CSCE 156: Project 2

Jstagram 2.0 Design Document

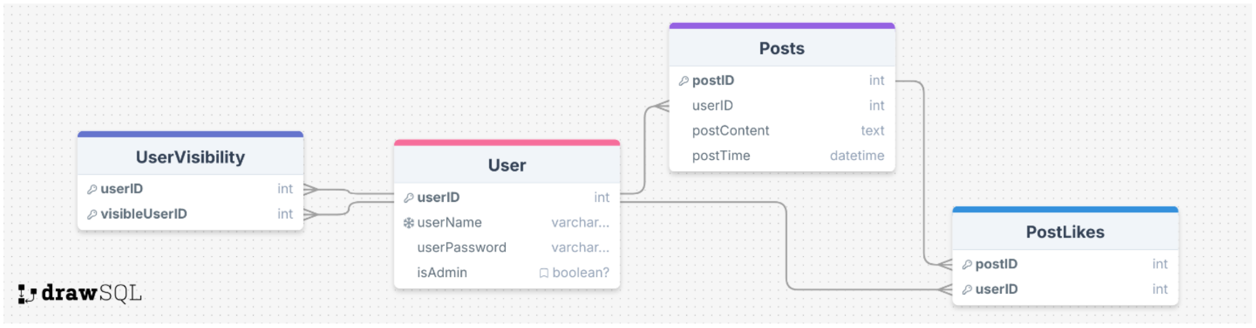
By Chua Zhen Keat

Introduction

Jstagram 2.0 is a social media application designed to offer core functionalities like user account management, post creation, and visibility control. The app supports two types of users: regular users and admins. At high level, regular users can manage their visibility lists, create and interact with posts, and like or unlike posts. Admins have additional powers, such as promoting users, creating or deleting accounts, and managing all posts in the system. Targeted audience include users looking for a lightweight, controlled social media platform and developers aiming to learn the integration of Java and MySQL in a real-world application.

ER Diagram

Jstagram 2.0 is a social media application designed to offer core functionalities like user



3NF Compliance

All database tables are designed to comply with the Third Normal Form (3NF):

1. User Table

Attributes: userID (PK), userName, userPassword, isAdmin.

1NF: All attributes are atomic; no repeating groups.

2NF: No partial dependencies; primary key is a single attribute (userID).

3NF: No transitive dependencies; all attributes depend solely on the primary key.

2. Posts Table

Attributes: postID (PK), userID (FK), postContent, postTime.

1NF: All attributes are atomic.

2NF: No partial dependencies; primary key is postID.

3NF: No transitive dependencies; each attribute depends only on postID.

3. UserVisibility Table

Attributes: userID (PK, FK), visibleUserID (PK, FK).

1NF: All attributes are atomic.

2NF: Composite primary key (userID, visibleUserID); no partial dependencies.

3NF: No transitive dependencies; attributes depend only on the composite primary key.

