BILKENT UNIVERSITY

CS 485 - Section 1: Verification and Validation

Project 1

Report

Group Members:
Tolgahan Arslan 22003061
Tolga Artun Koçak 22102132
Atilla Alp Yavuz 22102191
Tuna Cuma 22103156

Analysis:

Selenium is an open-source framework designed to automate web browsers. It helps developers and testers automate user interactions with web applications. This means it is used for different types of testing such as functional and performance. Selenium can be used with various programming languages and works on most browsers. It is flexible and widely used as it has a lot of community support as well.

10 Key Capabilities of Selenium

1. Selenium can run on different browsers and thus is versatile.
2. It can be used with different programming languages.
3. Provides the Webdriver API to interact with web elements and mimics user actions. This is why it is used to automate tests.
4. It has an IDE in which it has a tool that lets coders quickly create tests without writing code
5. Can execute tests in parallel to reduce test run time.
6. It is open source, continually being updated and free.
7. Can integrate with other testing frameworks and CI/CD tools.
8. Can interact with dynamic content, and asynchronous events commonly used in web apps.
9. Can also be used for mobile app testing (Appium).
10. It is scalable and extensible, fitting the requirements of modern corporations.

Structure:

Screenshots of the Application

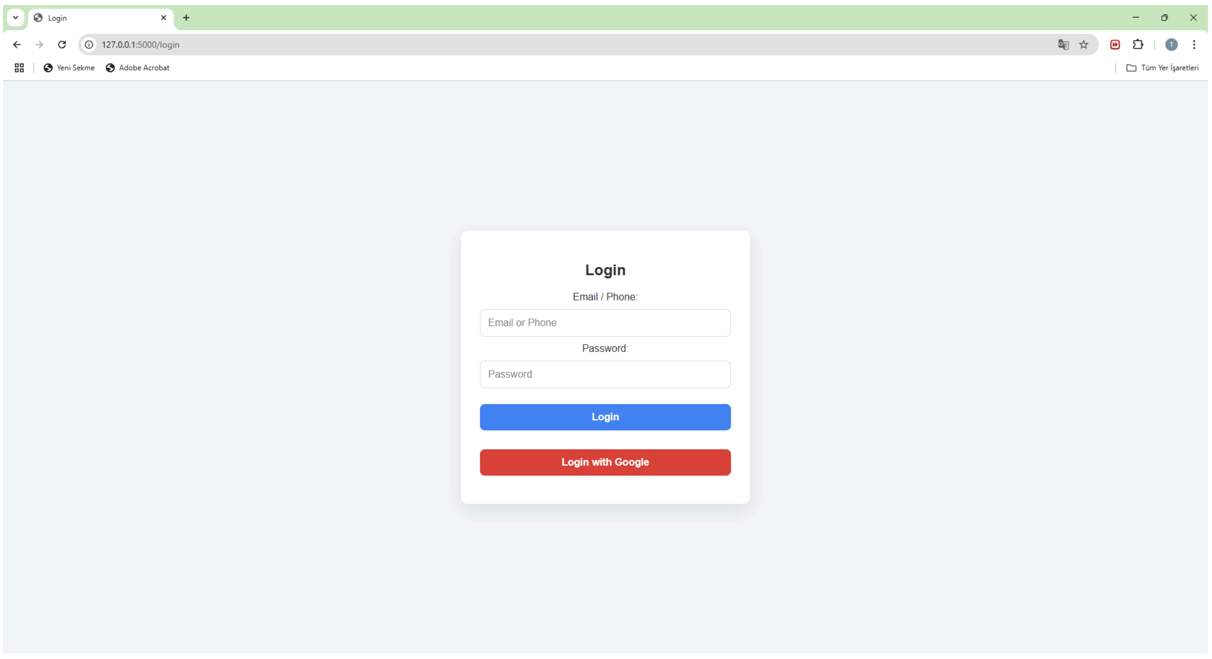
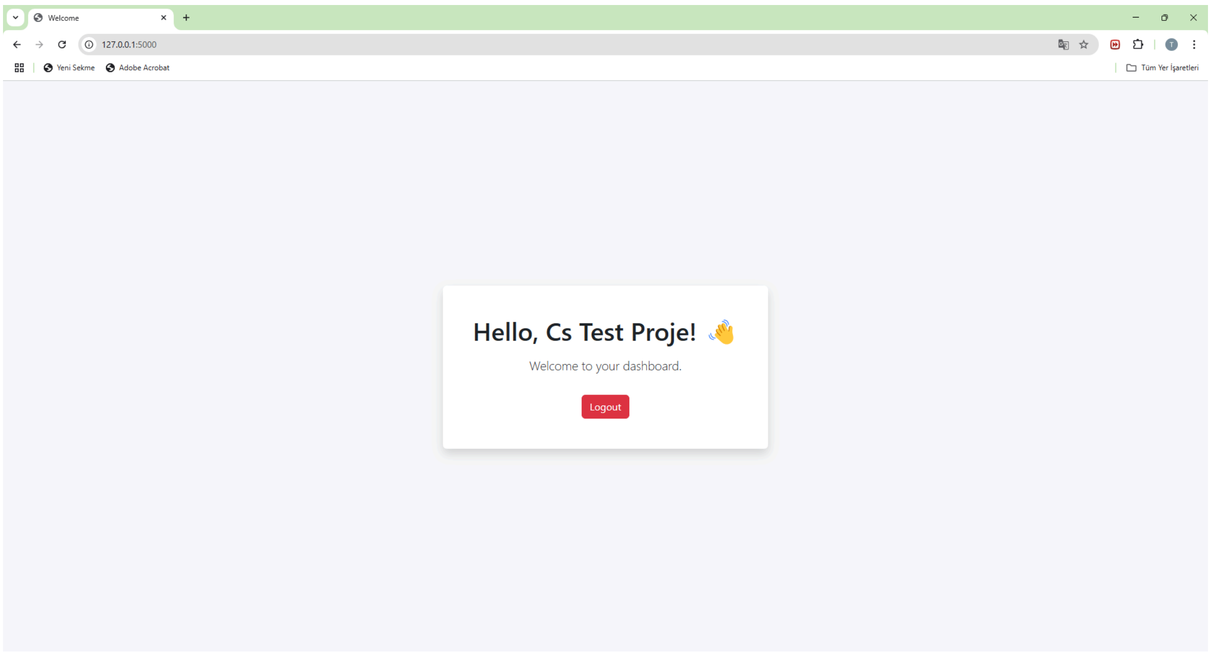


Figure 1: Login User Interface of the Application

Figure 1 is the login page of the application. Users can input their emails or phone numbers with the respective passwords from this page. If the related account exists in the system, login will be successful and it redirects the user to the dashboard. Moreover, users can login with their Google accounts by the respective option here.

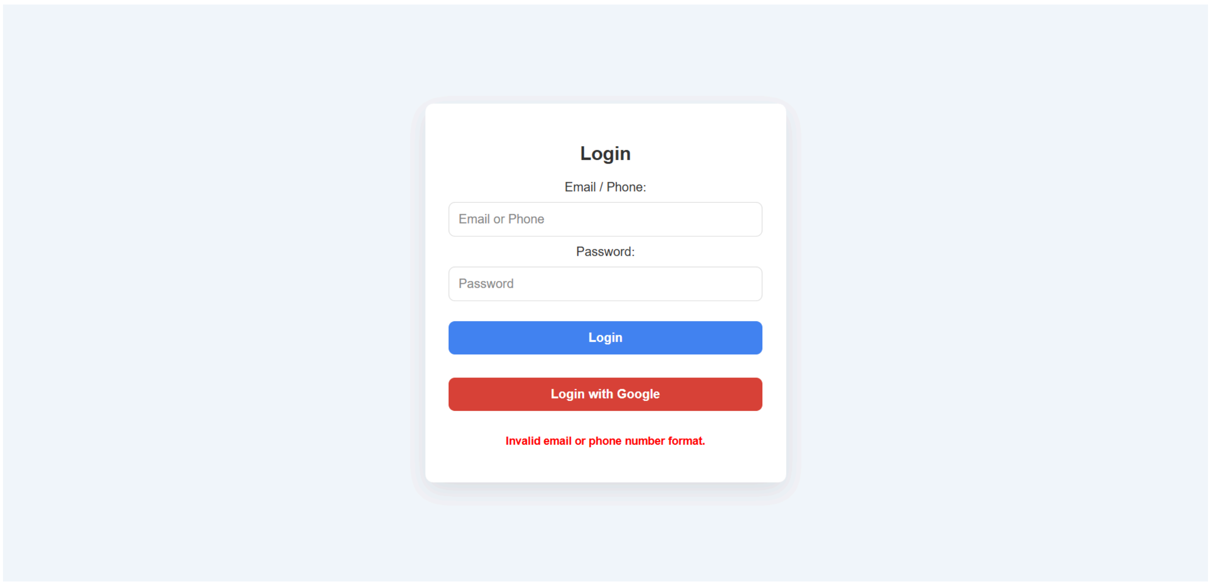


Hello, Cs Test Proje! 👋

Welcome to your dashboard.

Figure 2: Dashboard User Interface of the Application

This is the dashboard page after the login process. It greets the signed in user and provides a logout option. If the input is wrong however, the user will remain on the same page and will be given an error message related to the issue they caused. The error message appears as a red text below the form:



Invalid email or phone number format.

Figure 3: Failed Log in Error Response of the Application

Diagrams

Use Case Diagram:

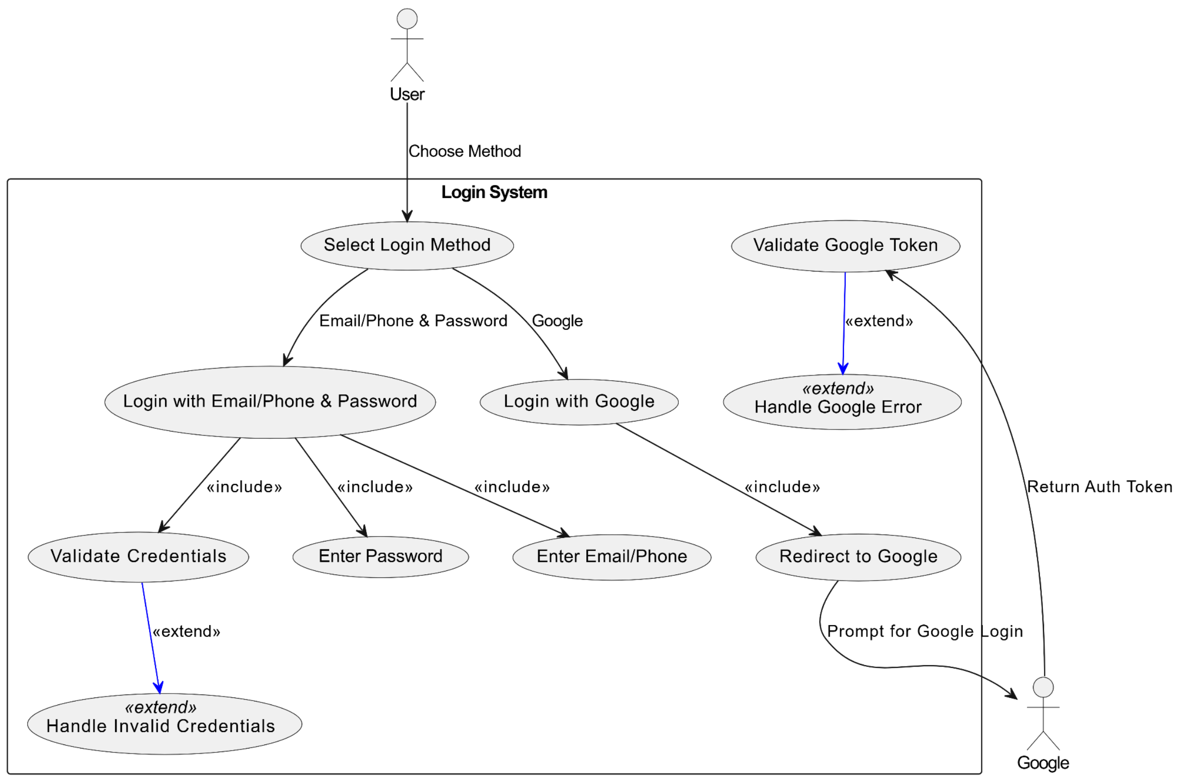


Figure 4: Use Case Diagram of the Application

This use case diagram shows a Login System where a User can choose to log in either by Email/Phone & Password or via Google. It breaks down the manual login flow into sub-use-cases for entering credentials and validating them, while the Google login flow involves redirecting the user to Google and validating the returned token. The diagram also includes error-handling use cases for invalid credentials and Google-specific errors.

