Bilkent University
Department of Computer Science
CS 458 - Software Validation & Verification

Project 1 Report

Deniz Polat 22103590
Çağatay Akpınar 22003508
Yamaç Yiğit Ozan 22003595
Efe Tokar 22103299

Instructor: Dr. Haluk Altunel
Teaching Assistants: Giray Akyol, Furkan Gön
03/03/2024

Abstract

This project includes an authentication system where users can log in with their email or phone number or with Google OAuth. The system, which was developed as a Flask-based web application, was designed using HTML, CSS, and JavaScript. Within the scope of the project, test automation was developed using Selenium for different scenarios and the accuracy of the system was tested. The tests included valid and invalid login attempts, password restrictions, security measures against SQL and XSS attacks, and verification of Google OAuth integration. As a result of the tests, it was observed that the system worked as expected, authentication processes were performed as desired, and possible security vulnerabilities were prevented. This report comprehensively covers the technical details of the project, test scenarios, and the results obtained.

1. Introduction

1.1 Purpose of the Application

This project's primary goal is to provide a login system that allows users to safely authenticate. Users can use their system-registered phone number or email address to log in, or they can use Google OAuth authentication to check in with an external account. The program features a user interface that is supported by HTML, CSS, and JavaScript and was created with the Flask framework.

1.2 How the Application Works

The system has two basic login methods. Users can log in directly by entering their email or phone number & password. In this method, the main page is opened after the login information is verified in the database.

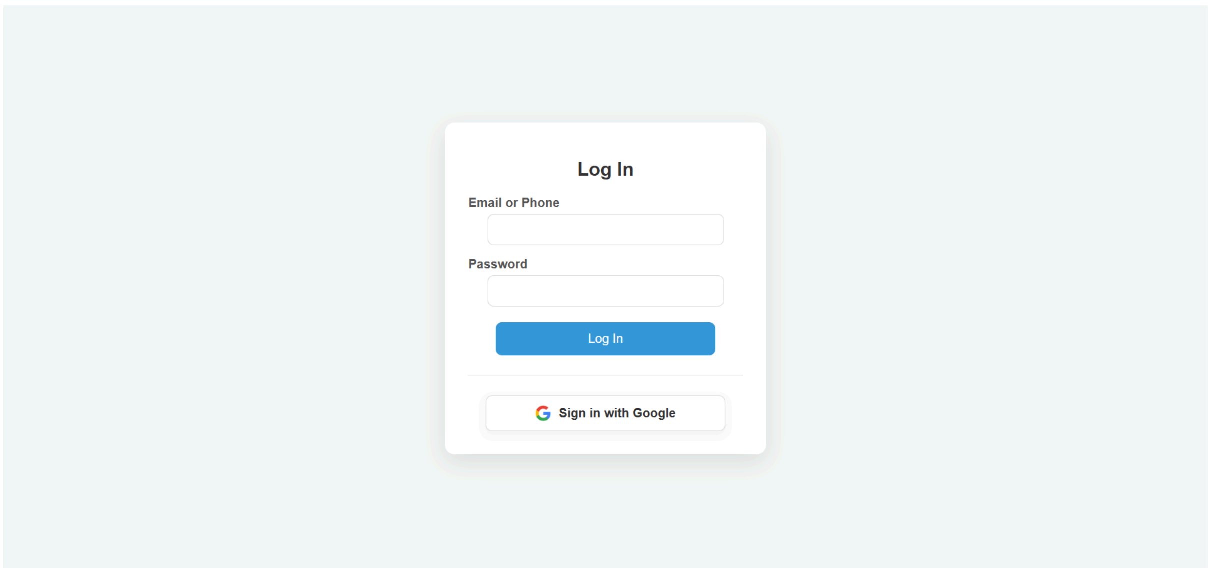


Image 1: Login Page

Alternatively, users can start the Google OAuth authentication process by clicking the "Sign in with Google" button. Users who are directed to the Google authentication page are recognized by the system after verifying their account information and the login process is completed. Google OAuth integration is implemented using the Authlib library.

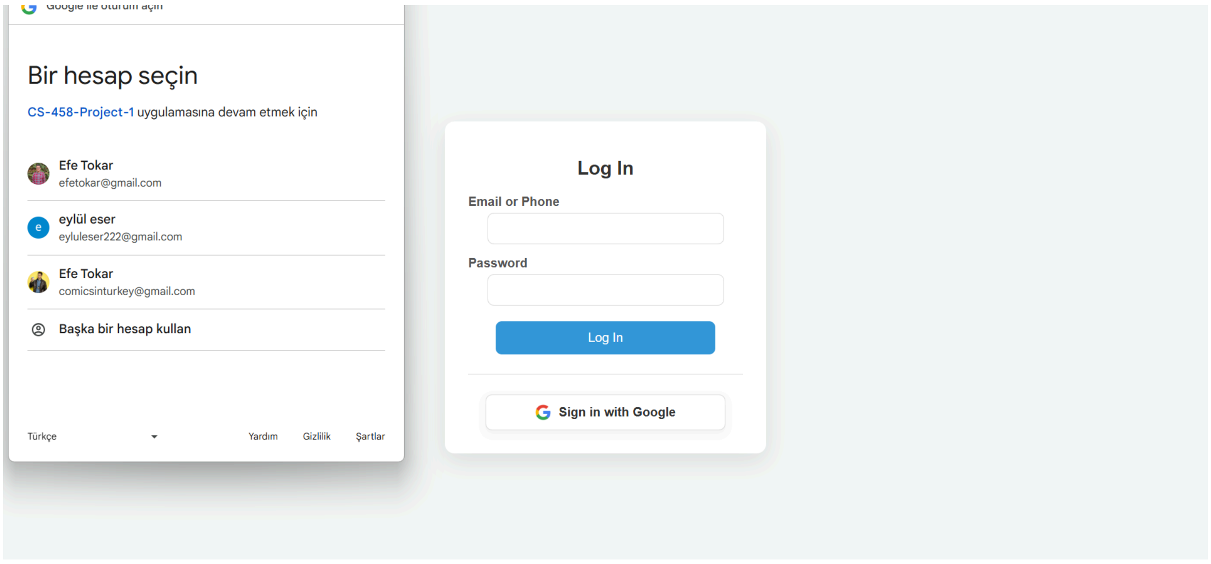
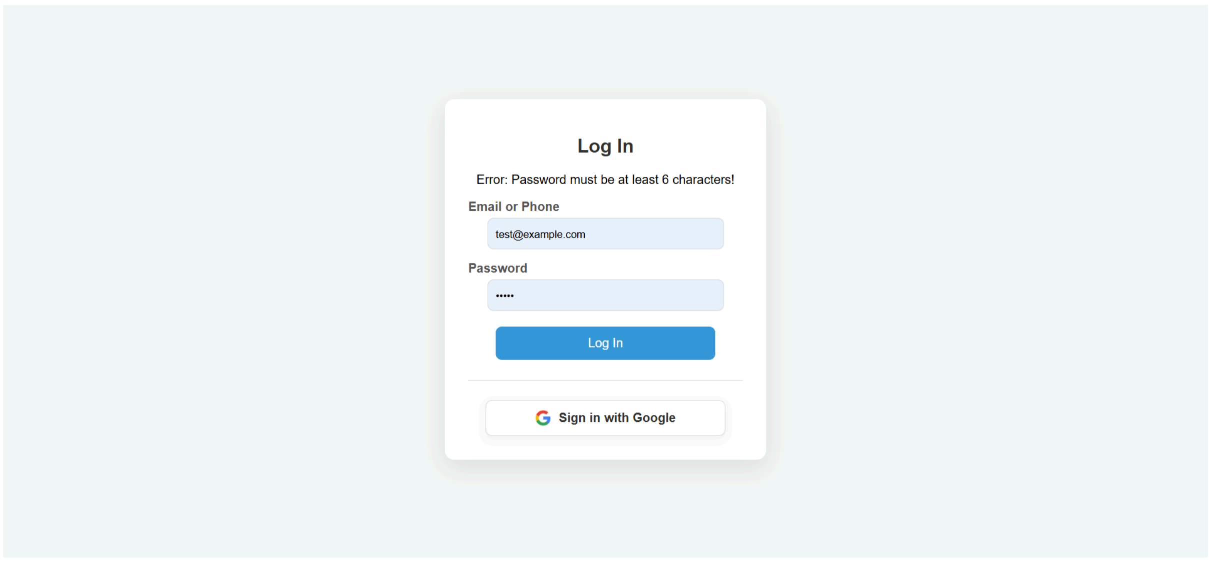


