Final Project Report: Severus Social Network Platform

Vu Binh Minh - V202100421
Nguyen Tung Lam - V202100571
Bui Huy Linh Phuc - V202100398

1 Introduction

The Severus Social Network Platform is a database-driven web application developed as a final project for the Database and Database Systems course at VinUniversity. The platform demonstrates the design and implementation of a relational database system to support core social networking features such as user authentication, media-rich content sharing, and real-time interaction.

The project is built using a modern and robust technology stack. Next.js, a React-based framework, powers the frontend and server-side rendering, enabling responsive and performant user interfaces. Prisma ORM is used to facilitate secure and efficient database interactions with a MySQL relational database, ensuring data integrity and relational consistency. Clerk is integrated for user authentication and session management, offering a streamlined and secure sign-up/sign-in experience. For media handling, Cloudinary is employed to manage and optimize user-uploaded images and videos, supporting rich media functionality that is essential for a social networking application.

2 Conceptual & Physical Design

2.1 Functional Requirements

2.1.1 User Account Management

User authentication and identity management are handled using **Clerk**, a modern authentication service tailored for full-stack applications. Clerk provides a robust and secure foundation through the following features:

- **Industry-Standard Security Practices:** Clerk follows best practices in security, including secure password hashing with `bcrypt`, encrypted token handling, and secure cookie management.
- **OAuth 2.0 and OpenID Connect (OIDC) Support:** Clerk integrates with trusted third-party identity providers (e.g., Google, GitHub) using standard authentication protocols.
- **JSON Web Tokens (JWTs):** For authenticated sessions, Clerk issues JWTs that are signed and optionally encrypted to ensure secure session management.
- **Multi-Factor Authentication (MFA):** Clerk supports optional MFA methods such as time-based one-time passwords (TOTP) and SMS verification, enhancing account security.
- **Role-Based Access Control (RBAC):** Developers can assign roles and permissions at the user level, enabling fine-grained access control.
- **Next.js Integration:** Clerk offers seamless integration with `Next.js`, allowing secure access to user information on both the client and server via middleware, API routes, and React components.

2.1.2 User Profile Operations

The system shall enable users to retrieve and modify their profile information, which includes fields such as: Full name, Unique username, Profile image,etc.

Profile data shall be stored in a normalized relational database schema with appropriate constraints to ensure data integrity, including a **UNIQUE** constraint on usernames and email addresses to prevent duplication.

Profile images shall be managed via **Cloudinary's** media management service, which provides secure, scalable storage and on-the-fly image optimization. Upon upload, the platform shall generate a unique URL for each image, which is persisted within the user profile record.

Access to profile images shall be controlled to prevent unauthorized retrieval, leveraging Cloudinary's signed URLs and secure delivery mechanisms.

2.1.3 Post Management

Authenticated users shall be empowered to perform Create, Read, Update, and Delete (CRUD) operations on posts. Each post may contain textual content and associated media files such as images. Media assets will be uploaded to Cloudinary, and their corresponding URLs will be stored within the database, linked to the posts via foreign key relationships to maintain referential integrity.

The post data model shall support timestamp metadata and ownership linkage, facilitating feed generation and user-specific content retrieval. All post modifications will be subject to validation rules and authorization checks to ensure only the post owners can modify or delete their content.

2.1.4 Social Interaction

The platform shall support user interactions on posts through liking and commenting functionalities. The system shall accurately record these interactions in dedicated relational tables with foreign key references to both users and posts.

Aggregate counts of likes and comments shall be maintained and efficiently queried for display in the user interface, using database indexing and optimized queries to ensure real-time performance. The platform shall enforce transactional consistency to prevent race conditions or data anomalies during concurrent interactions.

2.1.5 Content Feed and Search Functionality

The application shall generate and render a personalized content feed for each authenticated user, dynamically aggregating posts based on *relevance* and *recency*. Posts within the feed shall be ordered in descending order by their creation timestamps to prioritize the most recent content.

The feed retrieval process shall leverage optimized SQL queries and indexed database fields to ensure low-latency response times and scalability under concurrent access.

The content feed shall include comprehensive post details such as:

- Associated media URLs
- Author metadata (e.g., username, profile image)
- Real-time engagement metrics, including like and comment counts

Additionally, the system shall provide robust search functionality, enabling users to query the database for other users and posts based on exact or partial username matches.

Search queries shall utilize indexed columns and support case-insensitive matching to enhance usability and performance.

2.2 Non-functional Requirements

2.3 Performance

The system shall maintain an average response time of under 2 seconds for key user interactions, including feed loading, post creation, and liking, even under moderate concurrent user load. Database queries shall be optimized using indexing, performance tuning, and caching strategies to ensure efficient data retrieval.

2.3.1 Scalability

The platform architecture shall support horizontal scaling of the web application and database components to accommodate growth in user base and data volume. Media storage via Cloudinary ensures scalable handling of large volumes of images and videos without degrading application performance.

2.3.2 Security

User credentials shall be securely stored and managed using industry-standard encryption and hashing (bcrypt via Clerk). The application shall implement role-based access control (RBAC) to restrict functionality based on user roles (e.g., regular users vs. administrators). All data transmissions shall be encrypted using TLS/HTTPS to protect data confidentiality during network communication. The system shall prevent SQL injection and other common web vulnerabilities through the use of prepared statements and ORM safeguards.

2.3.3 Usability

The user interface shall be responsive and intuitive, accessible across multiple devices and screen sizes using responsive design principles. Error messages and validation feedback shall be clear, informative, and user-friendly to enhance user experience.

2.3.4 Maintainability

The codebase shall follow clean coding standards and be well-documented to facilitate ongoing development and debugging. Database schema changes and migrations shall be managed using version control and migration tools provided by Prisma.

2.3.5 Compliance and Privacy

The platform shall adhere to applicable data protection regulations, ensuring user data privacy and secure handling of personally identifiable information (PII). User data shall only be accessed and processed in accordance with user consent and clearly defined privacy policies.

2.4 Entity–Relationship Diagram

Our Entity–Relationship Diagram is provided in Firgure 1 & 2.

2.5 Normalization proof

- 1NF: All attributes are atomic, with with no repeating groups.
- 2NF: For tables with candidate keys, non-key attributes fully depend on the entire key.
- 3NF: No transitive dependencies exist; all non-key attributes depend directly on the primary or candidate keys.

3 Implementation of Database Entities

3.1 Data Definition Language

```
CREATE DATABASE IF NOT EXISTS social;

USE social;

CREATE TABLE `User` (
  id VARCHAR(191) PRIMARY KEY,
  email VARCHAR(191) UNIQUE NOT NULL,
  username VARCHAR(191) UNIQUE NOT NULL,
  avatar VARCHAR(191),
  cover VARCHAR(191),
  name VARCHAR(191),
  surname VARCHAR(191),
  description VARCHAR(191),
  city VARCHAR(191),
  school VARCHAR(191),
```

