showcase a clearer version of this picture.



4.0 UI bank

Here are images of our UI for the feature of our implementation that we did not already showcase.

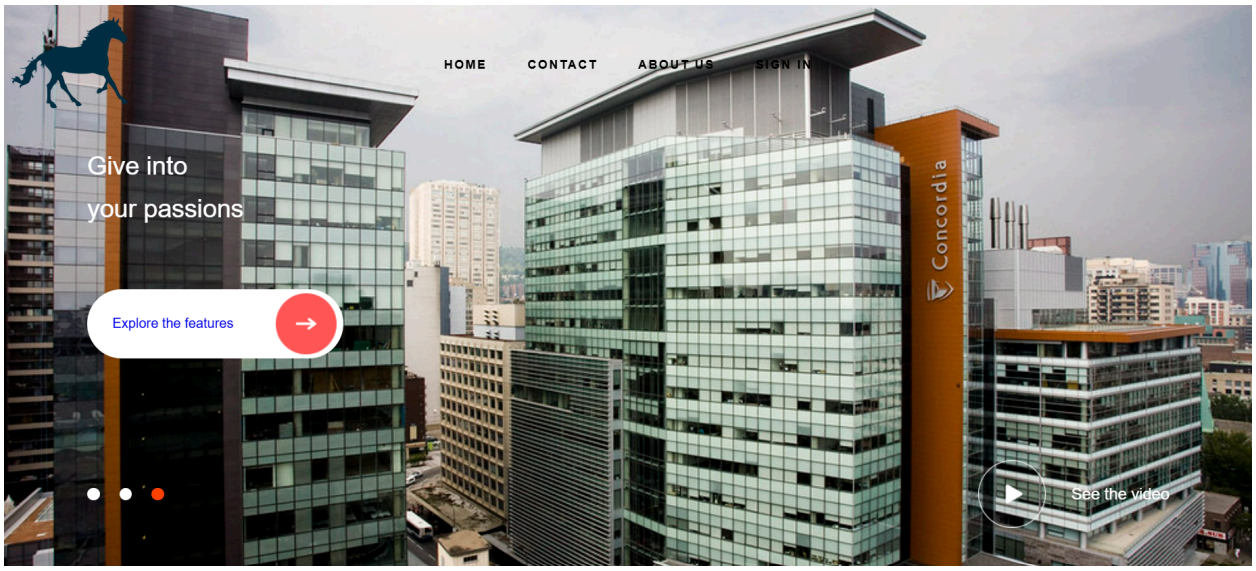


Fig 1: main landing page for our website

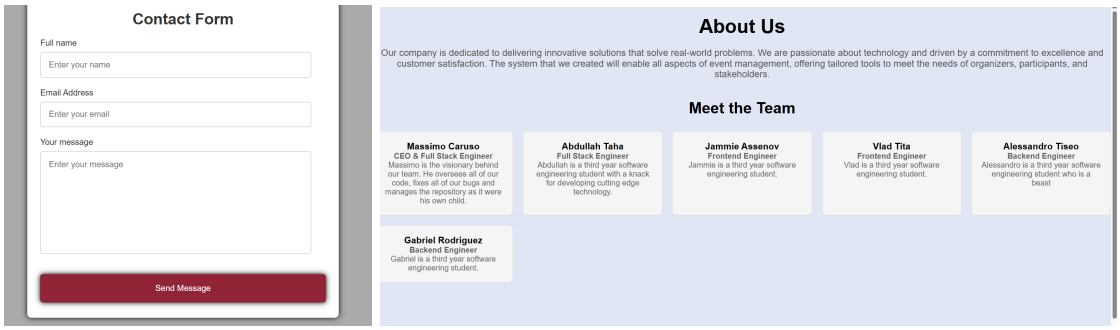