Activity Diagram:

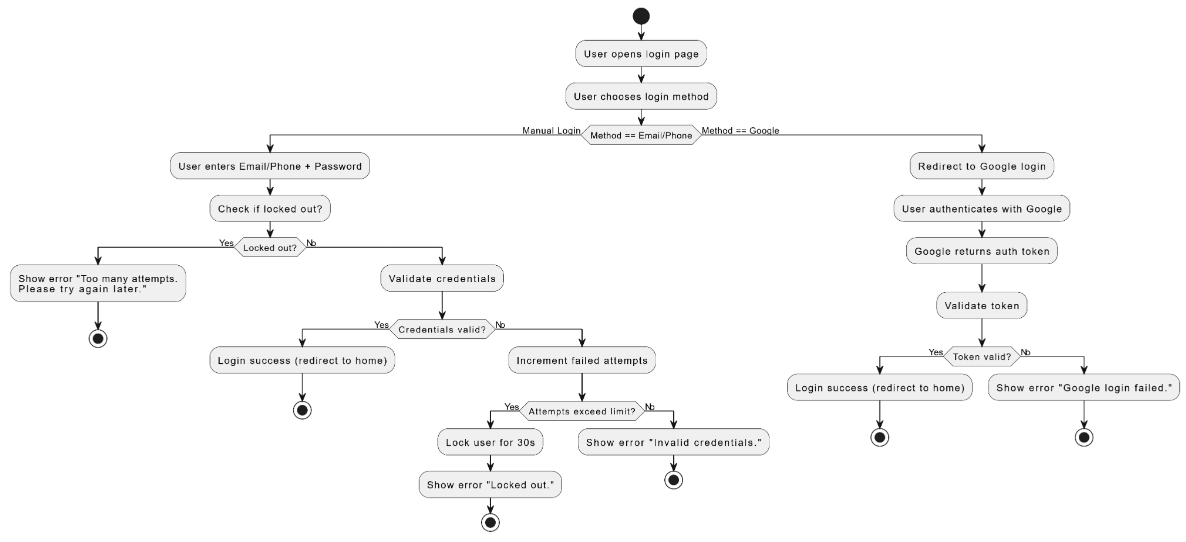

User opens login page
User chooses login method
Manual Login
Method == Email/Phone
Method == Google
User enters Email/Phone + Password
Check if locked out?
Yes
Locked out?
No
Show error "Too many attempts. Please try again later."
Validate credentials
Yes
Credentials valid?
No
Login success (redirect to home)
Increment failed attempts
Yes
Attempts exceed limit?
No
Lock user for 30s
Show error "Invalid credentials."
Show error "Locked out."
Redirect to Google login
User authenticates with Google
Google returns auth token
Validate token
Yes
Token valid?
No
Login success (redirect to home)
Show error "Google login failed."

Figure 5: Activity Diagram of the Login Process

This diagram depicts a two-path login process, where the user first chooses either manual or Google authentication. In the manual path, the system checks if the user is locked out, then validates their Email/Phone and Password; if attempts exceed the limit, the user is locked out. In the Google path, the user is redirected to Google for authentication, and the system validates the returned token. Both paths either result in a successful login and redirect to the home page or show an appropriate error message if authentication fails.

State Diagram:

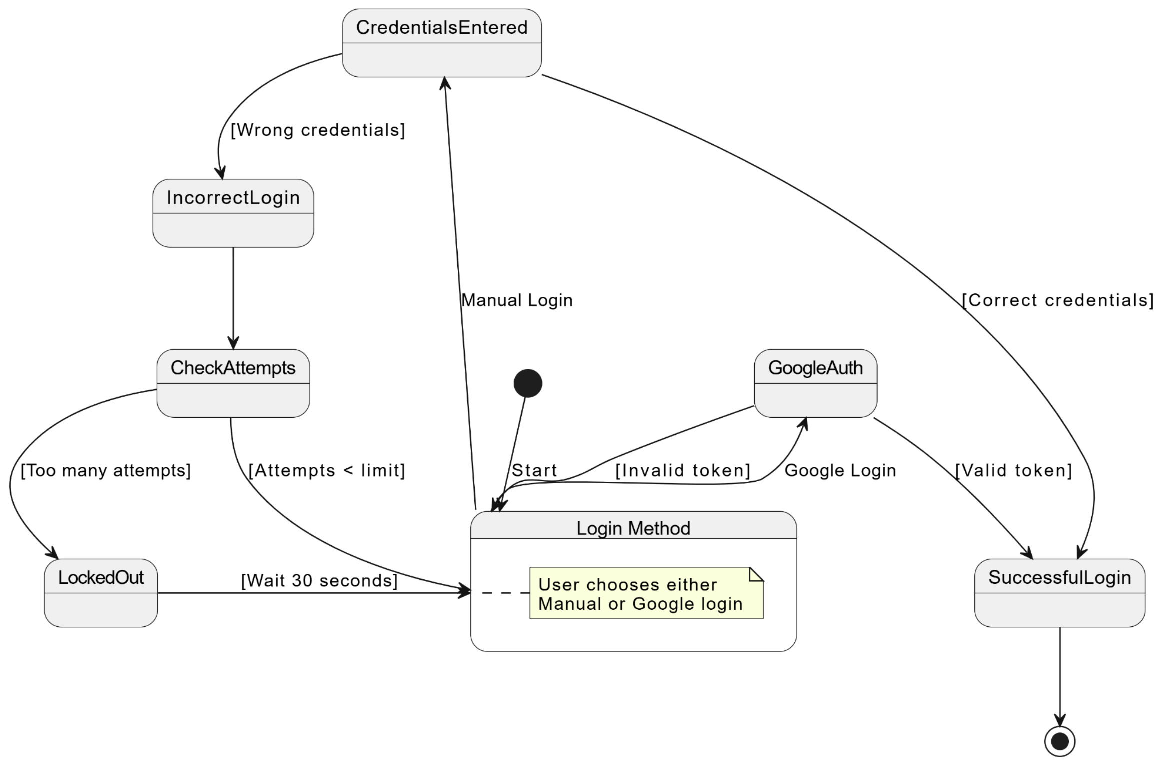
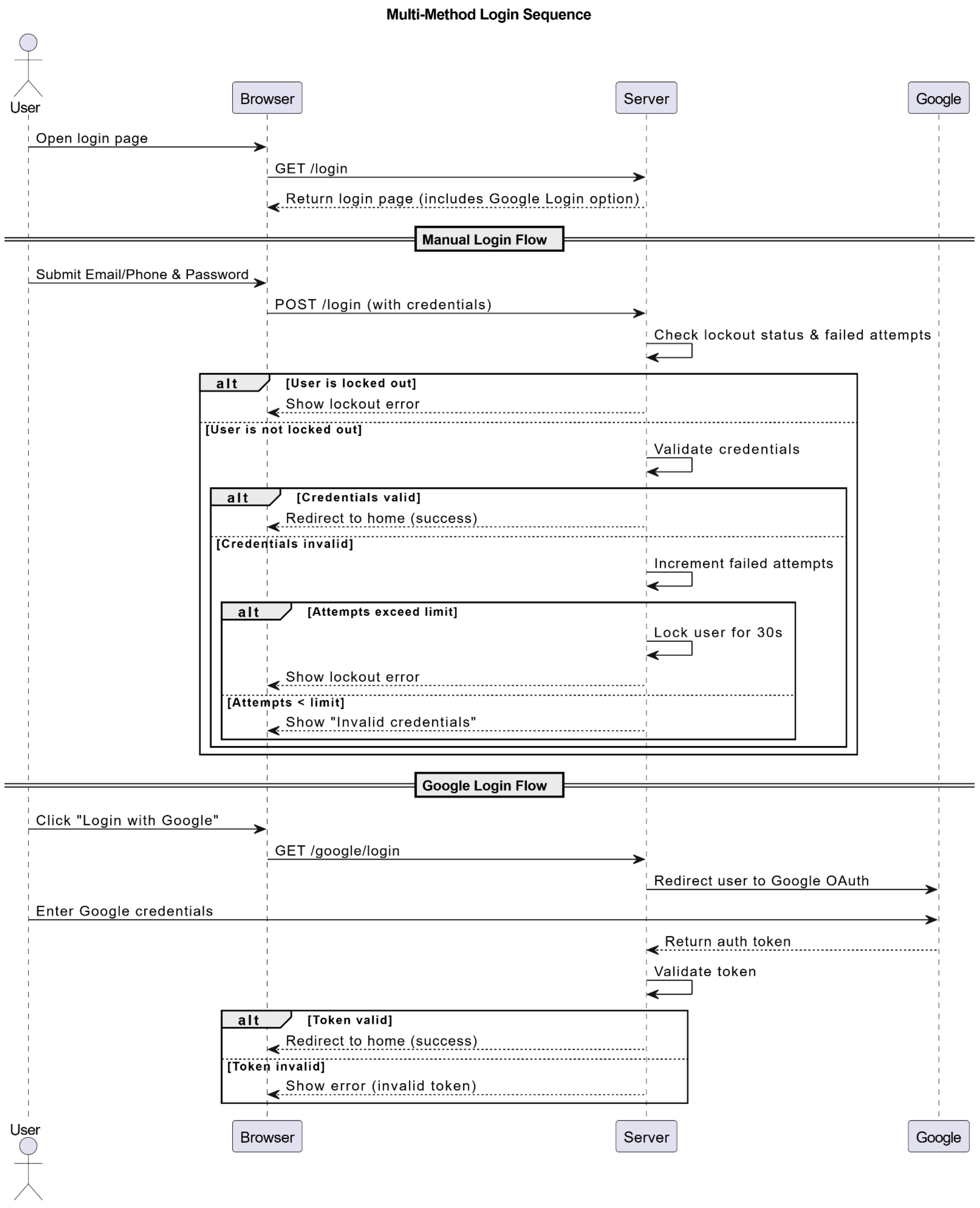


Figure 6: State Diagram of the Application

This state diagram shows how a user can either select to login manually or via using the Google Login and proceed through states like CredentialsEntered, IncorrectLogin, and LockedOut. If the user's credentials are invalid, the system checks the number of failed attempts; too many attempts lead to a LockedOut state, while fewer attempts let the user try again. In the GoogleAuth path, an invalid token sends the user back to the login method, and a valid token leads to SuccessfulLogin. Ultimately, both login methods converge on a SuccessfulLogin state when credentials or tokens are valid.

