Vietnam National University, HCMC

International University

School of Computer Science & Engineering

Report Project Final

Report: NexFlixx

Course: Web Application Development

Student:

Nguyen Quang Truc - ITCSIU23041

Le Canh Toan - ITCSIU23060

December 21, 2025

Contents

1 Introduction **3**
1.1 Project Background . . . 3
1.2 Programming Language . . . 3
1.3 Framework, Database, and Tools . . . 3

2 Requirement Analysis **4**
2.1 Use Case Diagram . . . 4
2.2 Functional Requirement & Non-functional Requirement . . . 6
2.2.1 Functional Requirements . . . 6
2.2.2 Non-Functional Requirements . . . 7

3 Design and Implementation **9**
3.1 Design . . . 9
3.2 Entity-Relationship Diagram (ERD) . . . 9
3.2.1 Sequence Diagram . . . 9
3.3 Implementation . . . 16
3.3.1 User's Account Management Functions . . . 16

4 Challenges **19**
4.1 Database Synchronization . . . 19
4.2 Insufficient Knowledge in Tools/Technologies . . . 20
4.3 GitHub Version Control Conflicts . . . 20
4.4 Communication and Coordination . . . 20
4.5 Balancing Learning and Building . . . 20
4.6 Overlooking Best Practices . . . 20

5 Conclusion **20**

Reference **22**

Listings

List of Figures

1 Usecase diagram of the website system 5
2 Entity-Relationship Diagram 9
3 Sequence diagram for register account 10
4 Sequence diagram for search video 11
5 Sequence diagram for find similar videos 12
6 Sequence diagram for create information 13
7 Sequence diagram for login account 14
8 Sequence diagram for change password 15
9 Register page 16
10 Login page 17
11 Login to the website successfully 17
12 Search video 17
13 Watch Video 18
14 Rating a video 18
15 Comment in a video 18
16 Reply a video 19
17 Profile page 19

1 Introduction

1.1 Project Background

Our web platform is a comprehensive video sharing and content management system designed to provide users with a rich, interactive experience for sharing and consuming video content. The platform combines modern web technologies with robust backend services to deliver a seamless user experience.

1.2 Programming Language

The platform is developed primarily using Java [2] as the core programming language. Java was chosen for its robustness, platform independence, and strong support for building scalable and maintainable web applications.

1.3 Framework, Database, and Tools

Since this is a web-based product, the project is conducted using modern web technologies following the MVC (Model-View-Controller) architecture pattern. In this project, we use:

- React.js [3] for the frontend, providing a component-based architecture for building interactive user interfaces
- Tailwind CSS for styling and responsive design, ensuring a modern and consistent visual appearance
- JavaScript/TypeScript for client-side logic and state management
- Spring Boot [4] as the backend framework, providing robust server-side functionality
- IntelliJ IDEA as the primary IDE for Java development
- MySQL [1] as the relational database for persistent data storage
- JPA (Java Persistence API) and Hibernate for object-relational mapping and database operations

- JWT (JSON Web Tokens) for secure authentication and authorization
- Axios for handling HTTP requests between frontend and backend
- Vercel for frontend deployment and hosting

The project follows modern development practices:

- RESTful API architecture for backend services
- Component-based frontend architecture using React
- Secure authentication using JWT tokens
- CORS (Cross-Origin Resource Sharing) configuration for secure cross-domain requests
- Environment-based configuration for different deployment stages

2 Requirement Analysis

2.1 Use Case Diagram

The use case diagram below represents the interactions between different actors and the website system functionalities. The primary actors involved are User, Guest, and Admin, each with specific roles and access to various features of the system.

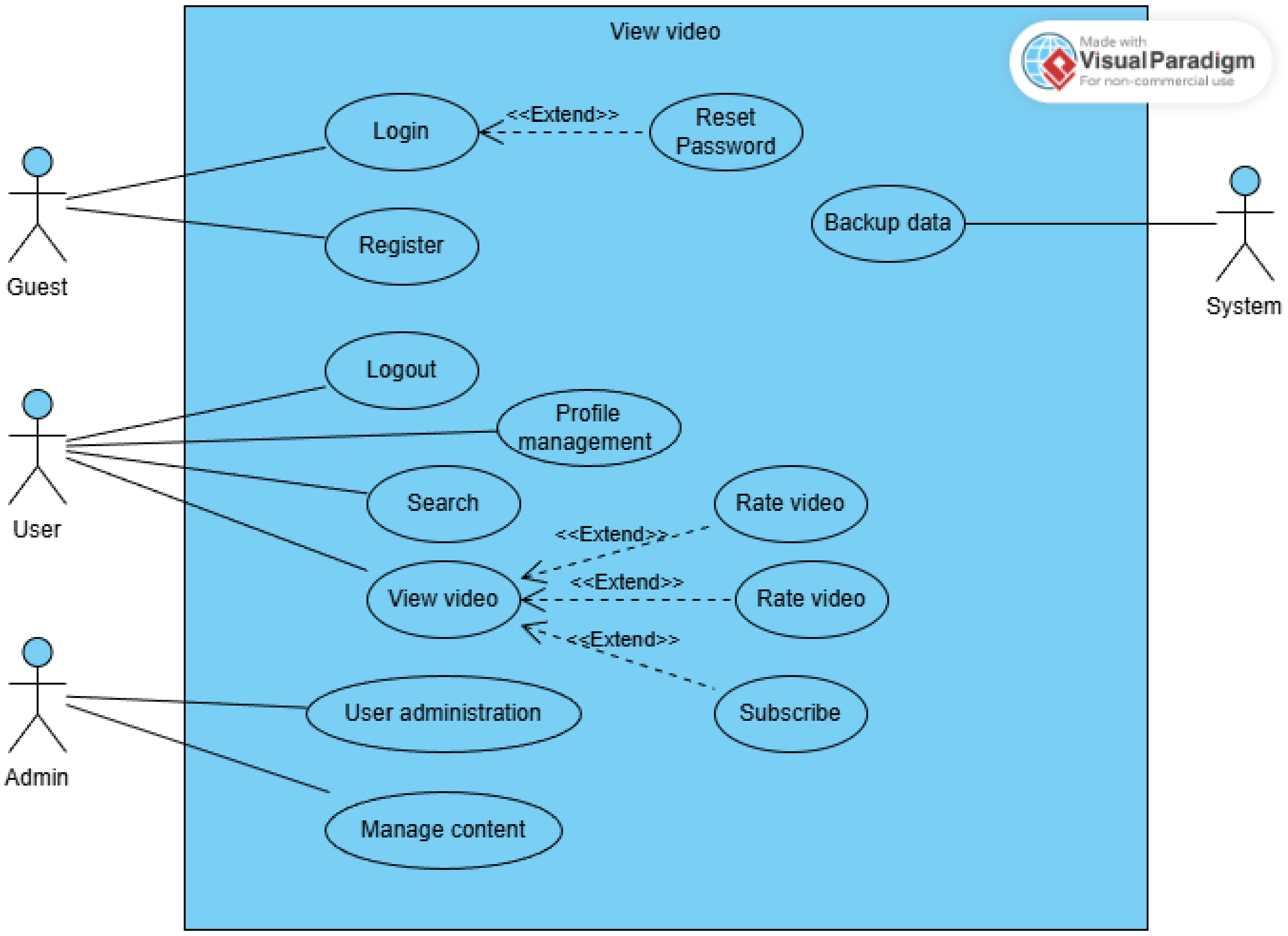

Figure 1: Usecase diagram of the website system

From Figure 1 we can see an overview about 4 actors and their roles in the project:

