



BILKENT UNIVERSITY

CS 458 - Software Verification And Validation

Section 2

Project 1 Report

08/02/2025

Muhammed Shayan Usman	22101343
Emir Kerem Şahin	22101882
Begüm Kunaç	22103838
Gün Taştan	22101850

Introduction.....	2
Selenium webdriver.....	2
Main Capabilities of Selenium.....	2
Automation Experience.....	3
Diagrams.....	4
Activity Diagram.....	4
State Diagram.....	5
Use-case Diagram.....	6
Sequence Diagram.....	7
Class Diagram.....	8
Test cases (Description + screenshots).....	9
Test Case 1: Login with Valid/Invalid Credentials.....	9
Test Case 2: Login with Empty Fields.....	11
Test Case 3: Login with Incorrect Input Format.....	12
Test Case 4: Third-Party Authentication Availability.....	13
Test Case 5: Login Persistence and Network Handling.....	14
Test automation in SDLC.....	18

Introduction

Test automation is one of the important areas in modern software development for ensuring the reliability, functionality, and performance of web applications. Web login automation is done using Selenium WebDriver in this project, which is among the well-known tools used for browser automation. The main objective is to automate the login functionality, check for different authentication scenarios, and identify the significance of automation in the Software Development Life Cycle (SDLC).

The project involves the implementation and testing of a web login system that accepts authentication using email and password, as well as third-party authentication using Google and Facebook login. The system validates whether the login is successful or whether it should throw an error message. Five test cases have been created to verify different functionalities of the login feature, including valid and invalid credential processing, empty fields, incorrect email formats, third-party authentication, network interruption and session persistence.

To automate these test cases, a test suite has been developed using Python and Selenium WebDriver to replicate user interaction in a smooth manner. The automated scripts execute different login scenarios, verifying system action under various conditions. The project also explores how test automation is advantageous to the SDLC by enhancing testing velocity, accuracy, efficiency and reduction of manual effort.

As part of documentation, different UML diagrams, such as Activity, State, Use-case, Sequence, and Class diagrams, are included to illustrate the workflow, interactions, and logical arrangement of the login system. The report also contains a description of the experience of automation, including insights, issues encountered (e.g., network processing and form validation), and the implemented solutions.

Using Selenium WebDriver, the project aims to demonstrate how significant automated testing is in modern software development and how it facilitates achieving high-quality, errorless software with effectiveness.

Selenium webdriver

Selenium is an open-source tool that automates web browsers, enabling developers and testers to simulate user interactions for web applications. It supports multiple programming languages, including Python, Java, and JavaScript, and is widely used for functional and regression testing.

Main Capabilities of Selenium

1. **Cross-Browser Testing** – Selenium supports multiple browsers like Chrome, Firefox, Edge, and Safari, ensuring compatibility across different environments.
2. **Multi-Language Support** – It provides bindings for various languages such as Python, Java, C#, and JavaScript, allowing flexibility in test automation.

3. **Headless Execution** – Tests can run in headless mode (the way we did it for our login app) without opening a visible browser window, making automation faster and suitable for CI/CD pipelines.
4. **Element Locators** – Selenium offers powerful locators such as ID, class name, name attribute, XPath, and CSS selectors to interact with web elements. We use name attribute to find components in our login app
5. **Handling Dynamic Elements** – It supports explicit and implicit waits to manage dynamic elements that load asynchronously. We used
6. **Keyboard and Mouse Actions** – With Selenium's Action API, users can simulate keypresses, mouse clicks, drag-and-drop, and hover actions.
7. **Integration with Testing Frameworks** – Selenium integrates with popular testing frameworks like JUnit, TestNG, and PyTest for efficient test execution and reporting.
8. **Parallel Test Execution** – Selenium Grid enables running tests on multiple browsers and machines simultaneously, reducing execution time.
9. **Support for Mobile Testing** – Using Appium, Selenium can automate mobile browsers and applications on Android and iOS devices.
10. **Capture Screenshots and Logs** – It can take screenshots on test failures and generate logs to help debug errors.

Selenium's versatility and broad feature set make it clear why it is a leading choice for web automation testing.