4. PostLikes Table

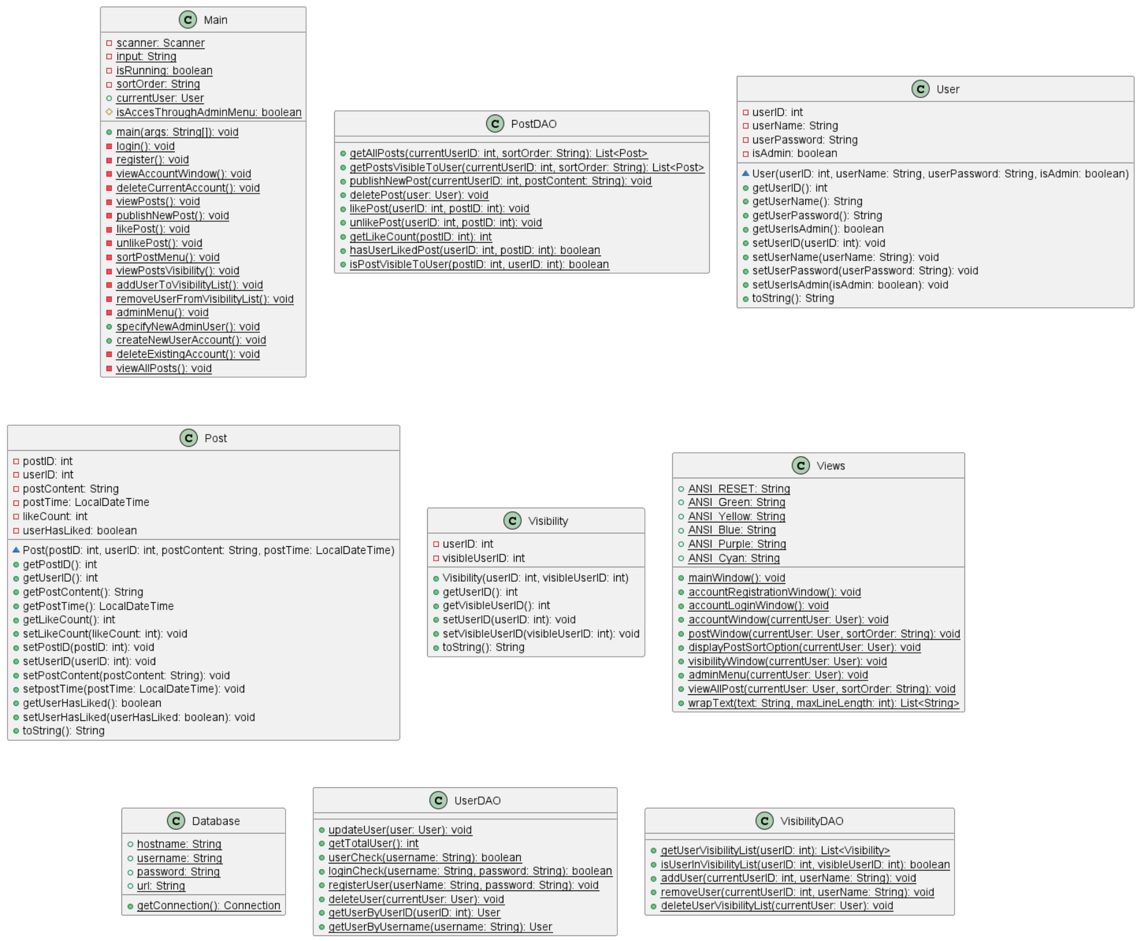
Attributes: postID (PK, FK), userID (PK, FK).

1NF: All attributes are atomic.

2NF: Composite primary key (postID, userID); no partial dependencies.

3NF: No transitive dependencies; attributes depend only on the composite primary key.