Sequence Diagram:



Multi-Method Login Sequence
User
Browser
Server
Google
Open login page
GET /login
Return login page (includes Google Login option)
Manual Login Flow
Submit Email/Phone & Password
POST /login (with credentials)
Check lockout status & failed attempts
alt [User is locked out]
Show lockout error
[User is not locked out]
Validate credentials
alt [Credentials valid]
Redirect to home (success)
[Credentials invalid]
Increment failed attempts
alt [Attempts exceed limit]
Lock user for 30s
Show lockout error
[Attempts < limit]
Show "Invalid credentials"
Google Login Flow
Click "Login with Google"
GET /google/login
Redirect user to Google OAuth
Enter Google credentials
Return auth token
Validate token
alt [Token valid]
Redirect to home (success)
[Token invalid]
Show error (invalid token)

Figure 7: Sequence Diagram of the Application

This sequence diagram illustrates two primary flows for logging in: Manual Login and Google Login. In the Manual Login flow, the user submits credentials, the server checks lockout status and attempts, and either grants access or locks the user out if too many failed attempts occur. In the Google Login flow, the user is redirected to Google for authentication, and upon receiving a token, the server validates it before granting or denying access. Overall, this sequence diagram captures each step from the user's initial request to the final success or error response.

Class Diagram:

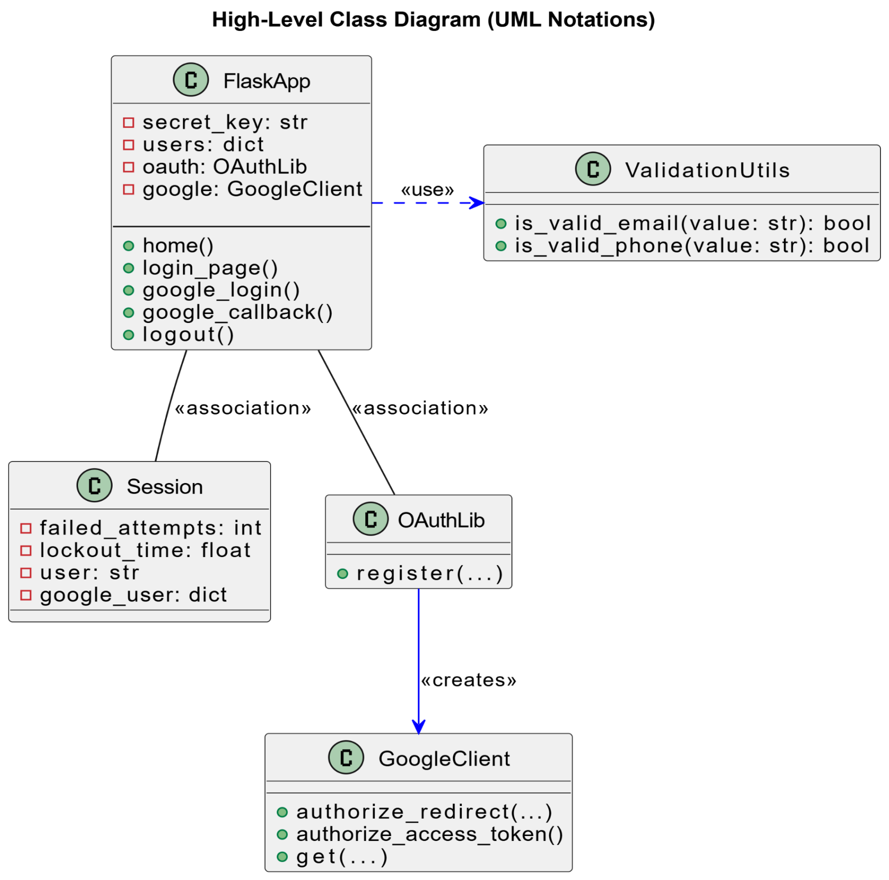
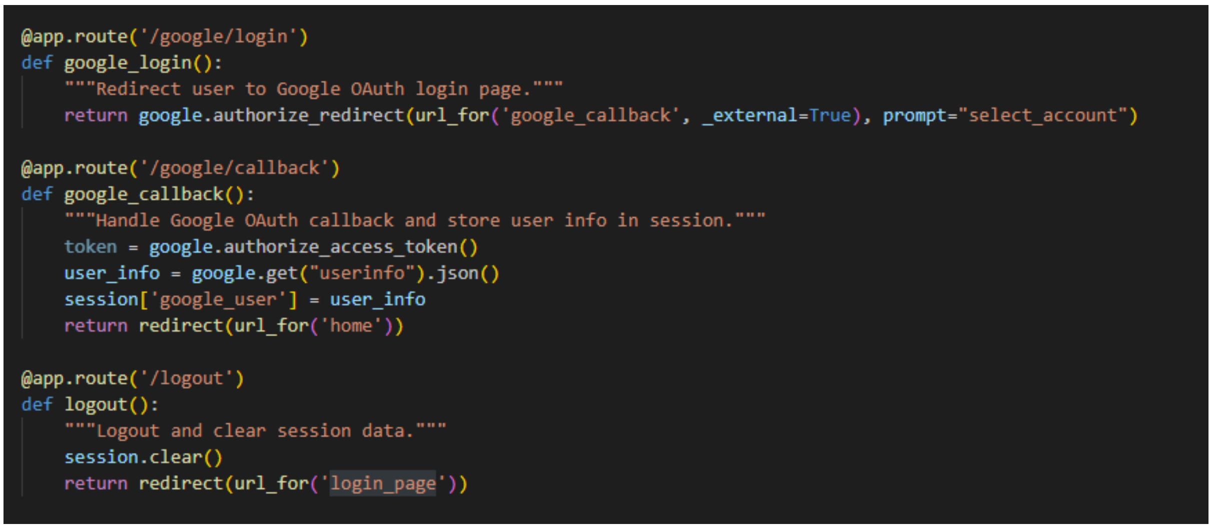


Figure 8: High Level Class Diagram of the Application

This class diagram shows the primary classes in a Flask-based login application. The FlaskApp class handles the main routes (home, login, Google login/callback, and logout) and references a Session object to track user data like failed attempts or lockout times. ValidationUtils provides helper methods to verify email or phone input, while OAuthLib and GoogleClient manage the Google OAuth flow. Overall, these classes work together to provide manual and Google-based authentication, along with basic lockout functionality.

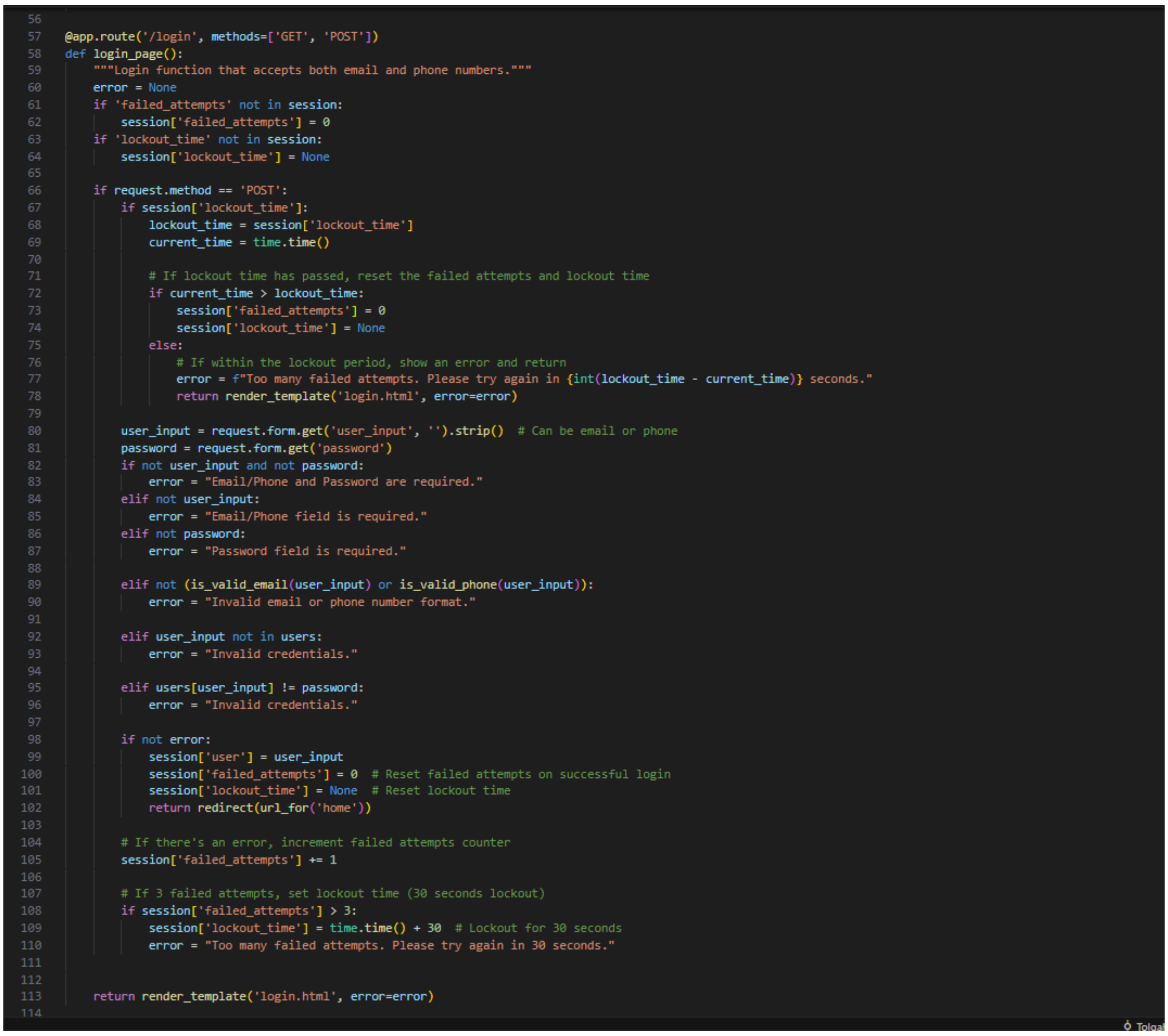


```python
@app.route('/google/login')
def google_login():
    """Redirect user to Google OAuth login page."""
    return google.authorize_redirect(url_for('google_callback', _external=True), prompt="select_account")

@app.route('/google/callback')
def google_callback():
    """Handle Google OAuth callback and store user info in session."""
    token = google.authorize_access_token()
    user_info = google.get("userinfo").json()
    session['google_user'] = user_info
    return redirect(url_for('home'))

@app.route('/logout')
def logout():
    """Logout and clear session data."""
    session.clear()
    return redirect(url_for('login_page'))
```