Automation Experience

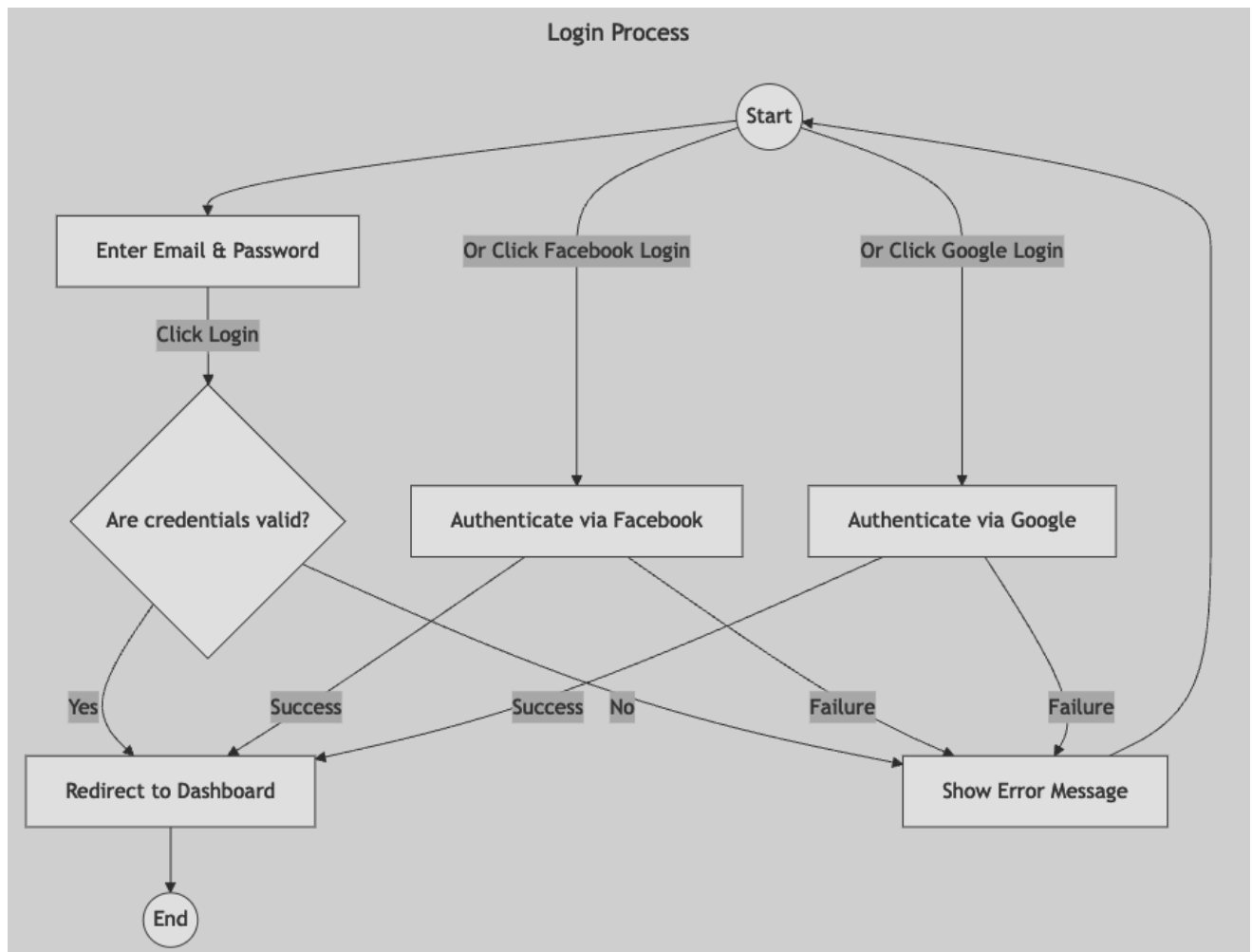
Automating the login functionality using Selenium provided valuable insights into web testing and error handling. The test suite I developed covered various scenarios, including valid and invalid logins, form validation, and external authentication via Google and Facebook.

One of the most challenging aspects was handling network disconnection scenarios. Initially, my test expected a popup message to appear when the internet was disconnected, but it failed because the application did not handle this case explicitly. To resolve this, I added an event listener in the React component to detect network loss and display a popup. This experience highlighted the importance of both frontend and test automation working together for reliable behavior.

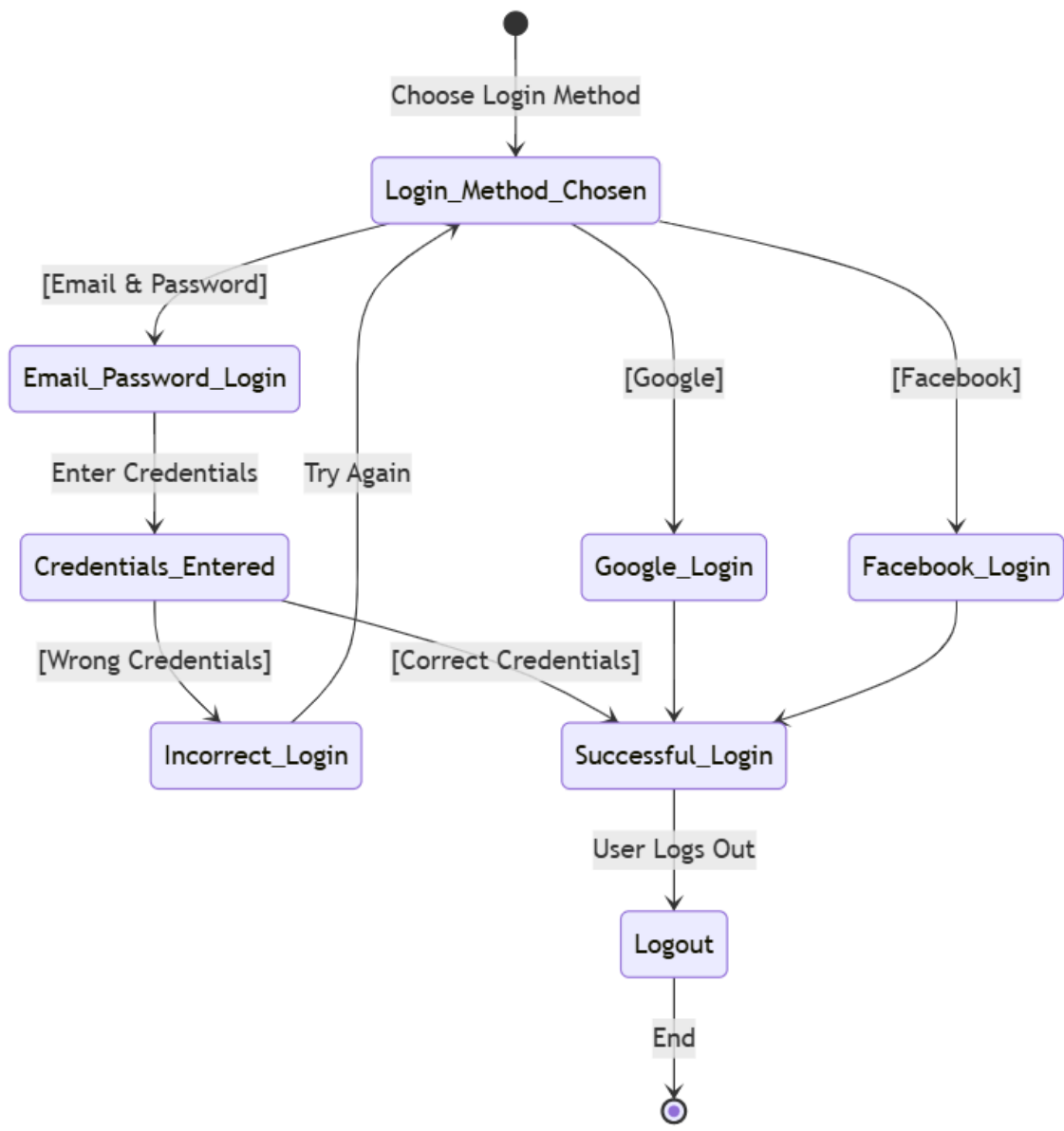
Writing automated tests also reinforced my understanding of form validation. The test for an incorrectly formatted email initially failed because the browser's HTML5 validation blocked submission instead of triggering a popup. I modified the test to check the `validity.valid` property, which correctly verified that the input was invalid.