Image 2: Login Page with Google Authentication

The basic operation of the system is as follows: After the user enters their information on the login screen, the system checks whether the entered email or phone number is in the database. If the user information is verified, the session is started and the user is directed to the main page. If an incorrect entry is made, error messages are displayed and feedback is provided to the user. When the Google OAuth login option is selected, the user is recognized by the system after verifying their Google account and logging in.

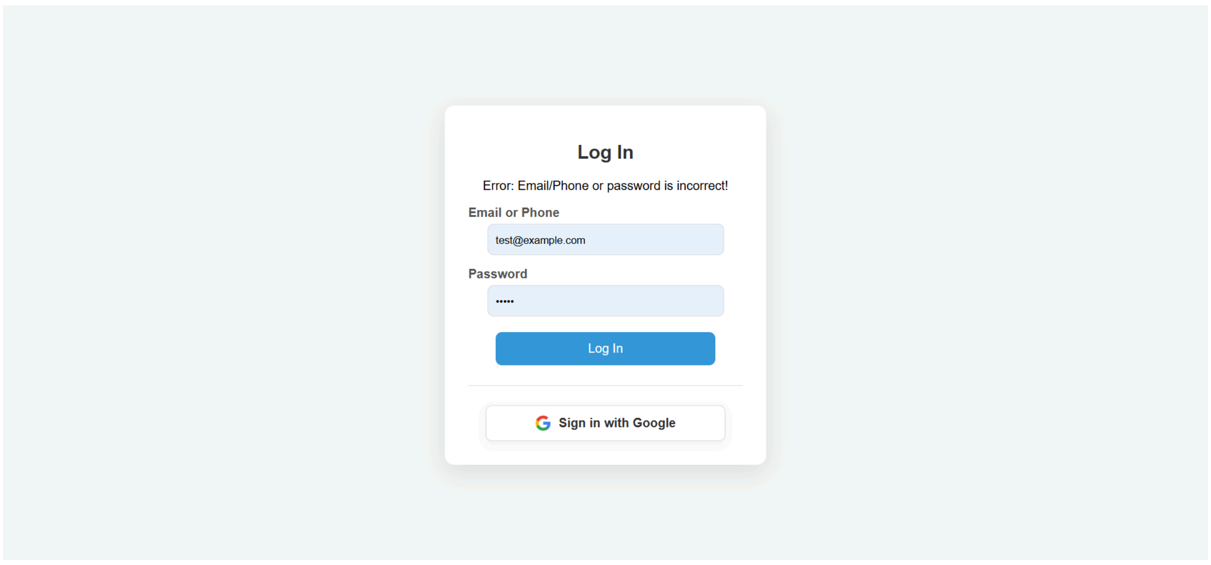

Log In

Error: Password must be at least 6 characters!

Email or Phone: test@example.com

Password: •••••

Image 3: Login Page (Example Error Message I)


Log In

Error: Email/Phone or password is incorrect!

Email or Phone: test@example.com

Password: •••••

Image 4: Login Page (Example Error Message II)

1.3 Used Technologies

The application has a Flask-based backend, while the SQLite database is used to store user information. The frontend is built with HTML, CSS, and JavaScript. Selenium test automation was used to verify that user logins and page redirects work as desired.

1.4 Security Precautions

In order to prevent SQL Injection attacks, the input data is validated with the regex filtering method. In order to prevent Cross-Site Scripting attacks, special characters are blocked from being processed in the input fields. Also, a minimum length of 6 and a maximum length of 50 characters are required to ensure password security. User sessions are managed with Flask's session module, providing an already secure authentication and session process.

1.5 Google OAuth Integration

Google OAuth integration has been integrated into the system to enable users to authenticate via their Google accounts. When the user chooses to log in with Google on the login screen, the system directs them to the Google authentication page. Here, the user completes the authentication process via Google by entering their credentials. After Google successfully authenticates the user, the system obtains an access token and obtains the user's basic information. The Flask backend performs the login process by obtaining this information and identifies the user to the system. After the login process is completed, the session information is securely stored and the user is directly directed to the welcome page.

During the Google OAuth integration, the JSON Web Token based authentication mechanism was used and the security of user data was ensured. By this, the system was only accessible to authorized and verified users. The addition of Google OAuth allowed users to log in directly to the system without having to create a new account (Although in the current state of the application, any sign up action is not possible as well.)

1.6 Test Automation

Test automation was used to analyze how the system performs in different scenarios and detect potential errors. The tests were divided into five categories to ensure the security of the system and optimize the user experience. Valid login tests verified that the system successfully logged in with the correct credentials, while invalid login tests checked that the system returned appropriate error messages when incorrect or incomplete login credentials were entered. Password restriction tests tested that the minimum and maximum character restrictions worked correctly and confirmed that passwords that were too long or too short were rejected by the system. To ensure the security of the system, its resistance to SQL and XSS attacks was also tested. In SQL Injection tests, it was observed that the system also responds as desired when malicious commands are entered into user input fields. Similarly, special characters were filtered to prevent XSS attacks and these inputs were safely rejected by the system. Google OAuth tests verified that the external authentication process works correctly and the login process is completed securely.

2. Application Design & Architecture

2.1 System Architecture

The application is built on a client-server architecture and consists of three main components. The frontend layer includes a web interface created with HTML, CSS and JavaScript that allows users to log in to the system. After the user enters their login information, this data is sent to the backend layer. The backend was developed using the Flask framework and is the main component that performs user authentication and manages database operations. The database layer stores user information using SQLite and provides the necessary data for authentication processes. Google OAuth integration is also positioned as an important component of the system. When users log in with Google authentication, the Flask backend receives an access token from Google using the OAuth protocol. With this token, the user's identity information is verified and the session is started. The system includes measures against SQL Injection and XSS attacks in terms of security.