- Guest:
  - Can log in, register for an account, and reset their password if needed.
  - Has access to view videos.
- User:
  - Can log out and manage their profile.
  - Can search for videos and view them.
  - Has the ability to rate videos, comment.
- Admin:

  - Has administrative privileges to manage users and content within the system.

- System:
  - Responsible for backing up data.

2.2 Functional Requirement & Non-functional Requirement

2.2.1 Functional Requirements

1. Authentication and User Management:
   - The system shall allow users to register an account using email and password.
   - The system shall provide secure login functionality with JWT token-based authentication.
   - The system shall maintain user sessions and handle token mechanisms.
2. Video Management:
   - The system allows users to upload and manage video content.
   - The system provides video playback functionality with proper streaming capabilities.
   - The system supports video metadata management (title, description, etc.).
3. User Profile Management:
   - The system allows users to view and update their profile information.
   - The system maintains user preferences and settings.
   - The system tracks user activity and history.
4. Security Features:
   - The system implements secure password hashing using BCrypt.
   - The system provides role-based access control for different user types.
   - The system handles CORS configuration for secure cross-origin requests.
5. API Integration:

- The system provides RESTful API endpoints for frontend-backend communication.
- The system implements proper error handling and response formatting.
- The system supports secure file uploads and media handling.

2.2.2 Non-Functional Requirements

1. Performance:
   - The system ensures fast API response times for user interactions.
   - The system optimizes video streaming performance.
   - The system implements efficient token validation and session management.
2. Usability:
   - The system provides a modern, responsive user interface using React.
   - The system offers clear error messages and loading states.
   - The system implements intuitive navigation and user flows.
3. Security:
   - The system implements JWT-based authentication with proper token management.
   - The system uses secure password hashing with BCrypt.
   - The system implements proper CORS policies for secure cross-origin requests.
   - The system protects sensitive routes and API endpoints.
4. Scalability:
   - The system is designed to handle multiple concurrent users.
   - The system supports future feature additions (e.g., social features, comments).
   - The system can be easily deployed to cloud platforms (Vercel).
5. Maintainability:
   - The system follows modern development practices with clear code organization.

- The system implements proper error handling and logging.
- The system uses environment-based configuration for different deployment stages.

6. Compatibility:

- The system supports modern web browsers.
- The system is responsive across different device sizes.
- The system implements proper CORS configuration for cross-origin requests.