Diagrams

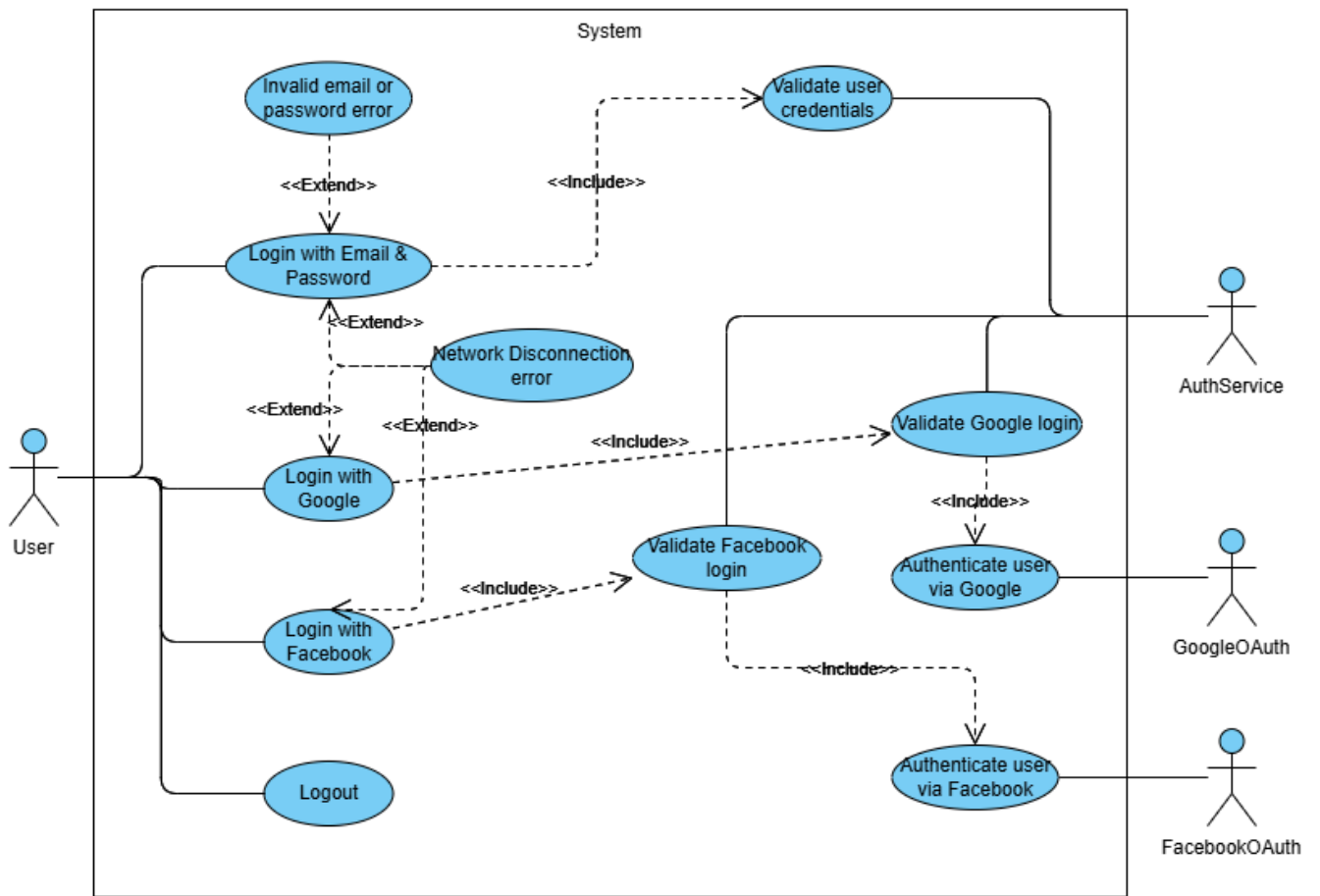
Activity Diagram



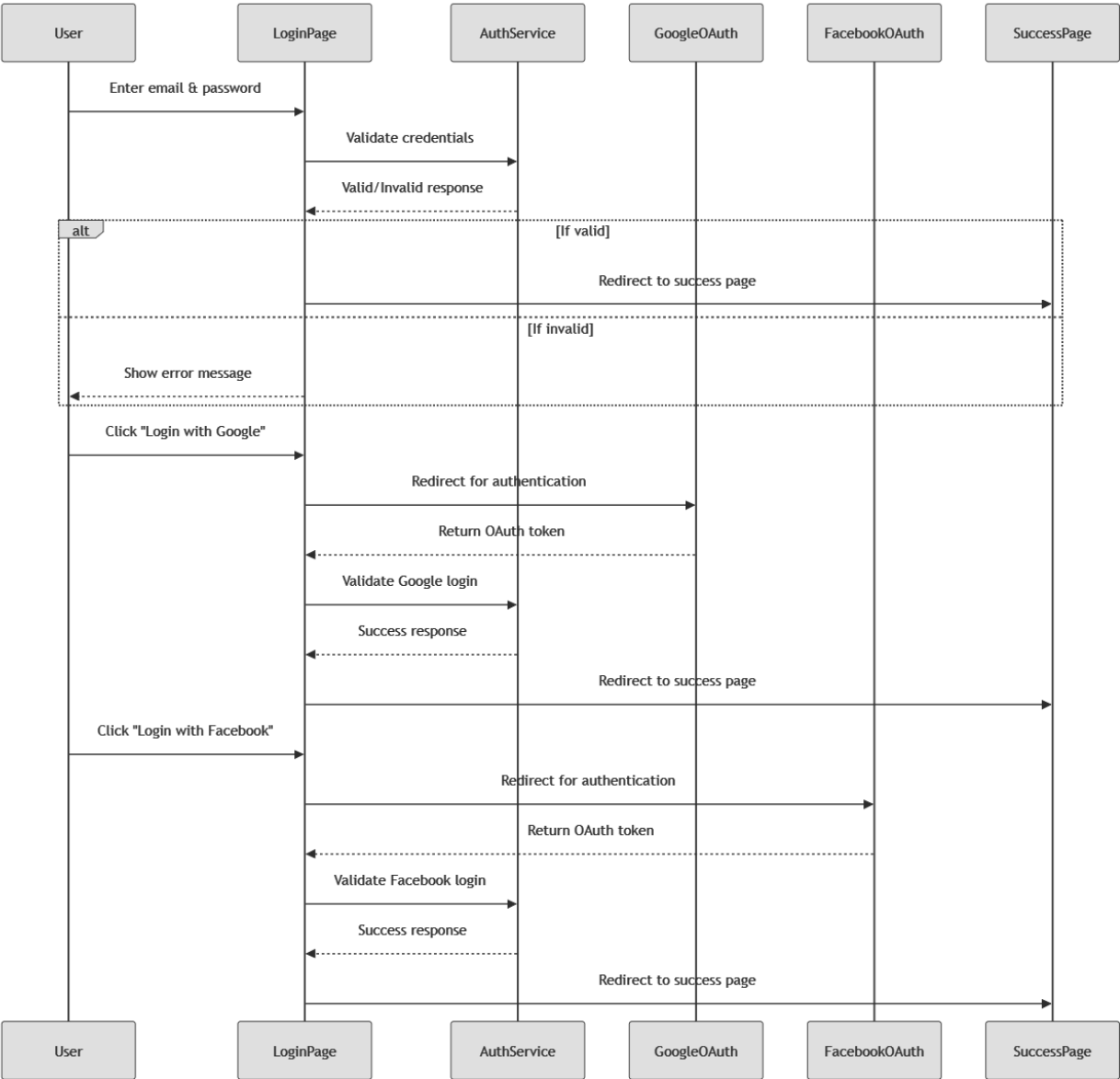
State Diagram



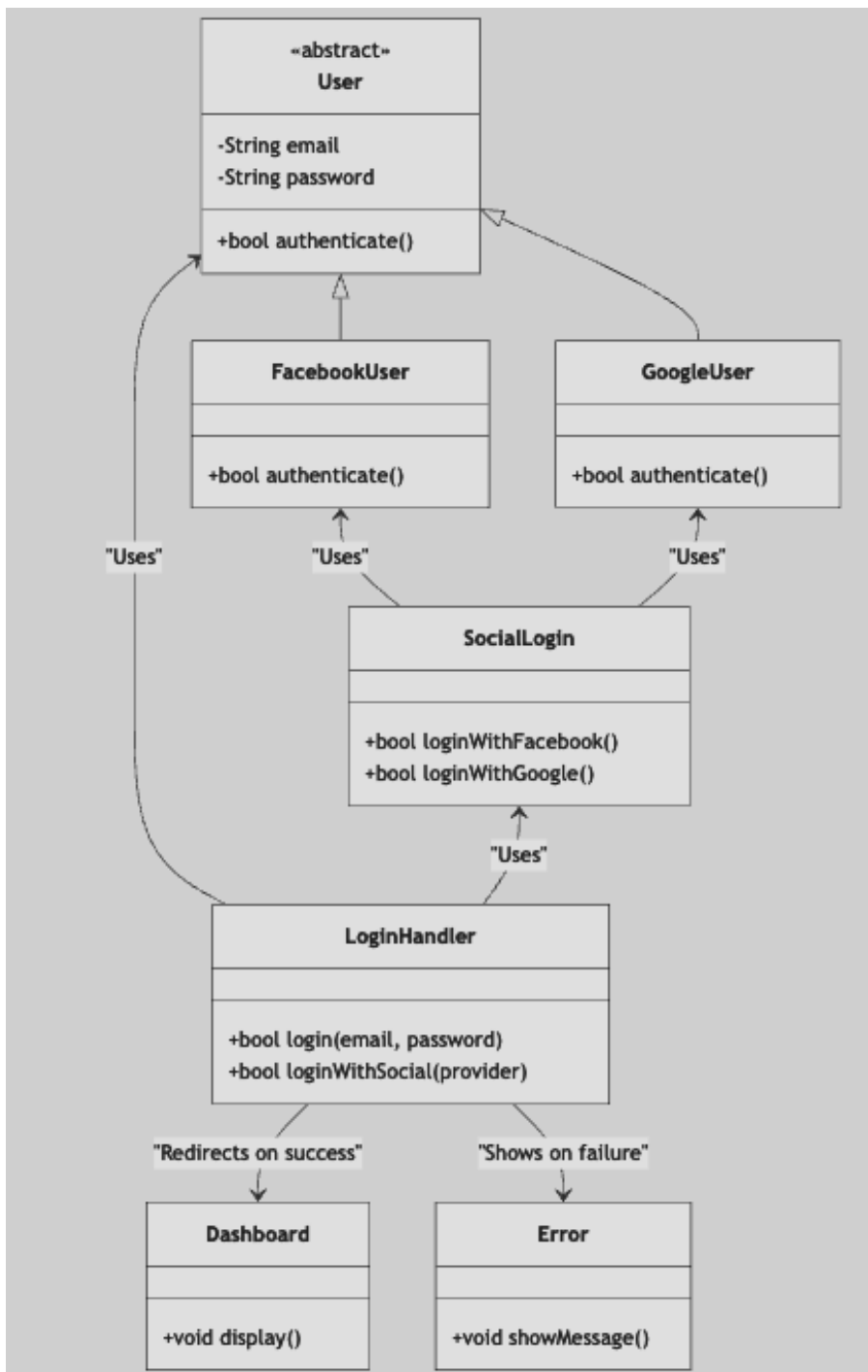
Use-case Diagram



Sequence Diagram



Class Diagram



Test cases (Description + screenshots)

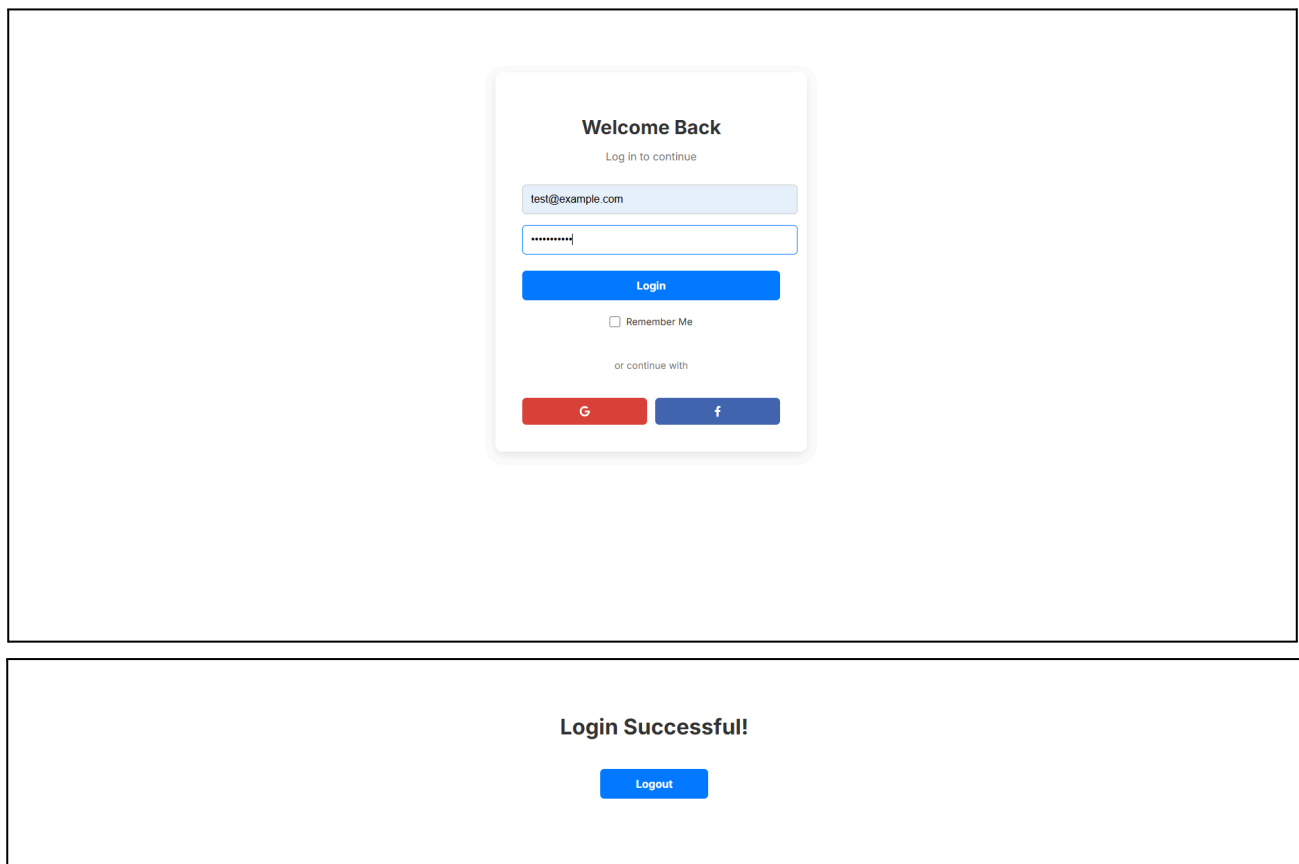
Test Case 1: Login with Valid/Invalid Credentials

1. Login with Valid Credentials

This test case verifies that the user can login with valid credentials.

- Write the valid input (email: test@example.com, password: password123)
- Click the login button
- Get redirected to the “success” page
- For resetting the session, logout.

Expected Result: The user is successfully logged in and redirected to /success.



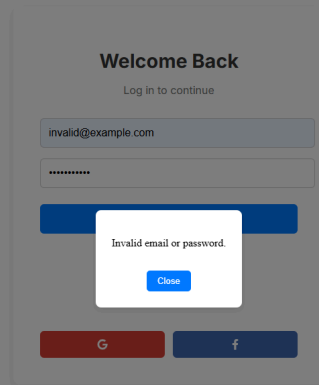
The first screenshot displays a login interface titled "Welcome Back" with the subtitle "Log in to continue". It features an email input field containing "test@example.com", a password input field with masked characters "password123", a blue "Login" button, a "Remember Me" checkbox, and social login options for Google (G) and Facebook (f). The second screenshot shows a confirmation message "Login Successful!" and a blue "Logout" button.

2. Login with Invalid Credentials

This test case verifies that the user cannot login with invalid credentials.