Figure 9: Code Excerpt for Google Login



```python

@app.route('/login', methods=['GET', 'POST'])
def login_page():
    """Login function that accepts both email and phone numbers."""
    error = None
    if 'failed_attempts' not in session:
        session['failed_attempts'] = 0
    if 'lockout_time' not in session:
        session['lockout_time'] = None

    if request.method == 'POST':
        if session['lockout_time']:
            lockout_time = session['lockout_time']
            current_time = time.time()

            # If lockout time has passed, reset the failed attempts and lockout time
            if current_time > lockout_time:
                session['failed_attempts'] = 0
                session['lockout_time'] = None
            else:
                # If within the lockout period, show an error and return
                error = f"Too many failed attempts. Please try again in {int(lockout_time - current_time)} seconds."
                return render_template('login.html', error=error)

        user_input = request.form.get('user_input', '').strip()  # Can be email or phone
        password = request.form.get('password')
        if not user_input and not password:
            error = "Email/Phone and Password are required."
        elif not user_input:
            error = "Email/Phone field is required."
        elif not password:
            error = "Password field is required."

        elif not (is_valid_email(user_input) or is_valid_phone(user_input)):
            error = "Invalid email or phone number format."

        elif user_input not in users:
            error = "Invalid credentials."

        elif users[user_input] != password:
            error = "Invalid credentials."

        if not error:
            session['user'] = user_input
            session['failed_attempts'] = 0  # Reset failed attempts on successful login
            session['lockout_time'] = None  # Reset lockout time
            return redirect(url_for('home'))

        # If there's an error, increment failed attempts counter
        session['failed_attempts'] += 1

        # If 3 failed attempts, set lockout time (30 seconds lockout)
        if session['failed_attempts'] > 3:
            session['lockout_time'] = time.time() + 30  # Lockout for 30 seconds
            error = "Too many failed attempts. Please try again in 30 seconds."


    return render_template('login.html', error=error)

```

Figure 10: Login Function Code Excerpt

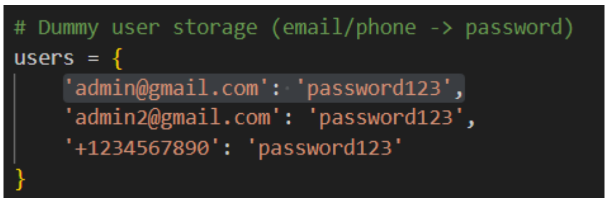
Figure 9 and 10 are the code excerpts of the login function and Google login function. The application takes track of failed login attempts on the same session, so that it can lockout any login attempt after 3 failed ones for a 30 seconds timestamp. It deletes any leading and trailing spaces in the email field and checks for other possible cases such as blank fields, invalid formats, and correct credentials.

Testing:

The tests all work if the user does not tab out of the pop up window that the test takes place in. We assume the processing related to changing windows causes issues especially in the google test. The reason for this assumption is that the code runs perfectly fine when not tabbing out and when tabbing out the errors are unpredictable. The code finishes with an error after completing successfully because of issues with an undetected chrome driver. This does not cause any issues to the tests, just results in an error message even if the code successfully runs.

Test Case #1:

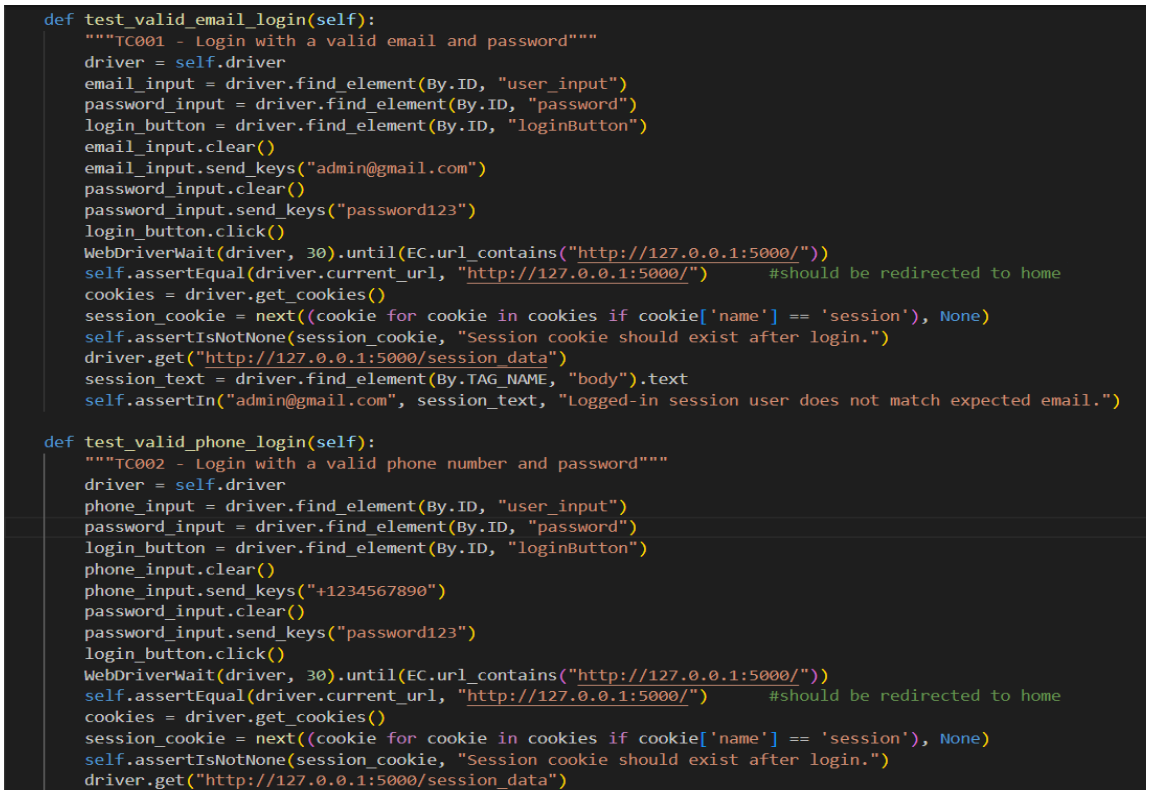
The first test focuses on the valid and invalid login using an email or a phone password. Firstly the valid logins are attempted, these valid identities are defined in the code for the login page in this dictionary:



```python
# Dummy user storage (email/phone -> password)
users = {
    'admin@gmail.com': 'password123',
    'admin2@gmail.com': 'password123',
    '+1234567890': 'password123'
}
```

Figure 11: Dummy User Accounts in the Application

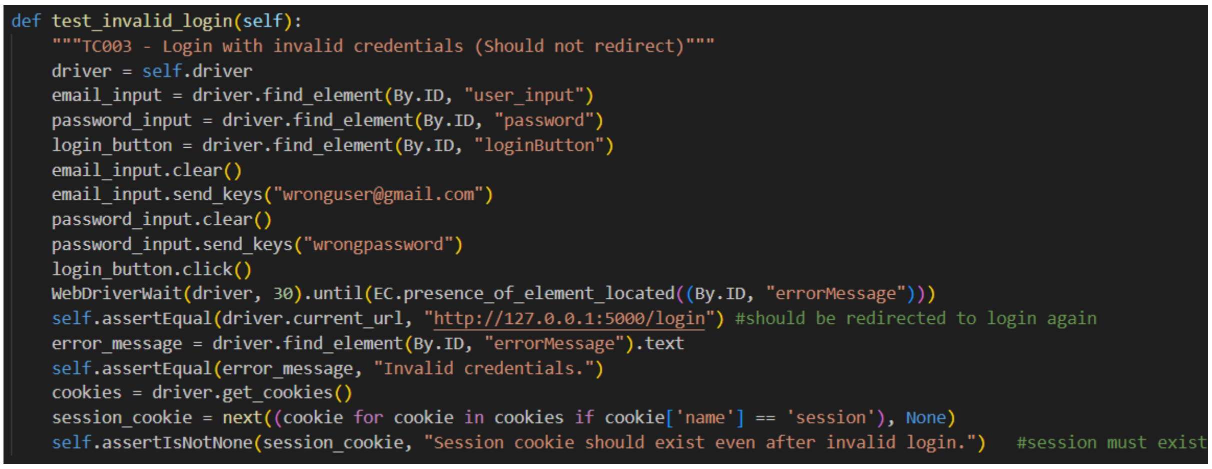
These can be tested in the log in and will work. The code for the valid test is below:



```python
def test_valid_email_login(self):
    """TC001 - Login with a valid email and password"""
    driver = self.driver
    email_input = driver.find_element(By.ID, "user_input")
    password_input = driver.find_element(By.ID, "password")
    login_button = driver.find_element(By.ID, "loginButton")
    email_input.clear()
    email_input.send_keys("admin@gmail.com")
    password_input.clear()
    password_input.send_keys("password123")
    login_button.click()
    WebDriverWait(driver, 30).until(EC.url_contains("http://127.0.0.1:5000/"))
    self.assertEqual(driver.current_url, "http://127.0.0.1:5000/")      #should be redirected to home
    cookies = driver.get_cookies()
    session_cookie = next((cookie for cookie in cookies if cookie['name'] == 'session'), None)
    self.assertIsNotNone(session_cookie, "Session cookie should exist after login.")
    driver.get("http://127.0.0.1:5000/session_data")
    session_text = driver.find_element(By.TAG_NAME, "body").text
    self.assertIn("admin@gmail.com", session_text, "Logged-in session user does not match expected email.")

def test_valid_phone_login(self):
    """TC002 - Login with a valid phone number and password"""
    driver = self.driver
    phone_input = driver.find_element(By.ID, "user_input")
    password_input = driver.find_element(By.ID, "password")
    login_button = driver.find_element(By.ID, "loginButton")
    phone_input.clear()
    phone_input.send_keys("+1234567890")
    password_input.clear()
    password_input.send_keys("password123")
    login_button.click()
    WebDriverWait(driver, 30).until(EC.url_contains("http://127.0.0.1:5000/"))
    self.assertEqual(driver.current_url, "http://127.0.0.1:5000/")      #should be redirected to home
    cookies = driver.get_cookies()
    session_cookie = next((cookie for cookie in cookies if cookie['name'] == 'session'), None)
    self.assertIsNotNone(session_cookie, "Session cookie should exist after login.")
    driver.get("http://127.0.0.1:5000/session_data")
```

Figure 12: Valid Logins with Email and Phone Number (Test Case 1)