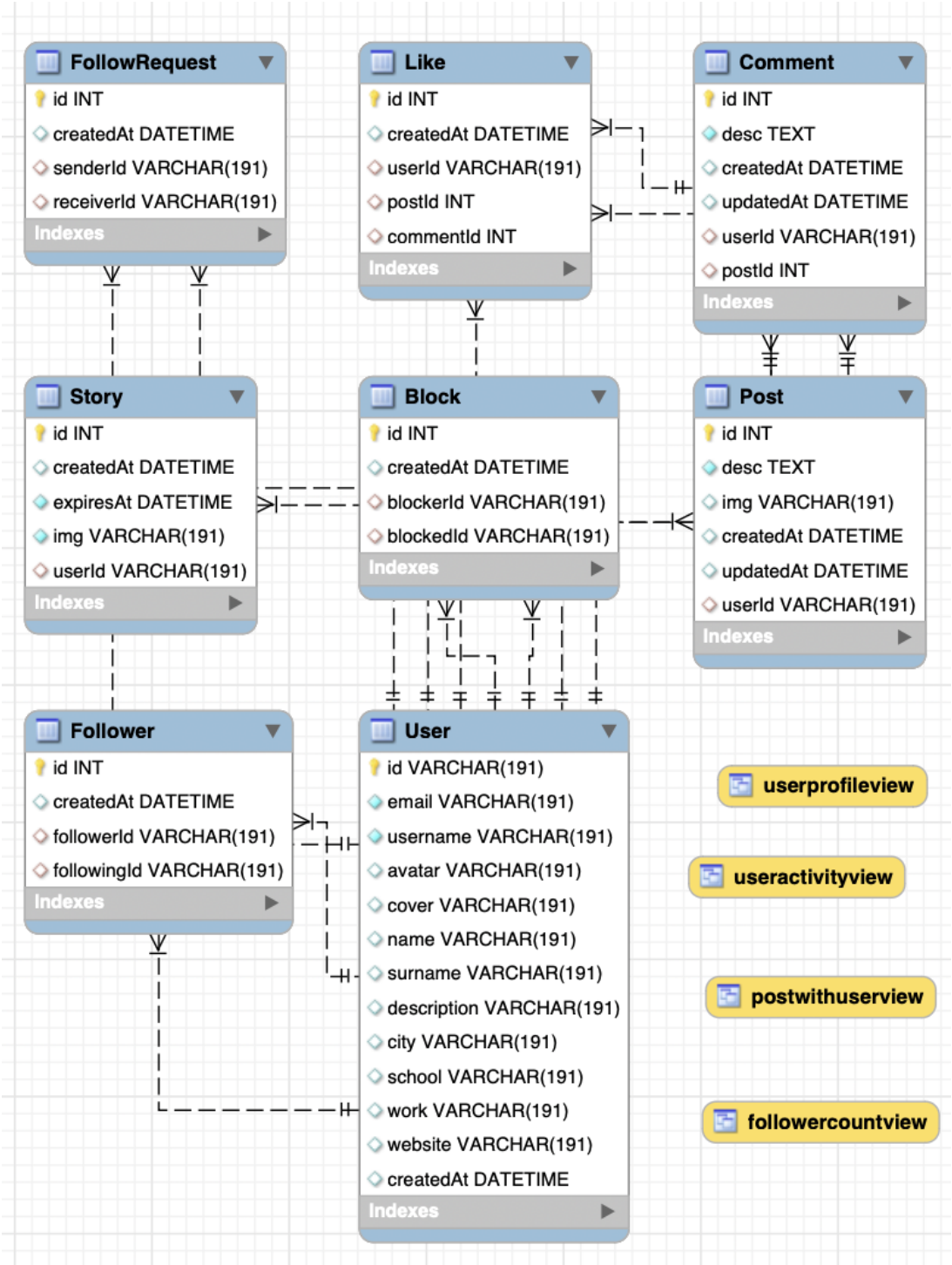

Figure 1: ERD in MySQL WorkBench

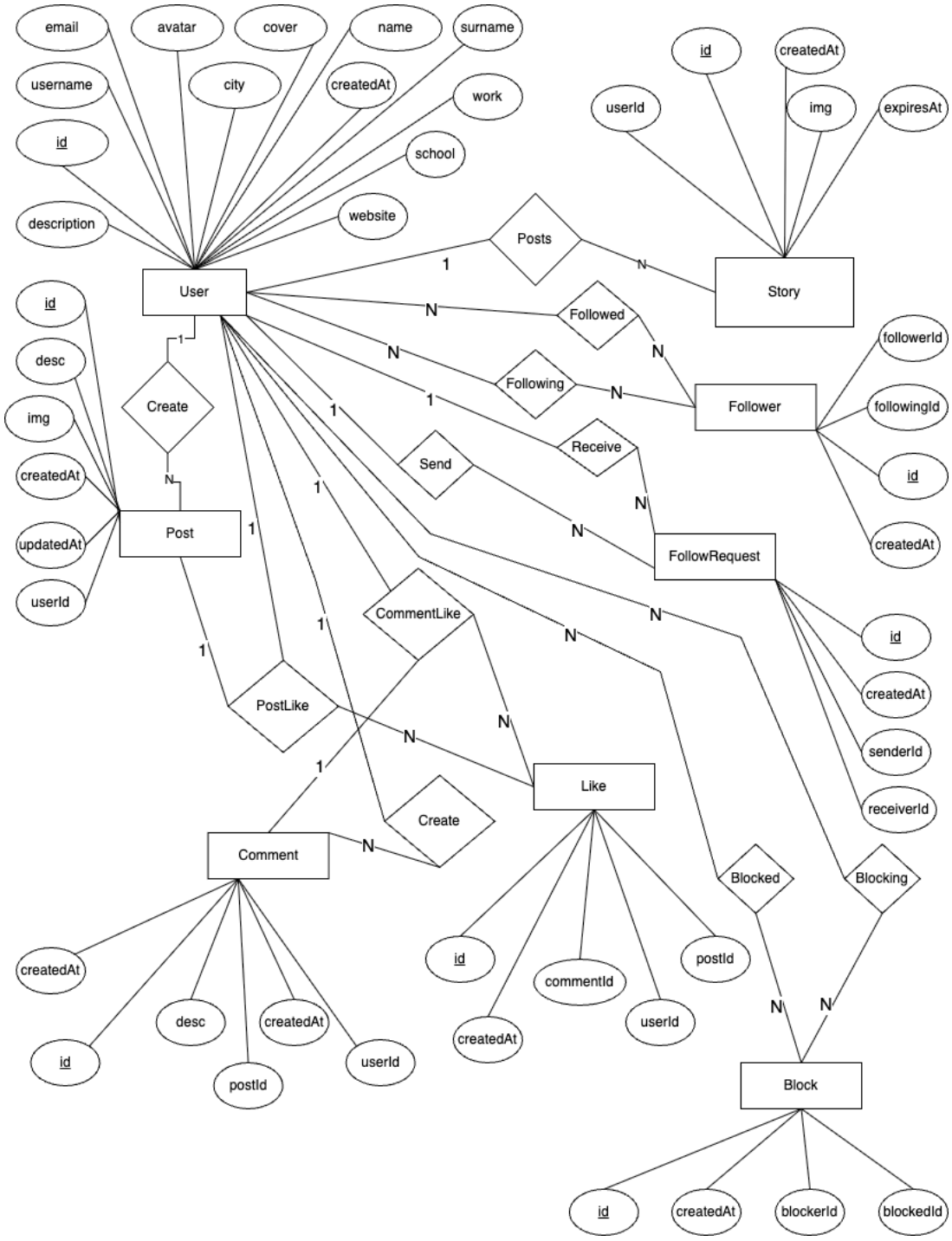

Figure 2: ERD in Chen Notation

```
  work VARCHAR(191),
  website VARCHAR(191),
  createdAt DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE Post (
  id INT AUTO_INCREMENT PRIMARY KEY,
  `desc` TEXT NOT NULL,
  img VARCHAR(191),
  createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
  updatedAt DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  userId VARCHAR(191),
  FOREIGN KEY (userId) REFERENCES `User`(id) ON DELETE CASCADE
);

CREATE TABLE Comment (
  id INT AUTO_INCREMENT PRIMARY KEY,
  `desc` TEXT NOT NULL,
  createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
  updatedAt DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  userId VARCHAR(191),
  postId INT,
  FOREIGN KEY (userId) REFERENCES `User`(id) ON DELETE CASCADE,
  FOREIGN KEY (postId) REFERENCES Post(id) ON DELETE CASCADE
);

CREATE TABLE `Like` (
  id INT AUTO_INCREMENT PRIMARY KEY,
  createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
  userId VARCHAR(191),
  postId INT,
  commentId INT,
  FOREIGN KEY (userId) REFERENCES `User`(id) ON DELETE CASCADE,
  FOREIGN KEY (postId) REFERENCES Post(id) ON DELETE CASCADE,
  FOREIGN KEY (commentId) REFERENCES Comment(id) ON DELETE CASCADE
);

CREATE TABLE Follower (
  id INT AUTO_INCREMENT PRIMARY KEY,
  createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
  followerId VARCHAR(191),
  followingId VARCHAR(191),
  FOREIGN KEY (followerId) REFERENCES `User`(id) ON DELETE CASCADE,
  FOREIGN KEY (followingId) REFERENCES `User`(id) ON DELETE CASCADE
);

CREATE TABLE FollowRequest (
  id INT AUTO_INCREMENT PRIMARY KEY,
  createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
  senderId VARCHAR(191),
  receiverId VARCHAR(191),
  FOREIGN KEY (senderId) REFERENCES `User`(id) ON DELETE CASCADE,
  FOREIGN KEY (receiverId) REFERENCES `User`(id) ON DELETE CASCADE,
  UNIQUE (senderId, receiverId)
);

CREATE TABLE Block (
  id INT AUTO_INCREMENT PRIMARY KEY,
  createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
  blockerId VARCHAR(191),
  blockedId VARCHAR(191),
```

```
  FOREIGN KEY (blockerId) REFERENCES `User`(id) ON DELETE CASCADE,
  FOREIGN KEY (blockedId) REFERENCES `User`(id) ON DELETE CASCADE,
  UNIQUE (blockerId, blockedId)
);

CREATE TABLE Story (
  id INT AUTO_INCREMENT PRIMARY KEY,
  createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
  expiresAt DATETIME NOT NULL,
  img VARCHAR(191) NOT NULL,
  userId VARCHAR(191) UNIQUE,
  FOREIGN KEY (userId) REFERENCES `User`(id) ON DELETE CASCADE
);
```

The DDL establishes a schema for a social media database with tables `User`, `Post`, `Comment`, `Like`, `Follower`, `FollowRequest`, `Block`, `Story` to manage core entities and relationships that satisfy the design.

```
-- Table for Statistics
CREATE TABLE UserStats (
    userId VARCHAR(191),
    statDate DATE,
    periodType ENUM('daily', 'monthly', 'annual'),
    postCount INT,
    commentCount INT,
    postRank INT,
    PRIMARY KEY (userId, statDate, periodType),
    FOREIGN KEY (userId) REFERENCES `User`(id) ON DELETE CASCADE
);
```

Additionally, we include a table to store aggregated statistics for frequent reporting.

3.2 View

```
-- UserProfileView
CREATE VIEW UserProfileView AS
SELECT id, username, name, surname, avatar, description, city, createdAt
FROM User;

-- PostWithUserView
CREATE VIEW PostWithUserView AS
SELECT p.id, p.desc, p.img, p.createdAt, p.updatedAt, u.username, u.avatar
FROM Post p
JOIN User u ON p.userId = u.id;

-- UserActivityView
CREATE VIEW UserActivityView AS
SELECT u.id, u.username,
       p.id AS postId, p.desc AS postDesc, p.createdAt AS postCreatedAt,
       c.id AS commentId, c.desc AS commentDesc, c.createdAt AS
           commentCreatedAt,
       l.postId AS likedPostId, l.createdAt AS likeCreatedAt
FROM User u
LEFT JOIN Post p ON u.id = p.userId
LEFT JOIN Comment c ON u.id = c.userId
LEFT JOIN `Like` l ON u.id = l.userId;

-- FollowerCountView
CREATE VIEW FollowerCountView AS