2.2 UML Diagrams

2.2.1 Class Diagram

As the class diagram shows, the system is built on a Flask server, with various components handling user authentication, session management, and OAuth integration. The user class stores user credentials and an authentication method. The GoogleAuth class manages the Google side of the authentication process. The session class is responsible for managing user sessions, offering methods to start and end user sessions securely. The User_DB class interacts with the database to validate user credentials. The server class serves as the central component that processes login requests, calls the appropriate authentication method, and manages session creation.

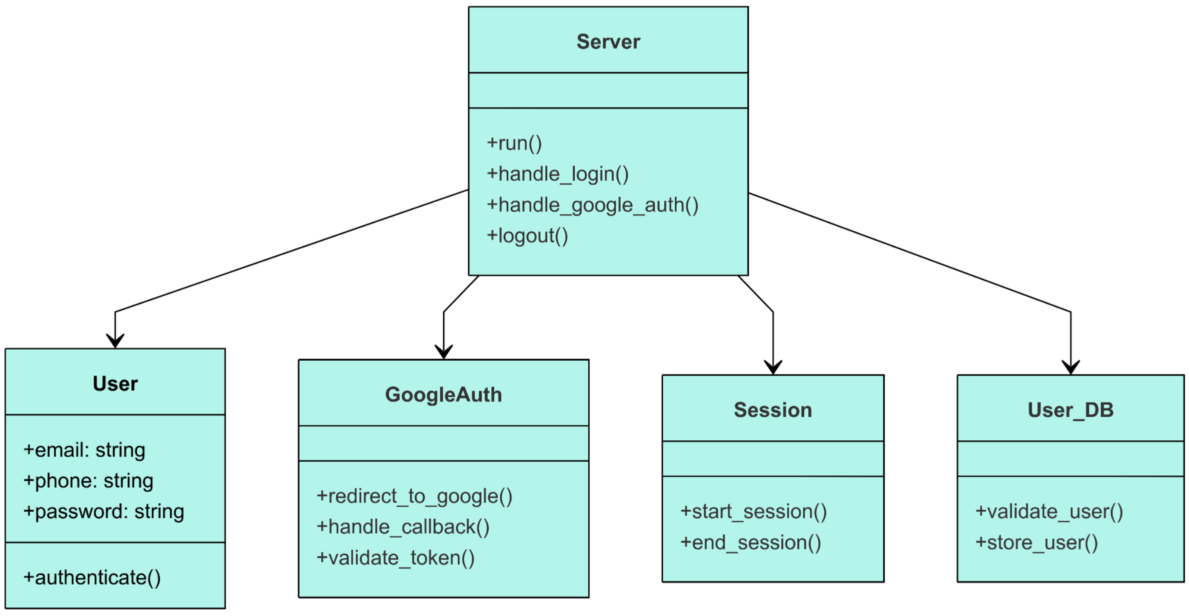


Image 5: Class Diagram

2.2.2 Sequence Diagram

The sequence diagram provides a step-by-step representation of how user login is processed through email/password authentication and Google OAuth. The authentication process begins when the user opens the login page. The browser sends a request to the Flask server to retrieve the login form. Once the form is displayed, the user enters their credentials and submits them to the server. Upon receiving the credentials, the Flask server verifies the user's information by checking the database. If the user credentials are valid, the server creates a session and redirects the user to the welcome page. If the credentials are incorrect, an error message is displayed on the login page, prompting the user to retry. In the case of Google authentication, the user clicks on the "Login with Google" button, and the system redirects them to Google's OAuth authentication page. The user enters their Google account credentials, and Google returns an authentication token to the Flask server. The server then retrieves the user's email and checks if it exists in the system. If the user is registered, authentication is completed, and they are redirected to the welcome page. If the user does not exist in the system, they remain on the login page.

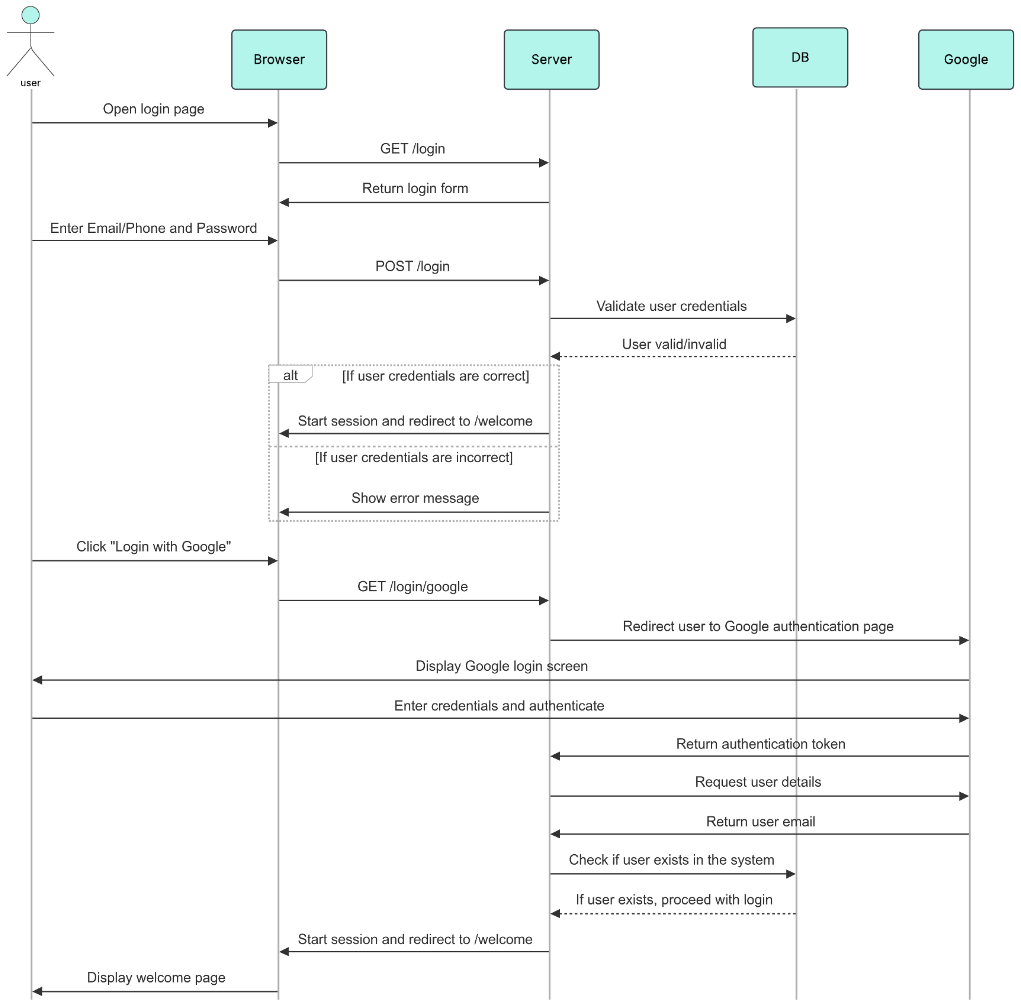

user
Browser
Server
DB
Google
Open login page
GET /login
Return login form
Enter Email/Phone and Password
POST /login
Validate user credentials
User valid/invalid
alt
[If user credentials are correct]
Start session and redirect to /welcome
[If user credentials are incorrect]
Show error message
Click "Login with Google"
GET /login/google
Redirect user to Google authentication page
Display Google login screen
Enter credentials and authenticate
Return authentication token
Request user details
Return user email
Check if user exists in the system
If user exists, proceed with login
Start session and redirect to /welcome
Display welcome page