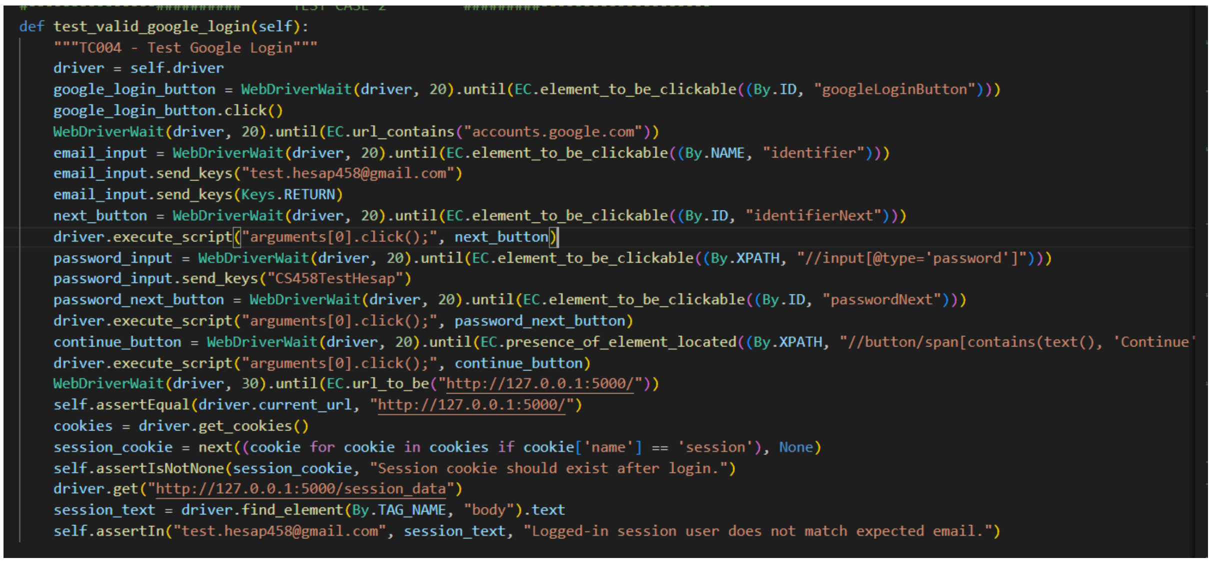


```python
def test_invalid_login(self):
    """TC003 - Login with invalid credentials (Should not redirect)"""
    driver = self.driver
    email_input = driver.find_element(By.ID, "user_input")
    password_input = driver.find_element(By.ID, "password")
    login_button = driver.find_element(By.ID, "loginButton")
    email_input.clear()
    email_input.send_keys("wronguser@gmail.com")
    password_input.clear()
    password_input.send_keys("wrongpassword")
    login_button.click()
    WebDriverWait(driver, 30).until(EC.presence_of_element_located((By.ID, "errorMessage")))
    self.assertEqual(driver.current_url, "http://127.0.0.1:5000/login") #should be redirected to login again
    error_message = driver.find_element(By.ID, "errorMessage").text
    self.assertEqual(error_message, "Invalid credentials.")
    cookies = driver.get_cookies()
    session_cookie = next((cookie for cookie in cookies if cookie['name'] == 'session'), None)
    self.assertIsNotNone(session_cookie, "Session cookie should exist even after invalid login.")   #session must exist
```

Figure 13: Invalid Login (Test Case 1)

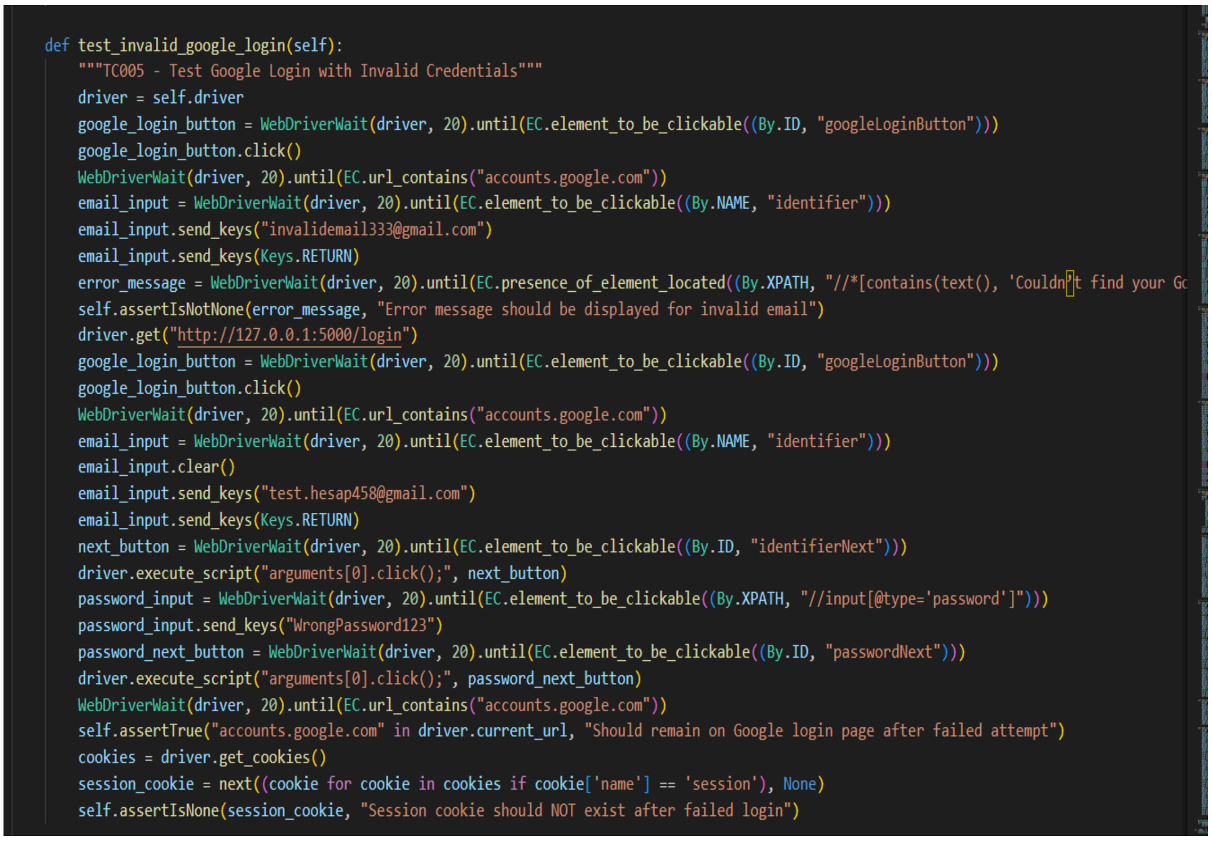
This test controls for both emails and login. It does so by opening a chrome tab, locating the input boxes, clearing them and entering the respective correct results. Then it checks if the page has rerouted to the logged in page and looks into the session data to find if the correct user has been logged in. On the invalid log in, the structure is very similar but it has to not be rerouted and remain in the same page and the session cookie must be saved in order to track incorrect login attempts.

Test Case #2:



```python
def test_valid_google_login(self):
    """TC004 - Test Google Login"""
    driver = self.driver
    google_login_button = WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.ID, "googleLoginButton")))
    google_login_button.click()
    WebDriverWait(driver, 20).until(EC.url_contains("accounts.google.com"))
    email_input = WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.NAME, "identifier")))
    email_input.send_keys("test.hesap458@gmail.com")
    email_input.send_keys(Keys.RETURN)
    next_button = WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.ID, "identifierNext")))
    driver.execute_script("arguments[0].click();", next_button)
    password_input = WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.XPATH, "//input[@type='password']")))
    password_input.send_keys("CS458TestHesap")
    password_next_button = WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.ID, "passwordNext")))
    driver.execute_script("arguments[0].click();", password_next_button)
    continue_button = WebDriverWait(driver, 20).until(EC.presence_of_element_located((By.XPATH, "//button/span[contains(text(), 'Continue'
    driver.execute_script("arguments[0].click();", continue_button)
    WebDriverWait(driver, 30).until(EC.url_to_be("http://127.0.0.1:5000/"))
    self.assertEqual(driver.current_url, "http://127.0.0.1:5000/")
    cookies = driver.get_cookies()
    session_cookie = next((cookie for cookie in cookies if cookie['name'] == 'session'), None)
    self.assertIsNotNone(session_cookie, "Session cookie should exist after login.")
    driver.get("http://127.0.0.1:5000/session_data")
    session_text = driver.find_element(By.TAG_NAME, "body").text
    self.assertIn("test.hesap458@gmail.com", session_text, "Logged-in session user does not match expected email.")
```

Figure 14: Valid Google Login (Test Case 2)



```python
def test_invalid_google_login(self):
    """TC005 - Test Google Login with Invalid Credentials"""
    driver = self.driver
    google_login_button = WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.ID, "googleLoginButton")))
    google_login_button.click()
    WebDriverWait(driver, 20).until(EC.url_contains("accounts.google.com"))
    email_input = WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.NAME, "identifier")))
    email_input.send_keys("invalidemail333@gmail.com")
    email_input.send_keys(Keys.RETURN)
    error_message = WebDriverWait(driver, 20).until(EC.presence_of_element_located((By.XPATH, "//*[contains(text(), 'Couldn't find your G
    self.assertIsNotNone(error_message, "Error message should be displayed for invalid email")
    driver.get("http://127.0.0.1:5000/login")
    google_login_button = WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.ID, "googleLoginButton")))
    google_login_button.click()
    WebDriverWait(driver, 20).until(EC.url_contains("accounts.google.com"))
    email_input = WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.NAME, "identifier")))
    email_input.clear()
    email_input.send_keys("test.hesap458@gmail.com")
    email_input.send_keys(Keys.RETURN)
    next_button = WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.ID, "identifierNext")))
    driver.execute_script("arguments[0].click();", next_button)
    password_input = WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.XPATH, "//input[@type='password']")))
    password_input.send_keys("WrongPassword123")
    password_next_button = WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.ID, "passwordNext")))
    driver.execute_script("arguments[0].click();", password_next_button)
    WebDriverWait(driver, 20).until(EC.url_contains("accounts.google.com"))
    self.assertTrue("accounts.google.com" in driver.current_url, "Should remain on Google login page after failed attempt")
    cookies = driver.get_cookies()
    session_cookie = next((cookie for cookie in cookies if cookie['name'] == 'session'), None)
    self.assertIsNone(session_cookie, "Session cookie should NOT exist after failed login")
```

Figure 15: Invalid Google Login (Test Case 2)

This test case consists of valid and invalid Google logins with the Google account we created. Since Google blocks automated Selenium login attempts, undetected_chromedriver library was used instead of the usual chrome driver. In the valid login case, Selenium clicks the login with Google button and inputs the credentials of the dummy Google account (test.hesap458@gmail.com CS458TestHesap). It checks whether the login is successful by the redirection on the home directory and checks the user in the session data. In the invalid login case, Selenium inputs an invalid gmail account, a valid gmail account and wrong password. Again, it checks the url redirections as well as the invalid login popups in the application.

Test Case #3:
This test controls for parallel logins from multiple tabs simultaneously. The first function creates two Chrome windows, one normal and one incognito. They enter different credentials, and login. The test then retrieves and compares their session cookies to ensure they're different. Both view the session data to confirm the log in credentials didn't get mixed up.