- Write an invalid input (email: invalid@example.com, password: wrongpass)
- Click the login button
- Verify that “Invalid email or password” popup message is displayed
- Close the popup message

Expected Result: The user is kept on the login page. “Invalid” error popup message is displayed and can be closed.



```
def test_valid_login(self): # TEST CASE 1.1
    """Test valid login redirects to /success and then logs out"""
    driver = self.driver

    WebDriverWait(driver,
15).until(EC.presence_of_element_located((By.TAG_NAME, "form")))

    email_input = driver.find_element(By.NAME, "email")
    password_input = driver.find_element(By.NAME, "password")
    login_button = driver.find_element(By.NAME, "login-button")

    email_input.clear()
    password_input.clear()
    email_input.send_keys("test@example.com")
    password_input.send_keys("password123")
    login_button.click()

    WebDriverWait(driver, 10).until(EC.url_contains("success"))
    self.assertIn("success", driver.current_url.lower())

    # Logout to reset session before next test
    self.logout()

def test_invalid_login(self): # TEST CASE 1.2
    """Test invalid login shows error message"""
    driver = self.driver

    WebDriverWait(driver,
15).until(EC.presence_of_element_located((By.TAG_NAME, "form")))

    email_input = driver.find_element(By.NAME, "email")
    password_input = driver.find_element(By.NAME, "password")
    login_button = driver.find_element(By.NAME, "login-button")

    email_input.clear()
    password_input.clear()
```

```

email_input.send_keys("invalid@example.com")
password_input.send_keys("wrongpass")
login_button.click()

# Verify popup appears
popup_container = WebDriverWait(driver, 10).until(
    EC.visibility_of_element_located((By.NAME, "popup-container"))
)
popup_message = driver.find_element(By.NAME, "popup-message").text
self.assertIn("Invalid", popup_message)

# Close popup
close_button = driver.find_element(By.NAME, "popup-close")
close_button.click()

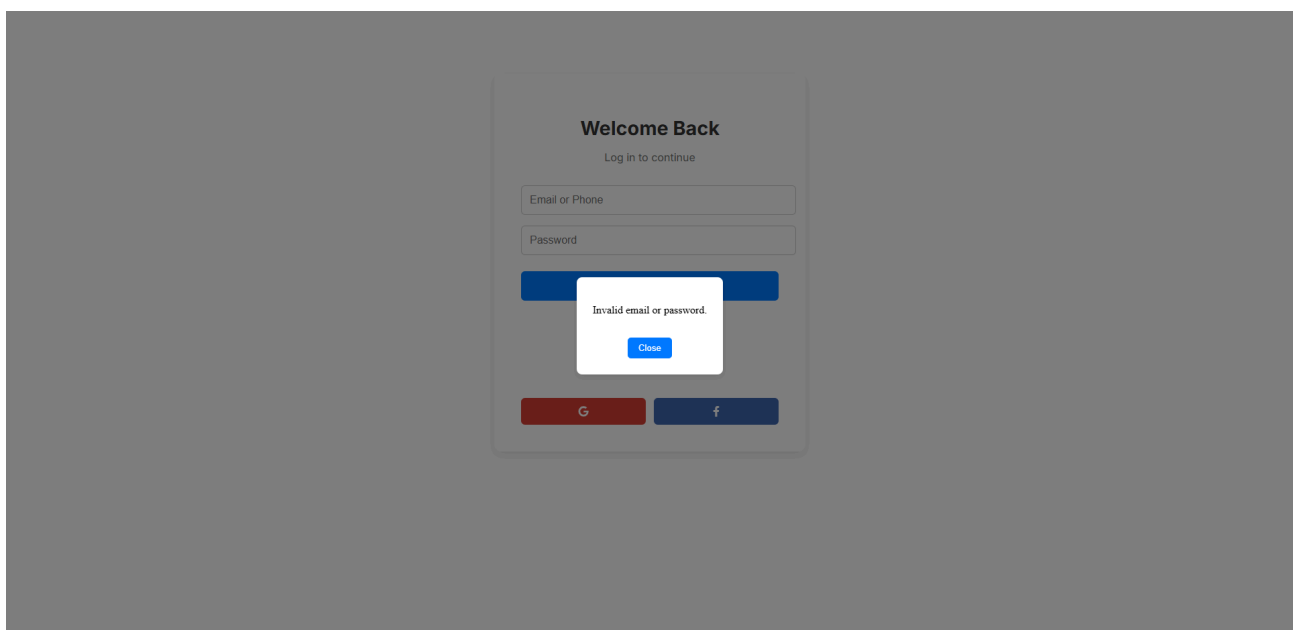
```

Test Case 2: Login with Empty Fields

This test case verifies that the user cannot login with empty fields.

- Click the login button (without providing input to the email and password fields)
- Wait for the error popup message
- Verify that “Invalid email or password” popup message is displayed
- Close the popup message

Expected Result: The user is kept on the login page. “Invalid” error popup message is displayed and can be closed.



```

def test_empty_fields(self): # TEST CASE 2
    """Test login with empty fields should not proceed"""
    driver = self.driver

    WebDriverWait(driver,
15).until(EC.presence_of_element_located((By.TAG_NAME, "form")))

    login_button = driver.find_element(By.NAME, "login-button")
    login_button.click()

```

```

# Verify popup appears
popup_container = WebDriverWait(driver, 10).until(
    EC.visibility_of_element_located((By.NAME, "popup-container"))
)
popup_message = driver.find_element(By.NAME, "popup-message").text
self.assertIn("Invalid", popup_message)

# Close popup
close_button = driver.find_element(By.NAME, "popup-close")
close_button.click()

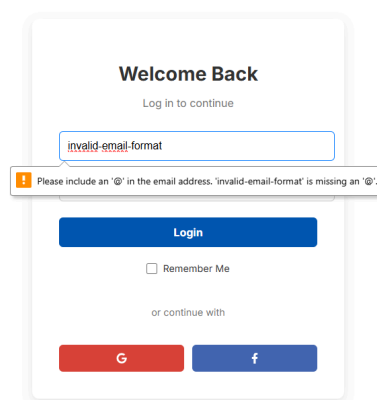
```

Test Case 3: Login with Incorrect Input Format

This test case verifies that incorrectly formatted email addresses are not accepted.