Image 6: Sequence Diagram

2.2.3 Use Case Diagram

The use case diagram represents the interaction between the user and the authentication system. The user can log in using either email/phone or Google authentication. The login process involves authentication through an email/phone number and password. If the credentials are correct, the user is granted access to the main page. In cases where the login attempt fails due to incorrect credentials, the system displays an error message and allows a retry. Google authentication follows a similar process, where the system redirects the user to Google's OAuth provider. If authentication is successful, the user is logged in; otherwise, they remain logged out. The authentication system also includes an email authentication mechanism that verifies credentials before granting access. The failed login scenario is treated as an extension of the email login use case, ensuring that incorrect authentication attempts are properly handled.

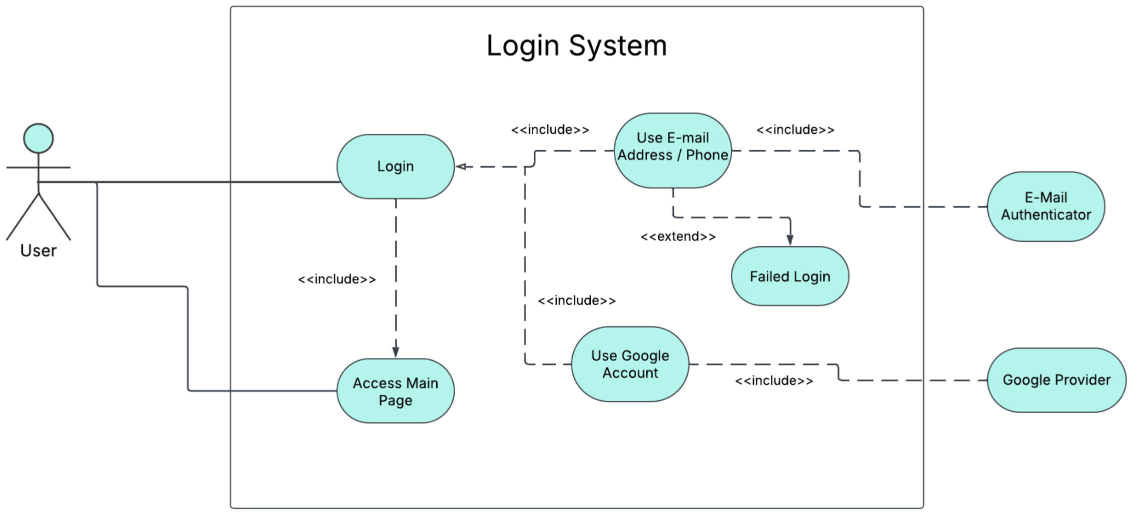


Image 7: Use Case Diagram

2.2.4 State Diagram

The process starts with the user in the logged-out state. If they enter an email or phone number along with a password, the system moves to the authentication state. If the credentials are invalid, the system transitions to the login failed state, from which the user is given the option to retry. When valid credentials are provided, the user transitions to the logged-in state. For users who choose Google authentication, the system enters the Google authentication state. If authentication is successful, the user transitions to the logged-in state. If authentication fails, the user remains in the logged-out state. Once the user is logged in, they can transition back to the logged-out state by selecting the logout option.

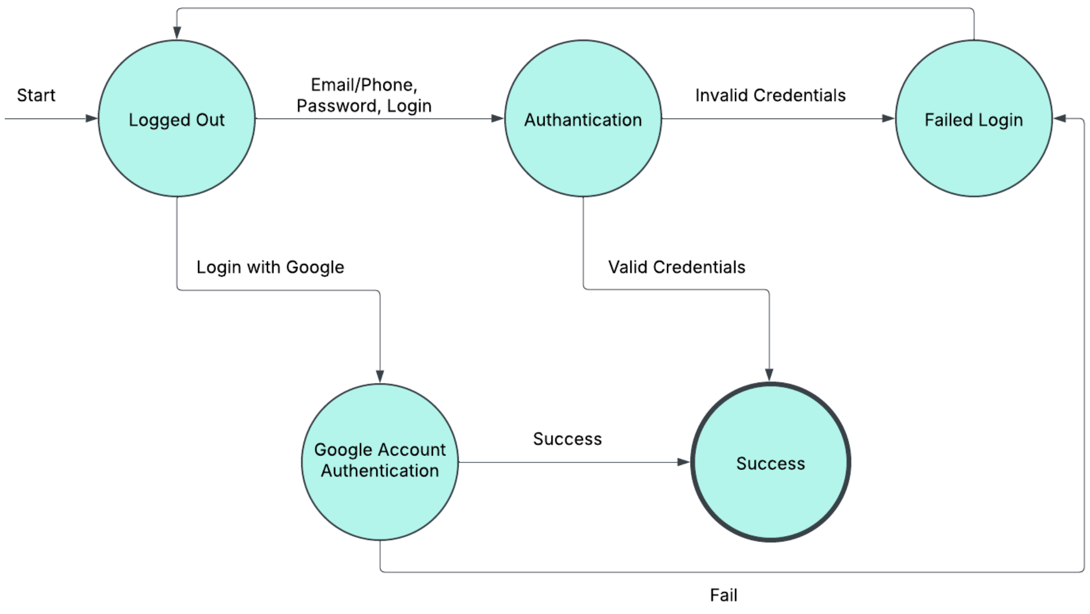
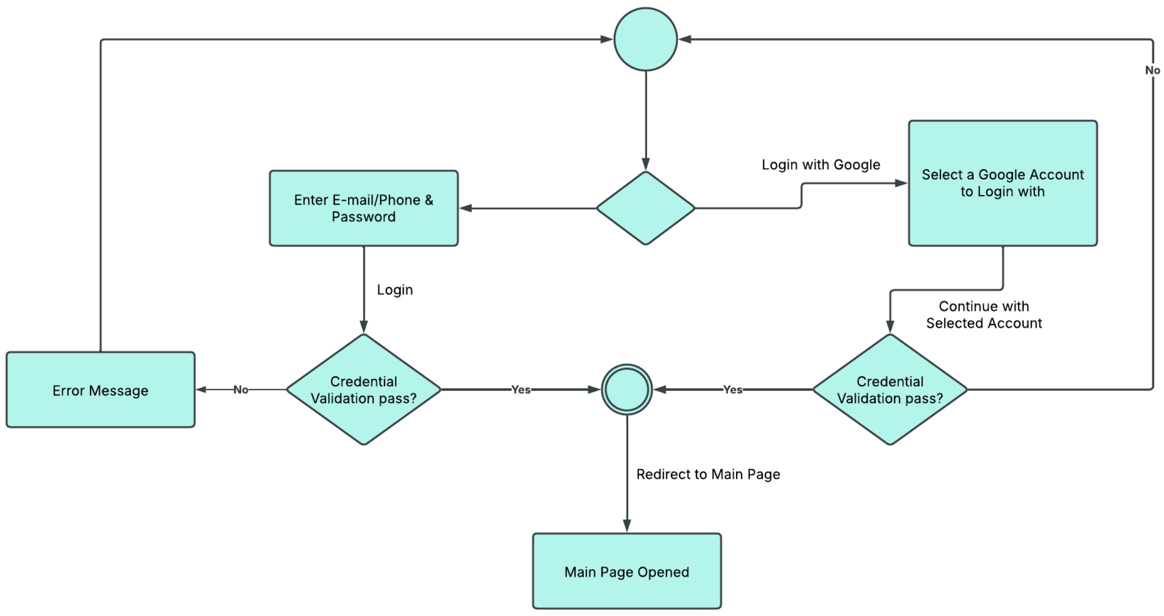


