CSE 360 Help System - Phase 1

Team Members

Dhruv Bhatt- Team Leader
Mansi Bhanushali
Vishwesh Karthikeyan
Amir Boshra
Nick Trinidad

Course Information

CSE 360 - Introduction to Software Engineering

Professor Name: Lynn Robert Carter

Date: 9th October, 2024

Project Overview

Purpose

The CSE 360 Help System is designed to give ASU students participating in CSE 360 a useful, individualized, and efficient tool to help them comprehend and apply software engineering concepts. The system aims to assist students with different programming skill levels by providing them with pertinent, precise, and focused information without being overbearing. The technology also makes it easier to manage personal information securely and privately. It guarantees that students have a reliable way to ask for assistance if they cannot locate the information they need. Additionally, it will enable the teaching staff to effortlessly maintain, update, and improve resources in response to student input and changing course materials.

Scope

The aid system will support administrators, teachers, and students in a variety of capacities. Features like role-based access control, secure personal data management, content search and retrieval, user authentication, and support for feedback loops to enhance content over time are all part of its scope. Additionally, the system will include Ed Discussion data to deliver pertinent, real-time responses to frequently asked student queries. Phase 1 of the system's development will concentrate on creating a safe user identity system, which will include managing roles, processing passwords, and registering users. The system must also allow future extensions, like introducing additional user roles and incorporating more complex content management tools.

Development Phase

There will be four main phases to the development process:

- Phase 1: Defines the core functionalities, such as password processing, user roles (Admin, Student, Instructor), and secure identity management. Setting up safe account creation, role assignment, and login features is the main priority.

- Phase 2: Enhances the help system by adding support for managing help articles, including creating, updating, deleting, and searching for articles. These tools will be simple for educators and administrators to administer.

- Phase 3: Ensures that sensitive and private data is protected and works on improving the user experience for students. There will be more testing done and security improvements made.

- Phase 4: Completes the system by making sure that the architecture, code, and requirements all work together seamlessly. The system will undergo comprehensive testing and documentation, and input from earlier stages will be integrated.

Tech Stack

The following technologies will be utilized in the development of the help system:

- Java: The primary programming language for backend features.
- JavaFX: For creating user-interactive graphical user interfaces (GUIs).
- GitHub: To facilitate teamwork and version control.
- JUnit: Used to test individual software modules to make sure they are reliable and functional.
- Security Features: Implementing password encryption, private data management, and secure user authentication.

Requirements

Functional Requirements

The following features need to be included in the system:

1. Management of User Accounts:
    - Users (Administrators, Students, and Instructors) must be able to create accounts on the system using distinct usernames and passwords.
    - In order to complete the account setup process, users must be able to log in and enter their full name, including their preferred first name, along with their email address.
    - During the account creation process, the system needs to enable administrators to give roles to users.
    - Admins need to be able to modify user roles, reset passwords, remove accounts, and invite people with a one-time invitation code.
    - Users must always be able to log out of the system.
    - Multiple roles per user must be supported by the system. When a user has many roles, they ought to be able to choose the role for that session.

2. Handling Passwords:

- When creating an account, passwords need to be entered twice to be accurate.
- One-time passwords for account resets must be supported by the system. There should be an expiration date and time for the one-time password.
- When using a one-time password to log in, the system needs to prompt users to create a new one.

3. Support for Content Management:
    - Help articles must be able to be created, edited, updated, deleted, and restored by administrators and instructors.
    - For ease of finding, every assistance article needs to have a title, a description, a body, and a list of keywords.
    - For ease of management, articles need to be organized according to their level (beginning, intermediate, advanced, and expert).
    - Students should be able to use keywords or phrases to look for help articles through the system.

4. Mechanism of Feedback:
    - When students are unable to locate relevant material, the system needs to give them a mechanism to send feedback to the instructional staff.
    - It must be possible for students to ask for assistance articles that are more in-depth or easier to understand.

5. Privacy and Security:
    - Sensitive data, such as passwords and private user information, must be encrypted by the system.
    - In order to prevent other users, including administrators, from accessing personal data, the system needs to implement privacy rules.

Non-functional Requirements

The following are the limitations and system-wide requirements that the project needs to follow:

1. Performance:
    - User input should be processed by the system in two seconds, with a maximum response time of five seconds during periods of high demand.
    - Retrieving content should not take longer than three seconds.

2. Usability:
    - Students with poor technical abilities should be able to navigate the system and access the necessary help materials with little to no training.

- A how-to screencast must be supplied for each user group: students, admins, and instructional team members.

3. Reliability:
    - A 99.5% uptime rate is required for the system, accounting for both planned and unforeseen downtime.
    - A minimum of six months should be the mean time to failure (MTTF).

4. Scalability:
    - Up to 500 concurrent users should be supported by the system without noticeably degrading performance.

5. Mobility:
    - The system must be able to run on many platforms, including Windows, macOS, and Linux, without change.

6. Safety:
    - Industry-standard encryption standards must be followed when it comes to user authentication and password storage.
    - Common online vulnerabilities like SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF) must be guarded against by the system.

7. Data Accuracy:
    - It is imperative to conduct routine backups of user accounts and help articles to guarantee that no data is lost in the event of a system breakdown.

User Stories

The following user stories are based on the necessary functionality:

1. Creation of User Accounts:
    - In order to have access to the help system, I would want to register as a new user and create an account with a special username and password.
    - Acceptance Standards: The user's name and email address must be entered correctly. The user must be able to select a username, choose a password, and double-enter the password to verify it on the system.

2. Choose Your Role:
    - I want to be able to choose my role when logging in so that I can use features that are unique to my role as a multi-role user.

- Acceptance Criteria: The user sees the possible roles after logging in. The user is taken to the appropriate homepage after selecting a role.

3. Reset Password:
    - I want to change a user's password as an administrator to a one-time password so they can change it the next time they log in.
    - Acceptance Standards: By creating a one-time password that expires after a predetermined amount of time, the administrator resets the user's password. The user needs to create a new password in order to access their account after using the one-time password to log in.

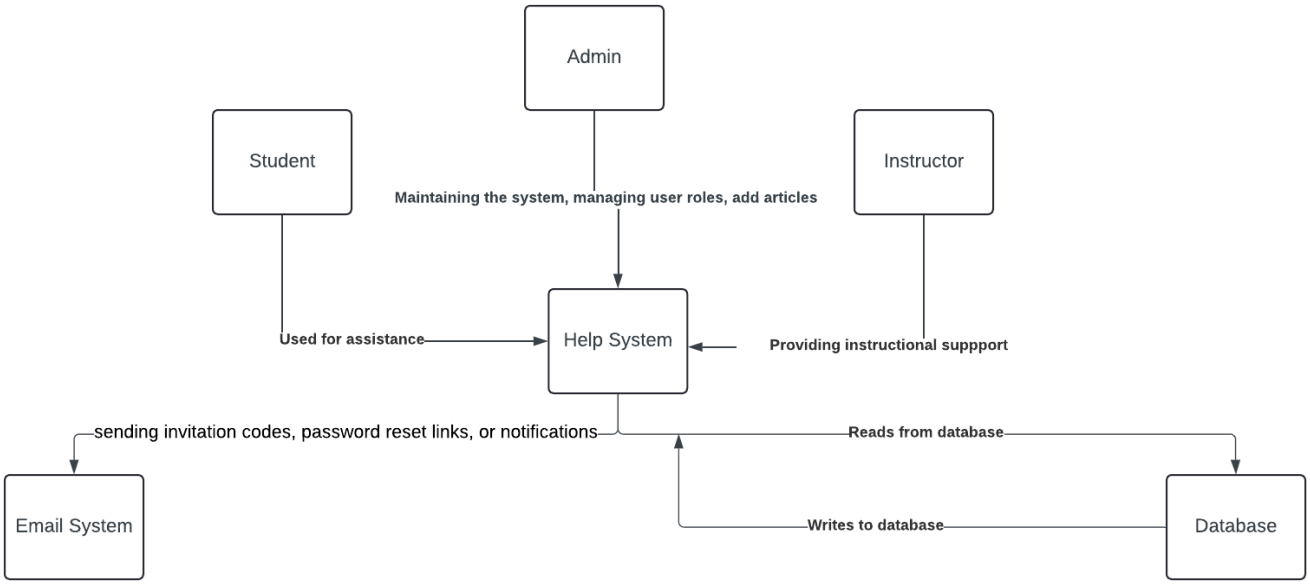
4. Supported Article Search:
    - In order to swiftly get the pertinent assistance I require, as a student, I wish to look for aid articles using particular keywords.
    - Acceptance Criteria: A list of articles that match the input keywords should appear in the search results, arranged according to relevancy.

