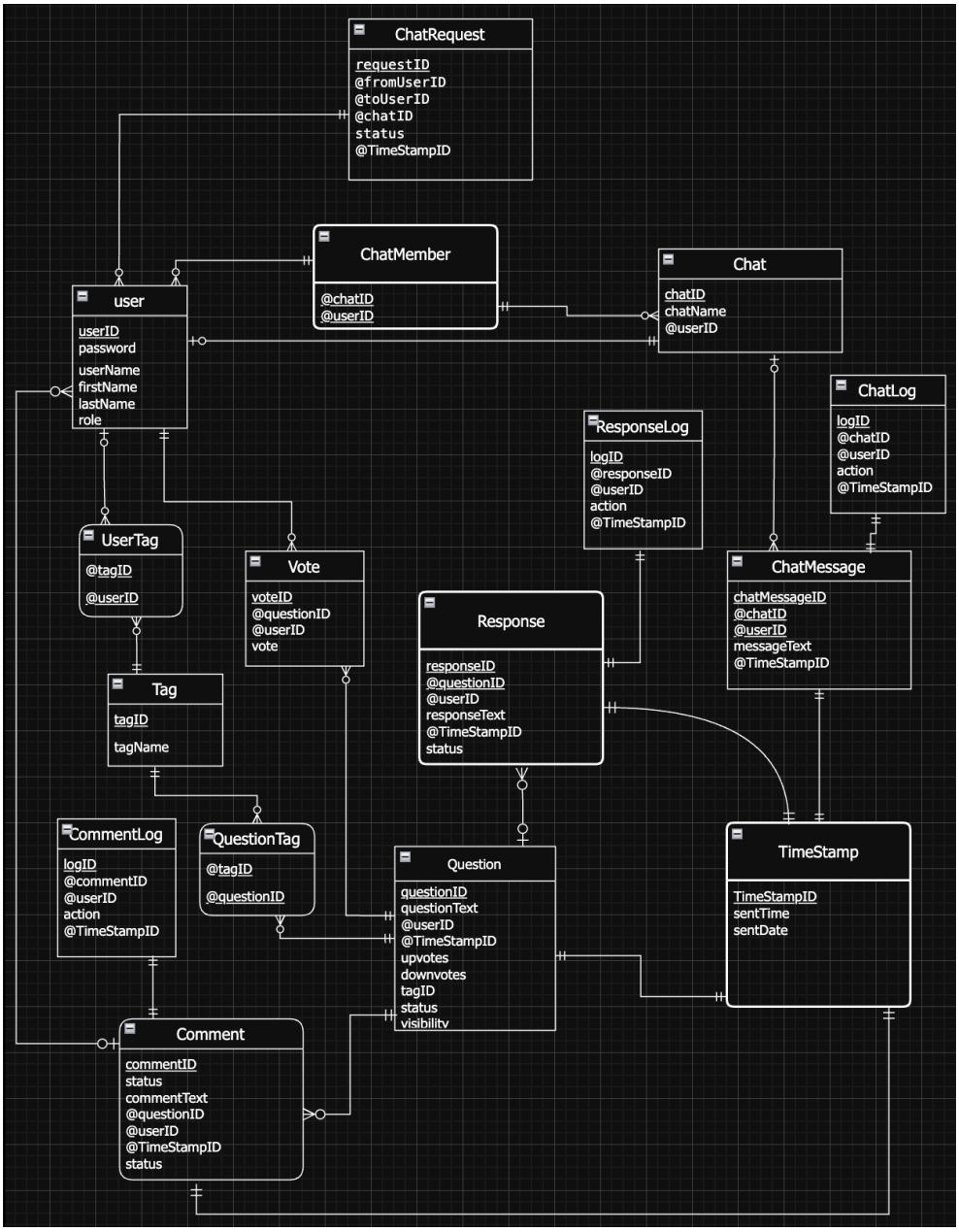
Project Repository - Kushal Mamillapalli (km6238@nyu.edu), Sihan Ma (sm11627@nyu.edu), Ayush Sarkar (as17812@nyu.edu)

Database Design

a) E-R Diagram



b) Schema Statements

i) Exact Schema included in COMMANDSV2.sql

- **User** (userID, userName, password, firstName, lastName, role)
- **Tag** (tagID, tagName)
- **Timestamp** (TimeStampID, sentTime, sentDate)
- **Question** (questionID, questionText, upvotes, downvotes, status, visibility, @userID, @tagID, @TimeStampID)