Image 8: State Diagram

2.2.4 Activity Diagram

The activity diagram represents the authentication process in the login system, outlining both traditional email/phone login and Google authentication. The flow starts from the initial state, where the user is presented with two options: logging in using email/phone credentials or selecting Google authentication. If the user opts to enter their email or phone along with a password, the system proceeds to credential validation. If the credentials are correct, the system redirects the user to the main page, indicating a successful login. However, if the validation fails due to incorrect credentials, the system displays an error message, and the user is given a chance to retry. For users choosing to log in via Google, they are redirected to a Google account selection screen. Once the user selects an account and continues, Google validates the credentials. If authentication is successful, the user is redirected to the main page. In case of failure, the user is returned to the initial login screen.



Login with Google
Enter E-mail/Phone & Password
Select a Google Account to Login with
No
Login
Continue with Selected Account
Error Message
No
Credential Validation pass?
Yes
Yes
Credential Validation pass?
Redirect to Main Page
Main Page Opened

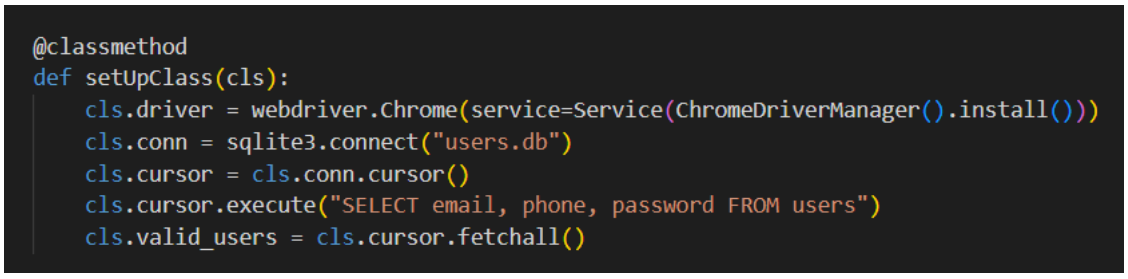
Image 9: Activity Diagram

3. Test Automation & Scenarios

This test code we prepared was created to test the security and accuracy of the user authentication system, which we have developed with Flask. The tests are run using Selenium and login operations are performed based on user information obtained from the SQLite database. **Five different categories** were determined to test how the system behaves in scenarios such as valid logins, invalid logins, password restrictions, SQL and XSS attacks, and Google OAuth login.

At the beginning of the test code, the Selenium WebDriver object was created to open the Chrome browser. A connection to the SQLite database was established and user information to be used in the tests was obtained. Selenium is a testing library used to automate web browsers, and in this project, Selenium's **Chrome WebDriver driver** was used to run the tests. Selenium's main capabilities include **browser automation**, allowing it to open, control, and close web browsers for automated testing. It supports **cross-browser testing**, enabling test execution across different browsers. Selenium provides various **locators** such as XPath, CSS Selector, ID, Name, and Class to accurately identify web elements. It can **simulate user interactions**, including clicking buttons, entering text, handling dropdowns, and keyboard inputs. Also, Selenium supports **headless execution**, allowing tests to run in the background without a visible browser window for improved speed and efficiency. It offers **wait mechanisms** like Implicit, Explicit, and Fluent Wait to handle dynamically loading web elements. Test reporting and **logging features** help capture errors and track execution results efficiently. Selenium also has **multiple window and tab handling**, allowing automated workflows across different browser contexts. It enables **cookie and session management**, which is essential for testing authentication and user sessions. Selenium also integrates with **Appium for mobile testing**, extending its capabilities to automated web application testing on mobile devices.

At this stage, in order to test the system, verifiable login information of multiple users should have been loaded into the test environment, but since only one user was added to our database for this project, only that user was loaded. Although we used a single user while performing the tests, we created a database to make the code extensible & maintainable, and valid login tests were performed for each user taken from this database, and invalid login scenarios were simulated.

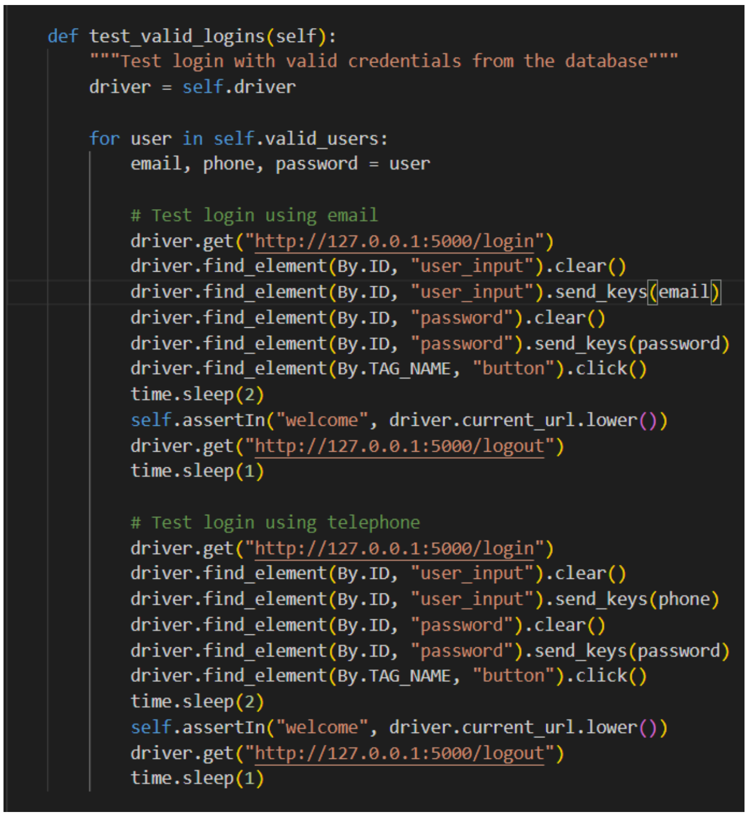


```python
@classmethod
def setUpClass(cls):
    cls.driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
    cls.conn = sqlite3.connect("users.db")
    cls.cursor = cls.conn.cursor()
    cls.cursor.execute("SELECT email, phone, password FROM users")
    cls.valid_users = cls.cursor.fetchall()
```

Image 10: Initial actions before the tests

3.1 Valid Entry Scenario

This test was written to verify that the system works successfully with the correct username and password combinations. Users should be able to log in with both email and phone number. The test first logs in with email and password, then repeats the same process with the phone number.