SELECT u.id, u.username,
```

```
        (SELECT COUNT(*) FROM Follower f WHERE f.followingId = u.id) AS
            followerCount,
        (SELECT COUNT(*) FROM Follower f WHERE f.followerId = u.id) AS
            followingCount
FROM User u;
```

These views streamline data retrieval by simplifying complex queries, improving performance for user profiles, feeds, activity tracking, and follower counts, while improving query maintainability.

3.3 Procedures

```
-- Procedure: Follow a user
DELIMITER $$

CREATE PROCEDURE FollowUser(
    IN p_followerId VARCHAR(191),
    IN p_followingId VARCHAR(191)
)
BEGIN
    IF NOT EXISTS (
        SELECT 1 FROM Block WHERE (blockerId = p_followingId AND blockedId =
            p_followerId)
    ) THEN
        INSERT IGNORE INTO Follower (followerId, followingId)
        VALUES (p_followerId, p_followingId);
    END IF;
END $$

DELIMITER ;

-- Procedure: Block a user
DELIMITER $$

CREATE PROCEDURE BlockUser(
  IN blockerId VARCHAR(191),
  IN blockedId VARCHAR(191)
)
BEGIN
  DELETE FROM Follower
  WHERE (followerId = blockerId AND followingId = blockedId)
     OR (followerId = blockedId AND followingId = blockerId);

  INSERT IGNORE INTO Block(blockerId, blockedId)
  VALUES (blockerId, blockedId);
END $$

DELIMITER ;

-- Procedure: Accept FollowRequest
DELIMITER $$

CREATE PROCEDURE AcceptFollowRequest(
  IN senderId VARCHAR(191),
  IN receiverId VARCHAR(191)
)
BEGIN
  DELETE FROM FollowRequest
  WHERE senderId = senderId AND receiverId = receiverId;

  INSERT IGNORE INTO Follower(followerId, followingId)
```

```
  VALUES (senderId, receiverId);
END $$

DELIMITER ;
```

These procedures centralize logic for following, blocking, and accepting follow requests, ensuring data integrity and reducing application-layer complexity through secure, atomic operations.

```
-- Procedure to Refresh Statistics
DELIMITER $$

CREATE PROCEDURE RefreshUserStats()
BEGIN
    -- Clear existing stats
    TRUNCATE TABLE UserStats;

    -- Daily Stats
    INSERT INTO UserStats (userId, statDate, periodType, postCount,
        commentCount, postRank)
    SELECT
        p.userId,
        DATE(p.createdAt) AS statDate,
        'daily' AS periodType,
        COUNT(p.id) AS postCount,
        (SELECT COUNT(*) FROM Comment c WHERE c.userId = p.userId AND
            DATE(c.createdAt) = DATE(p.createdAt)) AS commentCount,
        RANK() OVER (PARTITION BY DATE(p.createdAt) ORDER BY COUNT(p.id) DESC)
            AS postRank
    FROM Post p
    GROUP BY p.userId, DATE(p.createdAt);

    -- Monthly Stats
    INSERT INTO UserStats (userId, statDate, periodType, postCount,
        commentCount, postRank)
    SELECT
        p.userId,
        DATE_FORMAT(p.createdAt, '%Y-%m-01') AS statDate,
        'monthly' AS periodType,
        COUNT(p.id) AS postCount,
        (SELECT COUNT(*) FROM Comment c WHERE c.userId = p.userId AND
            DATE_FORMAT(c.createdAt, '%Y-%m') = DATE_FORMAT(p.createdAt,
            '%Y-%m')) AS commentCount,
        RANK() OVER (PARTITION BY DATE_FORMAT(p.createdAt, '%Y-%m') ORDER BY
            COUNT(p.id) DESC) AS postRank
    FROM Post p
    GROUP BY p.userId, DATE_FORMAT(p.createdAt, '%Y-%m');

    -- Annual Stats
    INSERT INTO UserStats (userId, statDate, periodType, postCount,
        commentCount, postRank)
    SELECT
        p.userId,
        DATE_FORMAT(p.createdAt, '%Y-01-01') AS statDate,
        'annual' AS periodType,
        COUNT(p.id) AS postCount,
        (SELECT COUNT(*) FROM Comment c WHERE c.userId = p.userId AND
            YEAR(c.createdAt) = YEAR(p.createdAt)) AS commentCount,
        RANK() OVER (PARTITION BY YEAR(p.createdAt) ORDER BY COUNT(p.id) DESC)
            AS postRank
    FROM Post p
    GROUP BY p.userId, YEAR(p.createdAt);
```

```
END $$

DELIMITER ;
```

We also include procedures to refresh the statistic table, calculating metrics like total posts and comments by user, grouped by time periods, using `DATE_FORMAT` and COUNT with window functions for ranking. This ensures that we can refresh and store the user statistic frequently.