- **UserTag** (tagID, @userID)
- **QuestionTag** (tagID, @questionID)
- **Chat** (chatID, chatName, userID)
- **Response** (responseID, responseText, status, @userID, @questionID, @TimeStampID)
- **Comment** (commentID, commentText, status, @userID, @questionID, @TimeStampID)
- **ChatMessage** (chatMessageID, messageText, @userID, @chatID, @TimeStampID)
- **ChatMember** (@userID, @chatID)
- **ChatRequest** (requestID, status, @userID, @userID, @chatID, @TimeStampID)
- **ChatLog** (logID, action, @userID, @chatID, @TimeStampID)
- **ResponseLog** (logID, action, @userID, @responseID, @TimeStampID)
- **CommentLog** (logID, action, @userID, @responseID, @TimeStampID)
- **Vote** (voteID, @userID, @questionID, vote)

Database Programming

a) The database is hosted with mysql locally. When running the app locally or using a web link, the local database is referenced for any operations.

b) The app is run with flask, which takes care of the routing for the endpoints in regards to the web pages. The environment requires flask, pandas, and some other small python packages to run with the mysql server. To run the app through the web, instead of localhost, we use ngrok to create a link that can be accessible from multiple devices and allow multiple users to connect and use the functionality of the Mango app.

c) Exact steps are described in the instructions.txt. The basic steps are to set up the database in mysql (non-xampp). This is done using the mysql commands outlined on COMMANDSV2.SQL, and PERMISSIONS.SQL. Commands sets up the relevant tables after creating the mango database and the permissions sets up the user used to access the database. After this is done, running main.py will start the server locally and it can be accessed by a single user. For the demonstration/multiple user functionality, ngrok will be used to run the app on the web, and more instructions to do that step as a test are outlined in the file. Basically, you setup ngrok and use a command that takes the localhost link and outputs a web link that can be accessed by multiple users.

d)

Triggers:

- **OnUserDelete** automatically deletes related data when a user is deleted. It ensures referential integrity by calling deletions across dependent tables. This trigger reflects the application's need to clean up related data when a user account is removed, preventing floating records.

- **BeforeInsertChatMessage, BeforeInsertComment,** and **BeforeInsertResponse** all generate a timestamp for messages/comments/responses respectively if not provided. This simplifies the insertion of chat messages by automating timestamp creation. This trigger reflects the application's focus on tracking the timing for user interactions.
- **AfterChatRequestAccepted** adds a user to a chat when their chat request is accepted. This automates the process of updating chat memberships. This trigger reflects the application's need to manage dynamic relationships between users and chats.

Checks:

- There is a check constraint to ensure that the role column in the **User** table only contains valid values (user or admin). This prevents invalid roles from being inserted into the database. This constraint reflects the application's need to differentiate between regular and administrative users.
- There is a check constraint in the **Vote** table to ensure that the vote column only contains up or down. This prevents invalid vote types from being recorded and reflects the application's need to maintain valid voting data
- In the **Question** table, there are check constraints to ensure that upvotes and downvotes cannot be negative. This feature prevents invalid vote counts and reflects the application's need to maintain accurate vote tallies.

Procedures:

The project includes over 30 stored procedures, which would be tedious to explain one by one. However, there isn't anything unique to each stored procedure. The reason stored procedures were used is to simplify the logic of the application. Instead of writing a specific command in the main.py, we can just call a procedure which keeps the line count shorter and also functionality more clear since each procedure has a name associated with it that can help explain the SQL behind it.

Security at a Database Level

a) Since our app's focus is around end users, it is primarily set up for them in mind. This means that we just have one "admin" user in SQL with restricted privileges to stop injection attacks that can harm the database and make sure that there is no way for standard users to engage in restricted actions.

b) The security was set up by using only one account, mango_user, to access the app/database, when logging in. This means that anyone accessing the app must be limited by the mango_user's permissions which is good for securing end users and the app. We intentionally do not have a way to promote a user to "admin", as

anyone who needs to edit the structure itself would just be a developer working on backend, and so they wouldn't be restricted by standard user privileges.

c) All the commands are described in the PERMISSIONS.SQL file, but they are organized in 4 groups:

```
CREATE USER 'mango_user'@'localhost' IDENTIFIED BY
'arfaouiRocks123';
```

This creates the user, and credentials are used in main.py to run the app with the account.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON `mango`.* TO
'mango_user'@'localhost';
```

This adds privileges to the user that are used in the app.

```
GRANT EXECUTE ON PROCEDURE `Mango`.`CancelComment` TO
'mango_user'@'localhost';
```

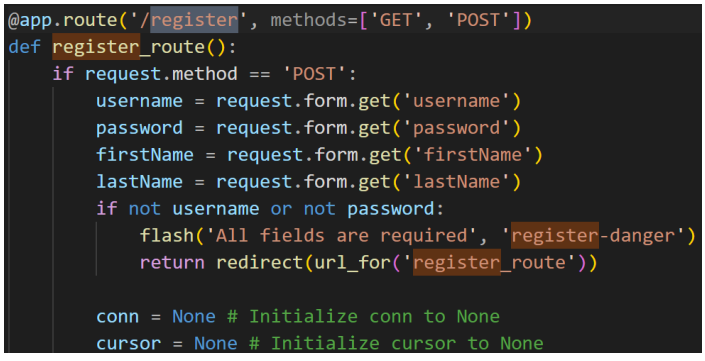
There are a bunch of statements like this, but the summary is that each procedure should be manually granted, and this can be restricted in ways so that procedures used by developers are not given to the mango_user, which is good for database security and not giving them unnecessary access, which allows for exploits.