5. Assist in Writing Articles:
    - In order to make the information easily accessible and useful to students, I want to, as an instructor, develop a new assistance article with pertinent content and keywords.
    - Acceptance Criteria: The article's title, summary, and body must be supplied by the instructor. They must be able to tag it with relevant keywords and classify it according to difficulty degree using the system.

6. Giving Remarks:
    - In order for the instructional team to enhance the support system, I as a student would want to provide feedback when I am unable to locate any helpful material.
    - Acceptance Criteria: The feedback submission form should be simple and quick to fill out, including the issue encountered and the results of the search.

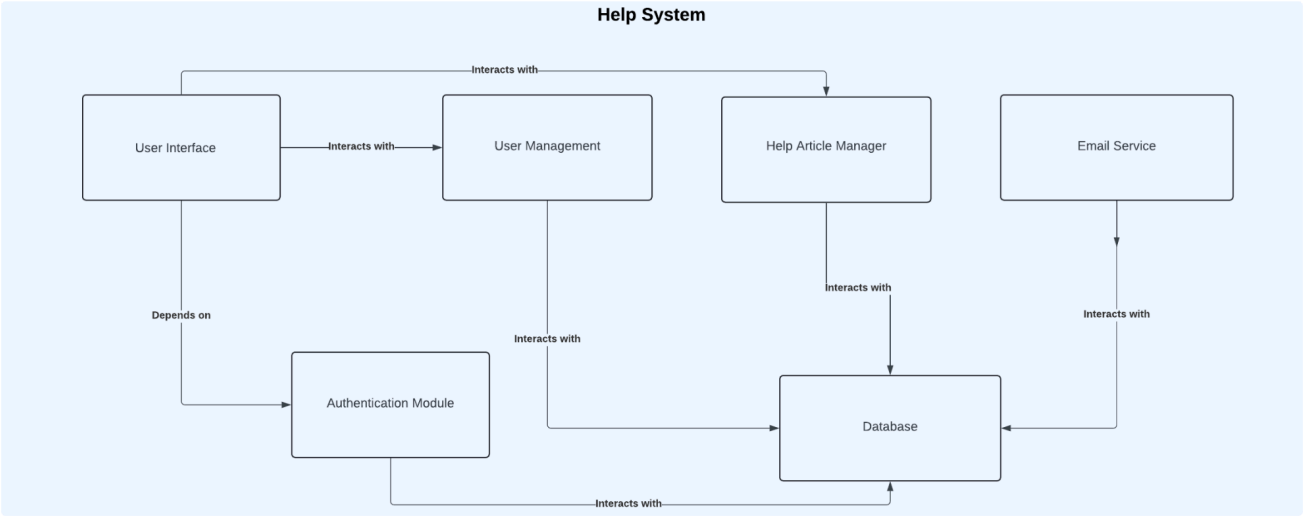
Architecture

Context Diagram



- Students: Use the help system to look for articles, choose subjects, ask for help, or submit comments.
- Admins: Manage users, change passwords, add and remove articles, control user roles and accounts, and other activities.
- Instructors: May also oversee student-related content or carry out administrative duties.
- Email System: When invitation codes or links to change passwords are sent, communication takes place.
- Database: Holds and retrieves all user information, session data, and assistance articles.
- The Help System serves as every user's primary point of contact.
- The main purposes of student and instructor interaction with the system are role selection and authentication.
- Phase 1 of the diagram shows how each type of user interacts with the system, with an emphasis on authentication and account management.

Major Components



The Help System's internal organization is illustrated in the Component Diagram, which consists of four primary parts:

- User Interface Component: This serves as the front-end, facilitating user interactions and presenting relevant screens based on user responsibilities and activities.
- Authentication Component: This component manages all aspects of user authentication, ensuring secure password management, handling user sessions, verifying user credentials, and granting access only to authorized users.
- User Management Component: This component empowers administrators to invite new users, reset passwords, and manage the creation, editing, and removal of user accounts. It also oversees user permissions and roles, and assigns roles to users.
- Data Storage Component: This acts as the central repository for all user data, ensuring efficient and secure storage of user, help article, and system configuration data.
- In addition to these primary components, the system also includes the Help Article Manager, responsible for creating, modifying, and deleting help articles, as well as an Email Service, which provides users with emails containing links to reset their passwords or notifications.

This architecture is designed to support scalability and modularity, making it easy to add and update features in the future, including the incorporation of help articles and strengthening security.

Relationships:

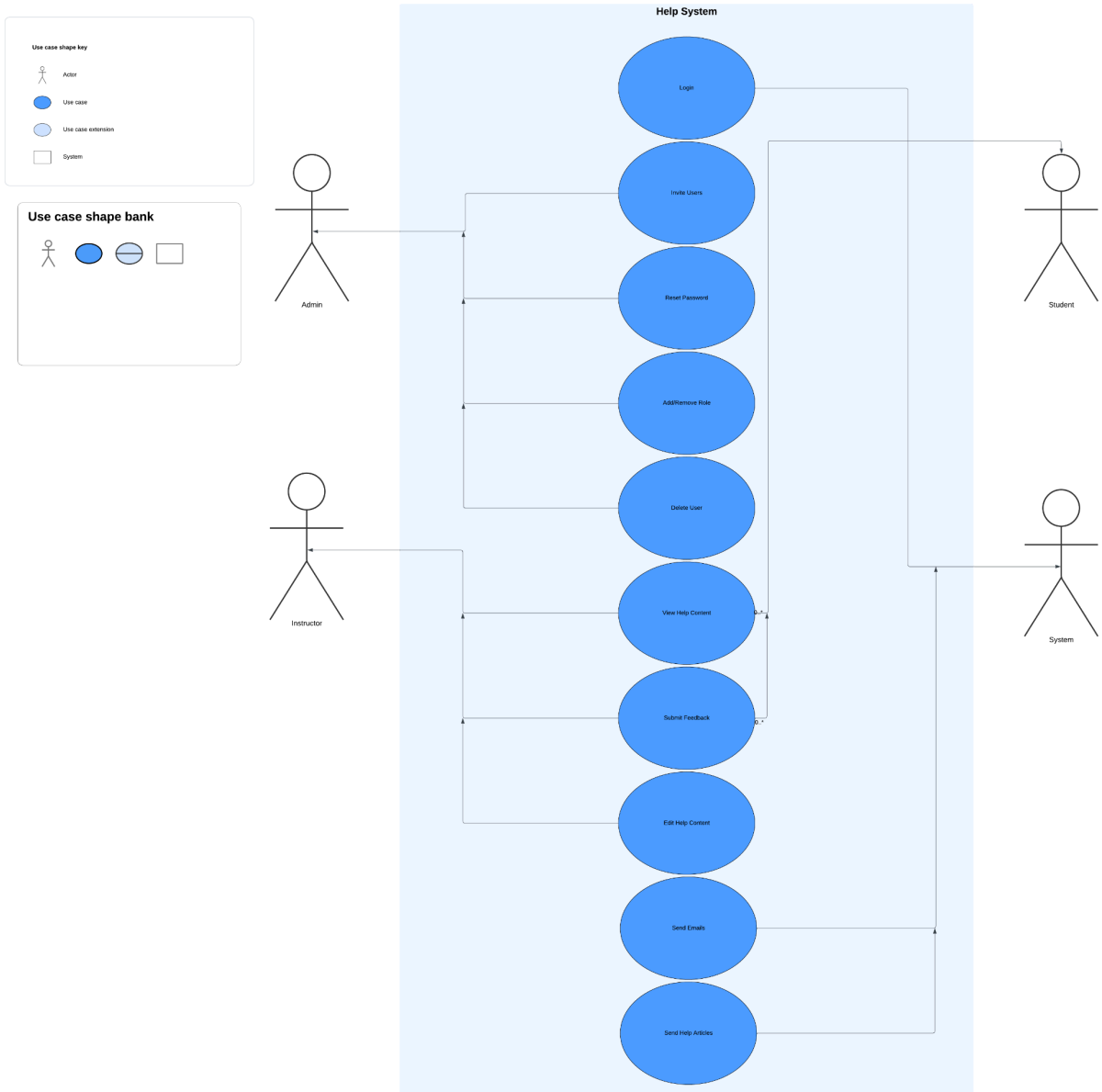
- The User Interface interacts with the following components:

    - Authentication Module: This component verifies user credentials before granting access.
    - User Management: This component displays user information and allows for account management.
    - Help Article Manager: This component displays and searches for help articles.
- The Authentication Module depends on the Database to store and retrieve user information.
- User Management interacts with the Database to store and retrieve user data.
- Help Article Manager interacts with the Database to store and retrieve help article data.
- The Email Service interacts with the Database to retrieve user email addresses for notifications.