3 Design and Implementation

3.1 Design

3.2 Entity-Relationship Diagram (ERD)

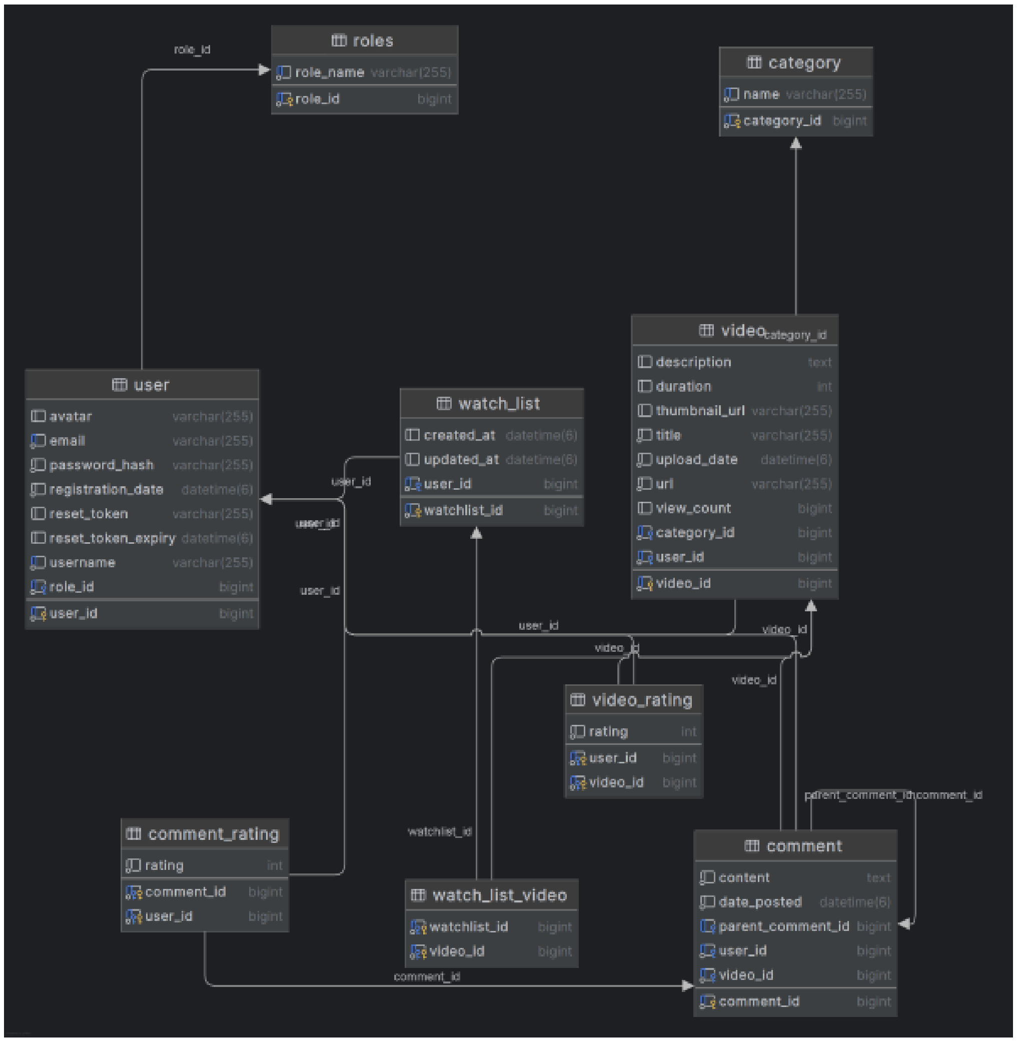

Figure 2: Entity-Relationship Diagram

3.2.1 Sequence Diagram

Register account

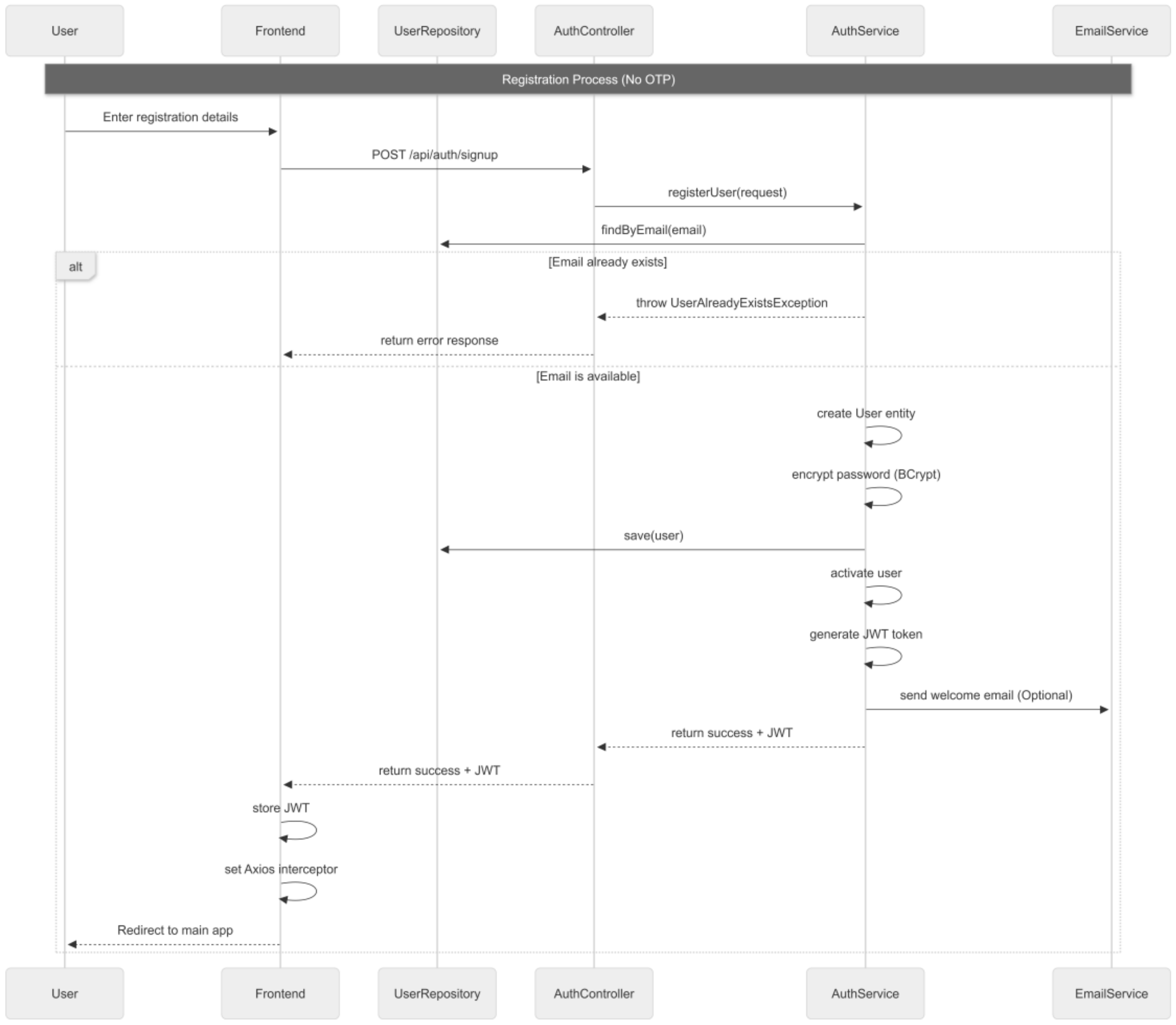

Figure 3: Sequence diagram for register account

This sequence diagram, shown in Figure 3, illustrates the streamlined user registration process. The workflow initiates when a user submits their credentials via the front-end interface, dispatching a POST request to the back-end API. The system first performs a validation check against the *UserRepository* to ensure the email address is unique; if a duplicate is detected, a *UserAlreadyExistsException* is thrown, and an error response is returned to the client.

Upon confirming the email's availability, the system instantiates a new user entity. Security is enforced by encrypting the password using the BCrypt hashing algorithm before persistence. Deviating from multi-step verification flows, the system immediately activates the user account and generates a signed JWT (JSON Web Token). This token is encapsulated in the success response and returned to the front-end, where it is stored for session management. The user is then automatically redirected to the main application, effectively establishing an authenticated session in a single transaction.

Search Video

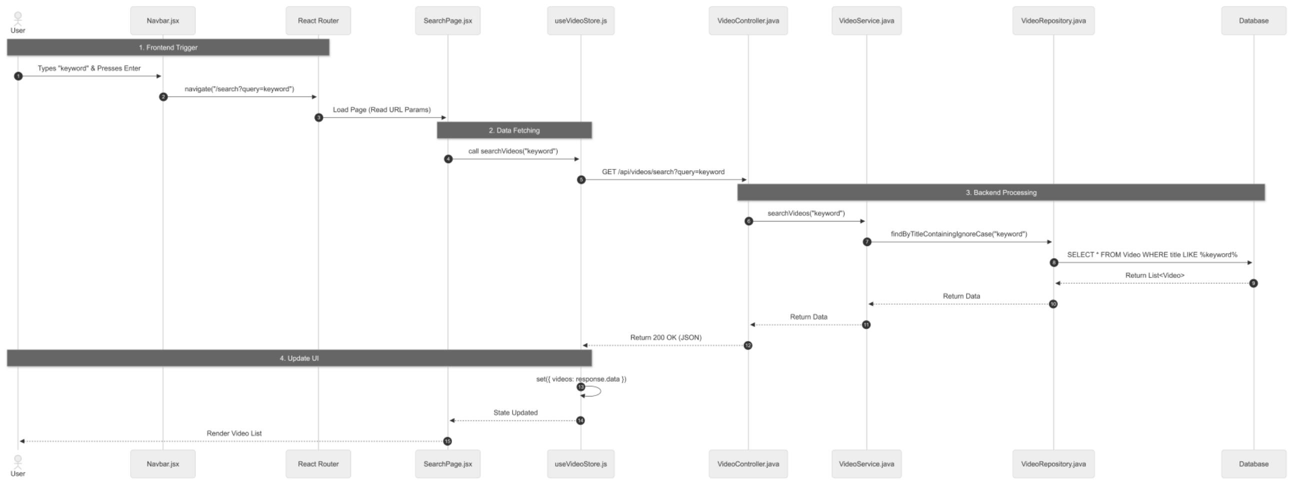
User
Navbar.jsx
React Router
SearchPage.jsx
useVideoStore.js
VideoController.java
VideoService.java
VideoRepository.java
Database
1. Frontend Trigger
1 Types "keyword" & Presses Enter
2 navigate("/search?query=keyword")
3 Load Page (Read URL Params)
2. Data Fetching
4 call searchVideos("keyword")
5 GET /api/videos/search?query=keyword
3. Backend Processing
6 searchVideos("keyword")
7 findByTitleContainingIgnoreCase("keyword")
8 SELECT * FROM Video WHERE title LIKE %keyword%
9 Return List<Video>
10 Return Data
11 Return Data
12 Return 200 OK (JSON)
4. Update UI
13 set({ videos: response.data })
14 State Updated
15 Render Video List