Fig 2: sections in our main landing page

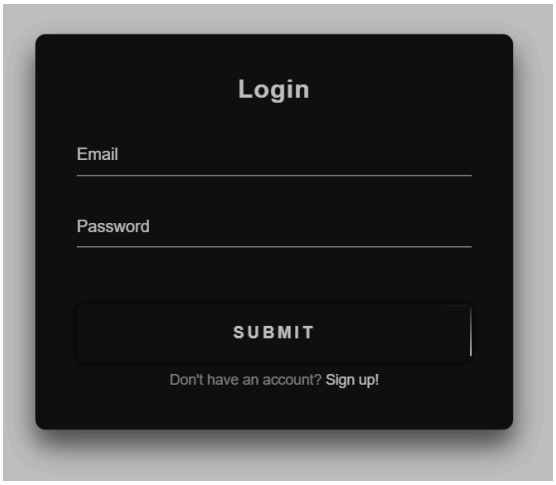


Fig 3: Login form

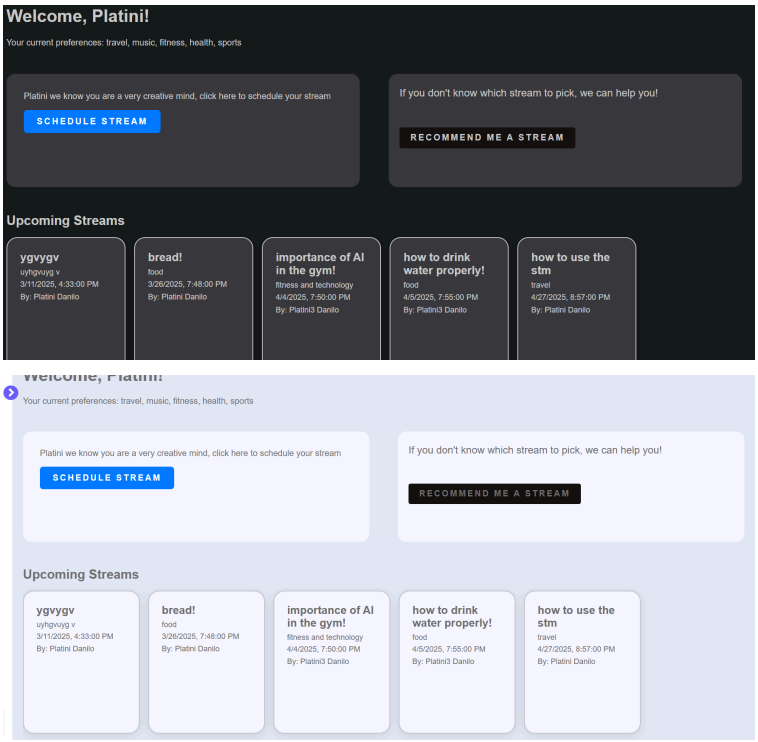


Fig 4: Main homepage light and dark mode view

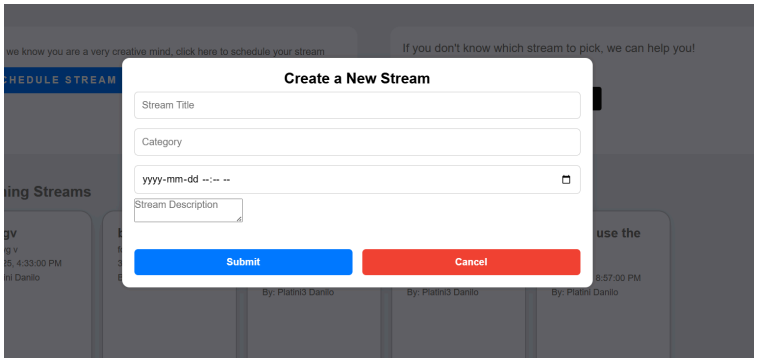