- Write email in incorrect format (email: invalid-email-format, missing the @ symbol, password: password123)
- Click the login button
- Wait for the error popup message
- Verify that “Invalid email or password” popup message is displayed
- Close the popup message

Expected Result: The user is kept on the login page. “Invalid” error popup message is displayed and can be closed.



```

def test_invalid_email_format(self):
    """Test login with an incorrectly formatted email and verify the form
    prevents submission."""
    driver = self.driver

    WebDriverWait(driver, 15).until(
        EC.presence_of_element_located((By.TAG_NAME, "form"))
    )

    email_input = driver.find_element(By.NAME, "email")
    password_input = driver.find_element(By.NAME, "password")
    login_button = driver.find_element(By.NAME, "login-button")

    email_input.clear()

```

```

password_input.clear()
email_input.send_keys("invalid-email-format") # No @ symbol
password_input.send_keys("password123")

# Try clicking login
login_button.click()

# Use JavaScript to check if form submission was prevented due to email
format
is_valid = driver.execute_script("return arguments[0].validity.valid;",
email_input)
self.assertFalse(is_valid, "Email field should be invalid")

# Check for the validation message
validation_message = driver.execute_script("return
arguments[0].validationMessage;", email_input)
self.assertTrue(validation_message, "Expected a validation message for an
invalid email.")

```

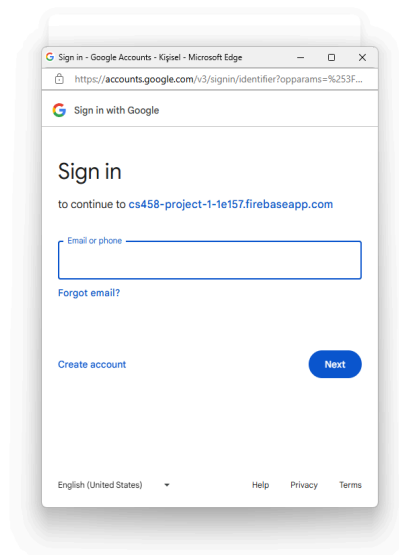
Test Case 4: Third-Party Authentication Availability

1. “Login with Google” Button Presence

This test case verifies that the “Login with Google” button is visible and clickable.

- Verify “Login with Google” button is displayed
- Click on “Login with Google” button
- Wait for the Google login popup window

Expected Result: “Login with Google” button is displayed and when clicked, the Google login popup window is displayed for the user to be able to login with their Google account.



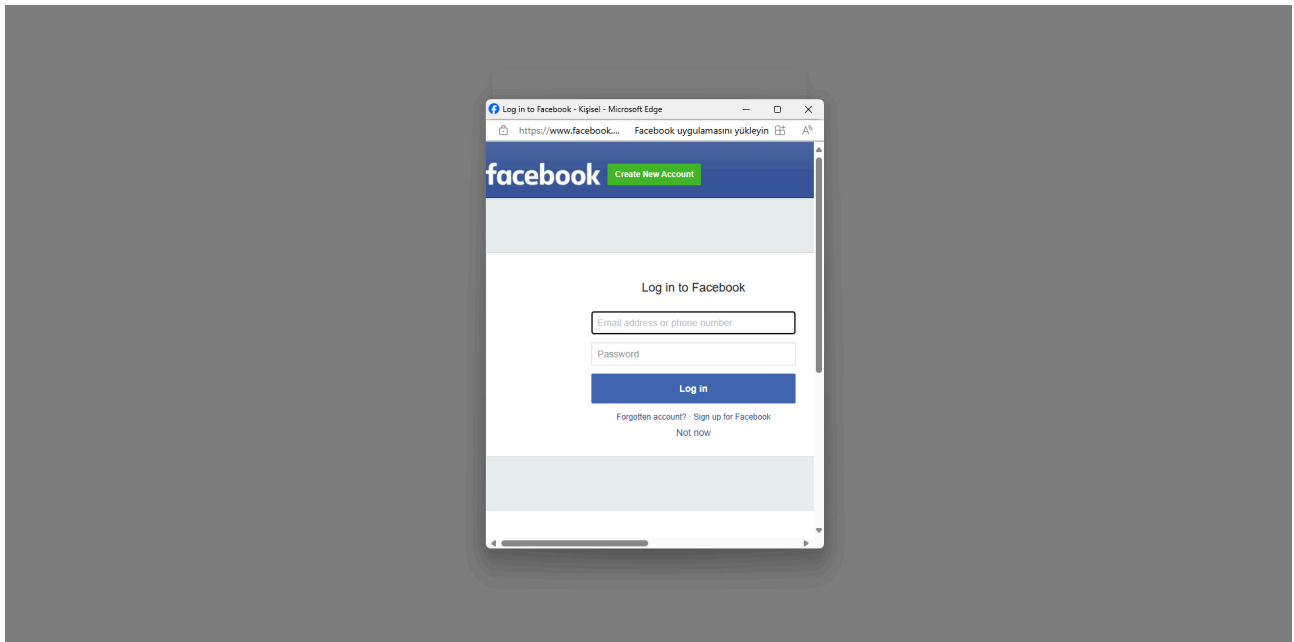
2. “Login with Facebook” Button Presence

This test case verifies that the “Login with Facebook” button is visible and clickable.