UML Diagram


Main
scanner: Scanner
input: String
isRunning: boolean
sortOrder: String
currentUser: User
isAccesThroughAdminMenu: boolean
main(args: String[]): void
login(): void
register(): void
viewAccountWindow(): void
deleteCurrentAccount(): void
viewPosts(): void
publishNewPost(): void
likePost(): void
unlikePost(): void
sortPostMenu(): void
viewPostsVisibility(): void
addUserToVisibilityList(): void
removeUserFromVisibilityList(): void
adminMenu(): void
specifyNewAdminUser(): void
createNewUserAccount(): void
deleteExistingAccount(): void
viewAllPosts(): void
PostDAO
getAllPosts(currentUserID: int, sortOrder: String): List<Post>
getPostsVisibleToUser(currentUserID: int, sortOrder: String): List<Post>
publishNewPost(currentUserID: int, postContent: String): void
deletePost(user: User): void
likePost(userID: int, postID: int): void
unlikePost(userID: int, postID: int): void
getLikeCount(postID: int): int
hasUserLikedPost(userID: int, postID: int): boolean
isPostVisibleToUser(postID: int, userID: int): boolean
User
userID: int
userName: String
userPassword: String
isAdmin: boolean
User(userID: int, userName: String, userPassword: String, isAdmin: boolean)
getUserID(): int
getUserName(): String
getUserPassword(): String
getUserIsAdmin(): boolean
setUserID(userID: int): void
setUserName(userName: String): void
setUserPassword(userPassword: String): void
setUserIsAdmin(isAdmin: boolean): void
toString(): String
Post
postID: int
userID: int
postContent: String
postTime: LocalDateTime
likeCount: int
userHasLiked: boolean
Post(postID: int, userID: int, postContent: String, postTime: LocalDateTime)
getPostID(): int
getUserID(): int
getPostContent(): String
getPostTime(): LocalDateTime
getLikeCount(): int
setLikeCount(likeCount: int): void
setPostID(postID: int): void
setUserID(userID: int): void
setPostContent(postContent: String): void
setpostTime(postTime: LocalDateTime): void
getUserHasLiked(): boolean
setUserHasLiked(userHasLiked: boolean): void
toString(): String
Visibility
userID: int
visibleUserID: int
Visibility(userID: int, visibleUserID: int)
getUserID(): int
getVisibleUserID(): int
setUserID(userID: int): void
setVisibleUserID(visibleUserID: int): void
toString(): String
Views
ANSI_RESET: String
ANSI_Green: String
ANSI_Yellow: String
ANSI_Blue: String
ANSI_Purple: String
ANSI_Cyan: String
mainWindow(): void
accountRegistrationWindow(): void
accountLoginWindow(): void
accountWindow(currentUser: User): void
postWindow(currentUser: User, sortOrder: String): void
displayPostSortOption(currentUser: User): void
visibilityWindow(currentUser: User): void
adminMenu(currentUser: User): void
viewAllPost(currentUser: User, sortOrder: String): void
wrapText(text: String, maxLineLength: int): List<String>
Database
hostname: String
username: String
password: String
url: String
getConnection(): Connection
UserDAO
updateUser(user: User): void
getTotalUser(): int
userCheck(username: String): boolean
loginCheck(username: String, password: String): boolean
registerUser(userName: String, password: String): void
deleteUser(currentUser: User): void
getUserByUserID(userID: int): User
getUserByUsername(username: String): User
VisibilityDAO
getUserVisibilityList(userID: int): List<Visibility>
isUserInVisibilityList(userID: int, visibleUserID: int): boolean
addUser(currentUserID: int, userName: String): void
removeUser(currentUserID: int, userName: String): void
deleteUserVisibilityList(currentUser: User): void

Mapping

User Table ↔ User Class

-Stores user details like userID, userName, userPassword, and isAdmin.

Posts Table ↔ Post Class

-Contains post information including postID, userID, postContent, and postTime.

UserVisibility Table ↔ Visibility Class

-Manages which users can see whose posts.

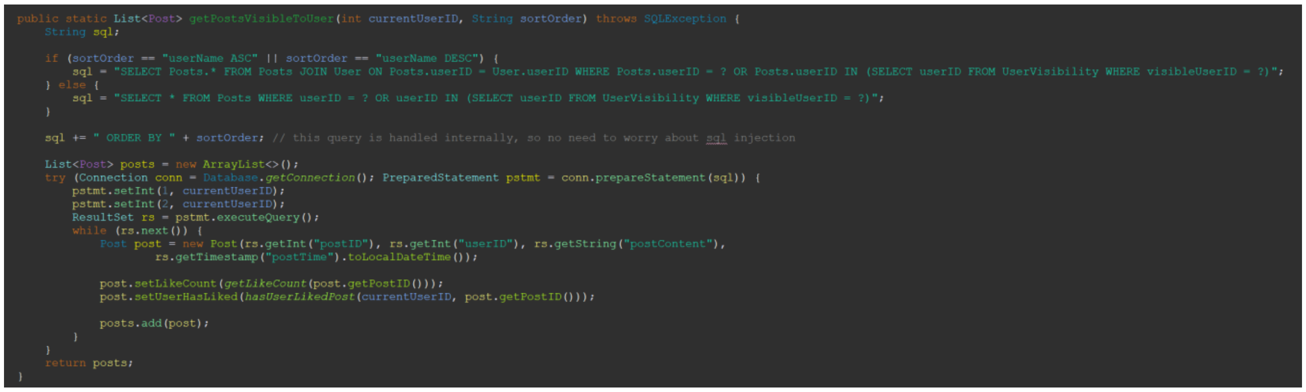
PostLikes Table ↔ Managed by PostDAO

-Handles liking and unliking posts, and tracks like counts.