Figure 5: Create a new stream creation

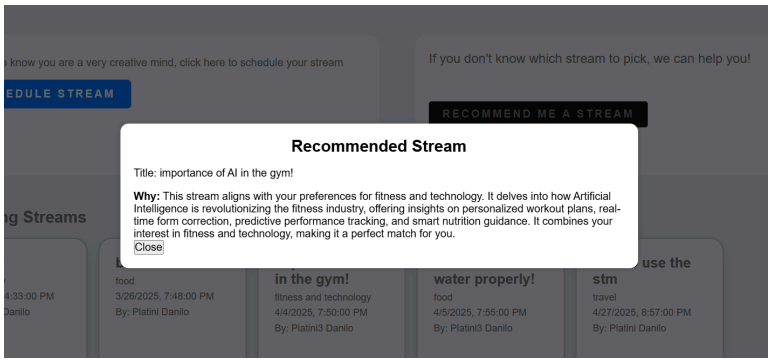


Fig 6: AI stream recommendation based on user preference

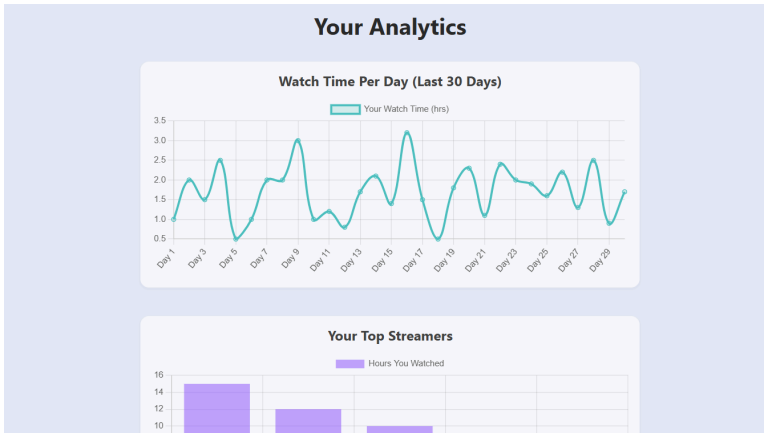


Fig 7: User analytics

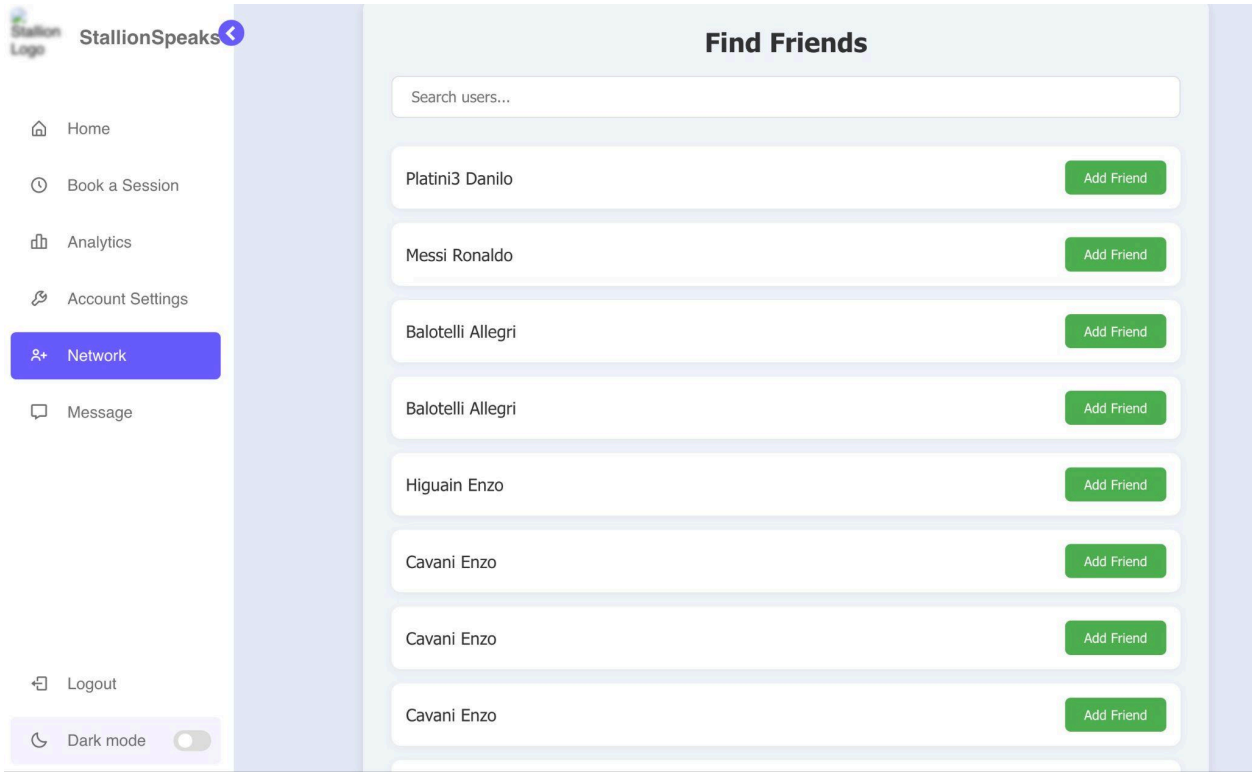


Figure 8: Find users to add as a friend.