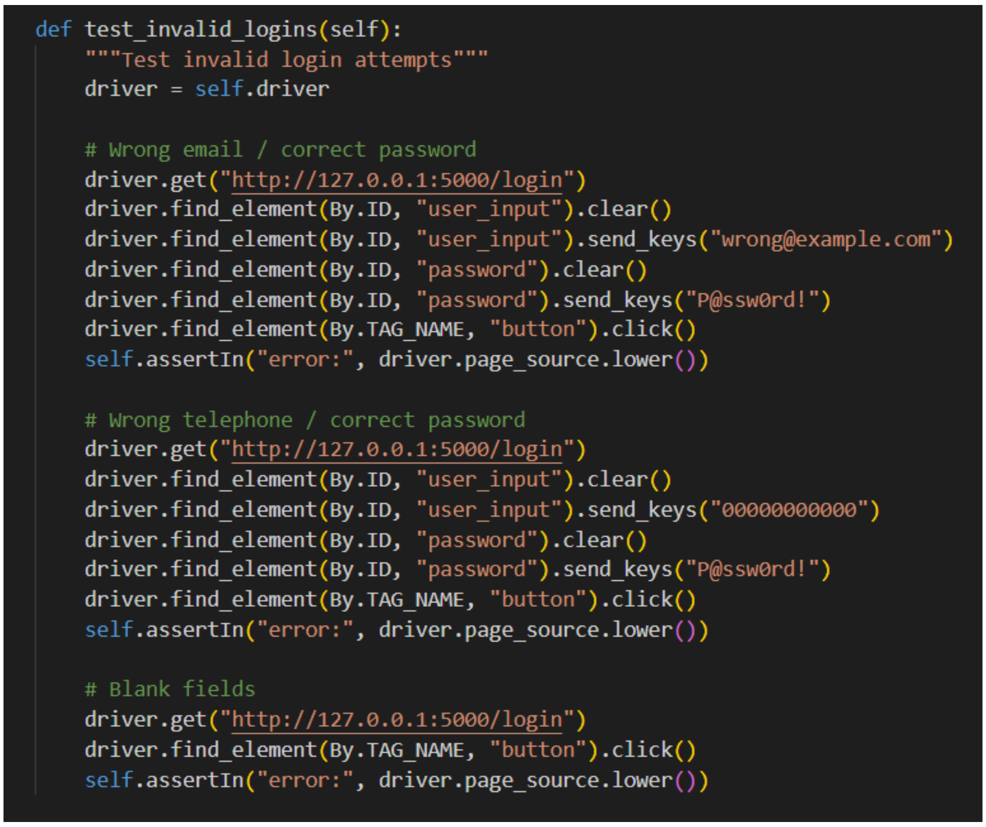
```python
def test_valid_logins(self):
    """Test login with valid credentials from the database"""
    driver = self.driver

    for user in self.valid_users:
        email, phone, password = user

        # Test login using email
        driver.get("http://127.0.0.1:5000/login")
        driver.find_element(By.ID, "user_input").clear()
        driver.find_element(By.ID, "user_input").send_keys(email)
        driver.find_element(By.ID, "password").clear()
        driver.find_element(By.ID, "password").send_keys(password)
        driver.find_element(By.TAG_NAME, "button").click()
        time.sleep(2)
        self.assertIn("welcome", driver.current_url.lower())
        driver.get("http://127.0.0.1:5000/logout")
        time.sleep(1)

        # Test login using telephone
        driver.get("http://127.0.0.1:5000/login")
        driver.find_element(By.ID, "user_input").clear()
        driver.find_element(By.ID, "user_input").send_keys(phone)
        driver.find_element(By.ID, "password").clear()
        driver.find_element(By.ID, "password").send_keys(password)
        driver.find_element(By.TAG_NAME, "button").click()
        time.sleep(2)
        self.assertIn("welcome", driver.current_url.lower())
        driver.get("http://127.0.0.1:5000/logout")
        time.sleep(1)
```

Image 11: Valid Entry Tests

In a successful login scenario, the user should be redirected to the "/welcome" page. The test checks the URL to verify that the login was completed successfully.

3.2 Invalid Entry Scenario

This test was written to verify that the system returned appropriate error messages when incorrect login information was entered. Login attempts were made with the correct password with the incorrect email, the correct password with the incorrect phone number, and with blank fields.



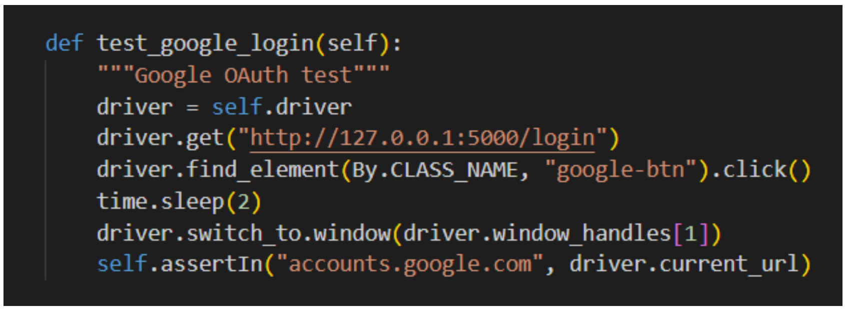
```python
def test_invalid_logins(self):
    """Test invalid login attempts"""
    driver = self.driver

    # Wrong email / correct password
    driver.get("http://127.0.0.1:5000/login")
    driver.find_element(By.ID, "user_input").clear()
    driver.find_element(By.ID, "user_input").send_keys("wrong@example.com")
    driver.find_element(By.ID, "password").clear()
    driver.find_element(By.ID, "password").send_keys("P@ssw0rd!")
    driver.find_element(By.TAG_NAME, "button").click()
    self.assertIn("error:", driver.page_source.lower())

    # Wrong telephone / correct password
    driver.get("http://127.0.0.1:5000/login")
    driver.find_element(By.ID, "user_input").clear()
    driver.find_element(By.ID, "user_input").send_keys("00000000000")
    driver.find_element(By.ID, "password").clear()
    driver.find_element(By.ID, "password").send_keys("P@ssw0rd!")
    driver.find_element(By.TAG_NAME, "button").click()
    self.assertIn("error:", driver.page_source.lower())

    # Blank fields
    driver.get("http://127.0.0.1:5000/login")
    driver.find_element(By.TAG_NAME, "button").click()
    self.assertIn("error:", driver.page_source.lower())
```

Image 12: Invalid Entry Tests

Error messages should appear on the screen as expected. This has tested the system to be safe from incorrect inputs.