3.4 Trigger

```
-- Trigger: Prevent FollowRequest if Block
DELIMITER $$

CREATE TRIGGER PreventBlockedFollowRequest
BEFORE INSERT ON FollowRequest
FOR EACH ROW
BEGIN
  IF EXISTS (
    SELECT 1 FROM Block
    WHERE blockerId = NEW.receiverId AND blockedId = NEW.senderId
  ) THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'USER NOT FOUND';
  END IF;
END $$

DELIMITER ;
```

The trigger enforces privacy by blocking follow requests from blocked users, ensuring data consistency and reducing application-layer validation overhead.

4 Performance Tuning

4.1 Indexing

```
-- Indexing
CREATE INDEX idx_user_name ON User(name);
CREATE INDEX idx_post_userId ON Post(userId);
CREATE INDEX idx_post_createdAt ON Post(createdAt);
CREATE INDEX idx_comment_userId ON Comment(userId);
CREATE INDEX idx_comment_postId ON Comment(postId);
CREATE INDEX idx_comment_createdAt ON Comment(createdAt);
CREATE INDEX idx_like_userId_postId ON `Like`(userId, postId);
CREATE INDEX idx_like_userId_commentId ON `Like`(userId, commentId);
```

These indexes optimize query performance for user searches, feed retrieval, and interaction tracking, balancing read efficiency with write overhead.

4.2 Query optimization

To test and demonstrate the performance benefits of our indexing, we will use the query `"SELECT * FROM User WHERE name LIKE 'Name1234%';"` to compare results before and after using the index `idx_user_name`. Without the index, the query needs to scan through the entire User table row by row 3, resulting in a full table scan and significantly higher resource usage 4. After creating the index, MySQL can efficiently locate matching records using the index tree 5, drastically reducing the number of rows accessed and improving query performance 6.

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	User	NULL	ALL	NULL	NULL	NULL	NULL	98610	11.11	Using where

Figure 3: Query explain before optimization

Variable_name	Value
Handler_read_first	1
Handler_read_key	1
Handler_read_last	0
Handler_read_next	0
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_next	100002

Figure 4: Query before optimization

5 Security Configuration

```
CREATE USER 'admin'@'localhost' IDENTIFIED BY 'secure_admin_pass';
CREATE USER 'normal_user'@'localhost' IDENTIFIED BY 'secure_user_pass';
CREATE USER 'read_only'@'localhost' IDENTIFIED BY 'secure_read_pass';

GRANT ALL PRIVILEGES ON social.* TO 'admin'@'localhost';
GRANT SELECT, INSERT, UPDATE, DELETE ON social.`User` TO
    'normal_user'@'localhost';
GRANT SELECT, INSERT, UPDATE, DELETE ON social.Post TO
    'normal_user'@'localhost';
GRANT SELECT, INSERT, UPDATE, DELETE ON social.Comment TO
    'normal_user'@'localhost';
GRANT SELECT, INSERT, DELETE ON social.`Like` TO 'normal_user'@'localhost';
GRANT SELECT, INSERT, DELETE ON social.Follower TO 'normal_user'@'localhost';
GRANT SELECT, INSERT, DELETE ON social.FollowRequest TO
    'normal_user'@'localhost';
GRANT SELECT, INSERT, DELETE ON social.Block TO 'normal_user'@'localhost';
GRANT SELECT, INSERT ON social.Story TO 'normal_user'@'localhost';
GRANT SELECT ON social.* TO 'read_only'@'localhost';
GRANT SELECT ON social.UserStats TO 'read_only'@'localhost';
```

The `admin@localhost` user has full access to the social database for administrative tasks, restricted to local connections for safety. The `normal_user@localhost` is granted `SELECT, INSERT, UPDATE`, and `DELETE` on tables like `User, Post, Comment, Like, Follower, FollowRequest, and Block`, with limited `INSERT, DELETE` access to Story, aligning with typical user actions while minimizing risks. The `read_only@localhost` user has SELECT access across all tables and `UserStats` for analytics, preventing any modifications.

6 End-to-End Testing & Web Integration

Our Web interface is integrated as shown in Figure 7.

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	User	NULL	range	idx_user_name	idx_user_name	767	NULL	11	100.00	Using index condition

Figure 5: Query explain after optimization

Variable_name	Value
Handler_read_first	0
Handler_read_key	1
Handler_read_last	0
Handler_read_next	11
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_next	0

Figure 6: Query after optimization

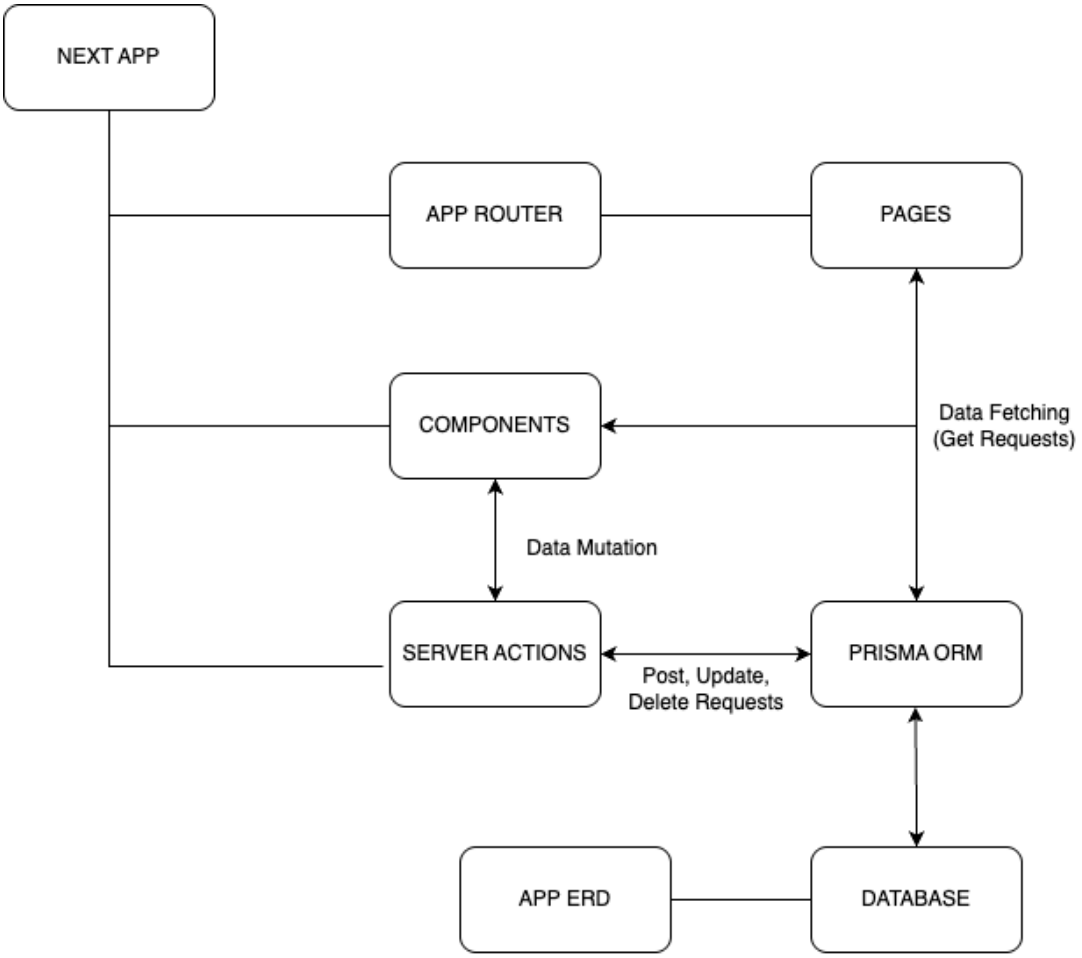

Figure 7: Severus Social Network Web Interface

Some of the core functionalities of the application include:

- User authentication and authorization (Figure 8).
- A homepage interface that supports image uploading and post management (Figure 9).
- Interactive post engagement features, such as liking and commenting (Figure 10).
- Post deletion functionality, allowing users to remove their own posts when desired (Figure 11).
- Story functionality allowing users to add and view stories (Figures 12 and 13).
- A dedicated view for displaying blocked users (Figure 14).
- User profile management, including profile viewing and editing (Figures 15 and 16).
- Functionality to update personal user information (Figure 17).
- Real-time handling of follow requests (Figure 18).
- Centralized user management through Clerk (Figure 19).