```
REVOKE DROP ON *.* FROM 'mango_user'@'localhost';
```

This command basically removes some dangerous privileges that could be used in exploits to attack the database. For example, if someone tried to sql inject drop commands with commonly used database names, they could potentially drop important tables and inhibit the application's functionality.

Security at Application Level

User Authentication: We ensure that only authorized users can access the full application. Passwords are hashed before being stored in the database to prevent exposure. Users also must be logged in to interact with the app.



Code Snippets: User Registration and Password Hashing

Role-Based Access Control: This ensures that only admin users can access certain features like user logs or perform specific actions like hiding questions.

```python
@app.route('/admin_dashboard')
def admin_dashboard():
    #only admins access this route
    if session.get('role') != 'admin':
        flash('Access denied: Admins only.', 'danger')
        return redirect(url_for('home'))
    #fetch admin-specific data (e.g., user/question management)
    conn = connect_db()
    cursor = conn.cursor(pymysql.cursors.DictCursor)
    try:
        cursor.execute("""
            SELECT questionID, questionText, visibility, userID
            FROM Question
            ORDER BY TimeStampID DESC
        """)
        questions = cursor.fetchall()
    except Exception as e:
        flash(f"Error loading admin dashboard: {e}")
        questions = []
    finally:
        cursor.close()
        conn.close()
    return render_template('admin_dashboard.html', questions=questions)
```

Code Snippet: Admin-Only Access

Session Management: A user must be logged in to access and respond to other users' posts. To manage unique instances of a user, we require that sessions be used to track logged-in users and their roles. Session expiration is enforced to prevent unauthorized access (session.pop is used to remove those variables and reset for the next login).

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        conn = connect_db()
        authenticated, role = verify_user(conn, username, password)
        if authenticated:
            cursor = conn.cursor(pymysql.cursors.DictCursor)
            cursor.execute("SELECT userID, role FROM User WHERE userName = %s", (username,))
            user = cursor.fetchone()
            session['username'] = username
            session['role'] = role
            session['userID'] = user['userID']
            return redirect(url_for('dashboard'))
        else:
            flash('Login failed: Invalid username or password.', 'login-danger')
            return redirect(url_for('login'))
    return render_template('user_login.html')
```

```python
@app.route('/logout')
def logout():
    session.pop('username', None)
    session.pop('role', None)
    session.pop('userID', None)
    return redirect(url_for('home'))
```

```python
# Redirect to login if user is not authenticated
if 'username' not in session:
    return redirect(url_for('login'))
```

Code Snippets: Protection through Login Checks and Logout Data Management

Flash Messages: We use flash messages to communicate to the user the current status of their request. This includes hiding/unhiding posts, publishing posts, accepting messages, and more. Each flash message has a different category that allows the html to get only the right ones; in the below example the register route uses the register-____ alerts to make sure only register alerts go through, not ones from chats, login, or something else.

```python
def register_route():
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')
        firstName = request.form.get('firstName')
        lastName = request.form.get('lastName')
        if not username or not password:
            flash('All fields are required', 'register-danger')
            return redirect(url_for('register_route'))

        conn = None # Initialize conn to None
        cursor = None # Initialize cursor to None
        try:
            conn = connect_db()
            cursor = conn.cursor(pymysql.cursors.DictCursor) # Use DictCursor for consistency
            cursor.execute("SELECT COUNT(*) AS count FROM User WHERE username = %s", (username,))
            user_exists = cursor.fetchone()['count'] > 0

            if user_exists:
                flash('Username already exists. Please choose a different username.', 'register-danger')
                return redirect(url_for('register_route')) # Redirect if user exists

            # Call register_user within the same try block
            register_user(conn, username, password, 'user', firstName, lastName)
            conn.commit() # Commit after successful registration
            flash('Registration successful! You can now log in.', 'register-success')
            return redirect(url_for('login')) # Redirect on success

        except Exception as e:
            if conn: # Check if connection exists before rollback
                conn.rollback()
            flash(f'Registration error: {e}', 'register-danger')
            # Redirect back to registration page on error
            return redirect(url_for('register_route'))
        finally:
            if cursor: # Check if cursor exists before closing
                cursor.close()
            if conn: # Check if connection exists before closing
                conn.close()
```

Code Snippet: Flash example