3.3 Google OAuth Entry Scenario

This test was written to verify that Google OAuth authentication is working properly in the system. The purpose of the test is to verify that when the user clicks the Google Sign In button, a new window opens and the user is directed to the Google account login page.



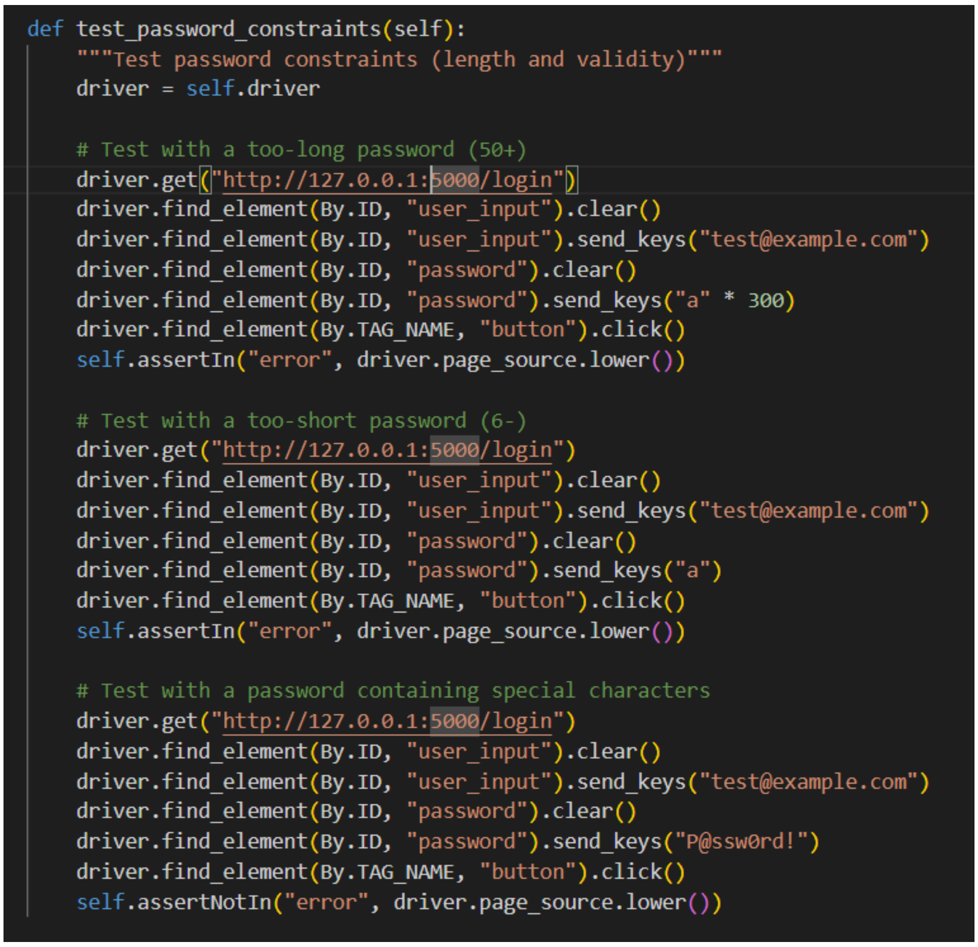
```python
def test_google_login(self):
    """Google OAuth test"""
    driver = self.driver
    driver.get("http://127.0.0.1:5000/login")
    driver.find_element(By.CLASS_NAME, "google-btn").click()
    time.sleep(2)
    driver.switch_to.window(driver.window_handles[1])
    self.assertIn("accounts.google.com", driver.current_url)
```

Image 13: Google OAuth Test

This test is required to verify that the system's Google OAuth integration is working properly. If the redirect to the Google login page fails, the new window never opens, or the URL is not as expected, the test will fail.

3.4 Password Validation Scenario

This test is required to verify that the system's Google OAuth integration is working properly. If the redirect to the Google login page fails, the new window never opens, or the URL is not as expected, the test will fail.



```python
def test_password_constraints(self):
    """Test password constraints (length and validity)"""
    driver = self.driver

    # Test with a too-long password (50+)
    driver.get("http://127.0.0.1:5000/login")
    driver.find_element(By.ID, "user_input").clear()
    driver.find_element(By.ID, "user_input").send_keys("test@example.com")
    driver.find_element(By.ID, "password").clear()
    driver.find_element(By.ID, "password").send_keys("a" * 300)
    driver.find_element(By.TAG_NAME, "button").click()
    self.assertIn("error", driver.page_source.lower())

    # Test with a too-short password (6-)
    driver.get("http://127.0.0.1:5000/login")
    driver.find_element(By.ID, "user_input").clear()
    driver.find_element(By.ID, "user_input").send_keys("test@example.com")
    driver.find_element(By.ID, "password").clear()
    driver.find_element(By.ID, "password").send_keys("a")
    driver.find_element(By.TAG_NAME, "button").click()
    self.assertIn("error", driver.page_source.lower())

    # Test with a password containing special characters
    driver.get("http://127.0.0.1:5000/login")
    driver.find_element(By.ID, "user_input").clear()
    driver.find_element(By.ID, "user_input").send_keys("test@example.com")
    driver.find_element(By.ID, "password").clear()
    driver.find_element(By.ID, "password").send_keys("P@ssw0rd!")
    driver.find_element(By.TAG_NAME, "button").click()
    self.assertNotIn("error", driver.page_source.lower())
```

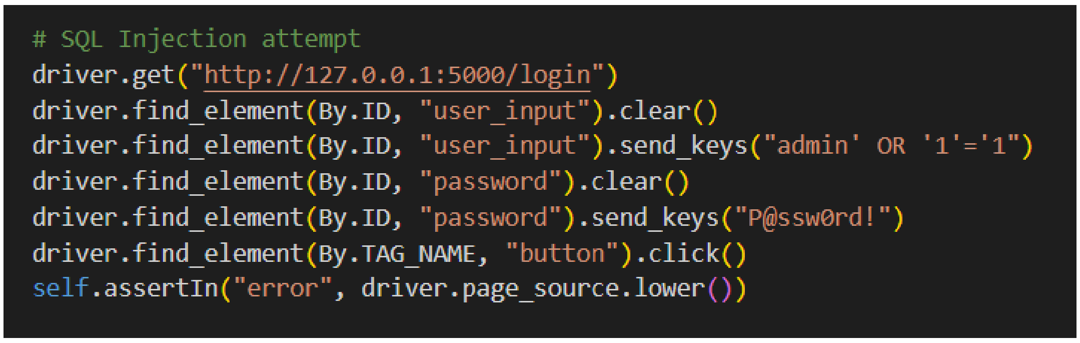
Image 14: Password Validation Tests

This test function shows that the system is correctly enforcing password length and content restrictions. Error messages are expected for excessively long and short passwords, but strong passwords containing special characters are expected to be accepted. A successful test shows that the system complies with security policies and protects users from incorrect password entries.