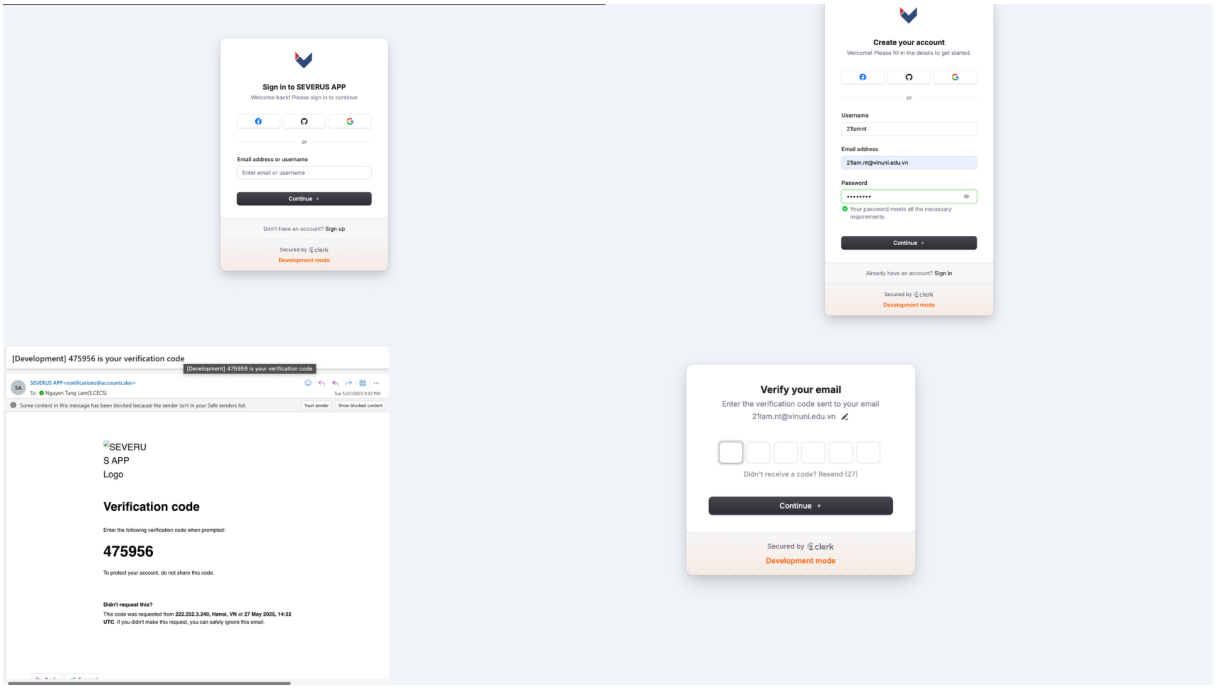

Figure 8: Sign in and Sign up Process

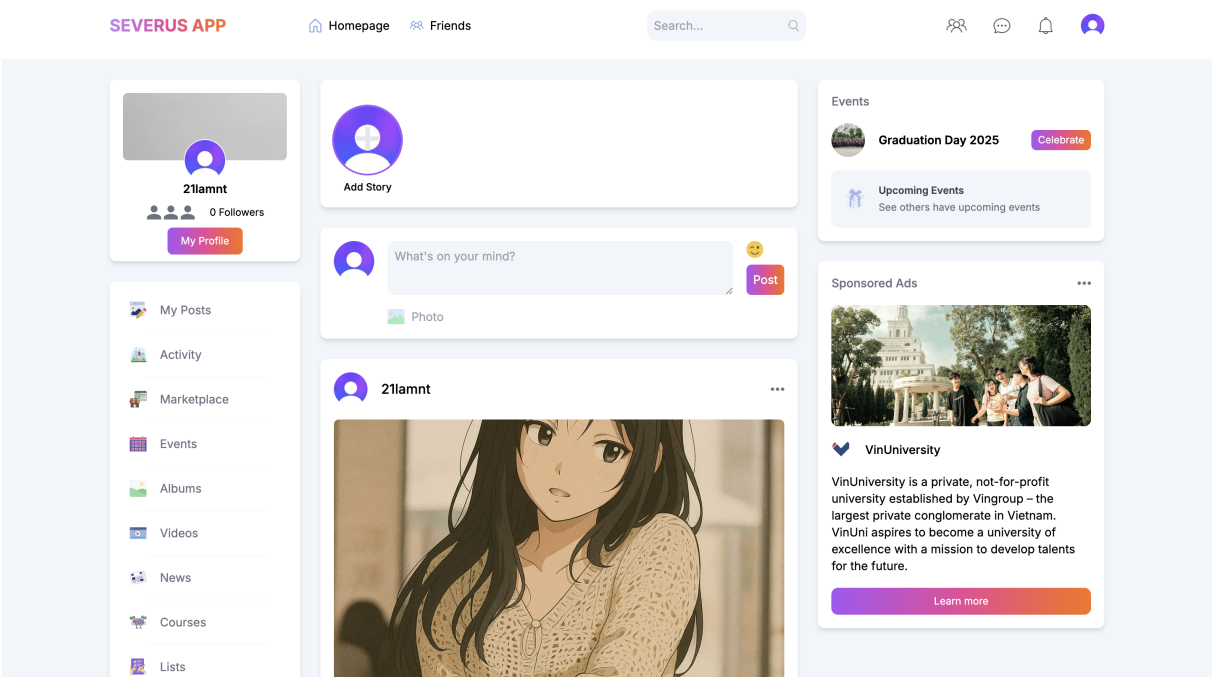

Figure 9: Homepage

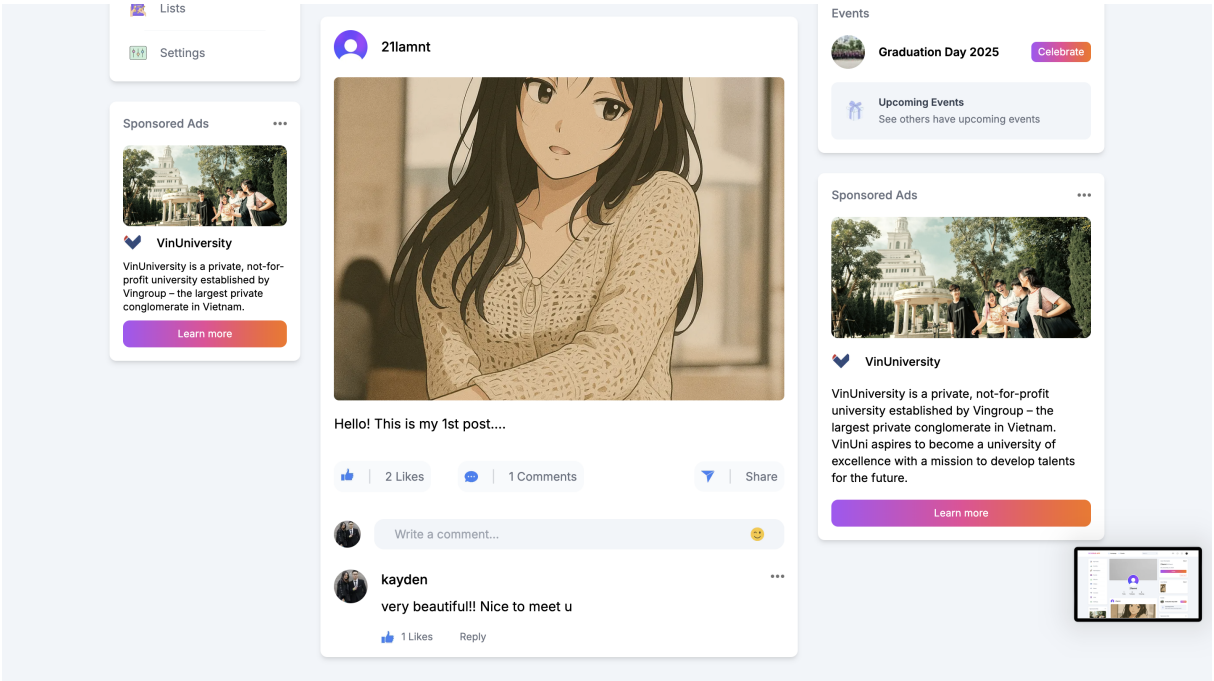

Figure 10: Comment & Like

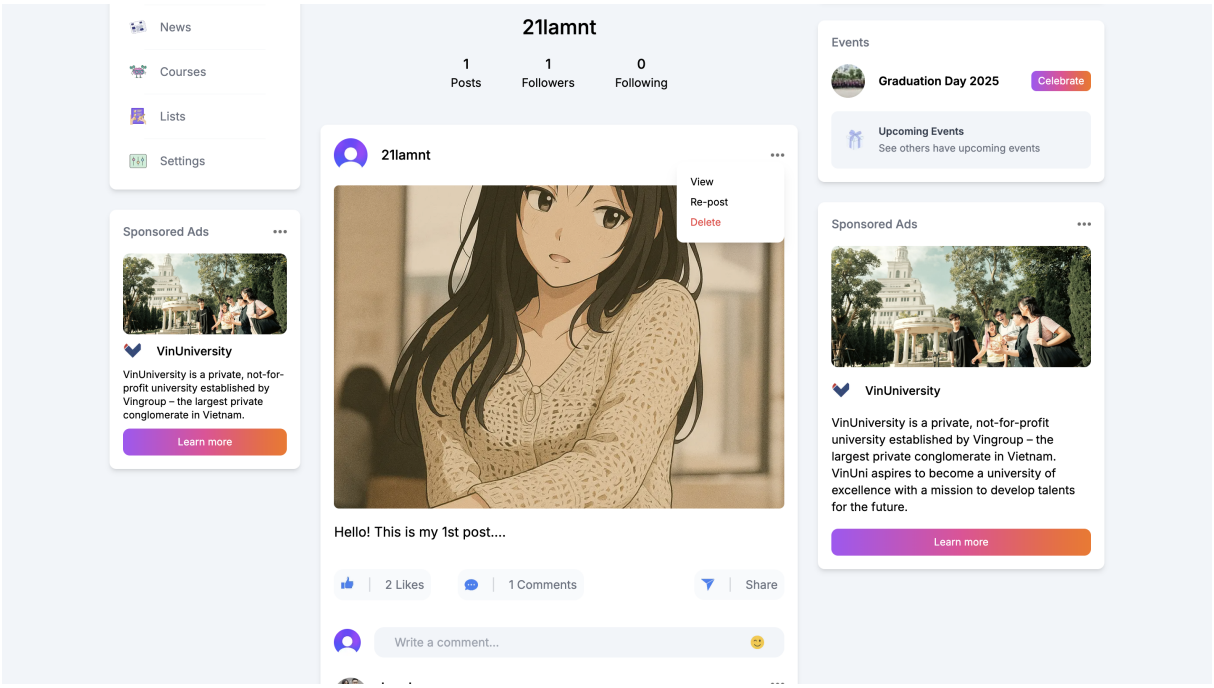

Figure 11: Post Delete

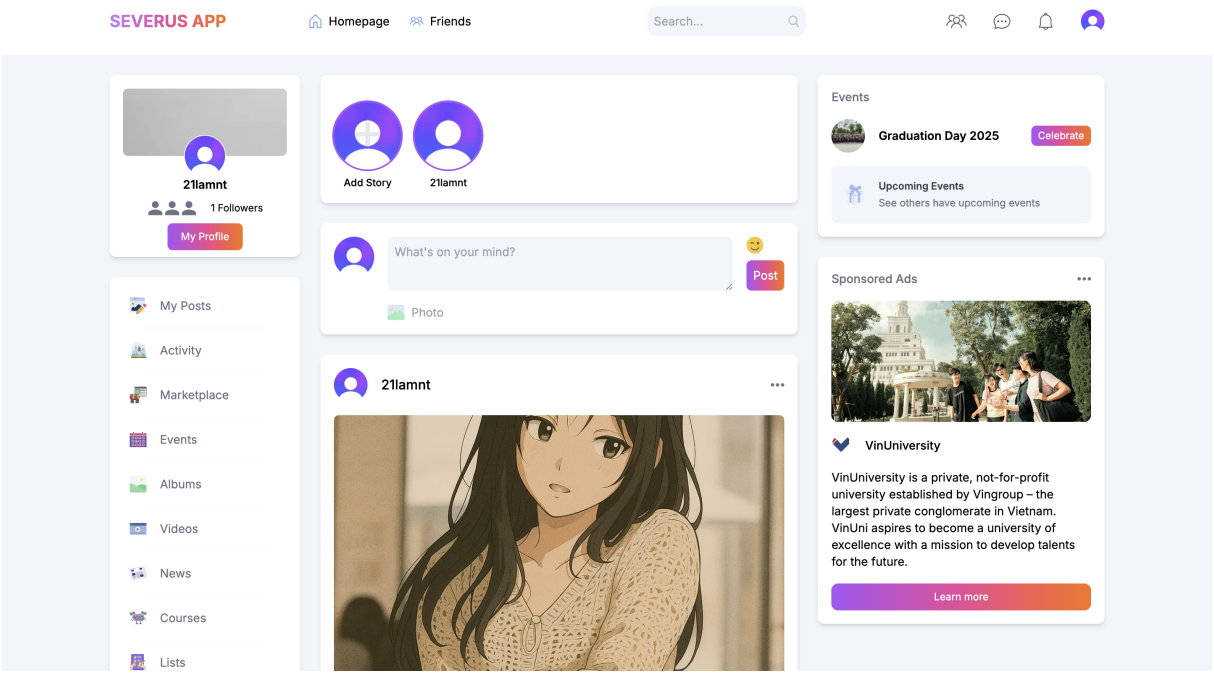

Figure 12: Add story

Figure 13: Story View

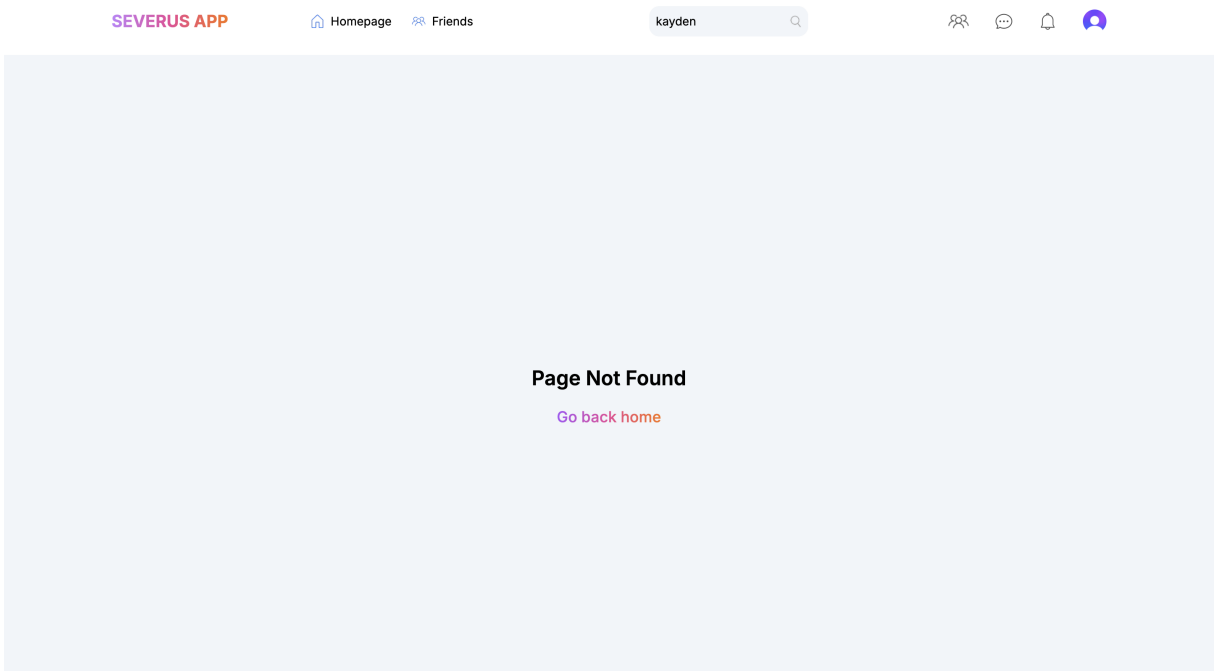

Figure 14: Blocked View

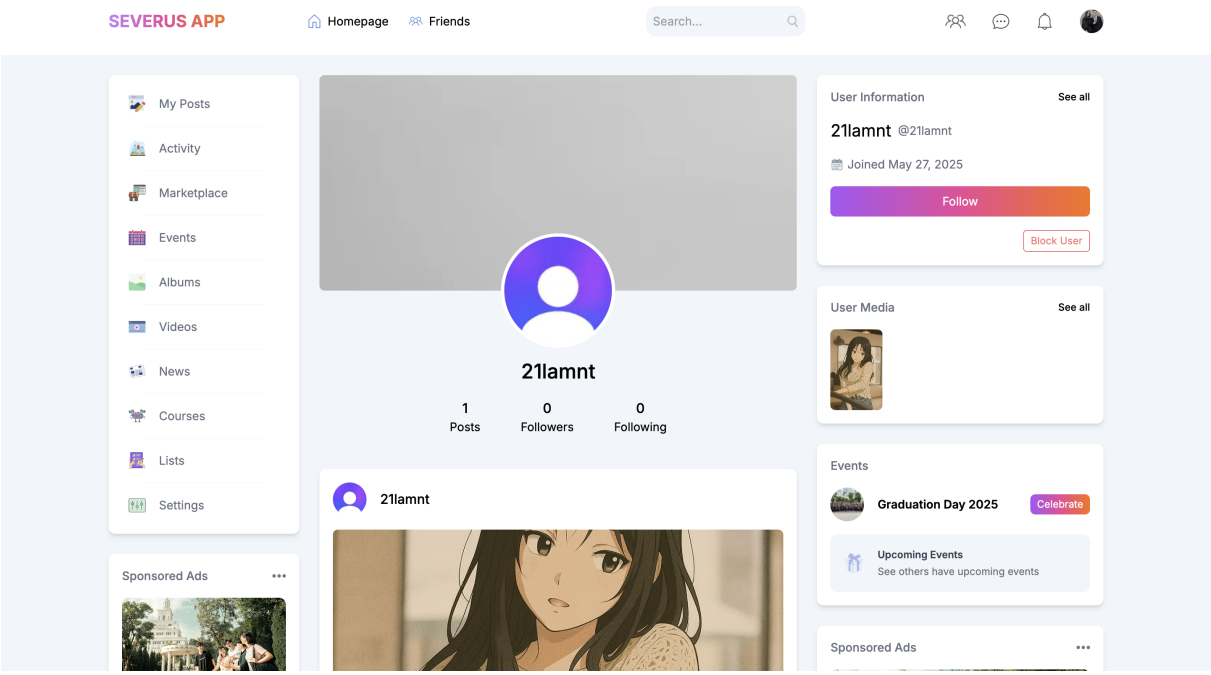

Figure 15: User 1 Profile