Data Access Objects (DAOs) like **UserDAO**, **PostDAO**, and **VisibilityDAO** are responsible for CRUD operations and database interactions for their respective entities.

Post Window

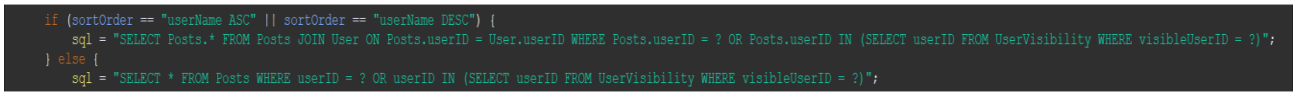
To display all posts visible to the current user, we uses the ***getPostsVisibleToUser()*** method in the **PostDAO class**. This method constructs a SQL query that retrieves posts based on the current user's visibility settings and the desired sort order.



```java
public static List<Post> getPostsVisibleToUser(int currentUserID, String sortOrder) throws SQLException {
    String sql;

    if (sortOrder == "userName ASC" || sortOrder == "userName DESC") {
        sql = "SELECT Posts.* FROM Posts JOIN User ON Posts.userID = User.userID WHERE Posts.userID = ? OR Posts.userID IN (SELECT userID FROM UserVisibility WHERE visibleUserID = ?)";
    } else {
        sql = "SELECT * FROM Posts WHERE userID = ? OR userID IN (SELECT userID FROM UserVisibility WHERE visibleUserID = ?)";
    }

    sql += " ORDER BY " + sortOrder; // this query is handled internally, so no need to worry about sql injection

    List<Post> posts = new ArrayList<>();
    try (Connection conn = Database.getConnection(); PreparedStatement pstmt = conn.prepareStatement(sql)) {
        pstmt.setInt(1, currentUserID);
        pstmt.setInt(2, currentUserID);
        ResultSet rs = pstmt.executeQuery();
        while (rs.next()) {
            Post post = new Post(rs.getInt("postID"), rs.getInt("userID"), rs.getString("postContent"),
                    rs.getTimestamp("postTime").toLocalDateTime());

            post.setLikeCount(getLikeCount(post.getPostID()));
            post.setUserHasLiked(hasUserLikedPost(currentUserID, post.getPostID()));

            posts.add(post);
        }
    }
    return posts;
}
```

The full method.

Firstly, we start by creating an SQL query that selects posts either made by the current user or by users who are on the current user's visibility list.



```java
    if (sortOrder == "userName ASC" || sortOrder == "userName DESC") {
        sql = "SELECT Posts.* FROM Posts JOIN User ON Posts.userID = User.userID WHERE Posts.userID = ? OR Posts.userID IN (SELECT userID FROM UserVisibility WHERE visibleUserID = ?)";
    } else {
        sql = "SELECT * FROM Posts WHERE userID = ? OR userID IN (SELECT userID FROM UserVisibility WHERE visibleUserID = ?)";
```

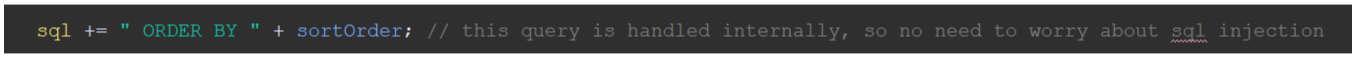
If the posts need to be sorted by the username (either ascending or descending), we include a JOIN with the User table

SELECT Posts.* FROM Posts JOIN User ON Posts.userID = User.userID WHERE Posts.userID = ? OR Posts.userID IN (SELECT userID FROM UserVisibility WHERE visibleUserID = ?)

If the sorting is by other criteria, we use a simpler query.

