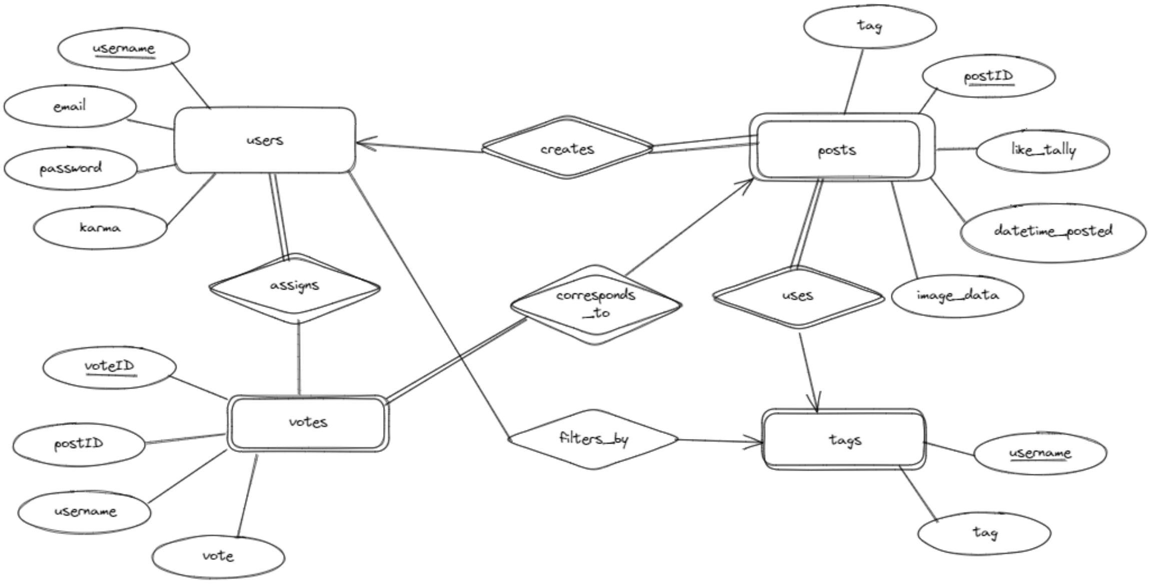
Final Project Report

Alex Kwong, Kurt Goerg, Darrion Chandler, Matthew Pezolt

1- **Database Design**

1a. ER-Diagram



1b. Schema Statements

- users(username, email, password, karma)
- tags(username, tag)
- votes(voteID, postID, username, vote)
- post(post_id, like_tally, datetime_posted, tag, image_data)
- creates(username, post_id)
- corresponds_to(vote_id, post_id)
- filters_by(username)
- assigns(username, vote_id)
- uses(post_id, username)

2- **Database Programming**

2a - We are hosting our database locally. In the database setup process we used phpMyAdmin to create the tables as it is more intuitive and allowed us to ensure that the database and tables were set up exactly how we wanted.

2b - We are hosting the app locally using XAMPP. The database is then connected to the app and both are run locally.

2c - Instructions for Deployment

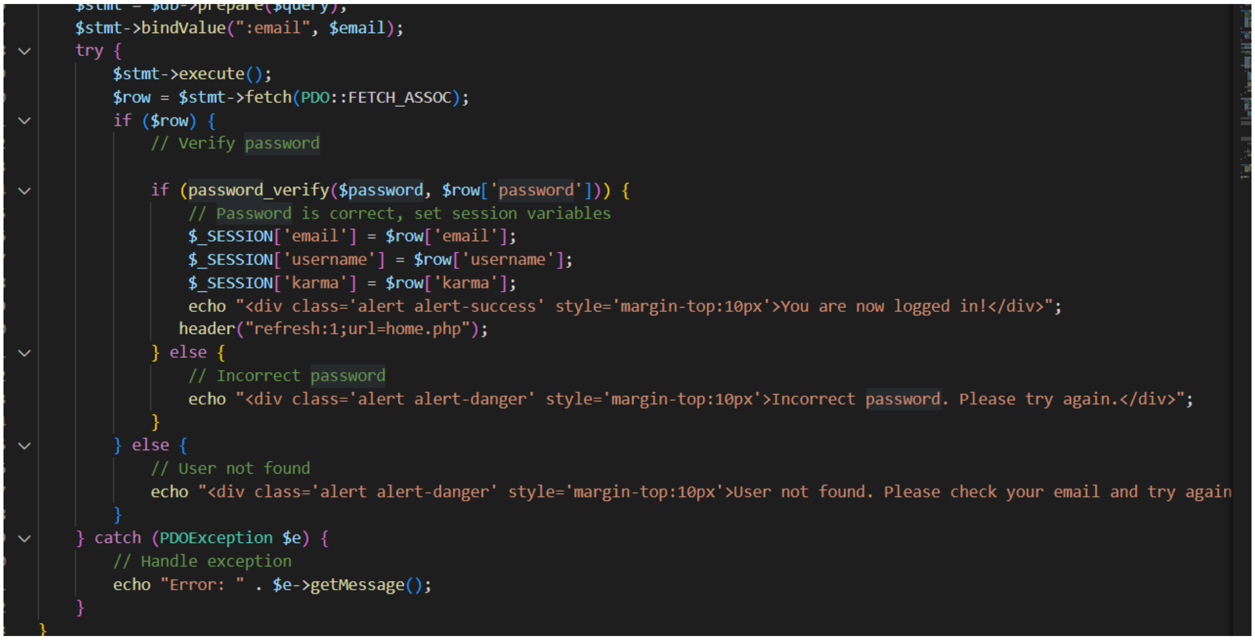
- The instructions for deployment are the standard instructions for hosting/running through XAMPP normally. Which were provided to us at the following links
    - Deploy a PHP app: https://www.cs.virginia.edu/~up3f/cs4750/supplement/php-deployment-xampp.html
    - XAMPP Setup: https://www.cs.virginia.edu/~up3f/cs4750/supplement/XAMPP-setup.html
    - The database is generated automatically by the php files