Figure 4: Sequence diagram for search video

This sequence diagram Figure 4 illustrates the flow of a video search process initiated by a user through a front-end interface. The interaction begins when the user enters a search query to search for videos by title. This action is captured by the frontend, which then sends an HTTP GET request to the backend endpoint $/api/videos/search?title = query$ to fetch relevant video content based on the input query.

The request is first received by the VideoController, which delegates the task to the VideoService by calling the method $searchVideosByTitleWithRatings(title)$. This service method is responsible for handling the business logic of retrieving and preparing the video data. It interacts with the $VideoRepository$, invoking the method $findByTitleContainingIgnoreCase(title)$ to perform a case-insensitive search for videos that match the title.

Once the repository fetches a list of video entities ($List < Video >$), the control returns to the $VideoService$, where each video object is processed individually in a loop. During this loop, each Video entity is converted into a data transfer object (DTO) using the method $convertToDTO(video)$.

These DTOs are collected into a list ($List < VideoDTO >$) which is then returned to the $VideoController$.

The controller sends the HTTP response back to the frontend with a status code of 200 (OK), including the list of video $DTOs$. The frontend processes the returned data and updates the user interface accordingly. If the list is empty, indicating that no matching videos were found, a message saying ”No videos found” is displayed. Otherwise, if results are found, they are presented in a video grid or list format for the user to view.

Find similar videos

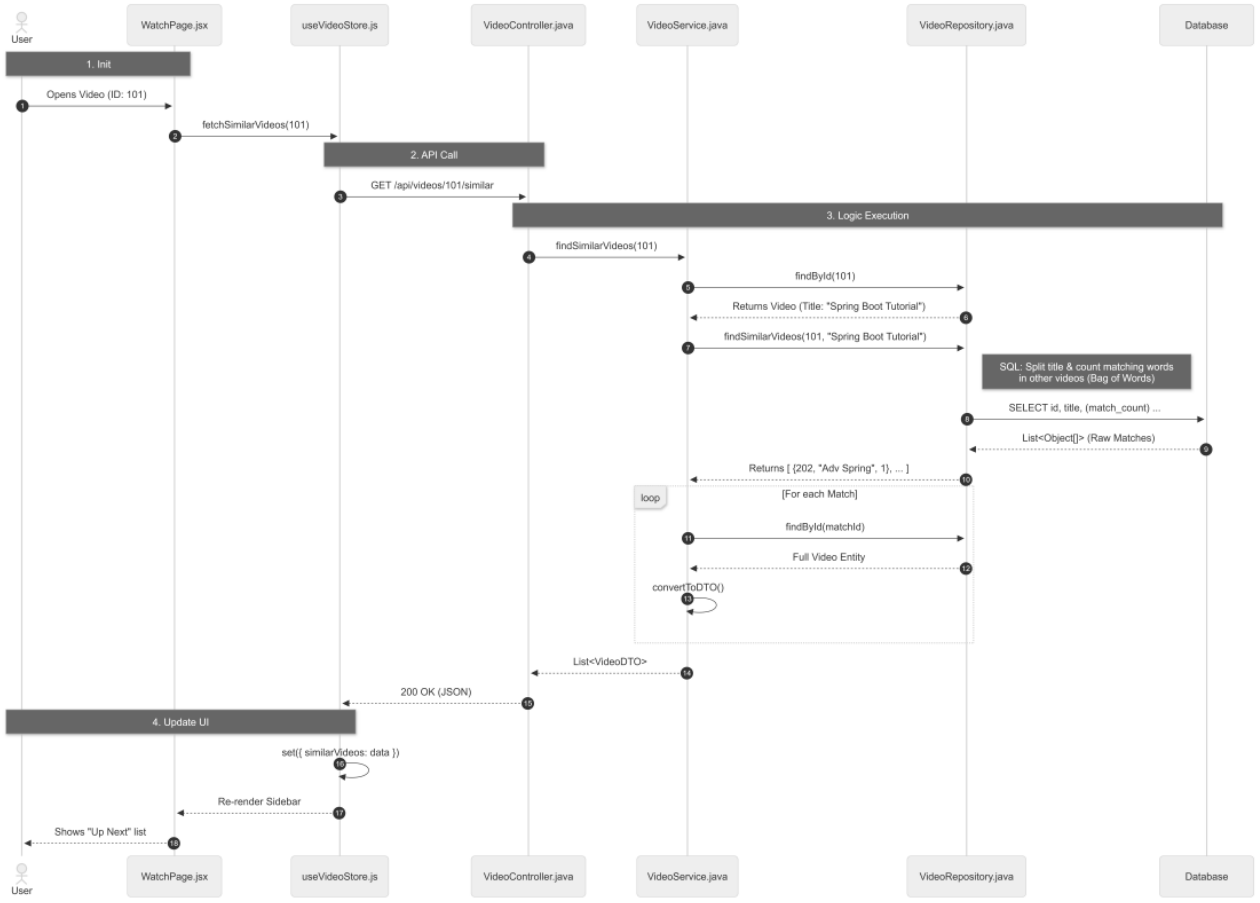

Figure 5: Sequence diagram for find similar videos

This sequence diagram Figure 4 shows how a user views a video and retrieves similar recommendations. When the user selects a video, the frontend sends a $GET/api/videos/id$ request. The backend retrieves the video by ID, increases its view count, and returns video details as a $VideoDTO$.