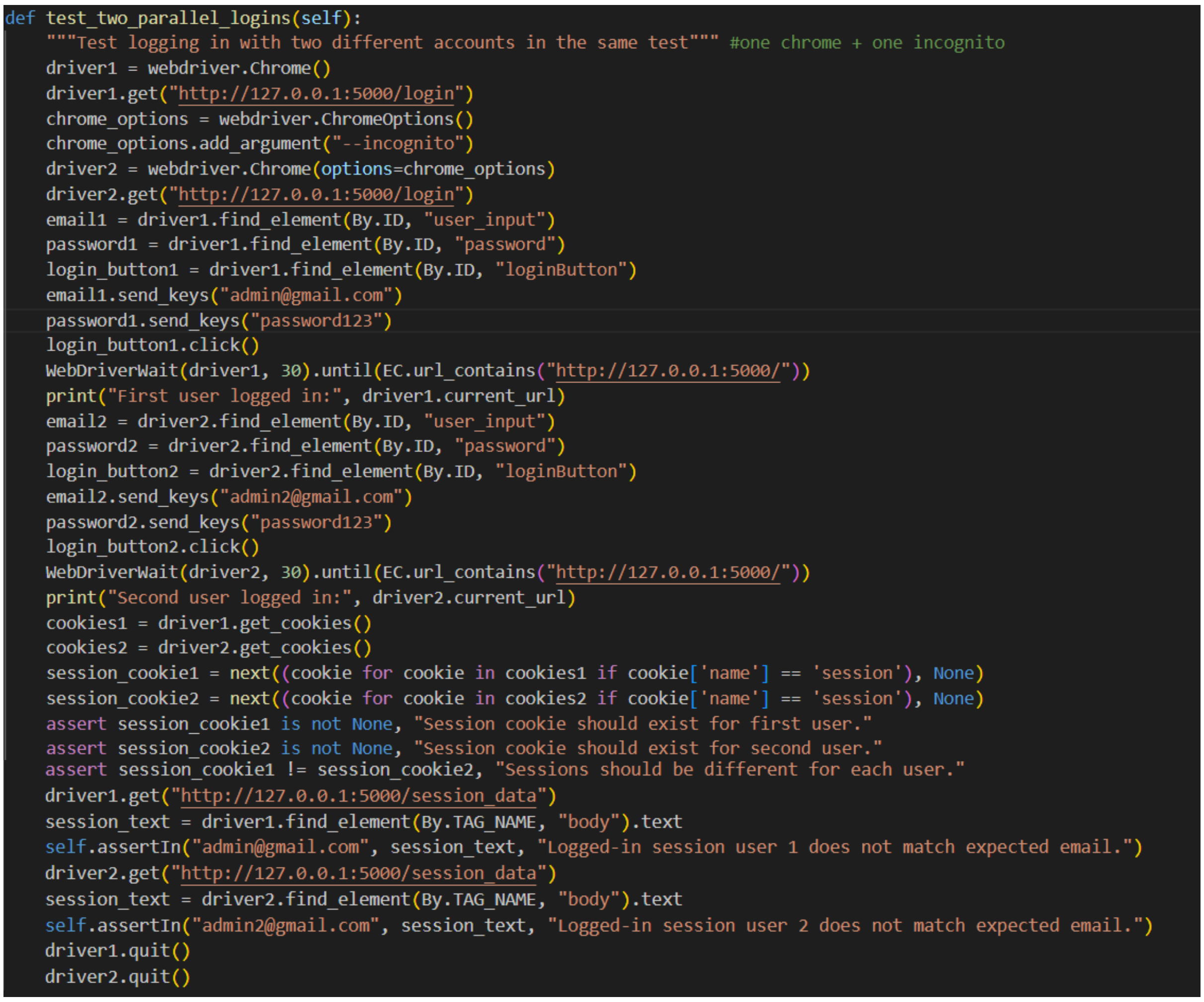
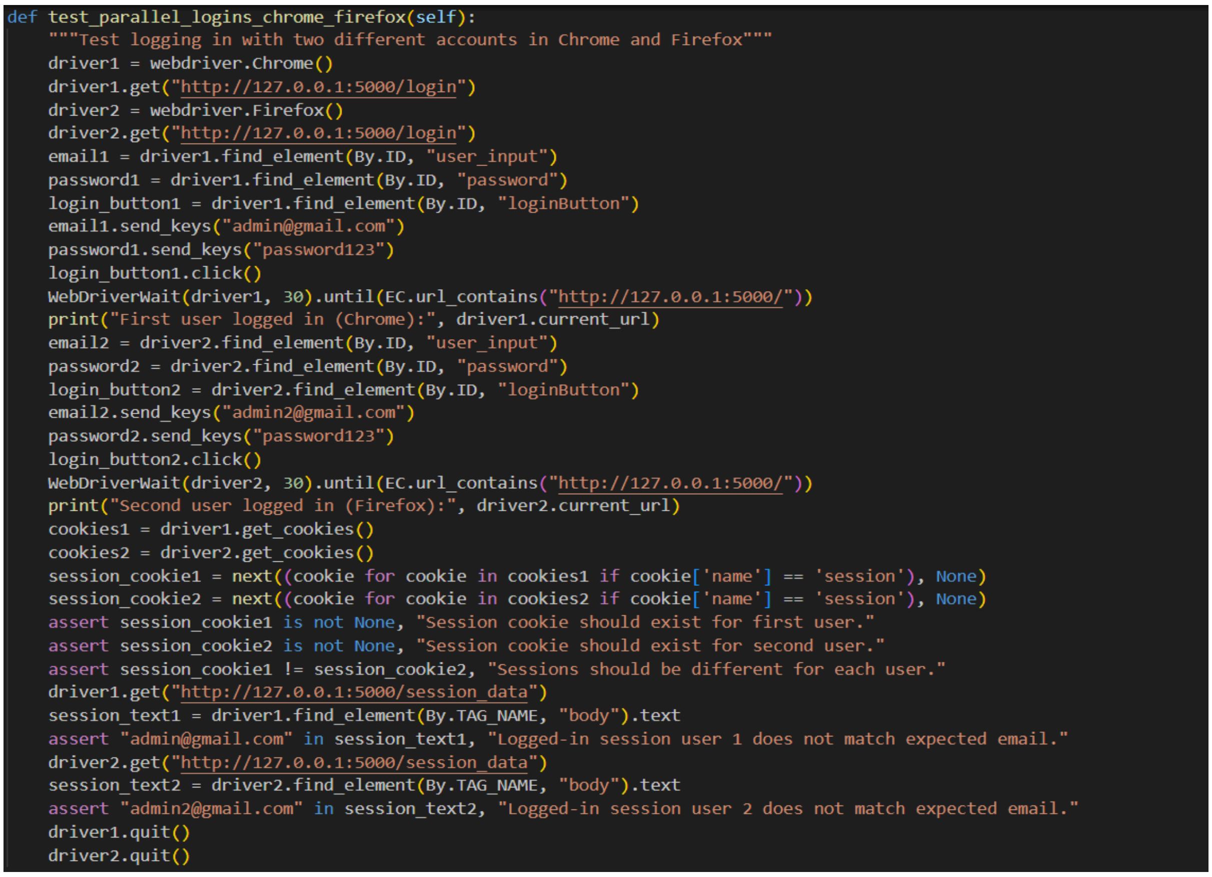

```python
def test_two_parallel_logins(self):
    """Test logging in with two different accounts in the same test""" #one chrome + one incognito
    driver1 = webdriver.Chrome()
    driver1.get("http://127.0.0.1:5000/login")
    chrome_options = webdriver.ChromeOptions()
    chrome_options.add_argument("--incognito")
    driver2 = webdriver.Chrome(options=chrome_options)
    driver2.get("http://127.0.0.1:5000/login")
    email1 = driver1.find_element(By.ID, "user_input")
    password1 = driver1.find_element(By.ID, "password")
    login_button1 = driver1.find_element(By.ID, "loginButton")
    email1.send_keys("admin@gmail.com")
    password1.send_keys("password123")
    login_button1.click()
    WebDriverWait(driver1, 30).until(EC.url_contains("http://127.0.0.1:5000/"))
    print("First user logged in:", driver1.current_url)
    email2 = driver2.find_element(By.ID, "user_input")
    password2 = driver2.find_element(By.ID, "password")
    login_button2 = driver2.find_element(By.ID, "loginButton")
    email2.send_keys("admin2@gmail.com")
    password2.send_keys("password123")
    login_button2.click()
    WebDriverWait(driver2, 30).until(EC.url_contains("http://127.0.0.1:5000/"))
    print("Second user logged in:", driver2.current_url)
    cookies1 = driver1.get_cookies()
    cookies2 = driver2.get_cookies()
    session_cookie1 = next((cookie for cookie in cookies1 if cookie['name'] == 'session'), None)
    session_cookie2 = next((cookie for cookie in cookies2 if cookie['name'] == 'session'), None)
    assert session_cookie1 is not None, "Session cookie should exist for first user."
    assert session_cookie2 is not None, "Session cookie should exist for second user."
    assert session_cookie1 != session_cookie2, "Sessions should be different for each user."
    driver1.get("http://127.0.0.1:5000/session_data")
    session_text = driver1.find_element(By.TAG_NAME, "body").text
    self.assertIn("admin@gmail.com", session_text, "Logged-in session user 1 does not match expected email.")
    driver2.get("http://127.0.0.1:5000/session_data")
    session_text = driver2.find_element(By.TAG_NAME, "body").text
    self.assertIn("admin2@gmail.com", session_text, "Logged-in session user 2 does not match expected email.")
    driver1.quit()
    driver2.quit()
```

Figure 16: Valid Parallel Log in Chrome and Incognito (Test Case 3)