Overall Functionality:
In this architecture, users interact with the user interface to access the system. The Authentication Module verifies their credentials and grants access. The User Management component allows administrators to manage user accounts and roles. The Help Article Manager provides access to help articles, which are stored in the database. The email service is used for sending notifications to users.

Design

UML Use Case Diagram



The Use Case Diagram illustrates the interactions between different system users (Admin, Student, Instructor). For instance:

- Admin: Invites new users, resets passwords, manages roles, and deletes users.
- Student and Instructor: Access help content, and provide feedback, and instructors can also edit help articles.

Class Responsibility Collaborator

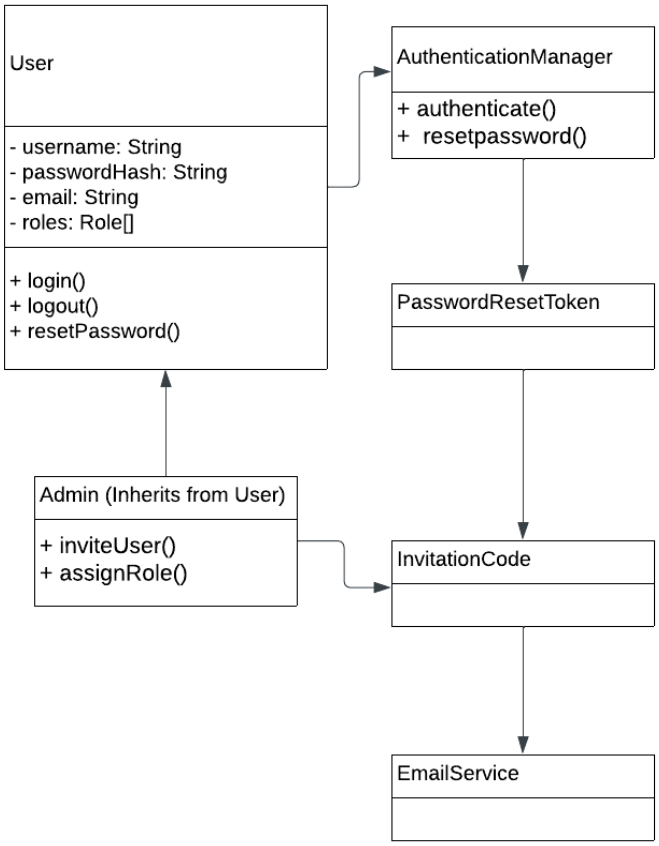
Class Name: Admin	
Description: The Admin class is a specific type of subclass within the User class. Apart from the standard user functions such as logging in, administrators can invite new users and allocate roles. Admins are tasked with overseeing the system's users and regulating access.	
Responsibilities	Collaborator
Manage user roles and permissions by inviting new users and assigning them roles.	**InvitationCode**: Generates and validates the codes sent to new users. **Role**: Manages the roles assigned to new users. **EmailService**: Sends invitation codes to users via email.
Ensure the system is populated with the right users who have the correct access levels.	**InvitationCode**: Generates and validates the codes sent to new users. **Role**: Manages the roles assigned to new users.

Class Name: User	
Description: The User class is a fundamental entity within the system, serving as the base for other user types like Admin, Student, and Instructor. It includes shared attributes and functionalities such as login details and standard actions like logging in and resetting passwords.	
Responsibilities	Collaborator
Store user credentials and roles.	**Role**: Defines what actions the user is allowed to perform based on their role.
Manage login/logout processes for all users.	**AuthenticationManager:** Authenticates the user during login and handles password resets.
Allow for password reset actions when required.	**PasswordResetToken:** Used to reset the user’s password.

Class Name: AuthenticationManager

Description: The AuthenticationManager class manages user authentication within the system. It works with the User class to facilitate user login and logout, as well as to oversee password reset capabilities.	
Responsibilities	Collaborator
Ensure that users can authenticate securely with their credentials.	**User**: Authenticates users based on their username and password.
Allow users to reset their passwords in case they forget their credentials.	**PasswordResetToken**: Handles the token sent to users to securely reset their password. **EmailService:** Sends out password reset emails to users.

Class Diagrams



In the context of object-oriented programming, the Admin class inherits from the User class. This is represented by a solid line with a hollow triangle. Additionally, the User class is linked

with the AuthenticationManager for activities such as login, logout, and password reset. The Admin class interacts with the InvitationCode and Role classes for user management. Furthermore, the User class utilizes the PasswordResetToken for resetting passwords. Both the Admin and User classes make use of the EmailService for sending invitation codes and password reset emails.

The diagram elements are as follows:

- User Class: This class includes common attributes (like username, email, etc.) and methods (such as login(), logout(), resetPassword()) that are applicable to all users.

- Admin Class: This class inherits from the User class and includes additional methods (like inviteUser() and assignRole()) that allow the Admin to manage other users.

- AuthenticationManager Class: This class is responsible for managing user authentication and password reset operations. It interacts with the User class to perform these functions.

- PasswordResetToken Class: This class is used by the User class in collaboration with the AuthenticationManager to facilitate password resets.

- InvitationCode Class: This class is used by the Admin class to generate invitation codes for inviting new users to the system.

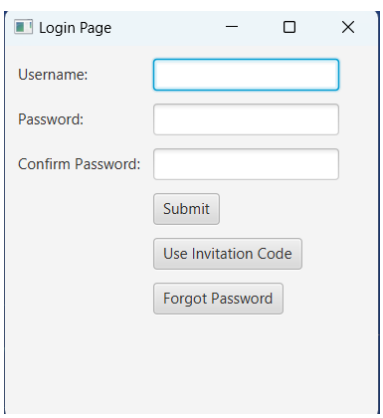
- EmailService Class: This class is used by both User and Admin for sending emails, such as those related to password resets or invitation code.

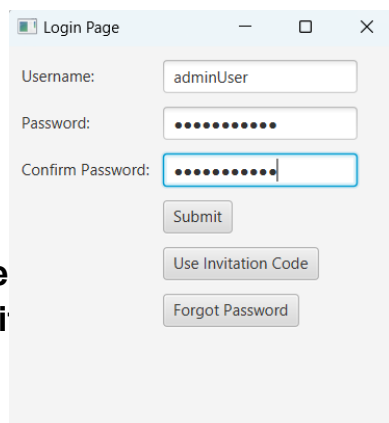
Code

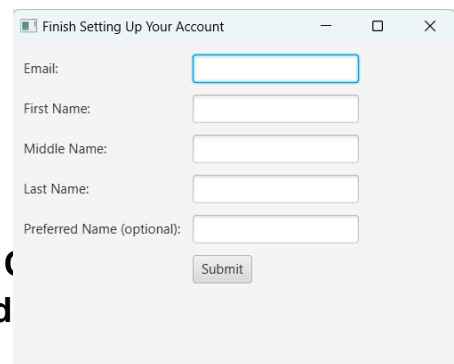
Testing

Test Case 1: Initial Admin Account Creation

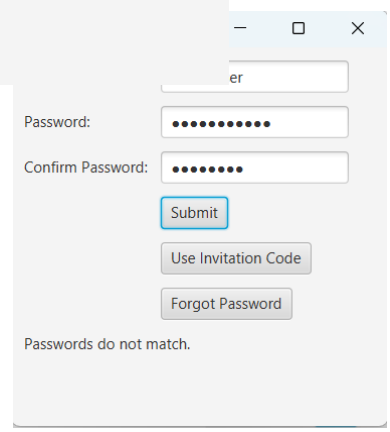
- **Screenshot**: Login page with fields for Username, Password, Confirm Password, and buttons for "Submit", "Use Invitation Code", and "Forgot Password".
- **Test Input**:
  - Username: `adminUser`
  - Password: `password123`
  - Confirm Password: `password123`
- **Expected Output**:
  - The system should create an admin account and redirect the user to the "Finish Setting Up Your Account" page.
- **Explanation**:
  - This test case ensures the initial setup for admin creation works as expected, with proper password confirmation validation. The admin is prompted to set up further details after submitting the login credentials.