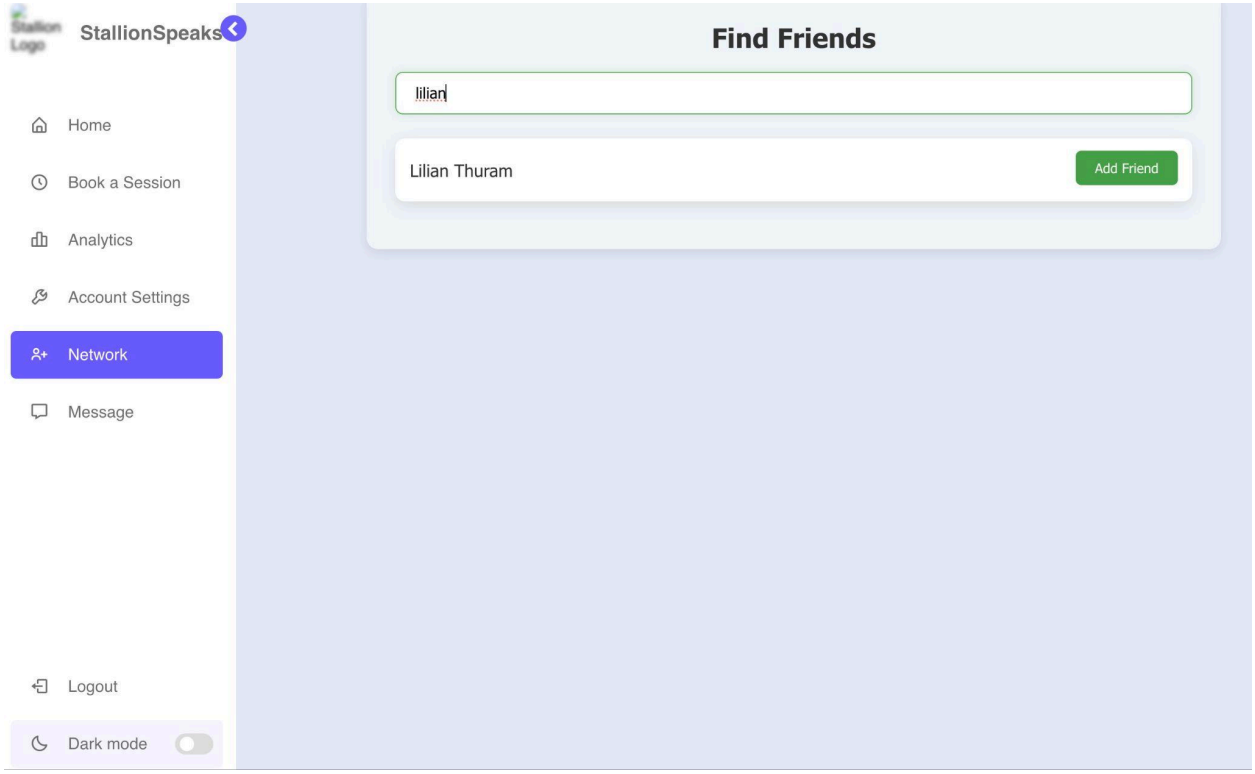


Figure 9: Search for a user to add as a friend.