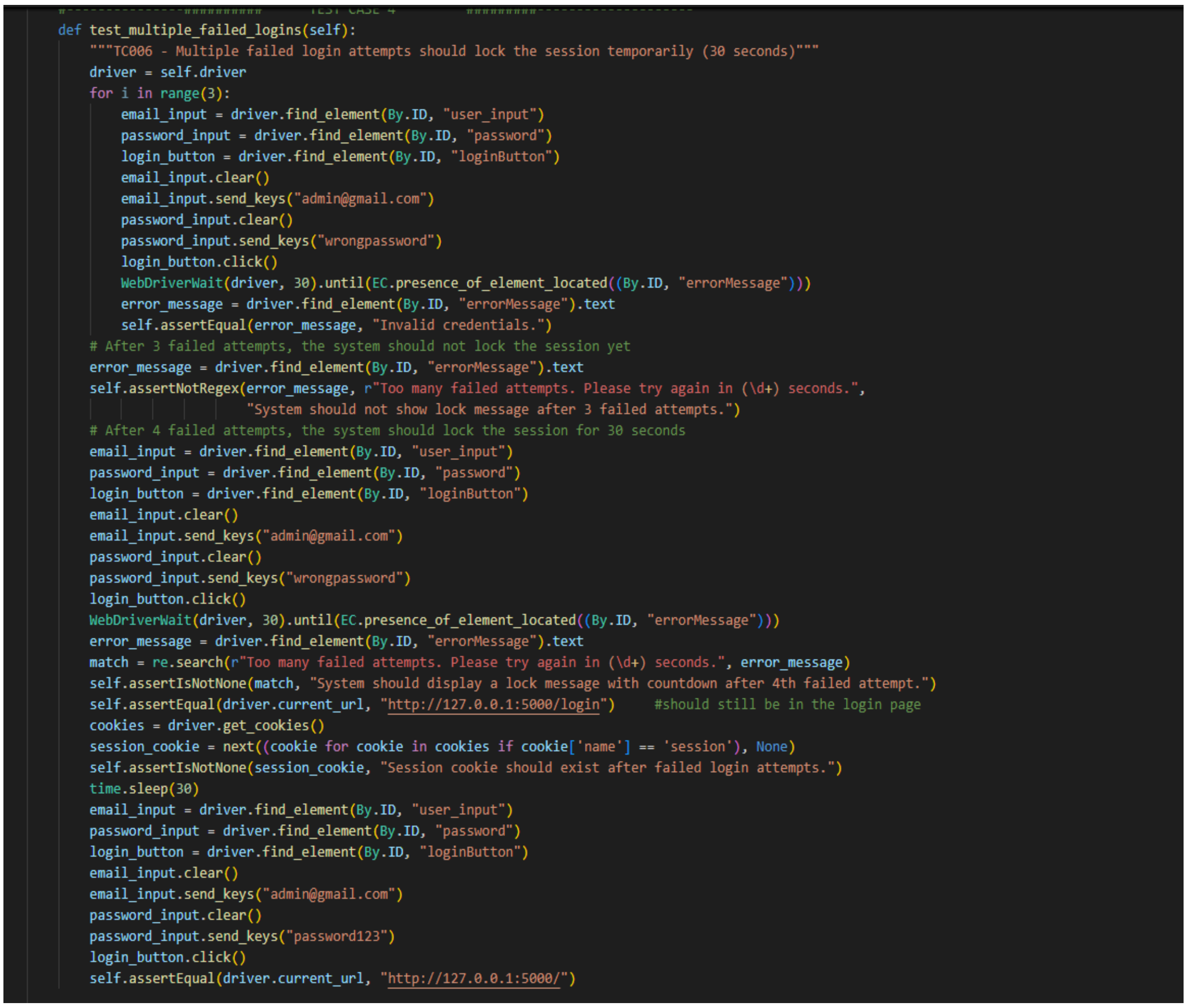
This test case showed no errors as well demonstrating a working app, the second function in this test case checks if those two tabs can be in different browsers. It does this by loading one tab in firefox and one tab in chrome and comparing those two. Again, the test shows no errors.



```python
def test_parallel_logins_chrome_firefox(self):
    """Test logging in with two different accounts in Chrome and Firefox"""
    driver1 = webdriver.Chrome()
    driver1.get("http://127.0.0.1:5000/login")
    driver2 = webdriver.Firefox()
    driver2.get("http://127.0.0.1:5000/login")
    email1 = driver1.find_element(By.ID, "user_input")
    password1 = driver1.find_element(By.ID, "password")
    login_button1 = driver1.find_element(By.ID, "loginButton")
    email1.send_keys("admin@gmail.com")
    password1.send_keys("password123")
    login_button1.click()
    WebDriverWait(driver1, 30).until(EC.url_contains("http://127.0.0.1:5000/"))
    print("First user logged in (Chrome):", driver1.current_url)
    email2 = driver2.find_element(By.ID, "user_input")
    password2 = driver2.find_element(By.ID, "password")
    login_button2 = driver2.find_element(By.ID, "loginButton")
    email2.send_keys("admin2@gmail.com")
    password2.send_keys("password123")
    login_button2.click()
    WebDriverWait(driver2, 30).until(EC.url_contains("http://127.0.0.1:5000/"))
    print("Second user logged in (Firefox):", driver2.current_url)
    cookies1 = driver1.get_cookies()
    cookies2 = driver2.get_cookies()
    session_cookie1 = next((cookie for cookie in cookies1 if cookie['name'] == 'session'), None)
    session_cookie2 = next((cookie for cookie in cookies2 if cookie['name'] == 'session'), None)
    assert session_cookie1 is not None, "Session cookie should exist for first user."
    assert session_cookie2 is not None, "Session cookie should exist for second user."
    assert session_cookie1 != session_cookie2, "Sessions should be different for each user."
    driver1.get("http://127.0.0.1:5000/session_data")
    session_text1 = driver1.find_element(By.TAG_NAME, "body").text
    assert "admin@gmail.com" in session_text1, "Logged-in session user 1 does not match expected email."
    driver2.get("http://127.0.0.1:5000/session_data")
    session_text2 = driver2.find_element(By.TAG_NAME, "body").text
    assert "admin2@gmail.com" in session_text2, "Logged-in session user 2 does not match expected email."
    driver1.quit()
    driver2.quit()
```

Figure 17: Valid Parallel Log in Chrome and Firefox (Test Case 3)

Test Case #4:

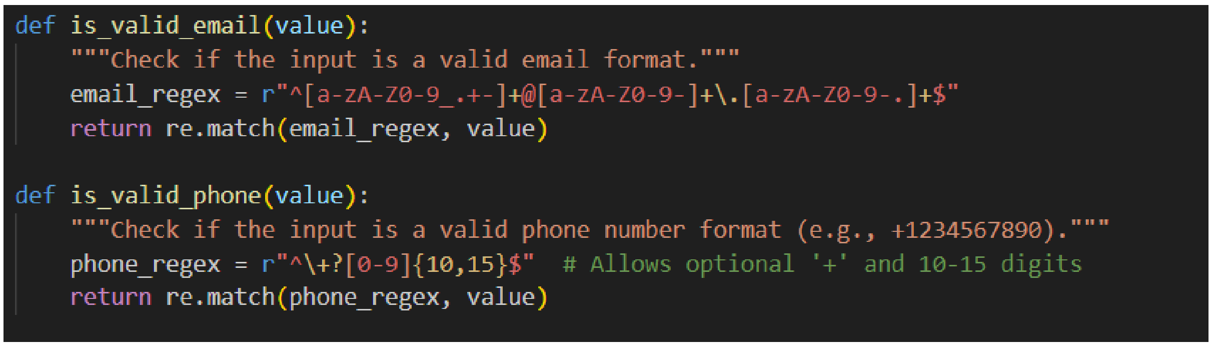


```python
    #---------------##########      TEST CASE 4      #########-------------------
    def test_multiple_failed_logins(self):
        """TC006 - Multiple failed login attempts should lock the session temporarily (30 seconds)"""
        driver = self.driver
        for i in range(3):
            email_input = driver.find_element(By.ID, "user_input")
            password_input = driver.find_element(By.ID, "password")
            login_button = driver.find_element(By.ID, "loginButton")
            email_input.clear()
            email_input.send_keys("admin@gmail.com")
            password_input.clear()
            password_input.send_keys("wrongpassword")
            login_button.click()
            WebDriverWait(driver, 30).until(EC.presence_of_element_located((By.ID, "errorMessage")))
            error_message = driver.find_element(By.ID, "errorMessage").text
            self.assertEqual(error_message, "Invalid credentials.")
        # After 3 failed attempts, the system should not lock the session yet
        error_message = driver.find_element(By.ID, "errorMessage").text
        self.assertNotRegex(error_message, r"Too many failed attempts. Please try again in (\d+) seconds.",
                                "System should not show lock message after 3 failed attempts.")
        # After 4 failed attempts, the system should lock the session for 30 seconds
        email_input = driver.find_element(By.ID, "user_input")
        password_input = driver.find_element(By.ID, "password")
        login_button = driver.find_element(By.ID, "loginButton")
        email_input.clear()
        email_input.send_keys("admin@gmail.com")
        password_input.clear()
        password_input.send_keys("wrongpassword")
        login_button.click()
        WebDriverWait(driver, 30).until(EC.presence_of_element_located((By.ID, "errorMessage")))
        error_message = driver.find_element(By.ID, "errorMessage").text
        match = re.search(r"Too many failed attempts. Please try again in (\d+) seconds.", error_message)
        self.assertIsNotNone(match, "System should display a lock message with countdown after 4th failed attempt.")
        self.assertEqual(driver.current_url, "http://127.0.0.1:5000/login")     #should still be in the login page
        cookies = driver.get_cookies()
        session_cookie = next((cookie for cookie in cookies if cookie['name'] == 'session'), None)
        self.assertIsNotNone(session_cookie, "Session cookie should exist after failed login attempts.")
        time.sleep(30)
        email_input = driver.find_element(By.ID, "user_input")
        password_input = driver.find_element(By.ID, "password")
        login_button = driver.find_element(By.ID, "loginButton")
        email_input.clear()
        email_input.send_keys("admin@gmail.com")
        password_input.clear()
        password_input.send_keys("password123")
        login_button.click()
        self.assertEqual(driver.current_url, "http://127.0.0.1:5000/")
```

Figure 18: Multiple Failed Login Attempts Account Lockout (Test Case 4)

This test case tries to sign in with invalid credentials several times and tests the lockout feature of the application. After 3 failed sign in attempts on the same session, the application should lockout any sign in attempt for 30 seconds. It inputs invalid credentials 3 times, and checks the respective popup for account lockout, as well as the login route to ensure that it does not redirect to the dashboard. Then, after waiting for 30 seconds, it sends a sign in attempt with valid credentials to ensure that the login function works correctly after lockout time passes.

Test Case #5:
The types of inputs of both fields are governed by these regular expressions:

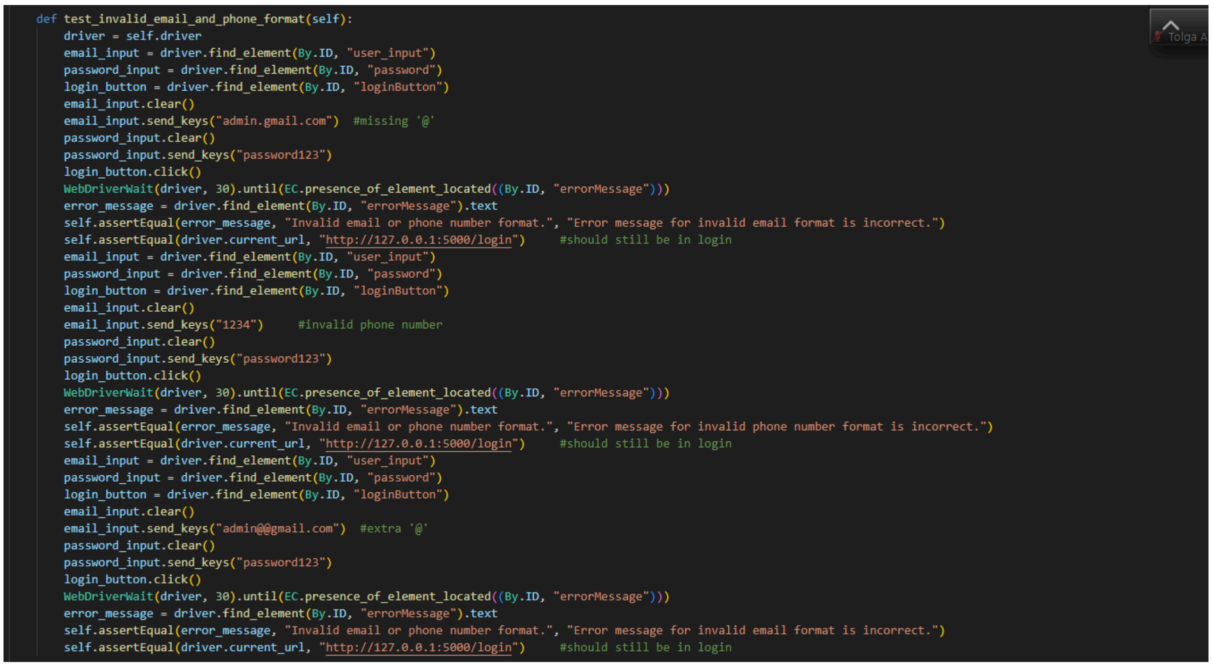


```python
def is_valid_email(value):
    """Check if the input is a valid email format."""
    email_regex = r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$"
    return re.match(email_regex, value)

def is_valid_phone(value):
    """Check if the input is a valid phone number format (e.g., +1234567890)."""
    phone_regex = r"^\+?[0-9]{10,15}$"  # Allows optional '+' and 10-15 digits
    return re.match(phone_regex, value)
```

