qHub Project

Ungur Paul Alexandru

1. Introduction

Welcome to qHub, the desktop application that rewards your knowledge! I am excited to introduce you to qHub, an application designed to challenge your understanding of the world around you.

Answering true or false questions about a range of subjects, such as history, physics, and popular culture, will earn you points on qHub. I think learning should be enjoyable and fulfilling, which is why I developed qHub. qHub is the ideal app for you if you enjoy trivia or are just trying to learn more.

qHub is a great tool for competition as well as a fun method to learn new things. You will gain points to move up the scoreboard for each question you successfully answer. Other methods of earning points include creating new questions with rewards for the other players.

2. Requirements

The application mainly functions with the use of a database, so you'll need:

- Database server
- DBMS
- Database schema

3. Usage

You will first need to create an account in order to utilize qHub. Only by contacting an admin or receiving an account as an invitation can accounts be created. You will see a box with all of the questions once you have been added and logged in. Simply click on a question to start playing.

You will receive points for providing a right response to a question. You will not score any points if you choose the incorrect response to a question, though. Each question only allows you one chance to respond, so pick your response wisely!

Simply click the "Add Questions" button and complete all the necessary details to add a new question. The question can then be seen on the list if it passes the admin check.

As an admin you'll have extra privileges, such as the power to add users. You will be in charge of moderating questions as an admin to make sure they are appropriate for the app. To add an user it's similar to adding a question.

4. Architecture

The application uses the Model-View-Controller (MVC) design pattern to separate the application logic into three interconnected components: the Model, View, and Controller. qHub also uses Java Database Connectivity (JDBC) to interact with a database that stores user and question information.

Model:

The application's logic is represented by the Model component of qHub. To read and write data to the qHub database, it uses JDBC to interact with the database. The methods for determining rankings,

points, and question moderation are all part of the Model component, along with classes for user and question data.

View:

The user interface of the program is represented by the View component of qHub. In order to show the proper data and take user input, it interfaces with the Controller. The login and registration screens, the question display and submission screen, and the leaderboard screen are all included in the View component.

Controller:

Between the Model and View, the Controller component of qHub serves as a bridge. It takes user input, analyzes it, and updates the Model and View as necessary. In order to carry out tasks like adding and deleting questions and users, calculating rankings and points, and moderating questions, the Controller component interacts with the Model.

JDBC:

qHub uses it to establish a connection to the database and to read and write data to it. The database schema includes tables for user information, question information, scores and interactions.

5. Design

The qHub application's main class first loads the FXML file for the StartController, which is the controller for the StartView file. Once the controller retrieves the necessary data from the view, it sends it to the SessionManager. The SessionManager creates an instance of the UserDAO, which is responsible for checking the data against the

database. The UserDAO handles different needs and is implemented using the singleton design pattern.

If the data is validated successfully, the MainController or MainAdminController and its corresponding FXML file are loaded. After loading, the QuestionsManager is initialized using dependency injection to decouple the components of the application. The QuestionsManager modifies the data from the database for use in the controller, and the QuestionsDAO performs much of the logic behind how the system works.

For the question response interface, a dialogue box was chosen instead of a new FXML file. This allows for more streamlined interaction and better user experience. The logout button nullifies the instance of the current user and loads the StartView. The ranking button opens the RankingView with its controller, which uses the RanksManager as a bridge between the RanksDAO and the controller, following the same design pattern as the QuestionsManager.

Finally, the Add User and Add Question options open their respective controllers and views. These forms collect information from either the admin or the user, which is verified in the controller and then sent to the QuestionManager/SessionManager. The information is then processed by the DAO files for server logic and any details that need to be verified with the database.

Overall, the application uses the Model-View-Controller (MVC) architecture and the Java Database Connectivity (JDBC) API to provide a robust and efficient experience for users. The decoupling of components through dependency injection and the use of design patterns such as the singleton and bridge patterns make the application more maintainable and extensible.

6. Code

```
1 usage  PaulUngur2
private SessionManager() {
}

7 usages  PaulUngur2
public static SessionManager getInstance() {
    if (instance == null) {
        instance = new SessionManager();
    }
    return instance;
}

1 usage  PaulUngur2
public User login(String username, String password) throws SQLException {
    UserDAO userDAO = new UserDAO();
    User user = userDAO.getUserByUsername(username);
    try {
        if (user != null && BCrypt.checkpw(password, user.getPassword())) {
            currentUser = user;
        }
    } catch (IllegalArgumentException e) {
        System.out.println("Error checking password: " + e.getMessage());
    }
    return currentUser;
}
```

The SessionManager class is a singleton class that manages the current user session. It has a private constructor and a static getInstance() method, which returns a single instance of the class. This ensures that there is only one instance of the SessionManager class throughout the application.