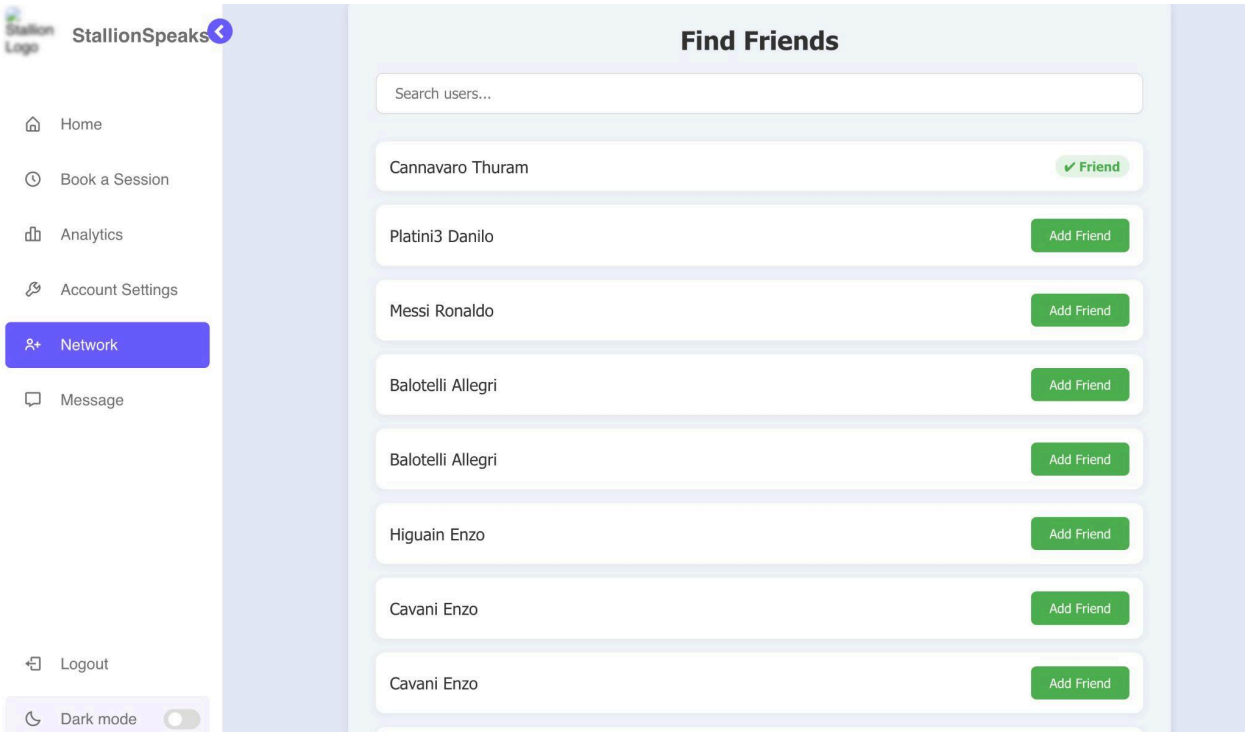


Figure 10: Friend request is accepted.