- Verify “Login with Facebook” button is displayed
- Click on “Login with Facebook” button

- Wait for the Facebook login popup window

Expected Result: “Login with Facebook” button is displayed and when clicked, the Facebook login popup window is displayed for the user to be able to login with their Facebook account.



```
def test_google_login_button(self): # TEST CASE 4
    """Test if Google login button is present and clickable"""
    driver = self.driver

    google_button = WebDriverWait(driver,
10).until(EC.element_to_be_clickable((By.NAME, "google-login")))
    self.assertTrue(google_button.is_displayed())

def test_facebook_login_button(self):
    """Test if Facebook login button is present and clickable"""
    driver = self.driver

    fb_button = WebDriverWait(driver,
10).until(EC.element_to_be_clickable((By.NAME, "facebook-login")))
    self.assertTrue(fb_button.is_displayed())
```

Test Case 5: Login Persistence and Network Handling

1. Network Disconnection Handling

This test case checks the behavior of the application when network disconnection occurs after clicking the login button.

- Write the valid input (email: test@example.com, password: password123)
- Click the login button
- Simulate Internet disconnection
- Verify that a network related error message is displayed

Expected Result: The application detects the network disconnection and shows the appropriate error message to the user.

Welcome Back

Log in to continue

user1@example.com

.....

No internet connection. Please check your network.

Close

or continue with

G

f

```
def test_internet_disconnect(self): # TEST CASE 5.1
    """Test login behavior when the internet disconnects immediately after
clicking login"""
    driver = self.driver

    WebDriverWait(driver,
15).until(EC.presence_of_element_located((By.TAG_NAME, "form")))

    email_input = driver.find_element(By.NAME, "email")
    password_input = driver.find_element(By.NAME, "password")
    login_button = driver.find_element(By.NAME, "login-button")

    email_input.clear()
    password_input.clear()
    email_input.send_keys("test@example.com")
    password_input.send_keys("password123")
```



```

# Click login, then immediately simulate a network disconnection
login_button.click()

# Enable network emulation and set the network to offline
driver.execute_cdp_cmd("Network.enable", {})
driver.execute_cdp_cmd("Network.emulateNetworkConditions", {
    "offline": True,
    "latency": 0,
    "downloadThroughput": 0,
    "uploadThroughput": 0
})

try:
    # Wait for error popup (or any network-related indicator)
    popup_container = WebDriverWait(driver, 10).until(
        EC.visibility_of_element_located((By.NAME, "popup-container"))
    )
    popup_message = driver.find_element(By.NAME, "popup-message").text
    self.assertIn("network", popup_message.lower())

except Exception:
    print("Network error popup did not appear. The application might not
handle disconnections properly.")

finally:
    # Re-enable network to avoid affecting other tests
    driver.execute_cdp_cmd("Network.emulateNetworkConditions", {
        "offline": False,
        "latency": 0,
        "downloadThroughput": 5000 * 1024,
        "uploadThroughput": 5000 * 1024
    })

```

2. Session Persistence

This test case checks if the user stays logged in after closing and reopening the login page.

- Write the valid input (email: test@example.com, password: password123)
- Select “Remember Me”
- Click the login button
- Close and reopen the application
- Verify that the user remains logged in

Expected Result: The application should automatically redirect the user to /success page.

Welcome Back

Log in to continue

test@example.com

password

Login

☒ Remember Me

or continue with

G f

Login Successful!

Logout

```
def test_remember_me(self):
    """Test if 'Remember Me' keeps the user logged in when returning to the
    site."""

    driver = self.driver # Use the existing WebDriver instance

    driver.get(LOGIN_URL)

    WebDriverWait(driver,
15).until(EC.presence_of_element_located((By.TAG_NAME, "form")))

    email_input = driver.find_element(By.NAME, "email")
    password_input = driver.find_element(By.NAME, "password")
    remember_me_checkbox = driver.find_element(By.NAME, "remember-me")
    login_button = driver.find_element(By.NAME, "login-button")

    email_input.clear()
    password_input.clear()
    email_input.send_keys("test@example.com")
    password_input.send_keys("password123")

    # Check "Remember Me"
    if not remember_me_checkbox.is_selected():
        remember_me_checkbox.click()

    login_button.click()

    # Wait until redirected to /success
    WebDriverWait(driver, 10).until(EC.url_contains("success"))
    self.assertIn("success", driver.current_url.lower())
```

```

        # Verify localStorage has saved the session
        logged_in_user = driver.execute_script("return
localStorage.getItem('loggedInUser');")
        self.assertIsNotNone(logged_in_user, "User session should be saved in
localStorage")

        # Navigate back to the login page
        driver.get(LOGIN_URL)

        # Wait and check if the app automatically redirects to /success
        WebDriverWait(driver, 10).until(EC.url_contains("success"))
        self.assertIn("success", driver.current_url.lower(), "User should be
redirected to /success after returning to /")

        # Cleanup: Logout and clear localStorage
        self.logout()

```

Test automation in SDLC

Test automation is crucial in ensuring the quality of a program while minimizing the amount of time within the software development lifecycle. Manual testing is time-consuming and, at times, error-prone for larger and more complex software. Manual testing also fails to account for how future changes to the software may change the results of previously done tests. On the other hand, automated testing ensures that whenever a new feature is added, the previous ones do not break, and the overall quality is maintained. However, test automation comes at the expense of taking time to write scripts that test the software automatically using tools such as Selenium. However, this is an acceptable tradeoff thanks to the quality and testing speed that it ensures. Automated testing also allows for parallel execution, different configurations, different environments, and different devices. These functionalities of automated testing provide insurance that the software works as expected under any circumstances, and they are much more costly to replicate through manual means.

Test users

```

{ email: "test@example.com", password: "password123" },
{ email: "user1@example.com", password: "securepass1" },
{ email: "admin@example.com", password: "admin123" },
{ email: "doctor@example.com", password: "medic987" },
{ email: "staff@example.com", password: "staffpass" },

```