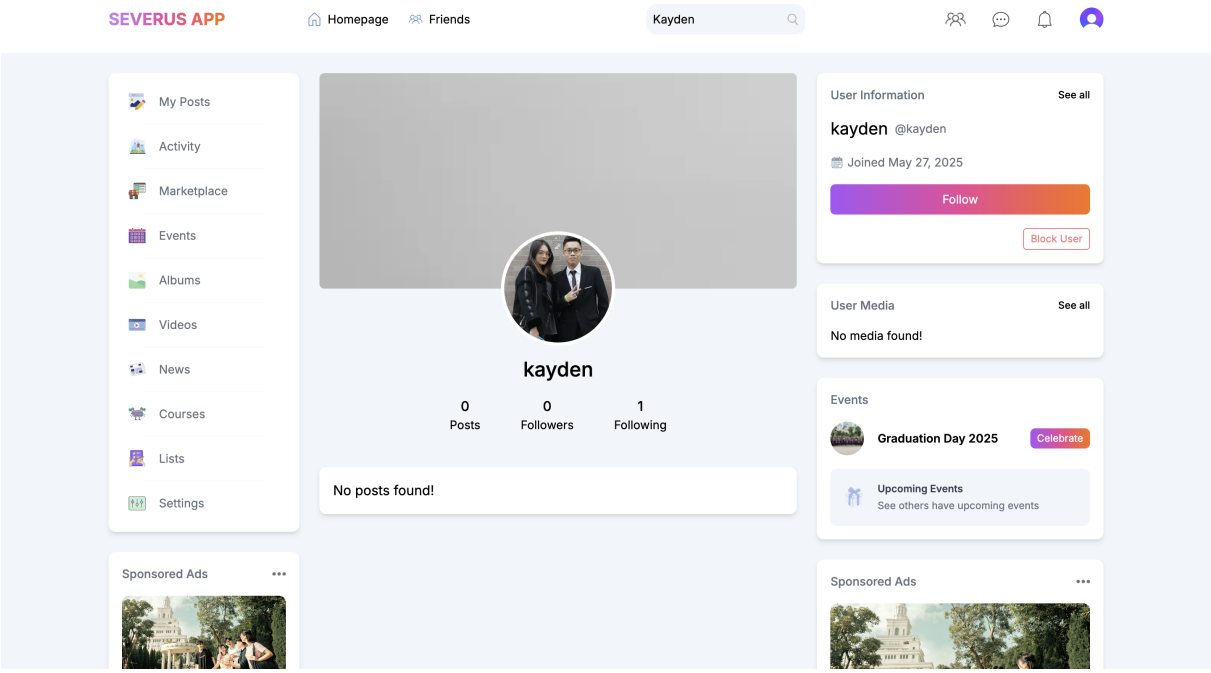

Figure 16: User 2 Profile

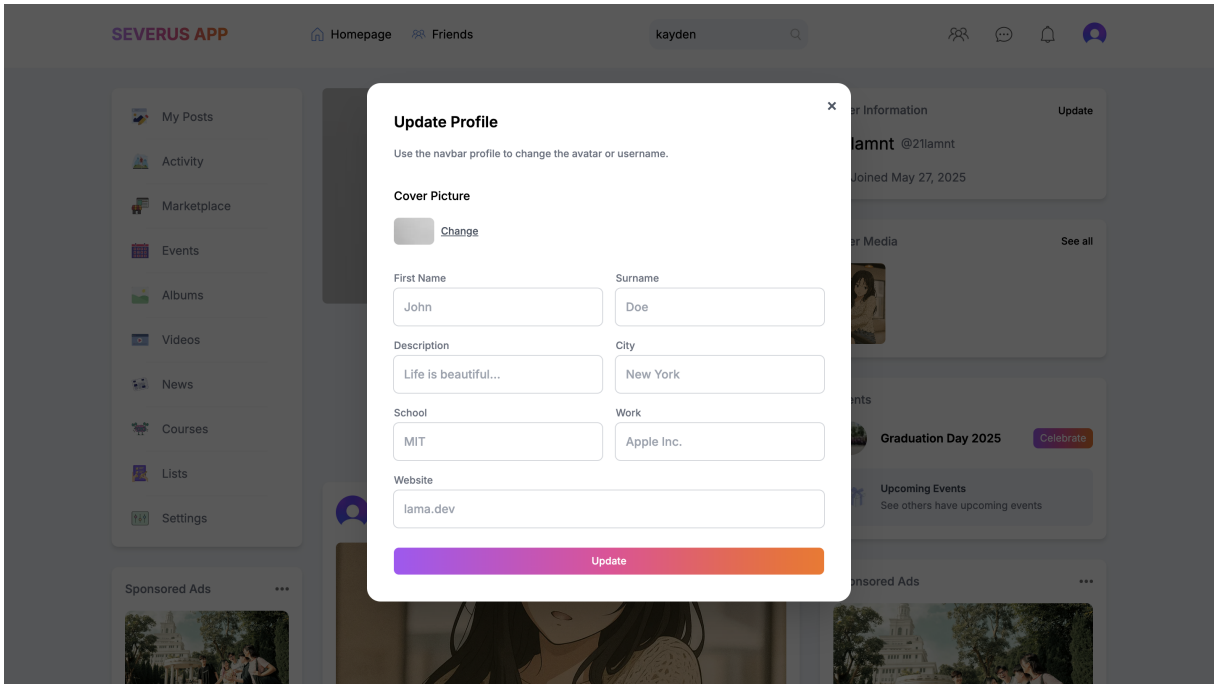

Figure 17: Update user information

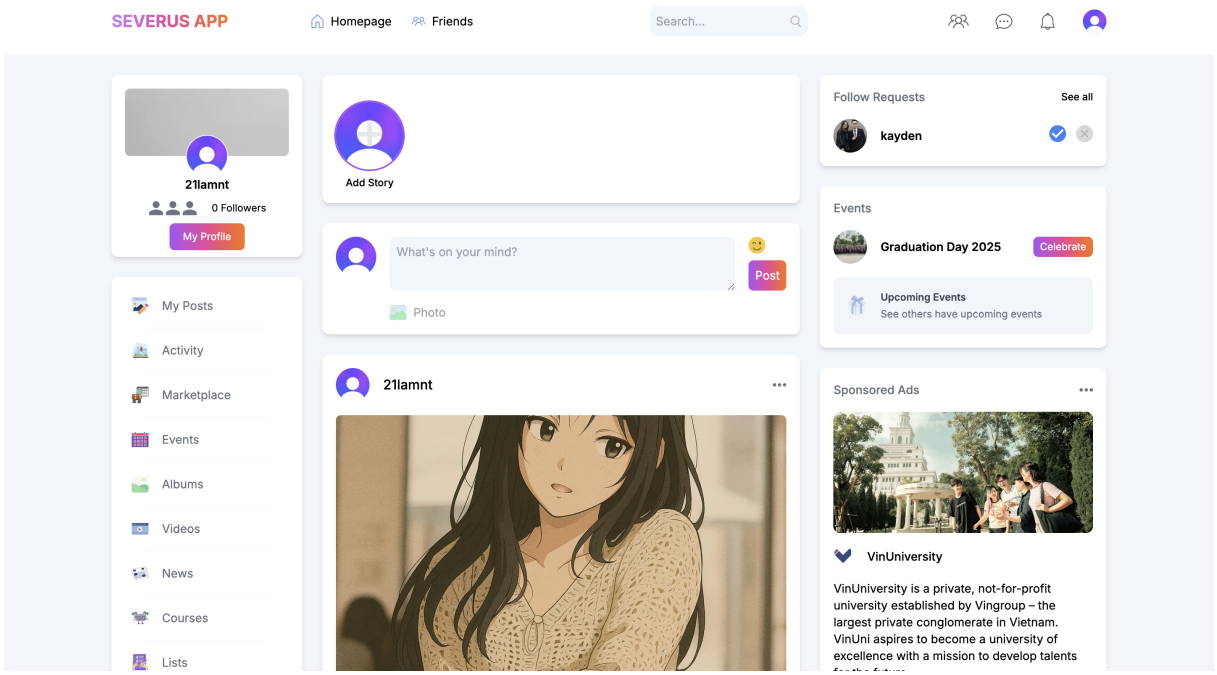

Figure 18: Follow Request

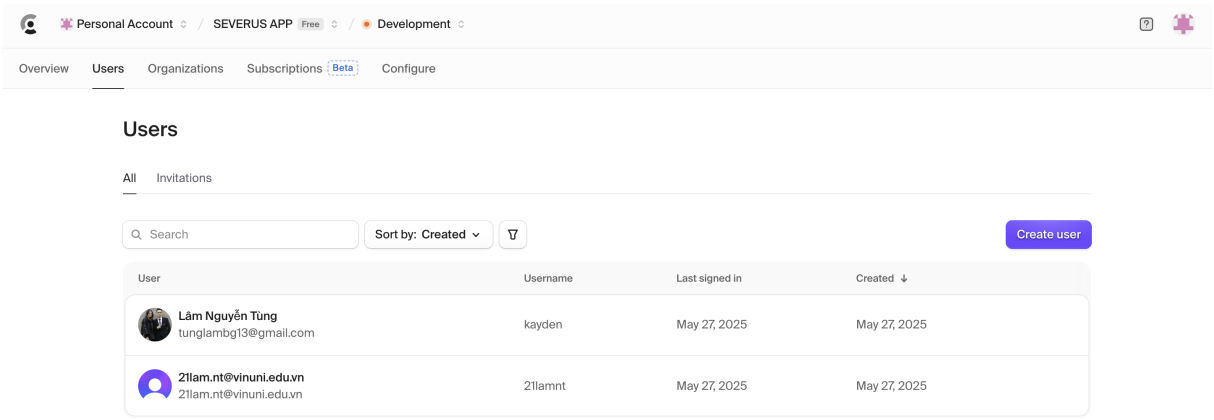

Figure 19: User management from Clerk

In addition to core UI functionality, the application fully supports Create, Read, Update, and Delete (CRUD) operations for all key resources such as users, posts, and comments. These operations are handled seamlessly through the web interface, providing users with intuitive controls and real-time feedback. Moreover, the application integrates basic analytics, including metrics such as post statistics, number of likes.

The system is designed with user experience in mind, ensuring smooth and responsive interaction across all features. All inputs and actions are validated, with appropriate success and error messages provided through toast notifications and modals from React.

7 Presentation & Documentation

The presentation materials and accompanying documentation are available on our GitHub repository.

Acknowledgements

We would like to express our sincere gratitude to our supervisor, **Prof.Le Duy Dung** and **Mr.Ta Quang Hieu** from CECS, for their invaluable guidance, continuous support, and insightful feedback throughout the course of this project.

Moreover, we would like to acknowledge our peers and friends for their feedback and support during the development of this project.