2d - Advanced SQL Commands

- The advanced SQL command we incorporated into our app uses a trigger. Essentially, for our application, we wanted it to work like the popular social media app “YikYak” where once a post reaches a certain number of dislikes it is removed from the database. This applies to both full posts and comments. For this we set a trigger that activates once a post/comment reaches a like_tally of -5. If the post in question is a comment, that comment is then “deleted” dropped from the DB. If it is a post, we delete both the posts and any/all comments corresponding to that post.

3- Database Security

At the database level our project is slightly different than most other projects by its nature. Essentially, because of the fact that our dataset is user generated (through posts and comments) everyone that uses the application has the same privileges. The database security that we implement is a login feature. This essentially requires everyone that wants to use our database (wants to upload and download pictures/leave comments) has to either log into an existing account (which is verified by checking against the account information in the database in the screenshot below), or they must create an account and have their account information added into the table of users within the database. From there all users have the ability to insert data through creating posts or writing comments (the later uses the Check Constraint described above to ensure that the comment text is not empty), update data (upvote/downvote a post) which can cause the tigger described above to occur depending on the current like tally of a post/comment.

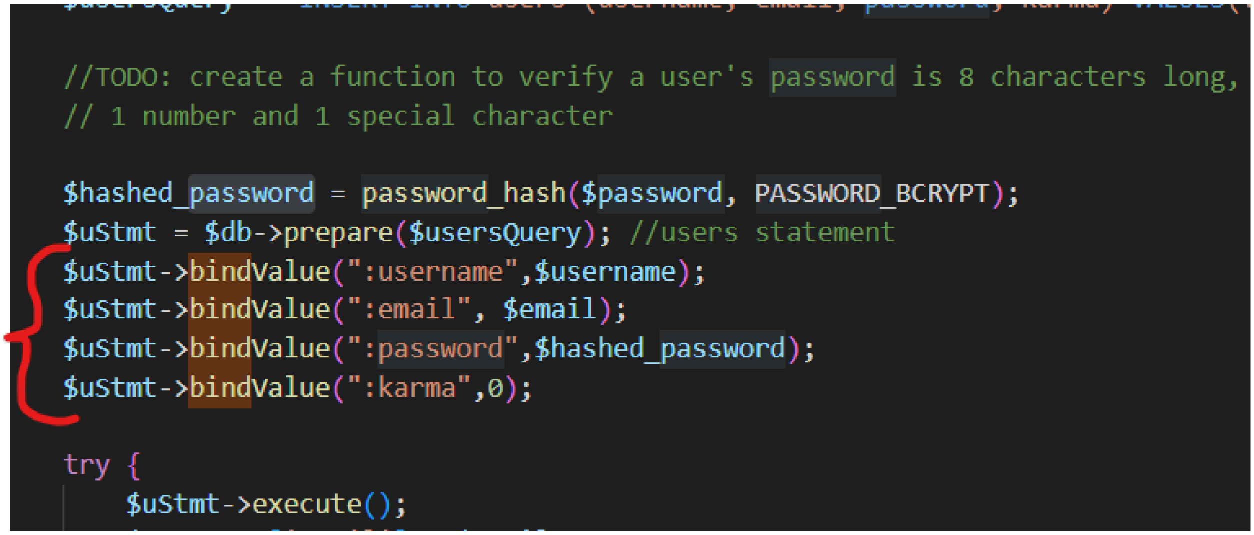


```php
    $stmt = $db->prepare($query);
    $stmt->bindValue(":email", $email);
    try {
        $stmt->execute();
        $row = $stmt->fetch(PDO::FETCH_ASSOC);
        if ($row) {
            // Verify password

            if (password_verify($password, $row['password'])) {
                // Password is correct, set session variables
                $_SESSION['email'] = $row['email'];
                $_SESSION['username'] = $row['username'];
                $_SESSION['karma'] = $row['karma'];
                echo "<div class='alert alert-success' style='margin-top:10px'>You are now logged in!</div>";
               header("refresh:1;url=home.php");
            } else {
                // Incorrect password
                echo "<div class='alert alert-danger' style='margin-top:10px'>Incorrect password. Please try again.</div>";
            }
        } else {
            // User not found
            echo "<div class='alert alert-danger' style='margin-top:10px'>User not found. Please check your email and try again
        }
    } catch (PDOException $e) {
        // Handle exception
        echo "Error: " . $e->getMessage();
    }
}
```