To find similar videos, the frontend makes two requests: one by category and one by tag. The backend fetches videos from the repository based on the category or tag, converts each video into a $VideoDTO$, and returns the lists. Finally, the frontend displays either a "No similar videos found" message or shows the similar videos section, depending on the results.

Count view video

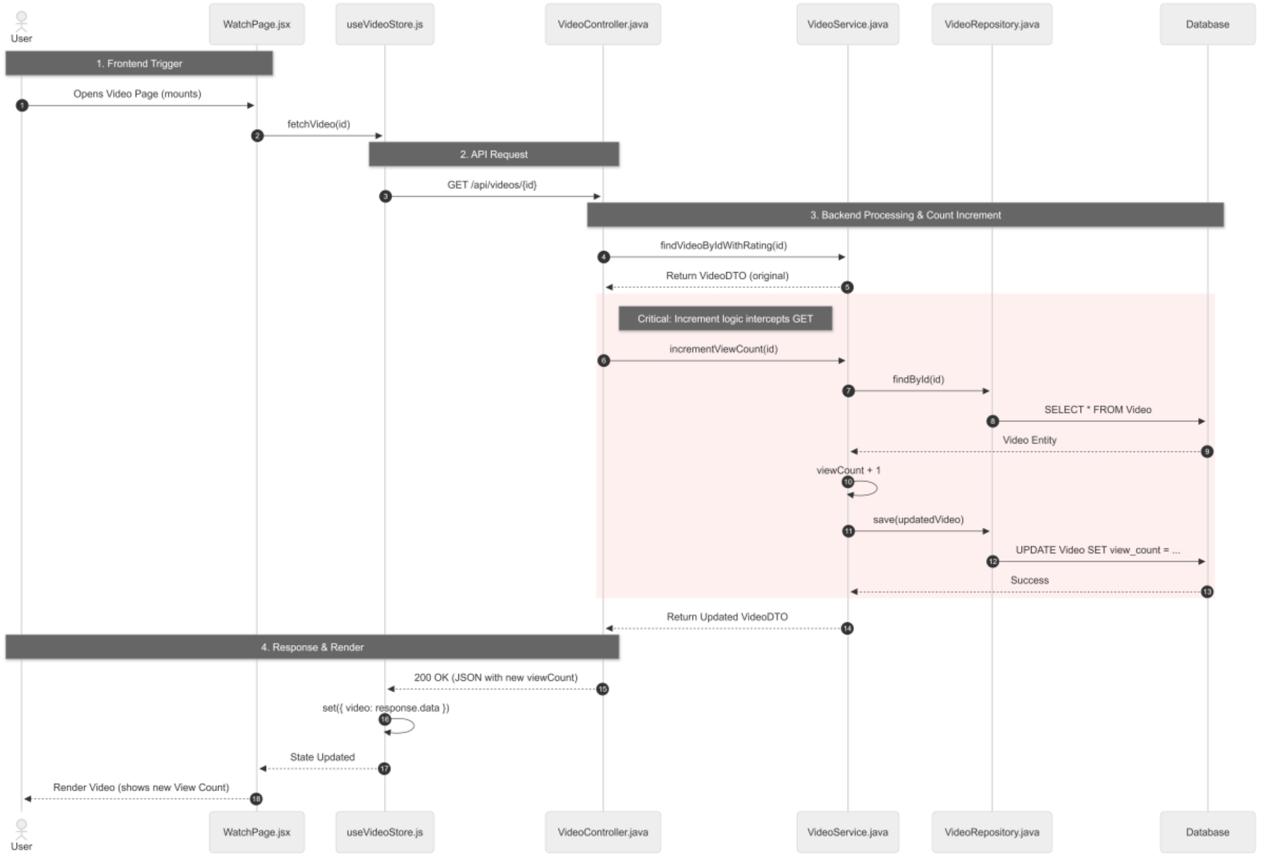

Figure 6: Sequence diagram for create information

This sequence diagram Figure 6 shows the process of updating a video's view count when a user watches it. The user triggers a GET request from the frontend to $/api/videos/id$. This request is handled by the $VideoController$, which calls the $VideoService$ to fetch the video and increment its view count.

The $VideoService$ retrieves the video entity from the $VideoRepository$ using $findById(id)$, then updates the view count and saves the updated entity. After that, the video is converted to a DTO and returned as an HTTP 200 response.

On the frontend, the view count is updated based on the response. If the update is successful, the new count is displayed; if it fails, the original count is shown instead.