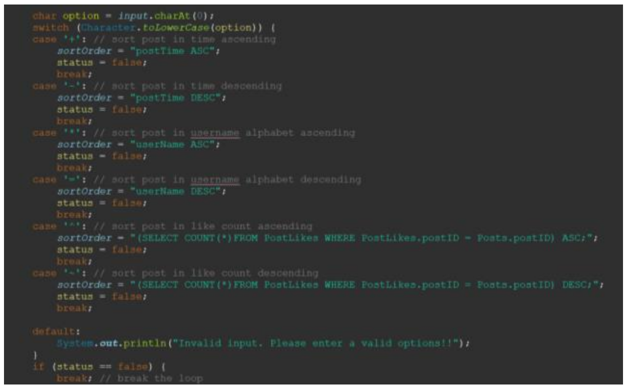
SELECT * FROM Posts WHERE userID = ? OR userID IN (SELECT userID FROM UserVisibility WHERE visibleUserID = ?)

Then we append the desired sort order to the SQL query later. This allows the user to sort the posts by time, username, or likes.



```java
sql += " ORDER BY " + sortOrder; // this query is handled internally, so no need to worry about sql injection
```

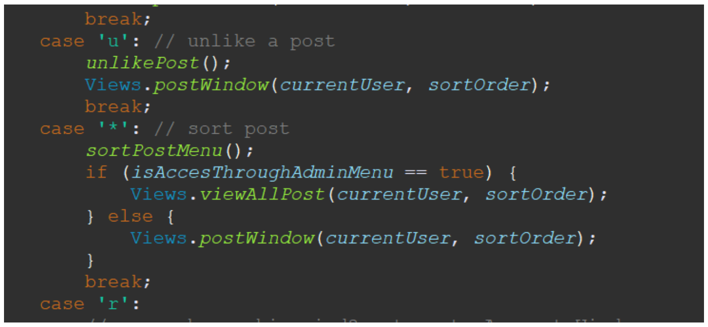
sortOrder is set by ***sortPostMenu()*** method from Main class,



```
char option = input.charAt(0);
switch (Character.toLowerCase(option)) {
case '+': // sort post in time ascending
    sortOrder = "postTime ASC";
    status = false;
    break;
case '-': // sort post in time descending
    sortOrder = "postTime DESC";
    status = false;
    break;
case '*': // sort post in username alphabet ascending
    sortOrder = "userName ASC";
    status = false;
    break;
case '=': // sort post in username alphabet descending
    sortOrder = "userName DESC";
    status = false;
    break;
case '^': // sort post in like count ascending
    sortOrder = "(SELECT COUNT(*)FROM PostLikes WHERE PostLikes.postID = Posts.postID) ASC;";
    status = false;
    break;
case '~': // sort post in like count descending
    sortOrder = "(SELECT COUNT(*)FROM PostLikes WHERE PostLikes.postID = Posts.postID) DESC;";
    status = false;
    break;

default:
    System.out.println("Invalid input. Please enter a valid options!!");
}
if (status == false) {
    break; // break the loop
```

Part of the code from SortPostMenu() in Main class

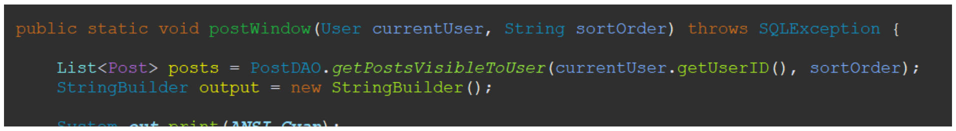
then pass over through ***ViewPost()*** method in main class



```
        break;
    case 'u': // unlike a post
        unlikePost();
        Views.postWindow(currentUser, sortOrder);
        break;
    case '*': // sort post
        sortPostMenu();
        if (isAccesThroughAdminMenu == true) {
            Views.viewAllPost(currentUser, sortOrder);
        } else {
            Views.postWindow(currentUser, sortOrder);
        }
        break;
    case 'r':
```

Part of the code from ViewPost() in Main class

ViewPost() method will then call ***View.postWindow(currentUser, sortOrder)*** passing the sortOrder then we finally call ***getPostsVisibleToUser()*** method to append the select statement