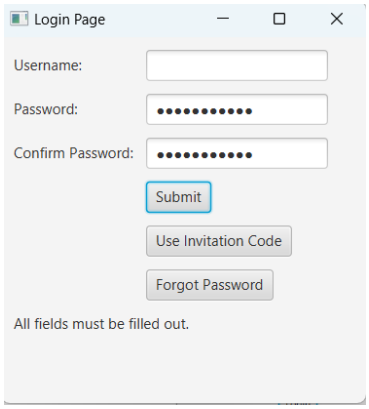


Te unt C
wi ord



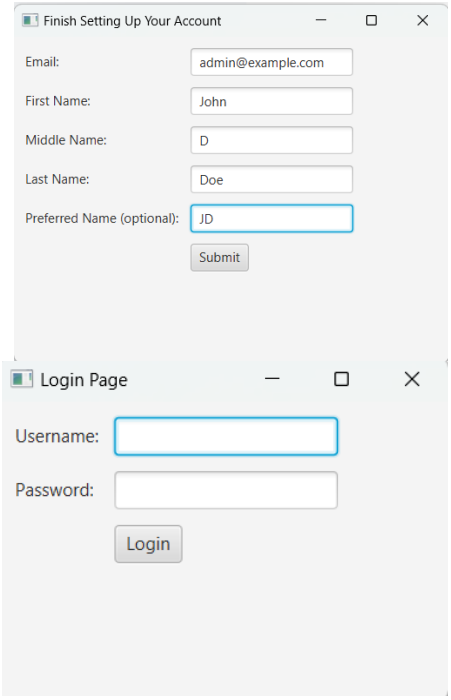
  - Username: `adminUser`
  - Password: `password123`
  - Confirm Password: `password`
- **Expected Output**:
  - The system displays an error message: `"Passwords do not match."`
  - The user remains on the login page and is not redirected.
- **Explanation**: This test ensures the system properly checks for matching passwords before allowing the user to proceed to the next step. Mismatching passwords should block progression to the setup page.

Test Case 3: Admin Account Creation with Empty Fields



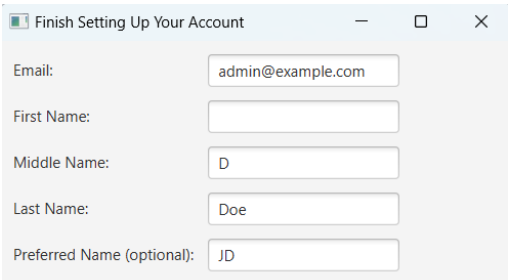
- **Test Input**:
  - Username: ``
  - Password: `password123`
  - Confirm Password: `password123`
- **Expected Output**:
  - The system displays an error message: `"All fields must be filled out."`
  - The user remains on the login page and is not redirected.
- **Explanation**: This test checks if the system validates that all required fields (username, password, confirm password) are filled in. Leaving any field blank should trigger an error and prevent progression.

Test Case 4: Account Setup with Valid Input



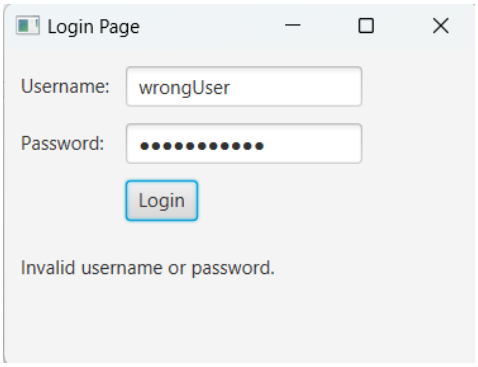
- **Test Input** on "Finish Setting Up Your Account" page:
  - Email: `admin@example.com`
  - First Name: `John`
  - Middle Name: `D`
  - Last Name: `Doe`
  - Preferred Name: `JD`
- **Expected Output**:
  - The system accepts the input and redirects the user back to the login page.
  - No error messages should appear.
- **Explanation**: This test ensures that the system correctly handles valid input on the account setup page and redirects the user back to the login page upon successful form submission.

Test Case 5: Account Setup with Missing Required Fields



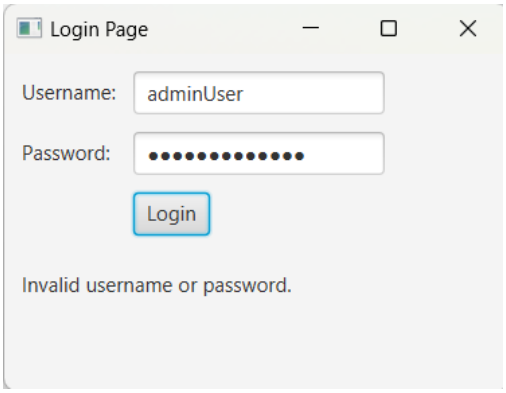
- **Test Input** on "Finish Setting Up Your Account" page:
  - Email: `admin@example.com`
  - First Name: ``
  - Middle Name: `D`
  - Last Name: `Doe`
  - Preferred Name: `JD`
- **Expected Output**:
  - The system displays an error message: `"All fields must be filled out."`
  - The user remains on the setup page and is not redirected.
- **Explanation**: This test ensures that the system validates that all required fields (email, first name, middle name, last name) are filled before allowing form submission. An empty required field should block progression.

Test Case 6: Login with Incorrect Username



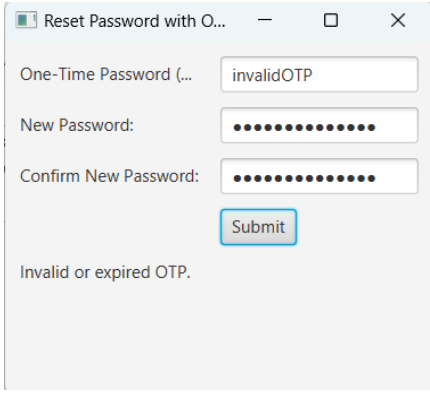
- **Test Input**:
  - Username: `wrongUser`
  - Password: `password123`
- **Expected Output**:
  - The system displays an error message: `"Invalid username or password."`
  - The user remains on the login page and is not redirected.
- **Explanation**: This test verifies that the system correctly handles invalid login attempts by checking both the username and password against stored values. Incorrect credentials should prevent access to the dashboard.

Test Case 7: Login with Incorrect Password



- **Test Input**:
  - Username: `adminUser`
  - Password: `wrongPassword`
- **Expected Output**:
  - The system displays an error message: `"Invalid username or password."`
  - The user remains on the login page and is not redirected.
- **Explanation**: This test ensures the system properly verifies passwords during login. An incorrect password should block access to the dashboard.

Test Case 8: Invalid OTP Entered by User



- **Test Input**:
  - OTP: `invalidOTP`
  - New Password: `newPassword123`
  - Confirm New Password: `newPassword123`
- **Expected Output**:
  - The system rejects the OTP and displays an error message: `"Invalid or expired OTP."`
- **Explanation**:
  - This test ensures that only valid OTPs generated by the admin are accepted. If the user enters an incorrect or expired OTP, the system should block the password reset.

Test Case 9: Password Reset with Mismatching Passwords

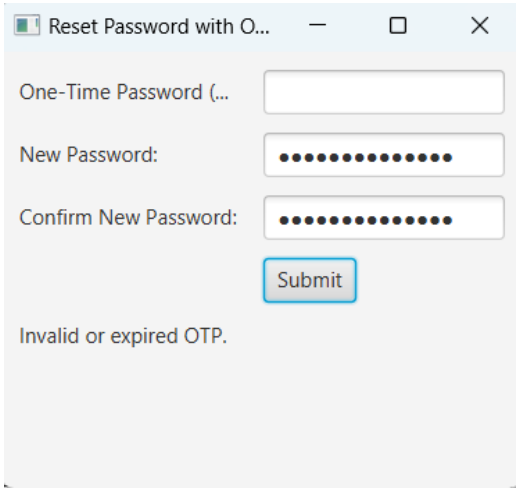
- **Test Input**:
  - OTP: `validOTPFromAdmin`
  - New Password: `newPassword123`
  - Confirm New Password: `newPassword321`
- **Expected Output**:
  - The system displays an error message: `"Passwords do not match."`, and the password reset fails.
- **Explanation**:
  - This test case ensures that the system properly validates matching passwords. Even with a valid OTP, mismatching passwords should block the reset process.

Test Case 10: Admin OTP Expiry

- **Scenario**: The user attempts to reset the password after the OTP has expired.
- **Test Input**:
  - OTP: `expiredOTP`
  - New Password: `newPassword123`
  - Confirm New Password: `newPassword123`
- **Expected Output**:
  - The system displays an error message: `"Invalid or expired OTP."`, and the password reset fails.

- **Explanation**:
  - This test ensures that OTPs are time-bound (e.g., valid for 15 minutes) and that the system correctly handles expired OTPs by preventing password resets with expired tokens.

Test Case 11: Empty OTP Field



- **Test Input**:
  - OTP: ``
  - New Password: `newPassword123`
  - Confirm New Password: `newPassword123`
- **Expected Output**:
  - The system displays an error message: `"OTP cannot be empty."`, and the password reset fails.
- **Explanation**:
  - This test case ensures that the OTP field cannot be left blank during password reset and the user must enter a valid OTP provided by the admin.