Login Account

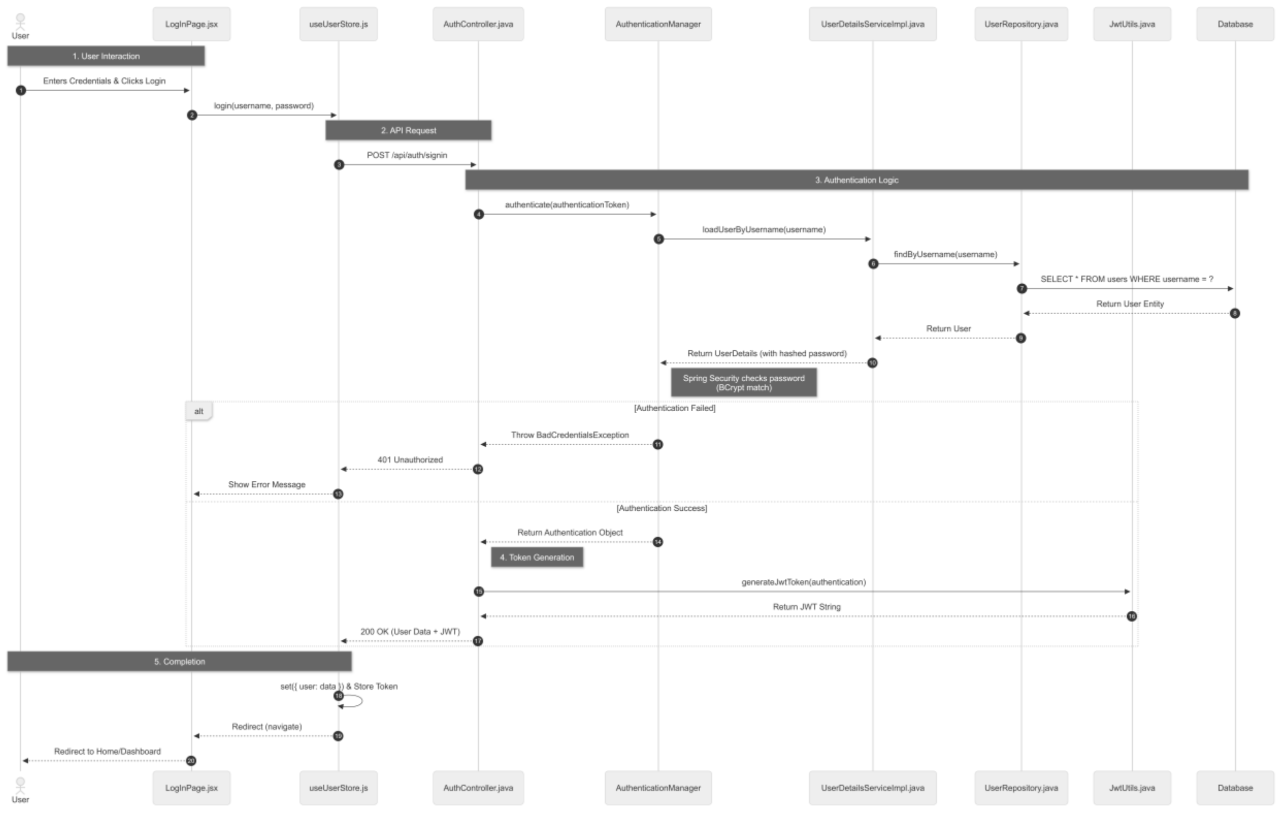
User
LogInPage.jsx
useUserStore.js
AuthController.java
AuthenticationManager
UserDetailsServiceImpl.java
UserRepository.java
JwtUtils.java
Database
1. User Interaction
Enters Credentials & Clicks Login
login(username, password)
2. API Request
POST /api/auth/signin
3. Authentication Logic
authenticate(authenticationToken)
loadUserByUsername(username)
findByUsername(username)
SELECT * FROM users WHERE username = ?
Return User Entity
Return User
Return UserDetails (with hashed password)
Spring Security checks password (BCrypt match)
alt
[Authentication Failed]
Throw BadCredentialsException
401 Unauthorized
Show Error Message
[Authentication Success]
Return Authentication Object
4. Token Generation
generateJwtToken(authentication)
Return JWT String
200 OK (User Data + JWT)
5. Completion
set({ user: data }) & Store Token
Redirect (navigate)
Redirect to Home/Dashboard

Figure 7: Sequence diagram for login account

This diagram Figure 7 describes the user login process using JWT authentication. The user enters their credentials, and the frontend sends a POST request to /api/auth/signin. The AuthController calls the AuthenticationManager to authenticate the username and password.

The AuthenticationManager retrieves the user details by calling UserDetailsService, which loads the user entity from the UserRepository. If the password is verified, an authentication object is returned.

A JWT token is then generated by JwtUtils, containing the username, issue time, expiration, and signature. The token is sent back and set in an HTTP-only cookie. The frontend receives a success response, stores the user info, sets up the Axios interceptor, and redirects to the main app.

If the credentials are invalid, a BadCredentialsException is thrown, resulting in an HTTP 400 error and an error message being displayed to the user.

Change Password

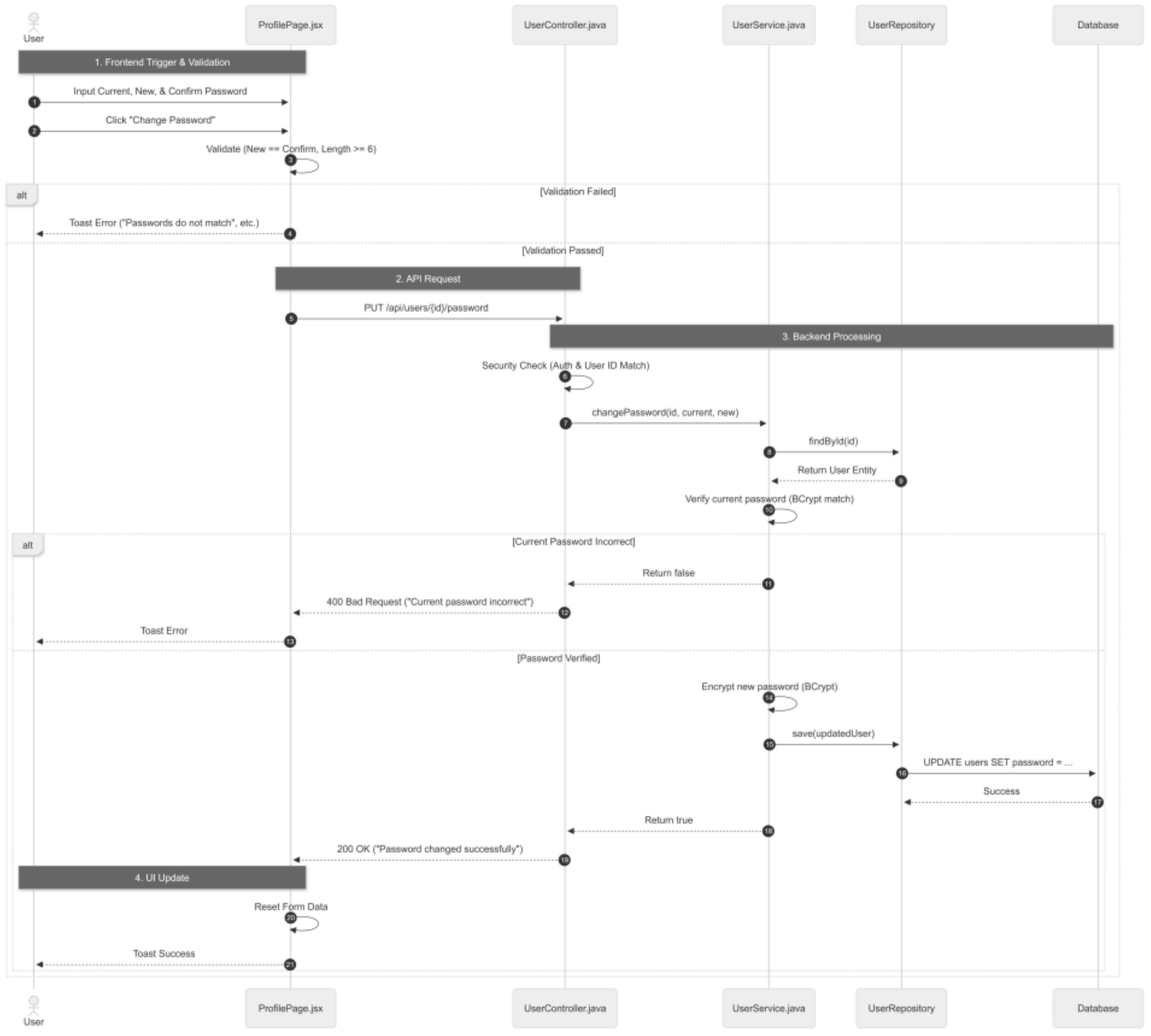
User
ProfilePage.jsx
UserController.java
UserService.java
UserRepository
Database
1. Frontend Trigger & Validation
Input Current, New, & Confirm Password
Click "Change Password"
Validate (New == Confirm, Length >= 6)
alt
[Validation Failed]
Toast Error ("Passwords do not match", etc.)
[Validation Passed]
2. API Request
PUT /api/users/{id}/password
3. Backend Processing
Security Check (Auth & User ID Match)
changePassword(id, current, new)
findById(id)
Return User Entity
Verify current password (BCrypt match)
alt
[Current Password Incorrect]
Return false
400 Bad Request ("Current password incorrect")
Toast Error
[Password Verified]
Encrypt new password (BCrypt)
save(updatedUser)
UPDATE users SET password = ...
Success
Return true
200 OK ("Password changed successfully")
4. UI Update
Reset Form Data
Toast Success