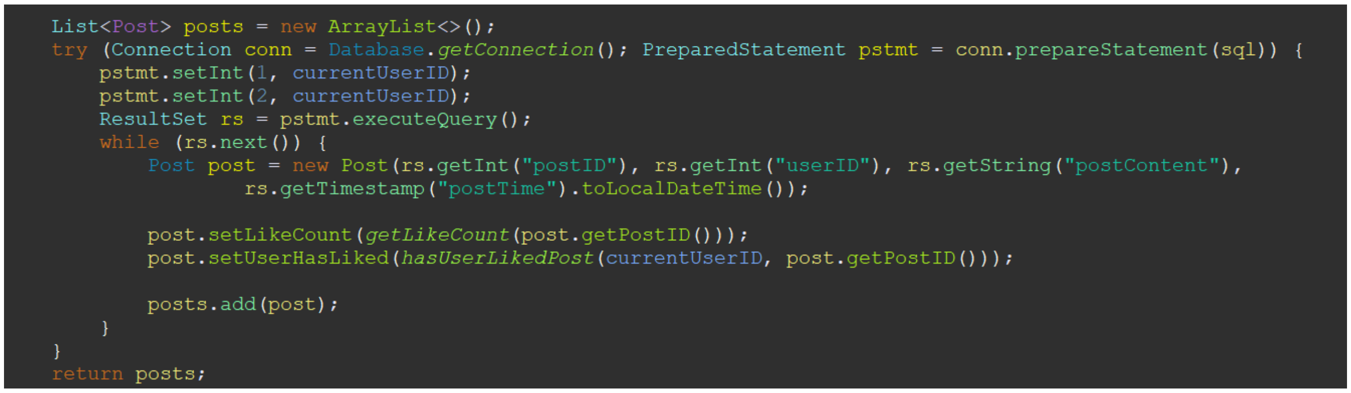


```
public static void postWindow(User currentUser, String sortOrder) throws SQLException {

    List<Post> posts = PostDAO.getPostsVisibleToUser(currentUser.getUserID(), sortOrder);
    StringBuilder output = new StringBuilder();
```

Part of the code from postWindow() in View class

We then use a prepared statement to execute the query. We replace the placeholders (?) with the current user's ID. This ensures that we're only fetching posts relevant to the user.

As we retrieve each post from the database, we create a Post object. We also get additional information like the number of likes on the post and whether the current user has liked it. Finally, we collect all the Post objects into a list and return them. This list is then used to display the posts in the user's post window.



```
    List<Post> posts = new ArrayList<>();
    try (Connection conn = Database.getConnection(); PreparedStatement pstmt = conn.prepareStatement(sql)) {
        pstmt.setInt(1, currentUserID);
        pstmt.setInt(2, currentUserID);
        ResultSet rs = pstmt.executeQuery();
        while (rs.next()) {
            Post post = new Post(rs.getInt("postID"), rs.getInt("userID"), rs.getString("postContent"),
                    rs.getTimestamp("postTime").toLocalDateTime());

            post.setLikeCount(getLikeCount(post.getPostID()));
            post.setUserHasLiked(hasUserLikedPost(currentUserID, post.getPostID()));

            posts.add(post);
        }
    }
    return posts;
```

By using this method, we can retrieve all the posts that the current user is allowed to see, based on their visibility settings. We use a single SQL query to get all the necessary data, which makes the process faster and reduces the load on the database.

Testing

Initialize database with *dbinit.sql* script

(use campus wifi for better performance, different network have GREAT DELAY when retrieving data)

Main Window: Check if the user count is displayed correctly by querying the database.

Account Login: Test both valid and invalid login attempts.

Post Window:

-Check if posts is correctly visible to the user.

-Adding new posts and checking the database for updates.

-Sorting posts by time, username, and likes.

-Like and unlike post

Visibility Window:

-Add and remove users from the visibility list.

-Check if post visibility table is correctly update on database server.

Admin Test: specify new admin, creating/deleting accounts and viewing all posts.

Bonus

1. **Create New Accounts:** user can register account in Main Menu page
2. **Delete Accounts:** user can delete it's own account in Account Window
3. **Sorting Posts:** can be access with ' * ' in Post Window
4. **Like and Unlike Posts:** can be access with ' L ' or ' U ' in Post Window
5. **Admin Accounts**: David is the admin user, when logging in there will be a special menu option called "Admin Menu" appear on the account window. By choosing it you will have access to admin features.