Screencasts

Technical Screencast
User Screencast

GitHub URL

https://github.com/mansii-28/CSE360_Project.git

```java
package phase1GUI;

import javafx.application.Application;

/**
 * Main class for the GUI application.
 * This class extends the JavaFX Application class to provide the main GUI window.
 */
public class MainGUI extends Application {
    private List<User> users = new ArrayList<>();
    private boolean isInitialLogin = true; // Flag to indicate if it's the initial login
    private Label messageLabel = new Label(); // Label to display messages

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Login Page");
        primaryStage.setScene(createLoginScene(primaryStage, isInitialLogin));
        primaryStage.show();
    }

    public List<User> getUsers() {
        return users;
    }

    private Scene createLoginScene(Stage primaryStage, boolean showConfirmPassword) {
        messageLabel.setText(""); // Clear old messages
        Label userLabel = new Label("Username:");
        TextField userTextField = new TextField();
        Label pwLabel = new Label("Password:");
        PasswordField pwTextField = new PasswordField();
        Label pwConfirmLabel = new Label("Confirm Password:");
        PasswordField pwConfirmTextField = new PasswordField();

        Button loginOrCreateButton = new Button(showConfirmPassword ? "Submit" : "Login");
        Button invitationCodeButton = new Button("Use Invitation Code");
        Button forgotPasswordButton = new Button("Forgot Password");

        // Layout for the login/registration form
        GridPane grid = new GridPane();
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(10, 10, 10, 10));
        grid.add(userLabel, 0, 0);
        grid.add(userTextField, 1, 0);
        grid.add(pwLabel, 0, 1);
        grid.add(pwTextField, 1, 1);

        if (showConfirmPassword) {
            // Add Confirm Password field and Submit button for account creation
            grid.add(pwConfirmLabel, 0, 2);
            grid.add(pwConfirmTextField, 1, 2);
            grid.add(loginOrCreateButton, 1, 3);
            grid.add(invitationCodeButton, 1, 4);
```

```java
            grid.add(forgotPasswordButton, 1, 5);
        } else {
            // Standard login (no Confirm Password)
            grid.add(loginOrCreateButton, 1, 2);
            grid.add(invitationCodeButton, 1, 3);
            grid.add(forgotPasswordButton, 1, 4);
        }

        grid.add(messageLabel, 0, 6, 2, 1);

        // Set event handlers for buttons
        loginOrCreateButton.setOnAction(e -> {
            String username = userTextField.getText();
            byte[] password = pwTextField.getText().getBytes();

            if (username.isEmpty() || pwTextField.getText().isEmpty() || (showConfirmPassword && pwConfirmTextField.getText().isEmpty())) {
                messageLabel.setText("All fields must be filled out.");
                return;
            }

            if (showConfirmPassword) {
                byte[] confirmPassword = pwConfirmTextField.getText().getBytes();
                if (!new String(password).equals(new String(confirmPassword))) {
                    messageLabel.setText("Passwords do not match.");
                    return;
                }

                // Create a new admin user or another role
                User admin = new Admin("", username, password, LocalDateTime.now().plusDays(1), "", "", "", "", new ArrayList<>());
                users.add(admin);
                messageLabel.setText("Account created for: " + username);
                userTextField.clear();
                pwTextField.clear();
                pwConfirmTextField.clear();
                isInitialLogin = false; // No longer the first login

                // Redirect to account setup
                showSetupPage(primaryStage, admin);
            } else {
                // Handle user login
                boolean userFound = false;
                for (User user : users) {
                    if (user.getUsername().equals(username) && new String(user.getPassword()).equals(new String(password))) {
                        if (user.getInviteCode() == null) {
                            messageLabel.setText("Welcome, " + username + "! Role: " + user.getRoles());
                            showRoleSelectionPage(primaryStage, user);
                        } else {
                            messageLabel.setText("Please complete your account setup using the invitation code.");
                        }
                        userFound = true;
                        break;
                    }
```

```
                }
                if (!userFound) {
                    messageLabel.setText("Invalid username or password.");
                }
            }
        });

        // Invitation code and OTP event handlers
        invitationCodeButton.setOnAction(e -> showInvitationCodePage(primaryStage));
        forgotPasswordButton.setOnAction(e -> showForgotPasswordPage(primaryStage));

        return new Scene(grid, 300, 300);
    }



    private void showSetupPage(Stage primaryStage, User user) {
        primaryStage.setTitle("Finish Setting Up Your Account");

        // Create labels and text fields
        Label emailLabel = new Label("Email:");
        TextField emailTextField = new TextField();
        Label firstNameLabel = new Label("First Name:");
        TextField firstNameTextField = new TextField();
        Label middleNameLabel = new Label("Middle Name:");
        TextField middleNameTextField = new TextField();
        Label lastNameLabel = new Label("Last Name:");
        TextField lastNameTextField = new TextField();
        Label preferredNameLabel = new Label("Preferred Name (optional):");
        TextField preferredNameTextField = new TextField();
        Label setupMessageLabel = new Label(); // Label to display messages on the setup page

        // Create submit button
        Button submitButton = new Button("Submit");

        submitButton.setOnAction(e -> {
            if (emailTextField.getText().isEmpty() ||
                firstNameTextField.getText().isEmpty() ||
                middleNameTextField.getText().isEmpty() ||
                lastNameTextField.getText().isEmpty()) {
                setupMessageLabel.setText("All fields must be filled out.");
                return;
            }

            user.setEmail(emailTextField.getText());
            user.setFirstName(firstNameTextField.getText());
            user.setMiddleName(middleNameTextField.getText());
            user.setLastName(lastNameTextField.getText());
            String preferredName = preferredNameTextField.getText().isEmpty() ? firstNameTextField.getText() : preferredNameTextField.getText();
            user.setPreferredName(preferredName);

            setupMessageLabel.setText("Account setup complete for: " + user.getUsername());

            // Redirect back to login page
            primaryStage.setTitle("Login Page");
            primaryStage.setScene(createLoginScene(primaryStage, false));
```

```java
        });

        // Create layout and add components
        GridPane grid = new GridPane();
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(10, 10, 10, 10));
        grid.add(emailLabel, 0, 0);
        grid.add(emailTextField, 1, 0);
        grid.add(firstNameLabel, 0, 1);
        grid.add(firstNameTextField, 1, 1);
        grid.add(middleNameLabel, 0, 2);
        grid.add(middleNameTextField, 1, 2);
        grid.add(lastNameLabel, 0, 3);
        grid.add(lastNameTextField, 1, 3);
        grid.add(preferredNameLabel, 0, 4);
        grid.add(preferredNameTextField, 1, 4);
        grid.add(submitButton, 1, 5);
        grid.add(setupMessageLabel, 0, 6, 2, 1); // Add the message label to the grid

        Scene scene = new Scene(grid, 400, 300);
        primaryStage.setScene(scene);
    }

    private void showRoleSelectionPage(Stage primaryStage, User user) {
        List<String> roles = new ArrayList<>(); // Collect all the roles the user has
        if (user.getRoles().contains("Admin")) roles.add("Admin");
        if (user.getRoles().contains("Student")) roles.add("Student");
        if (user.getRoles().contains("Instructor")) roles.add("Instructor");
        // If the user has only one role, go directly to the corresponding page
        if (roles.size() == 1) {
            String selectedRole = roles.get(0);
            switch (selectedRole) {
                case "Admin":
                    showAdminPage(primaryStage);
                    break;
                case "Student":
                    showStudentPage(primaryStage, user);
                    break;
                case "Instructor":
                    showInstructorPage(primaryStage, user);
                    break;
            }
            return;
        }
        // Multiple roles, show role selection page
        primaryStage.setTitle("Select Role");
        Label roleLabel = new Label("Select your role for this session:");
        Button adminButton = new Button("Admin");
        Button studentButton = new Button("Student");
        Button instructorButton = new Button("Instructor");

        // Add event handlers to switch to the appropriate page
        adminButton.setOnAction(e -> showAdminPage(primaryStage));
        studentButton.setOnAction(e -> showStudentPage(primaryStage, user));
        instructorButton.setOnAction(e -> showInstructorPage(primaryStage, user));
        // Create layout and add components
```

```java
        GridPane grid = new GridPane();
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(10, 10, 10, 10));
        grid.add(roleLabel, 0, 0, 2, 1);

        if (roles.contains("Admin")) {
            grid.add(adminButton, 0, 1);
        }
        if (roles.contains("Student")) {
            grid.add(studentButton, 1, 1);
        }
        if (roles.contains("Instructor")) {
            grid.add(instructorButton, 0, 2);
        }
        Scene scene = new Scene(grid, 300, 200);
        primaryStage.setScene(scene);
    }

    private void showAdminPage(Stage primaryStage) {
        primaryStage.setTitle("Admin Page");
        Label adminLabel = new Label("Admin Functions");
        Button inviteUserButton = new Button("Invite User");
        Button resetAccountButton = new Button("Reset Account");
        Button deleteAccountButton = new Button("Delete Account");
        Button listUsersButton = new Button("List User Accounts");
        Button addRemoveRolesButton = new Button("Add/Remove Roles");
        Button logoutButton = new Button("Logout");
        Label adminMessageLabel = new Label(); // Label for admin messages
        inviteUserButton.setOnAction(e -> {
            // Implement your invite user functionality here
            String inviteCode = generateInviteCode();
            User invitedUser = new User(null, null, null, false, LocalDateTime.now().plusDays(1), null, null, null, null, new ArrayList<>(), new ArrayList<>(List.of("Student")), inviteCode);
            users.add(invitedUser);
            messageLabel.setText("User invited with code: " + inviteCode);
            adminMessageLabel.setText("Invite user with this code: " + inviteCode);
        });
        resetAccountButton.setOnAction(e -> resetAccount(primaryStage, adminMessageLabel));
        deleteAccountButton.setOnAction(e -> deleteAccount(primaryStage, adminMessageLabel));
        listUsersButton.setOnAction(e -> listUserAccounts(adminMessageLabel));
        addRemoveRolesButton.setOnAction(e -> addRemoveRoles(primaryStage, adminMessageLabel));

        logoutButton.setOnAction(e -> {
            primaryStage.setTitle("Login Page");
            primaryStage.setScene(createLoginScene(primaryStage, false));
        });
        GridPane grid = new GridPane();
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(10, 10, 10, 10));
        grid.add(adminLabel, 0, 0);
        grid.add(inviteUserButton, 0, 1);
        grid.add(resetAccountButton, 0, 2);
        grid.add(deleteAccountButton, 0, 3);
```

```java
        grid.add(listUsersButton, 0, 4);
        grid.add(addRemoveRolesButton, 0, 5);
        grid.add(logoutButton, 0, 6);
        grid.add(adminMessageLabel, 0, 7); // Add the message label for admin functions
        primaryStage.setScene(new Scene(grid, 300, 400));
    }

    private void showInstructorPage(Stage primaryStage, User user) {
        // Add your Instructor page components here
        primaryStage.setTitle("Instructor Page");
        Label instructorLabel = new Label("Welcome to the Instructor Page, " + user.getUsername() + "!");
        Button logoutButton = new Button("Logout");
        logoutButton.setOnAction(e -> {
            primaryStage.setTitle("Login Page");
            primaryStage.setScene(createLoginScene(primaryStage, false));
        });
        GridPane grid = new GridPane();
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(10, 10, 10, 10));
        grid.add(instructorLabel, 0, 0);
        grid.add(logoutButton, 0, 1);
        Scene scene = new Scene(grid, 300, 200);
        primaryStage.setScene(scene);
    }
    private void showStudentPage(Stage primaryStage, User user) {
        // Add your Student page components here
        primaryStage.setTitle("Student Page");
        Label studentLabel = new Label("Welcome to the Student Page, " + user.getUsername() + "!");
        Button logoutButton = new Button("Logout");
        logoutButton.setOnAction(e -> {
            primaryStage.setTitle("Login Page");
            primaryStage.setScene(createLoginScene(primaryStage, false));
        });
        GridPane grid = new GridPane();
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(10, 10, 10, 10));
        grid.add(studentLabel, 0, 0);
        grid.add(logoutButton, 0, 1);
        Scene scene = new Scene(grid, 300, 200);
        primaryStage.setScene(scene);
    }


    private void showInvitationCodePage(Stage primaryStage) {
        primaryStage.setTitle("Enter Invitation Code");

        // Invitation code input
        Label invitationLabel = new Label("Invitation Code:");
        TextField invitationTextField = new TextField();

        // Username input
        Label usernameLabel = new Label("Create Username:");