Figure 8: Sequence diagram for change password

This diagram Figure 8 illustrates the secure password update mechanism. The process commences when the user submits the password form on the profile page. The frontend first executes client-side validation to ensure the new password meets length requirements and matches the confirmation field. Upon validation, a `PUT` request is dispatched to `/api/users/{id}/password`.

On the server side, the `UserController` enforces strict authorization safeguards. It verifies that the authenticated principal's ID matches the path ID, preventing unauthorized modifications unless the requester holds administrative privileges. The request is then delegated to the `UserService`.

The service layer performs the critical cryptographic verification. It retrieves the user entity and utilizes the `PasswordEncoder` to validate the provided current password against the stored BCrypt hash. If the match is confirmed, the new password is hashed and persisted to the database.

Upon success, a 200 OK response is returned, triggering a success notification on the UI. Conversely, if the current password verification fails, the system returns a 400 Bad Request, and the user is informed of the credential mismatch.

3.3 Implementation

3.3.1 User's Account Management Functions

Register account

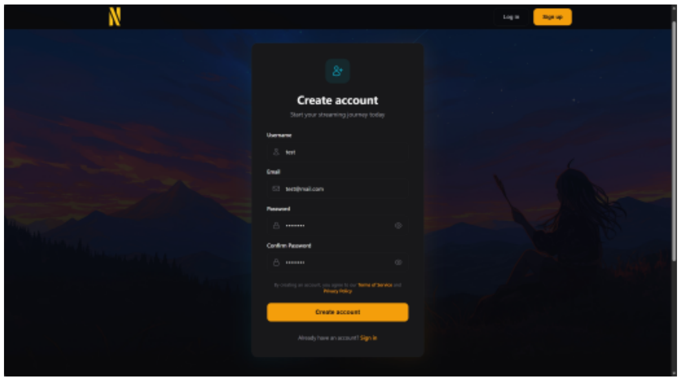
Log in
Sign up
Create account
Start your streaming journey today
Username
test
Email
test@mail.com
Password
Confirm Password
By creating an account, you agree to our Terms of Service and Privacy Policy
Create account
Already have an account? Sign in

Figure 9: Register page

Login Account

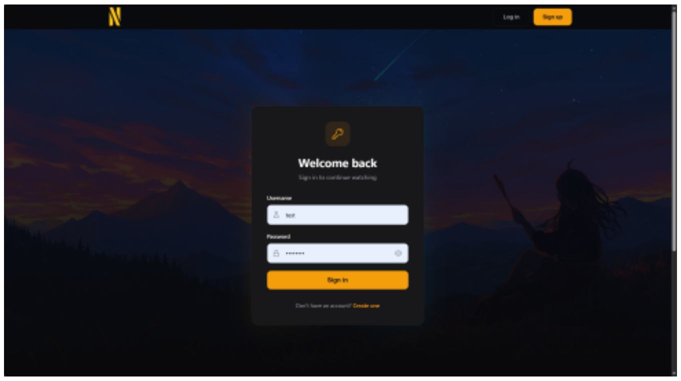
Log in
Sign up
Welcome back
Sign in to continue watching
Username
Password
Sign in
Don't have an account? Create one

Figure 10: Login page

Log in to the account on the website.