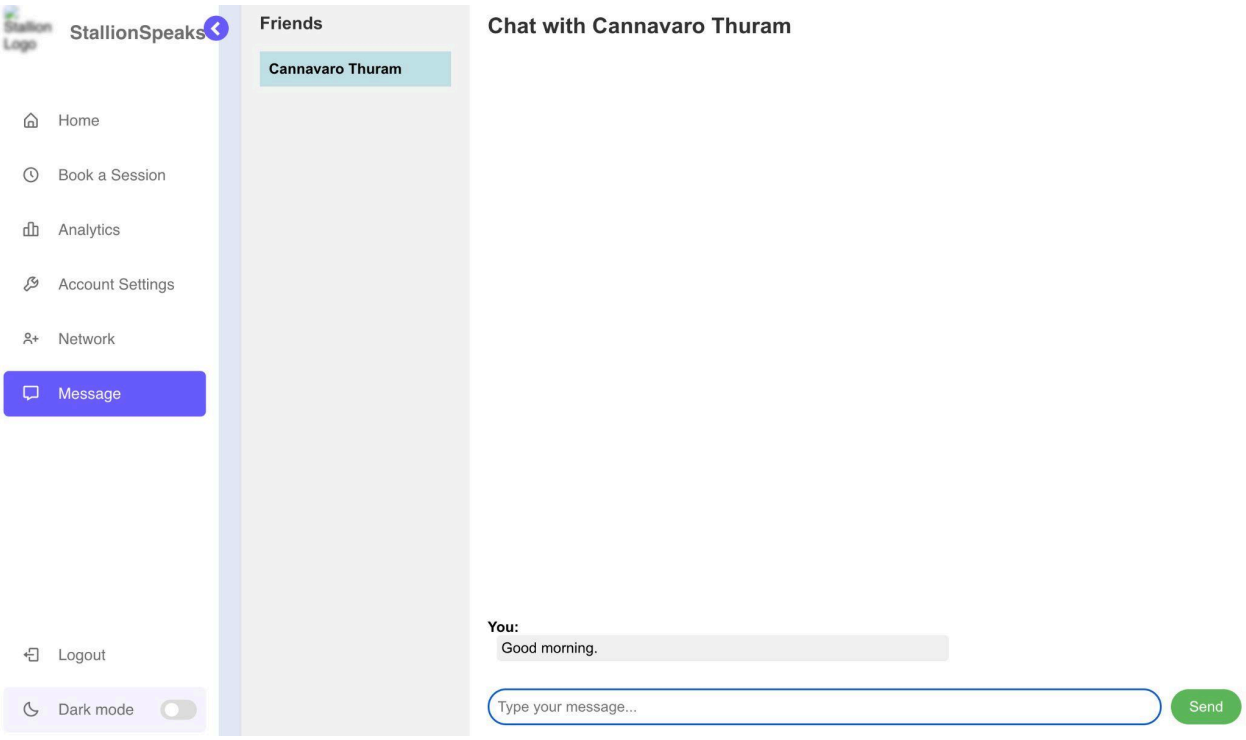


Figure 11: Chat with the user.

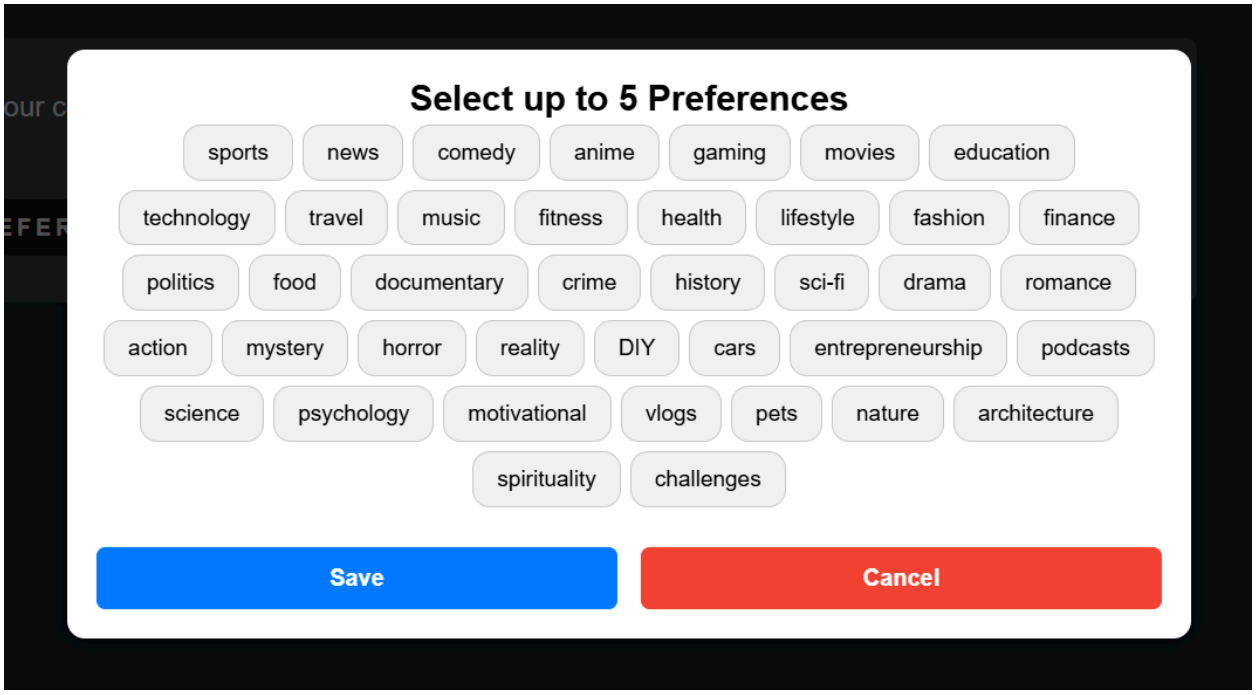


Fig.12: choose preferences.