        TextField usernameField = new TextField();
```

```java

        // Password input
        Label passwordLabel = new Label("Create Password:");
        PasswordField passwordField = new PasswordField();

        // Confirm password input
        Label confirmPasswordLabel = new Label("Confirm Password:");
        PasswordField confirmPasswordField = new PasswordField();

        // Message label for submission
        Label invitationMessageLabel = new Label();
        Button submitButton = new Button("Submit");

        submitButton.setOnAction(e -> {
            String enteredInviteCode = invitationTextField.getText();
            String newUsername = usernameField.getText();
            String newPassword = passwordField.getText();
            String confirmPassword = confirmPasswordField.getText();

            // Validate the invitation code
            boolean validCode = false;
            User invitedUser = null;

            // Searches for users with the same invite code
            for (User user : users) {
                if (user.getInviteCode() != null && user.getInviteCode().equals(enteredInviteCode)) {
                    validCode = true;
                    invitedUser = user;
                    break;
                }
            }

            // Proceed with account setup if the invitation code is valid
            if (validCode && invitedUser != null) {
                if (newPassword.equals(confirmPassword)) {
                    invitedUser.setUsername(newUsername);
                    invitedUser.setPassword(newPassword.getBytes());
                    invitedUser.setInviteCode(null);  // Clear the invite code now that it's used

                    invitationMessageLabel.setText("Account setup complete! Welcome, " + newUsername);
                    showRoleSelectionPage(primaryStage, invitedUser); // Redirect to role selection
                } else {
                    invitationMessageLabel.setText("Passwords do not match.");
                }
            } else {
                invitationMessageLabel.setText("Invalid invitation code.");
            }
        });

        GridPane grid = new GridPane();
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(10, 10, 10, 10));
```

```java
        grid.add(invitationLabel, 0, 0);
        grid.add(invitationTextField, 1, 0);
        grid.add(usernameLabel, 0, 1);
        grid.add(usernameField, 1, 1);
        grid.add(passwordLabel, 0, 2);
        grid.add(passwordField, 1, 2);
        grid.add(confirmPasswordLabel, 0, 3);
        grid.add(confirmPasswordField, 1, 3);
        grid.add(submitButton, 1, 4);
        grid.add(invitationMessageLabel, 0, 5, 2, 1);

        Scene scene = new Scene(grid, 400, 250);
        primaryStage.setScene(scene);
    }

    private void showForgotPasswordPage(Stage primaryStage) {
        primaryStage.setTitle("Reset Password with OTP");

        // Labels and fields for OTP and new password
        Label otpLabel = new Label("One-Time Password (OTP):");
        TextField otpTextField = new TextField();
        Label newPasswordLabel = new Label("New Password:");
        PasswordField newPasswordTextField = new PasswordField();
        Label confirmNewPasswordLabel = new Label("Confirm New Password:");
        PasswordField confirmNewPasswordTextField = new PasswordField();
        Label otpMessageLabel = new Label();
        Button submitButton = new Button("Submit");

        submitButton.setOnAction(e -> {
            String otp = otpTextField.getText();
            String newPassword = newPasswordTextField.getText();
            String confirmNewPassword = confirmNewPasswordTextField.getText();
            boolean otpValid = false;
            User userToUpdate = null;

            // Validate the OTP and password match
            if (!newPassword.equals(confirmNewPassword)) {
                otpMessageLabel.setText("Passwords do not match.");
                return;
            }

            // Check if the OTP is valid
            for (User user : users) {
                if (user.isOneTimePassword() && user.isOtpValid()) {
                    userToUpdate = user;
                    otpValid = true;
                    break;
                }
            }

            if (otpValid && userToUpdate != null) {
                userToUpdate.resetPassword(newPassword.getBytes());
                otpMessageLabel.setText("Password successfully reset.");
                primaryStage.setScene(createLoginScene(primaryStage, false)); // Return to login page
            } else {
                otpMessageLabel.setText("Invalid or expired OTP.");
```

```
            }
        });

        // Layout for the reset password form
        GridPane grid = new GridPane();
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(10, 10, 10, 10));
        grid.add(otpLabel, 0, 0);
        grid.add(otpTextField, 1, 0);
        grid.add(newPasswordLabel, 0, 1);
        grid.add(newPasswordTextField, 1, 1);
        grid.add(confirmNewPasswordLabel, 0, 2);
        grid.add(confirmNewPasswordTextField, 1, 2);
        grid.add(submitButton, 1, 3);
        grid.add(otpMessageLabel, 0, 4, 2, 1);

        // Create and set the scene
        Scene scene = new Scene(grid, 300, 250);  // Adjusted scene height for extra field
        primaryStage.setScene(scene);
    }
    public String generateInviteCode() {
        String chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
        StringBuilder code = new StringBuilder(6);
        Random rnd = new Random();
        for (int i = 0; i < 6; i++) {
            code.append(chars.charAt(rnd.nextInt(chars.length())));
        }
        String inviteCode = code.toString();
        System.out.println("Invite code generated: " + inviteCode); // Prints to console
        return inviteCode;
    }
    private void resetAccount(Stage primaryStage, Label adminMessageLabel) {
        TextInputDialog dialog = new TextInputDialog();
        dialog.setTitle("Reset Account");
        dialog.setHeaderText("Enter the username of the account to reset:");
        dialog.setContentText("Username:");
        Optional<String> result = dialog.showAndWait();
        if (result.isPresent()) {
            String username = result.get();
            User userToReset = null;
            // Find user by username
            for (User user : users) {
                if (user.getUsername().equals(username)) {
                    userToReset = user;
                    break;
                }
            }
            if (userToReset != null) {
                // Implement logic to reset the account (e.g., set a new password, send OTP)
                adminMessageLabel.setText("Account reset for: " + username);
                // Additional logic goes here...
            } else {
                adminMessageLabel.setText("User not found.");
            }
        } else {
            adminMessageLabel.setText("Reset cancelled.");
```

```
        }
    }
    private void deleteAccount(Stage primaryStage, Label adminMessageLabel) {
        // Show a dialog to delete an account
        TextInputDialog dialog = new TextInputDialog();
        dialog.setTitle("Delete Account");
        dialog.setHeaderText("Enter the username to delete:");
        dialog.setContentText("Username:");
        Optional<String> result = dialog.showAndWait();
        if (result.isPresent()) {
            String username = result.get();
            users.removeIf(user -> user.getUsername().equals(username));
            adminMessageLabel.setText("Account deleted for user: " + username);
        } else {
            adminMessageLabel.setText("Deletion cancelled.");
        }
    }
    private void listUserAccounts(Label adminMessageLabel) {
        StringBuilder userList = new StringBuilder("User Accounts:\n");
        for (User user : users) {
            userList.append(user.getUsername()).append("\n");
        }
        adminMessageLabel.setText(userList.toString());
    }
    private void addRemoveRoles(Stage primaryStage, Label adminMessageLabel) {
        // Show a dialog to add/remove roles
        TextInputDialog dialog = new TextInputDialog();
        dialog.setTitle("Add/Remove Roles");
        dialog.setHeaderText("Enter the username and role (format: username, role):");
        dialog.setContentText("Username, Role:");
        Optional<String> result = dialog.showAndWait();
        if (result.isPresent()) {
            String[] input = result.get().split(",\\s*");
            if (input.length == 2) {
                String username = input[0].trim();
                String role = input[1].trim();
                for (User user : users) {
                    if (user.getUsername().equals(username)) {
                        if (user.getRoles().contains(role)) {
                            user.removeRole(role); // Assuming you have a removeRole method
                            adminMessageLabel.setText("Removed role: " + role + " from user: " + username);
                        } else {
                            user.addRole(role); // Assuming you have an addRole method
                            adminMessageLabel.setText("Added role: " + role + " to user: " + username);
                        }
                        return;
                    }
                }
                adminMessageLabel.setText("User not found.");
            } else {
                adminMessageLabel.setText("Invalid input format.");
            }
        } else {
            adminMessageLabel.setText("Operation cancelled.");
        }
```

```
    }


}
```

```
 * <p>Admin Class.</p>
package phase1GUI;

import java.sql.PreparedStatement;

/**
 * Admin class extends the User class and provides additional admin functionalities.
 */
public class Admin extends User {

    /**
     * Constructor to initialize an Admin user with provided attributes and assigns the "Admin" role by default.
     *
     * @param email         The email address of the admin.
     * @param username      The username of the admin.
     * @param password      The password for the admin's account (stored as a byte array).
     * @param otpExpiry     The expiration time for the one-time password (OTP).
     * @param firstName     The first name of the admin.
     * @param middleName    The middle name of the admin.
     * @param lastName      The last name of the admin.
     * @param preferredName The preferred name of the admin (optional).
     * @param topics        A list of topics associated with the admin.
     */
    public Admin(String email, String username, byte[] password, LocalDateTime otpExpiry, String firstName,
                 String middleName, String lastName, String preferredName, List<String> topics) {
        super(email, username, password, false, otpExpiry, firstName, middleName, lastName, preferredName, topics, new ArrayList<>(List.of("Admin")), null);
    }

    /**
     * Generates a secure invitation code for a user based on their username and roles.
     *
     * @param username The username of the user to generate an invitation for.
     * @param roles    The list of roles assigned to the user.
     * @return The generated invitation code.
     */
    public String generateInvitationCode(String username, List<String> roles) {
        // Generate a secure random UUID
        String invitationCode = UUID.randomUUID().toString();
        // You can store the invitation in a database or data structure if needed
        System.out.println("Invitation code generated for " + username + ": " + invitationCode);
        return invitationCode;
    }

    /**
     * Resets the account of a user by setting a new one-time password and expiry.
     *
     * @param user The user whose account is being reset.
     */
    public void resetUserAccount(User user) {
        // Set a one-time password and expiry time
        String oneTimePassword = "newOTP_" + System.currentTimeMillis(); // Example OTP
        user.setPassword(oneTimePassword.getBytes());
```

```
        user.setOneTimePassword(true);
        user.setOtpExpiry(LocalDateTime.now().plusDays(1)); // Example expiry
        System.out.println("Password reset for user: " + user.getUsername());
    }

    /**
     * Deletes a user from the system's list of users.
     *
     * @param users       The list of all users.
     * @param userToDelete The user to be deleted.
     * @return True if the user was successfully deleted, false if the user was not found.
     */
    public boolean deleteUser(List<User> users, User userToDelete) {
        if (users.contains(userToDelete)) {
            users.remove(userToDelete);
            System.out.println("User deleted: " + userToDelete.getUsername());
            return true;
        }
        System.out.println("User not found: " + userToDelete.getUsername());
        return false;
    }

    /**
     * Retrieves all user accounts.
     *
     * @param users The list of all users.
     * @return The list of all user accounts.
     */
    public List<User> getUserAccounts(List<User> users) {
        return users; // Return the list of users passed from MainGUI
    }

    /**
     * Adds a new role to a user if they do not already have it.
     *
     * @param user The user to whom the role will be added.
     * @param role The role to add to the user.
     */
    public void addRoleToUser(User user, String role) {
        if (!user.getRoles().contains(role)) {
            user.addRole(role);
            System.out.println("Role added to user " + user.getUsername() + ": " + role);
        } else {
            System.out.println("User " + user.getUsername() + " already has role: " + role);
        }
    }



    /**
     * Removes a role from a user if they currently have it.
     *
     * @param user The user from whom the role will be removed.
     * @param role The role to remove from the user.
     */
    public void removeRoleFromUser(User user, String role) {
        if (user.getRoles().contains(role)) {
            user.removeRole(role);
```

```
            System.out.println("Role removed from user " + user.getUsername() + ": " + role);
        } else {
            System.out.println("User " + user.getUsername() + " does not have role: " + role);
        }
    }

}