4- Application Level Security

At the application level we have two aspects of security in place. Firstly, we are using prepared SQL statements while also binding user input data to ensure that any inputs are not running as scripts. This helps guard against SQL injection attacks and is able to keep sensitive information safe that way.



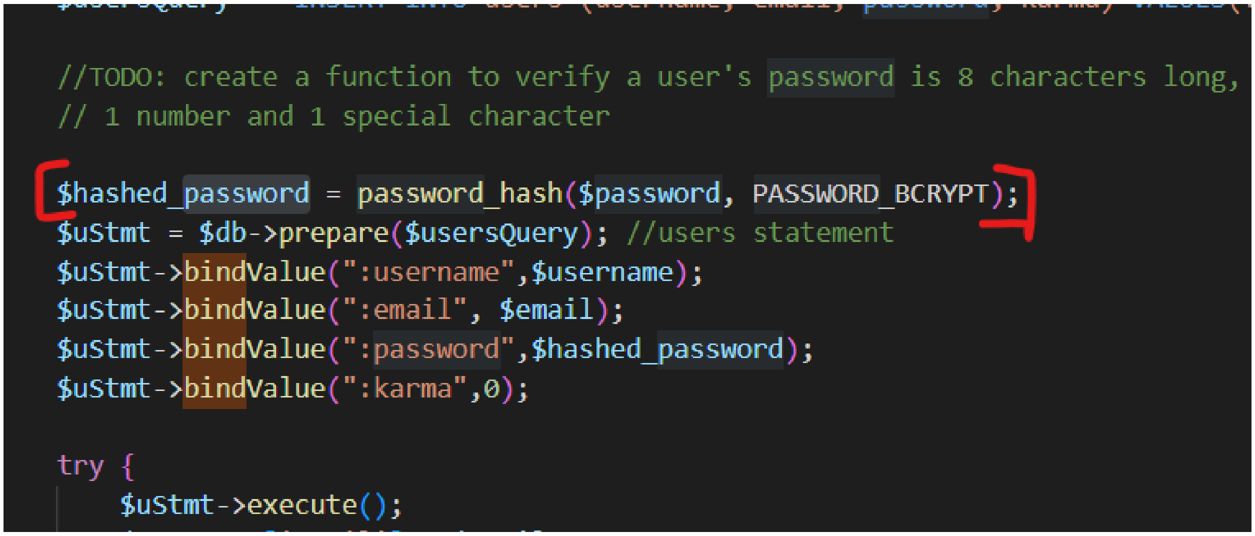
```php
$usersQuery = "INSERT INTO users (username, email, password, karma) VALUES(

//TODO: create a function to verify a user's password is 8 characters long,
// 1 number and 1 special character

$hashed_password = password_hash($password, PASSWORD_BCRYPT);
$uStmt = $db->prepare($usersQuery); //users statement
$uStmt->bindValue(":username",$username);
$uStmt->bindValue(":email", $email);
$uStmt->bindValue(":password",$hashed_password);
$uStmt->bindValue(":karma",0);

try {
    $uStmt->execute();
```

Secondly, we are protecting information at the application level by encrypting (or hashing) passwords. This helps keep user passwords safe in the event of a hacking/attack.



```
$usersQuery = "INSERT into users (username, email, password, karma) VALUES(

//TODO: create a function to verify a user's password is 8 characters long,
// 1 number and 1 special character

$hashed_password = password_hash($password, PASSWORD_BCRYPT);
$uStmt = $db->prepare($usersQuery); //users statement
$uStmt->bindValue(":username",$username);
$uStmt->bindValue(":email", $email);
$uStmt->bindValue(":password",$hashed_password);
$uStmt->bindValue(":karma",0);

try {
    $uStmt->execute();
```