3.5 Injection Attacks Scenario

This test scenario was created to verify whether the system is secure against SQL Injection and Cross-Site Scripting attacks. SQL Injection is a security vulnerability that allows malicious users to gain unauthorized access by manipulating SQL queries sent to the database. With such attacks, unauthorized access can be gained in the system, critical information in the database can be stolen, or damaging operations can be performed. Cross-Site Scripting, or XSS, is a security vulnerability that relies on attackers injecting malicious JavaScript codes into a website and causing these codes to run in the user's browser. XSS attacks are usually carried out by entering malicious code such as `<script>alert('XSS')</script>` into a login form or URL parameter. If the web application does not filter such inputs, the browser executes this code and the attacker can steal the user's cookie information, hijack their sessions, or upload malicious content. This test function measures how the system responds to these attacks by performing malicious login attempts.

First, the SQL Injection attack is tested. Normally, the system runs a query like `SELECT * FROM users WHERE username='admin' AND password='P@ssw0rd!';` to check if a user is registered in the database. If the username and password match, the login is successful. However, when the attacker enters a special entry like `admin' OR '1'='1'` in the login field, the logic of the query changes and becomes: `SELECT * FROM users WHERE username='admin' OR '1'='1' AND password='P@ssw0rd!';`. Since `OR '1'='1'` is always true, the system then grants the user access without checking the password. In this way, the attacker can log in to the system even if it is not a valid user account.

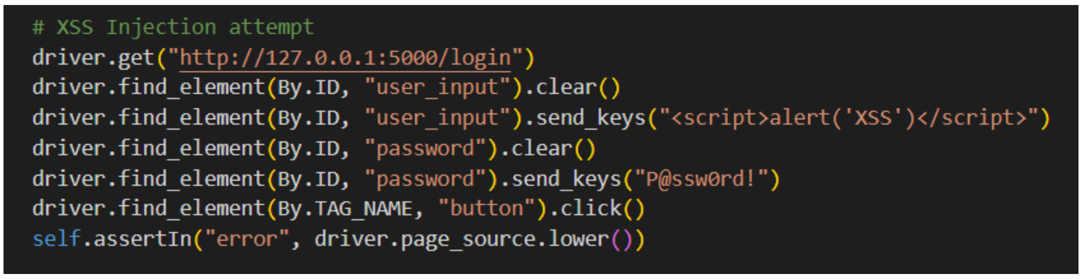


```
# SQL Injection attempt
driver.get("http://127.0.0.1:5000/login")
driver.find_element(By.ID, "user_input").clear()
driver.find_element(By.ID, "user_input").send_keys("admin' OR '1'='1")
driver.find_element(By.ID, "password").clear()
driver.find_element(By.ID, "password").send_keys("P@ssw0rd!")
driver.find_element(By.TAG_NAME, "button").click()
self.assertIn("error", driver.page_source.lower())
```

Image 15: SQL Injection Test

This test checks whether the system is resistant to SQL Injection attacks. The username field in the login form is entered as "admin' OR '1'='1" as mentioned in the previous example. If the system is not secure, it is possible for this login to be successful and for the attacker to gain unauthorized access. If an error message is found, it is assumed that the system has prevented the attack. Otherwise, the test fails and the system is vulnerable to SQL Injection attacks.

This test also checks whether the web application provides protection against XSS attacks or not. The `<script>alert('XSS')</script>` expression is entered in the username field in the login form. This expression is used to test whether the system processes user input directly in HTML. If the system does not filter user input, the browser will open an "alert" window by running this code. If the system detects the attack and returns an error message, the test is considered successful and it is verified that the security measures against XSS attacks are working. If no error message is found and the code written by the attacker is run in the browser, it is understood that the system is vulnerable to XSS attacks.



```python
# XSS Injection attempt
driver.get("http://127.0.0.1:5000/login")
driver.find_element(By.ID, "user_input").clear()
driver.find_element(By.ID, "user_input").send_keys("<script>alert('XSS')</script>")
driver.find_element(By.ID, "password").clear()
driver.find_element(By.ID, "password").send_keys("P@ssw0rd!")
driver.find_element(By.TAG_NAME, "button").click()
self.assertIn("error", driver.page_source.lower())
```

Image 16: XSS Injection Test

4. Evaluation on Test Automation

Automation testing assures us of the reliability and safety of the authentication system we have created. With automation testing, we could effectively test the functionality of our login features and reveal any issues early in the development cycle. Using Selenium specifically allowed us to simulate real user interaction with the web application, providing us with a comprehensive assessment of the system's behavior under different conditions. Thanks to test automation, we were able to execute a large number of tests quickly and consecutively. Additionally, testing automation provided a reliable way of verifying the system's security features, such as SQL Injection and XSS attack prevention. Our test cases covered a wide range of scenarios, valid and invalid logins, password policy, and Google OAuth integration. Automating these tests enabled us to verify the system behavior in a reproducible and timely manner. This not only improved the quality of the software as a whole but also gave us greater assurance of its robustness and security. Furthermore, with a database-based approach to testing, it was simple to maintain and update the test data, and the tests were more scalable and maintainable. Seeding the database with test users and testing their credentials with the system gave our tests a realistic and representative feel of real users interacting with the system. Overall, the utilization of test automation improved our development process in terms of obtaining rapid feedback about the system's functionality and security. It enabled us to identify and resolve issues early, reducing the chance of defects in the final product and ensuring a high level of quality for our authentication system.

5. Test Automation and SDLC Impacts

Automation testing has a significant impact on velocity and quality in the Software Development Life Cycle. Including automated testing as a component of our development cycle allowed us to gain multiple salient advantages, playing a vital role in making the project successful.

- **Time Advantage & Speed:** Automated tests helped us easily control the changes made in the codebase, and testing time was reduced compared to manual testing. These tests allowed quick iteration and integration while we clearly saw their changes reflected instantaneously in how they affected the system and quickly fixed faults so that they could proceed in an assured way.

- **Early Defect Detection:** When we run automated tests via the continuous integration pipeline, we were able to detect and resolve defects in the initial stages of the development process. This prevented us from fixing unexpected faults at the end of the process and led us to solve problems while they were still superficial and without any technical debt. While this was a relatively simple project, in the case where a more complicated project is required, an early defect detection system would have aided us in maintaining a much higher code quality.

- **Scalability & Maintainability:** Automated tests would also make it easier to scale the testing if the project grows in later stages. Additional test cases could be added easily to the existing suite without significantly increasing the testing expenses. Automated tests were also easier to maintain compared to manual tests since they could be updated and executed with less effort.

6. Conclusion

In short, test automation greatly enhanced our development and testing of our authentication system. With the help of Selenium and other testing tools, we were able to efficiently test the functionality and security of our login functionalities, such that the system met the highest standards of quality and reliability. Automated tests allowed us to easily identify and fix problems immediately, reducing the likelihood of defects and vulnerabilities on the end product. The strong test coverage that our automated tests provided us with allowed us to have confidence in the security and resilience of the system.