```

```java

package phase1GUI;

import java.time.LocalDateTime;

 * <p>User Class.</p>

/**
 * This class represents a user in the system. Each user has attributes such as email, username, password, and roles.
 * It includes methods to manage user roles and personal information.
 */
public class User {
    private String email;
    private String username;
    private byte[] password;
    private boolean isOneTimePassword;
    private LocalDateTime otpExpiry;
    private String firstName;
    private String middleName;
    private String lastName;
    private String preferredName;
    private List<String> topics;
    private String expertiseLevel;
    private ArrayList<String> roles; // Changed to ArrayList
    private String inviteCode;  // Field to store the invitation code

    /**
     * Constructor to initialize a user with the given attributes.
     *
     * @param email             The email address of the user.
     * @param username          The username of the user.
     * @param password          The user's password, stored as a byte array.
     * @param isOneTimePassword A flag to indicate if the password is a one-time password.
     * @param otpExpiry         The expiration time for the one-time password.
     * @param firstName         The user's first name.
     * @param middleName        The user's middle name.
     * @param lastName          The user's last name.
     * @param preferredName     The user's preferred name (optional).
     * @param topics            A list of topics associated with the user.
     * @param roles             A list of roles assigned to the user.
     * @param inviteCode        Invite code for user
     */
    public User(String email, String username, byte[] password, boolean isOneTimePassword, LocalDateTime otpExpiry,
            String firstName, String middleName, String lastName, String preferredName, List<String> topics, ArrayList<String> roles, String inviteCode) {
        this.email = email;
        this.username = username;
        this.password = password;
        this.isOneTimePassword = isOneTimePassword;
        this.otpExpiry = otpExpiry;
        this.firstName = firstName;
        this.middleName = middleName;
        this.lastName = lastName;
        this.preferredName = preferredName;
        this.topics = topics;
```

```java
        this.expertiseLevel = "Intermediate"; // Default expertise level
        this.roles = roles; // Initialize directly
        this.inviteCode = inviteCode;
    }

    /**
     * Gets the email of the user.
     *
     * @return The email address.
     */
    public String getEmail() {
        return email;
    }

    /**
     * Sets the email of the user.
     *
     * @param email The new email address.
     */
    public void setEmail(String email) {
        this.email = email;
    }

    /**
     * Gets the username of the user.
     *
     * @return The username.
     */
    public String getUsername() {
        return username;
    }

    /**
     * Sets the username of the user.
     *
     * @param username The new username.
     */
    public void setUsername(String username) {
        this.username = username;
    }

    /**
     * Gets the user's password.
     *
     * @return The password as a byte array.
     */
    public byte[] getPassword() {
        return password;
    }

    /**
     * Sets the user's password.
     *
     * @param password The new password.
     */
    public void setPassword(byte[] password) {
        this.password = password;
```

```
    }

    /**
     * Checks if the user's password is a one-time password.
     *
     * @return True if the password is a one-time password, false otherwise.
     */
    public boolean isOneTimePassword() {
        return isOneTimePassword;
    }

    /**
     * Sets whether the user's password is a one-time password.
     *
     * @param oneTimePassword True if the password is a one-time password, false otherwise.
     */
    public void setOneTimePassword(boolean oneTimePassword) {
        isOneTimePassword = oneTimePassword;
    }

    /**
     * Gets the expiration time of the one-time password.
     *
     * @return The expiration time of the OTP.
     */
    public LocalDateTime getOtpExpiry() {
        return otpExpiry;
    }

    /**
     * Sets the expiration time of the one-time password.
     *
     * @param otpExpiry The new expiration time.
     */
    public void setOtpExpiry(LocalDateTime otpExpiry) {
        this.otpExpiry = otpExpiry;
    }

    /**
     * Gets the first name of the user.
     *
     * @return The first name.
     */
    public String getFirstName() {
        return firstName;
    }

    /**
     * Sets the first name of the user.
     *
     * @param firstName The new first name.
     */
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    /**
```

```
     * Gets the middle name of the user.
     *
     * @return The middle name.
     */
    public String getMiddleName() {
        return middleName;
    }

    /**
     * Sets the middle name of the user.
     *
     * @param middleName The new middle name.
     */
    public void setMiddleName(String middleName) {
        this.middleName = middleName;
    }

    /**
     * Gets the last name of the user.
     *
     * @return The last name.
     */
    public String getLastName() {
        return lastName;
    }

    /**
     * Sets the last name of the user.
     *
     * @param lastName The new last name.
     */
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    /**
     * Gets the preferred name of the user.
     *
     * @return The preferred name.
     */
    public String getPreferredName() {
        return preferredName;
    }

    /**
     * Sets the preferred name of the user.
     *
     * @param preferredName The new preferred name.
     */
    public void setPreferredName(String preferredName) {
        this.preferredName = preferredName;
    }

    /**
     * Gets the list of topics associated with the user.
     *
     * @return A list of topics.
```

```java
     */
    public List<String> getTopics() {
        return topics;
    }

    /**
     * Sets the list of topics associated with the user.
     *
     * @param topics A new list of topics.
     */
    public void setTopics(List<String> topics) {
        this.topics = topics;
    }

    /**
     * Gets the expertise level of the user.
     *
     * @return The expertise level (default is "Intermediate").
     */
    public String getExpertiseLevel() {
        return expertiseLevel;
    }

    /**
     * Sets the expertise level of the user.
     *
     * @param expertiseLevel The new expertise level.
     */
    public void setExpertiseLevel(String expertiseLevel) {
        this.expertiseLevel = expertiseLevel;
    }

    /**
     * Gets the list of roles assigned to the user.
     *
     * @return A list of roles.
     */
    public ArrayList<String> getRoles() {
        return roles;
    }


    /**
     * Adds a role to the user if the role is not already present.
     *
     * @param role The role to add.
     */
    public void addRole(String role) {
        if (!roles.contains(role)) {
            roles.add(role);
        } else {
            System.out.println("Role " + role + " already exists.");
        }
    }

    /**
     * Removes a role from the user if the role exists.
```

```
     *
     * @param role The role to remove.
     */
    public void removeRole(String role) {
        if (roles.contains(role)) {
            roles.remove(role);
        } else {
            System.out.println("Role " + role + " does not exist.");
        }
    }


// Method to check if OTP is valid based on expiry
    public boolean isOtpValid() {
        if (isOneTimePassword && LocalDateTime.now().isBefore(otpExpiry)) {
            return true;
        } else {
            System.out.println("OTP has expired or is not valid.");
            return false;
        }
    }

    // Method to reset password using OTP
    public void resetPassword(byte[] newPassword) {
        if (isOneTimePassword && isOtpValid()) {
            setPassword(newPassword);
            setOneTimePassword(false); // Disable OTP after resetting
            System.out.println("Password successfully reset.");
        } else {
            System.out.println("Password cannot be reset. OTP is invalid or expired.");
        }
    }

    public String getInviteCode() {
        return inviteCode;
    }

    public void setInviteCode(String inviteCode) {
        this.inviteCode = inviteCode;
    }
}