5.0 Implementation

Networking and engagement:

The implementation of the network and engagement consist of two core features which are the friend request and chat system. In addition, the friend request system shown in *Figure 8* displays to the user all the accounts in the system. Consider *Figure 9*, where the user can search for a specific account to add. Once the other user accepts the friend request a check mark appears to show that the user accepted the friend request shown in *Figure 10.* Reflect on *Figure 11*, where once the user accepts the friend request, the user is allowed to directly message the other user.

The factor that played a massive part in the design of the chat system is that it is locked behind the friend request system to ensure privacy among the users. In contrast, social media apps have this feature where a user must accept the friend request before messaging one another. In the current implementation, a user can have the ability to send requests to all users. However, a user must accept the request to allow the messaging feature.

In the code base, there's a small deviation from the OOP principles since Node.js uses asynchronous messaging (non - blocking) for the url route. In other words, the system waits until a user does something and once the user does something the route is activated and performs an action which deviates from the original OOP principle of Encapsulation. Encapsulation requires the asynchronous messaging to be a part of an object.

AI recommendation:

To implement the AI-based stream recommendation feature, we had to first create a way to collect meaningful input from our users. We created a user interface that allows each user to select their personal preferences (fig.12), like topics, categories, or styles of content they enjoy. At the same time, we developed an interface for stream creation (fig.5) where users can schedule a stream and categorize it with relevant information like the title, category, description, and timing.These two sets of data are then fed into our AI model which then determines the best stream for the user and also generates a small paragraph on why would that be the case. An example of this can be found in fig.6.

Report / analytics

We implemented a basic report generation feature to display analytics such as attendance, watchtime history, top streamers, favorite categories and more using Chart.js on the frontend. The charts help visualize how users interact with streams in a simple and clear way. Currently since we do not have the features needed to record the data, we are using placeholder data. However, the charts are fully functional and ready to display real data once it becomes available. This allows us to test and demonstrate the feature while keeping it easy to update later with actual values from our users. Examples of the charts that we have can be found in fig.7

Event scheduling and management:

The implementation of the event scheduling and management consist of creating a stream shown in *Figure 5*. When a user creates a stream it is available to all users in the system. More importantly, a user can select a list of preferences shown in *Figure 4.* Once the user selects the preferences of the account, the user can click on " Recommend Me A Stream" where AI will be used to select a stream shown in *Figure 6*.

At the beginning of the design instead of creating a list of preferences the AI would recommend a stream based on description of the user account. However, preferences seem to be more ideal since a description is different from user preferences. Conversely, the code for this feature deviates from OOP principles since the system calls to API do not follow a class structure.

Multi-role authentication:

The implementation of the multi-role authentication consists of a selection of Organizer, Attendee, Administrator and Stakeholder. Consider Figure 1.1, where it shows how the user can select different types of users at the sign up page. In addition , this increases transparency since a user can see the available types of user. A previous design choice was to allow the user to select the type of user in the user settings. In other words, the previous design lacks a form of clarity.

Contact Form Submission:

We implemented a basic contact form submission feature that allows users to send messages, which are then forwarded to an email address using an automated notification system. When a user submits a message through the form, the system processes the input and notifies all registered handlers, such as an email service and a message logger. This ensures that messages are both delivered via email and recorded for future reference.

Currently, since we do not have integrations with additional messaging platforms, the system is set up to send emails using a predefined email service. However, the architecture follows the Observer Pattern, allowing us to easily extend functionality in the future. For example, we can add SMS notifications, store messages in a database, or integrate with a CRM system without modifying the core form logic. This approach ensures flexibility while keeping the implementation simple and maintainable. This can be seen in section 2.1.2.