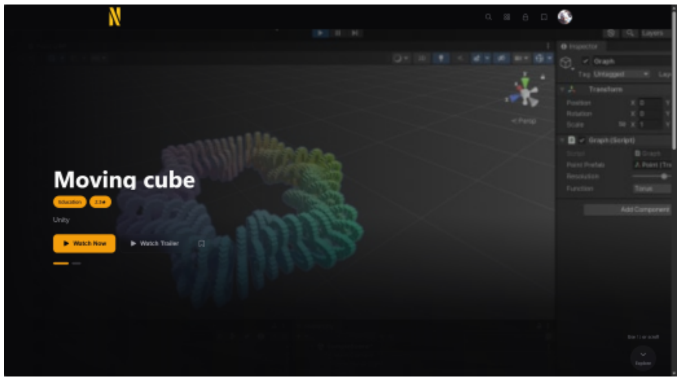

Figure 11: Login to the website successfully

Search Video

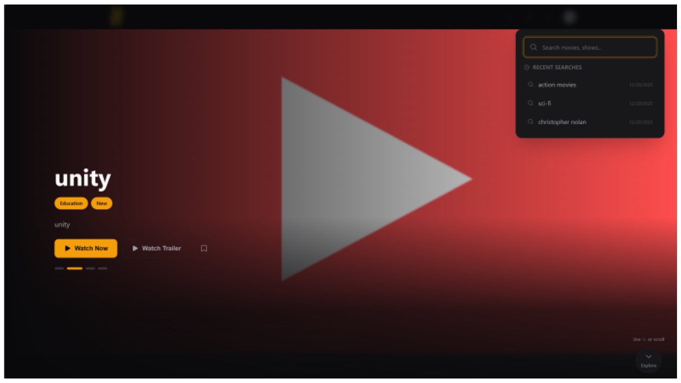

Figure 12: Search video

Watch Video

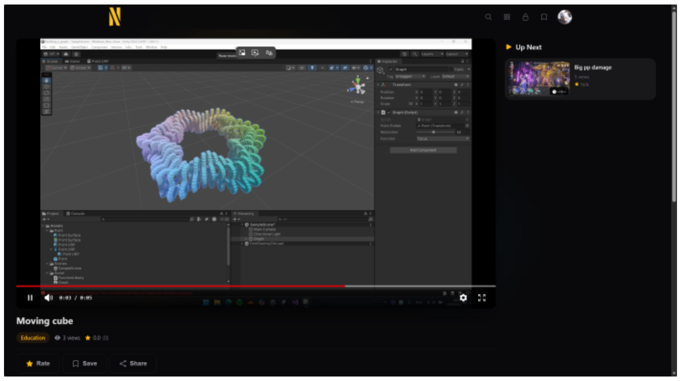

Figure 13: Watch Video

Rate Video

In above section, the video has zero rating, now we rating it as 5

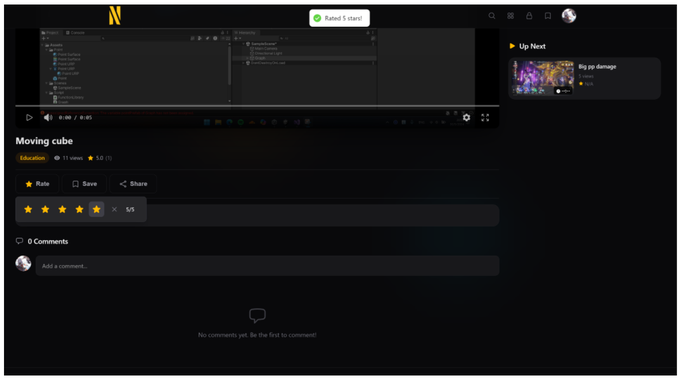

Figure 14: Rating a video

Comment in video

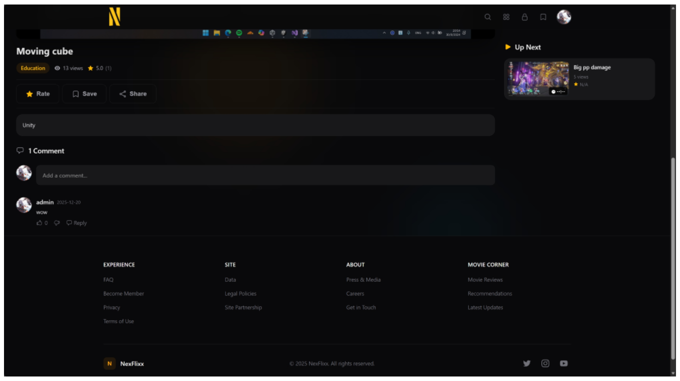

Figure 15: Comment in a video

We can also reply to a comment.

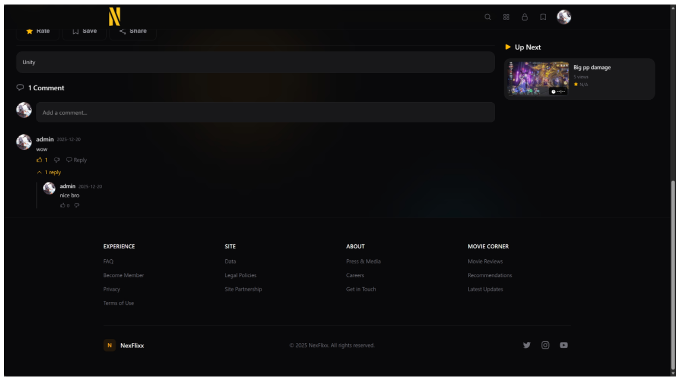

Figure 16: Reply a video

Profile page

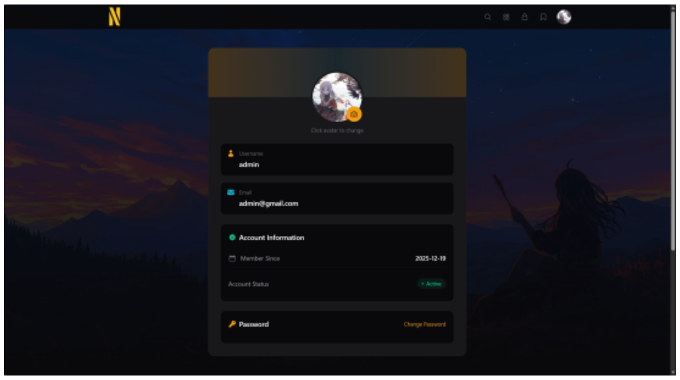

Figure 17: Profile page

4 Challenges

Below are some challenging problems commonly encountered by everyone working on the same project. We acknowledge these issues and aim to address them in more stable future projects.

4.1 Database Synchronization

- Challenge: Keeping the database schema consistent across team members.
- Symptoms: Updates to the database (adding tables, changing fields,...) might not be managed effectively, causing mismatches between the code and the database.

4.2 Insufficient Knowledge in Tools/Technologies

- Challenge: Team members might lack sufficient knowledge about essential tools like MySQL have difficulty in writing optimized queries, designing proper schemas, or debugging database issues because of limited prior exposure to database management or web development concepts.

4.3 GitHub Version Control Conflicts

- Challenge: When multiple people work on the same files, conflicts arise when merging code because of lacking of experience with Git workflows (e.g., branching, committing, resolving conflicts).
- Symptoms: Have to resolve conflict when occurring.

4.4 Communication and Coordination

- Challenge: Ensuring all team members stay updated on changes to the codebase and database.

4.5 Balancing Learning and Building

- Challenge: Needing to learn new skills (MySQL, SpringBoot, thymleaf) while simultaneously building the project due to lack of prior experience or formal guidance.
- Symptoms: Slow progress as time is spent researching and experimenting with technologies.

4.6 Overlooking Best Practices

- Challenge: Skipping best practices in favor of quick fixes, leading to technical debt.
- Symptoms: Messy codebase, poor database design, and fragile systems.

5 Conclusion

Our team has gained valuable experience in building a modern web application using Spring Boot for the backend and React with Vite for the frontend. We successfully implemented a secure JWT-based authentication system, integrated Cloudinary for media handling, and utilized Docker for containerization. The project featured a responsive frontend with a YouTube-style loading bar, efficient state management using Zustand, and a dynamic video upload system. We overcame challenges like CORS issues and deployment configurations, successfully deploying the application to production using Vercel. The experience enhanced our technical skills in handling real-world development challenges, from security implementations to collaborative development using Git and GitHub, providing us with practical knowledge in modern web application development.

References

[1] Oracle Corporation. Mysql 8.0 reference manual. *MySQL Documentation*, 2023.

[2] David J. Eck. *Introduction to Programming Using Java.* Hobart and William Smith Colleges, 8th edition, 2018.

[3] Facebook Inc. React: A javascript library for building user interfaces. *React Documentation*, 2013.

[4] Inc. Pivotal Software. Spring boot: Spring framework for java. *Spring Boot Documentation*, 2023.