The login() method takes a username and password as input parameters and attempts to log the user in. It does this by first retrieving the user with the given username from the database using the UserDAO class. If a user is found, the method then checks whether the provided password matches the user's stored password using the BCrypt.checkpw() method. If the password is correct, the currentUser variable is set to the logged-in user and returned. If the password is incorrect, null is returned.

It's important to note that the password is safely checked using the BCrypt.checkpw() method. Instead of keeping the passwords in plain text, this method compares the supplied password to the hashed password kept in the database. Even in the case that the database is compromised, this aids in preventing illegal access to user accounts.

```
private void serverLogic(int idUser, boolean status) throws SQLException {
    PreparedStatement statement = connection.prepareStatement( sql: "SELECT badgesPoints_qHub, badges_qHub, tokens_qHub, username_qHub FROM qHub_accounts WHERE id_qHub = ?");
    statement.setInt( parameterIndex: 1, idUser);
    ResultSet resultSet = statement.executeQuery();
    if (resultSet.next()) {
        int badgesPoints = resultSet.getInt( columnLabel: "badgesPoints_qHub");
        int badges = resultSet.getInt( columnLabel: "badges_qHub");
        int tokens = resultSet.getInt( columnLabel: "tokens_qHub");
        String username = resultSet.getString( columnLabel: "username_qHub");
        if (status) {
            badgesPoints += 1;
            updateScore(username, tokens);
        }

        if (badgesPoints >= 10) {
            int badgesEarned = badgesPoints / 10;
            badges += badgesEarned;
            badgesPoints %= 10;
            updateScore(username,  score: tokens + (badges * 500));
        }

        PreparedStatement statement2 = connection.prepareStatement( sql: "UPDATE qHub_accounts SET badgesPoints_qHub = ?, badges_qHub = ? WHERE id_qHub = ?");
        statement2.setInt( parameterIndex: 1, badgesPoints);
        statement2.setInt( parameterIndex: 2, badges);
        statement2.setInt( parameterIndex: 3, idUser);
        statement2.executeUpdate();
    }
}

3 usages  PaulUngur2
private void updateScore(String nameUser, int score) throws SQLException {
    PreparedStatement statement = connection.prepareStatement( sql: "UPDATE qHub_rankings SET ranking_score = ? WHERE ranking_name = ?");
    statement.setInt( parameterIndex: 1, score);
    statement.setString( parameterIndex: 2, nameUser);
    statement.executeUpdate();
```

idUser and a status boolean value are input arguments for the serverLogic() private function. It is in charge of managing the ranking system's logic for the qHub application. It initially does a SQL query on the database to retrieve the user's account data, including username, badges, points, and badges.

The method runs the updateScore() method and increases the badgesPoints variable by 1 if the status boolean value is true,

indicating that the user properly responded to a question. This updates the user's score in the rankings.

The procedure counts the badges acquired by dividing badgesPoints by 10 and adding the result to the badges variable if the badgesPoints variable is greater than or equal to 10. After that, it multiplies the quantity of badges by 500 to determine how the score has changed, and it adds that value to the tokens variable. It then makes another call to the updateScore() method to add the new token value and update the user's score in the rankings.

The serverLogic() method then uses a SQL UPDATE statement to update the user's account information in the database with the new badgesPoints and badges values.

7. Technologies and Tools

I utilized a variety of tools and technologies to create a robust and efficient application. I used Java as the main programming language and relied on JDBC to establish a connection with the MariaDB database. Where I used DataGrip to view the data in an easier way.

To ensure clean code and easy maintenance, I implemented the Model-View-Controller (MVC) design pattern, which helped to keep the application organized and modular. I also used Git for version control.

For the front-end of the application, I used JavaFX and Scene Builder to create a user-friendly interface with a modern design. I developed the application in Intellij, a powerful IDE that allowed me to streamline the development process and increase productivity.

Additionally, I utilized Apache, a web server that provided a robust and reliable platform for the application to run on. Overall, these tools and technologies were instrumental in creating a successful and efficient qHub application.