```

```
 * <p>Instructor Class.</p>
package phase1GUI;
import java.time.LocalDateTime;

/**
 * Constructor to initialize the Instructor object with provided attributes and assigns the "Admin" role by default.
 */
public class Instructor extends User {
    /**
     * Constructor to create a new Instructor object.
     *
     * @param email         The email address of the instructor.
     * @param username      The username chosen by the instructor.
     * @param password      The password for the instructor's account (stored as a byte array).
     * @param otpExpiry     The expiration time for the one-time password (OTP).
     * @param firstName     The first name of the instructor.
     * @param middleName    The middle name of the instructor.
     * @param lastName      The last name of the instructor.
     * @param preferredName The preferred name of the instructor (optional).
     * @param topics        A list of topics the instructor is associated with.
     */
    public Instructor(String email, String username, byte[] password, LocalDateTime otpExpiry, String firstName,
                    String middleName, String lastName, String preferredName, List<String> topics, String inviteCode) {
        super(email, username, password, false, otpExpiry, firstName, middleName, lastName, preferredName, topics, new ArrayList<>(List.of("Admin")), inviteCode);
    }


}
```

```
 * <p>Student Class.</p>
package phase1GUI;
import java.time.LocalDateTime;

/**
 * Constructor to initialize the Student object with provided attributes and assigns the "Admin" role by default.
 */
public class Student extends User {
    /**
     * Constructor to create a new Student object.
     *
     * @param email         The email address of the student.
     * @param username      The username chosen by the student.
     * @param password      The password for the student's account (stored as a byte array).
     * @param otpExpiry     The expiration time for the one-time password (OTP).
     * @param firstName     The first name of the student.
     * @param middleName    The middle name of the student.
     * @param lastName      The last name of the student.
     * @param preferredName The preferred name of the student (optional).
     * @param topics        A list of topics the student is associated with.
     */
    public Student(String email, String username, byte[] password, LocalDateTime otpExpiry, String firstName,
                   String middleName, String lastName, String preferredName, List<String> topics, String inviteCode) {
        super(email, username, password, false, otpExpiry, firstName, middleName, lastName, preferredName, topics, new ArrayList<>(List.of("Admin")), inviteCode);
    }


}
```

```
 2 * <p>Role Enum.</p>
14 package phase1GUI;
15
16 /**
17  * Enum representing the roles within the system.
18  * Each role has a display name.
19  */
20 public enum Role {
21     /**
22      * Administrator role with higher privileges.
23      */
24     ADMIN("Admin"),
25
26     /**
27      * Student role with access to learning materials.
28      */
29     STUDENT("Student"),
30
31
32     /**
33      * Instructor role with teaching privileges.
34      */
35     INSTRUCTOR("Instructor");
36
37     private final String displayName;
38
39     /**
40      * Constructor to initialize the enum with the role's display name.
41      *
42      * @param displayName The string representation of the role.
43      */
44     Role(String displayName) {
45         this.displayName = displayName;
46     }
47
48     /**
49      * Gets the display name of the role.
50      *
51      * @return The display name of the role.
52      */
53     public String getDisplayName() {
54         return displayName;
55     }
56
57
58     /**
59      * Converts a string to the corresponding Role enum.
60      * The method performs a case-insensitive match.
61      *
62      * @param roleName The string representation of the role.
63      * @return The Role corresponding to the string.
64      * @throws IllegalArgumentException If the provided string doesn't match any role.
65      */
66     public static Role fromString(String roleName) {
67         for (Role role : Role.values()) {
68             if (role.displayName.equalsIgnoreCase(roleName)) {
69                 return role;
```

```
            }
        }
        throw new IllegalArgumentException("Invalid role name: " + roleName);
    }

    /**
     * Returns the string representation of the role.
     *
     * @return The display name of the role.
     */
    @Override
    public String toString() {
        return displayName;
    }
}

```