Figure 19: Regular Expressions For Expected Field Values

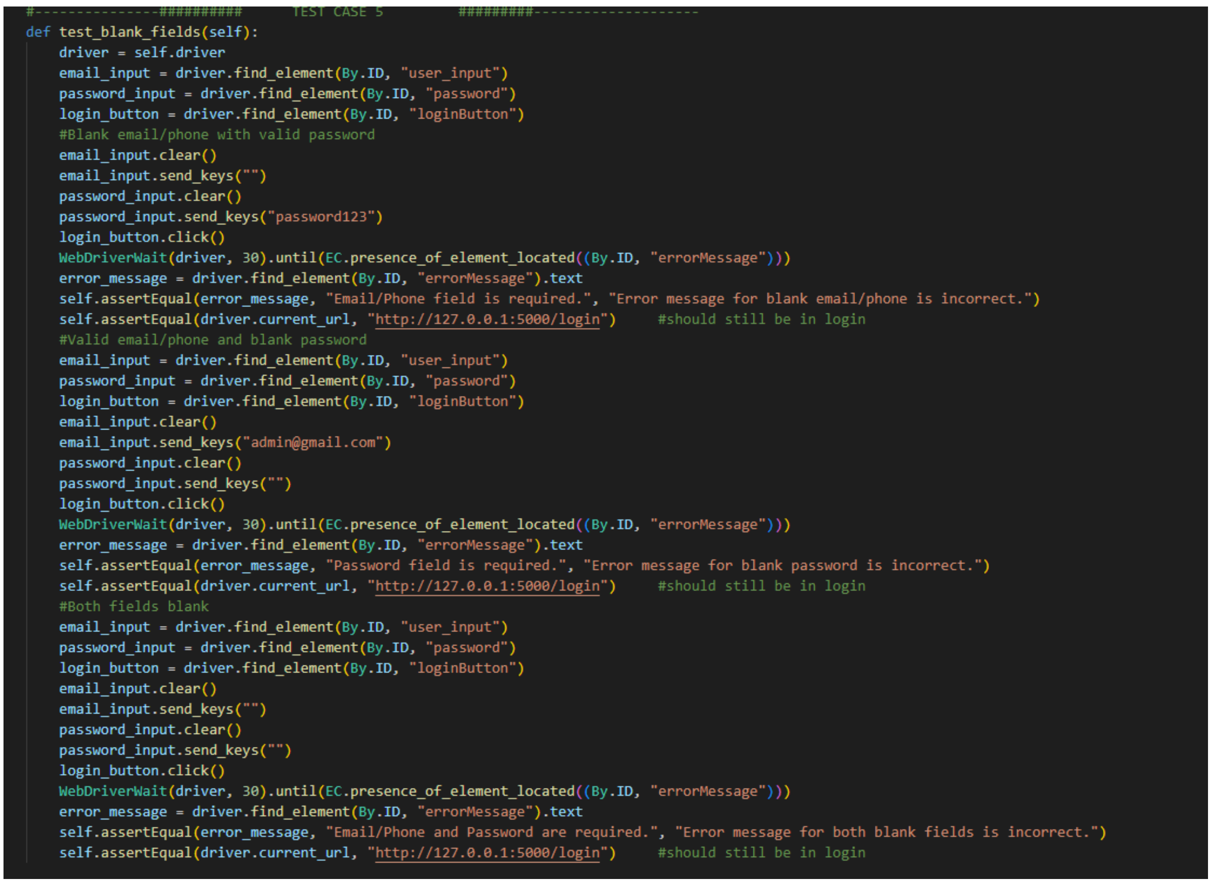
Meaning that there is an additional layer of filtration when the user is trying to log in. These tests determine if the given regular expressions work. When these regular expressions are not adhered to on the input the error message reads the given asserted expected results.



```python
def test_invalid_email_and_phone_format(self):
    driver = self.driver
    email_input = driver.find_element(By.ID, "user_input")
    password_input = driver.find_element(By.ID, "password")
    login_button = driver.find_element(By.ID, "loginButton")
    email_input.clear()
    email_input.send_keys("admin.gmail.com")  #missing '@'
    password_input.clear()
    password_input.send_keys("password123")
    login_button.click()
    WebDriverWait(driver, 30).until(EC.presence_of_element_located((By.ID, "errorMessage")))
    error_message = driver.find_element(By.ID, "errorMessage").text
    self.assertEqual(error_message, "Invalid email or phone number format.", "Error message for invalid email format is incorrect.")
    self.assertEqual(driver.current_url, "http://127.0.0.1:5000/login")     #should still be in login
    email_input = driver.find_element(By.ID, "user_input")
    password_input = driver.find_element(By.ID, "password")
    login_button = driver.find_element(By.ID, "loginButton")
    email_input.clear()
    email_input.send_keys("1234")     #invalid phone number
    password_input.clear()
    password_input.send_keys("password123")
    login_button.click()
    WebDriverWait(driver, 30).until(EC.presence_of_element_located((By.ID, "errorMessage")))
    error_message = driver.find_element(By.ID, "errorMessage").text
    self.assertEqual(error_message, "Invalid email or phone number format.", "Error message for invalid phone number format is incorrect.")
    self.assertEqual(driver.current_url, "http://127.0.0.1:5000/login")     #should still be in login
    email_input = driver.find_element(By.ID, "user_input")
    password_input = driver.find_element(By.ID, "password")
    login_button = driver.find_element(By.ID, "loginButton")
    email_input.clear()
    email_input.send_keys("admin@@gmail.com")  #extra '@'
    password_input.clear()
    password_input.send_keys("password123")
    login_button.click()
    WebDriverWait(driver, 30).until(EC.presence_of_element_located((By.ID, "errorMessage")))
    error_message = driver.find_element(By.ID, "errorMessage").text
    self.assertEqual(error_message, "Invalid email or phone number format.", "Error message for invalid email format is incorrect.")
    self.assertEqual(driver.current_url, "http://127.0.0.1:5000/login")     #should still be in login
```

Figure 20: Invalid Email and Phone Format Login Attempts (Test Case 5)

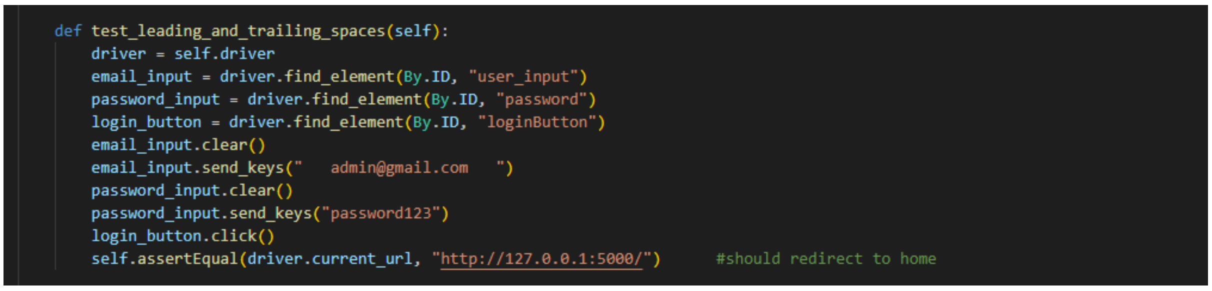
This code attempts to enter invalid email and phone formats and checks the corresponding errors.



```python
#---------------##########      TEST CASE 5       #########--------------------
def test_blank_fields(self):
    driver = self.driver
    email_input = driver.find_element(By.ID, "user_input")
    password_input = driver.find_element(By.ID, "password")
    login_button = driver.find_element(By.ID, "loginButton")
    #Blank email/phone with valid password
    email_input.clear()
    email_input.send_keys("")
    password_input.clear()
    password_input.send_keys("password123")
    login_button.click()
    WebDriverWait(driver, 30).until(EC.presence_of_element_located((By.ID, "errorMessage")))
    error_message = driver.find_element(By.ID, "errorMessage").text
    self.assertEqual(error_message, "Email/Phone field is required.", "Error message for blank email/phone is incorrect.")
    self.assertEqual(driver.current_url, "http://127.0.0.1:5000/login")     #should still be in login
    #Valid email/phone and blank password
    email_input = driver.find_element(By.ID, "user_input")
    password_input = driver.find_element(By.ID, "password")
    login_button = driver.find_element(By.ID, "loginButton")
    email_input.clear()
    email_input.send_keys("admin@gmail.com")
    password_input.clear()
    password_input.send_keys("")
    login_button.click()
    WebDriverWait(driver, 30).until(EC.presence_of_element_located((By.ID, "errorMessage")))
    error_message = driver.find_element(By.ID, "errorMessage").text
    self.assertEqual(error_message, "Password field is required.", "Error message for blank password is incorrect.")
    self.assertEqual(driver.current_url, "http://127.0.0.1:5000/login")     #should still be in login
    #Both fields blank
    email_input = driver.find_element(By.ID, "user_input")
    password_input = driver.find_element(By.ID, "password")
    login_button = driver.find_element(By.ID, "loginButton")
    email_input.clear()
    email_input.send_keys("")
    password_input.clear()
    password_input.send_keys("")
    login_button.click()
    WebDriverWait(driver, 30).until(EC.presence_of_element_located((By.ID, "errorMessage")))
    error_message = driver.find_element(By.ID, "errorMessage").text
    self.assertEqual(error_message, "Email/Phone and Password are required.", "Error message for both blank fields is incorrect.")
    self.assertEqual(driver.current_url, "http://127.0.0.1:5000/login")     #should still be in login
```

Figure 21: Blank Field Login Attempts (Test Case 5)

This test case checks if the responses for blank fields are given when there are blank fields. The error messages are classic info required messages as can be seen in the assertions in the code.



```python
def test_leading_and_trailing_spaces(self):
    driver = self.driver
    email_input = driver.find_element(By.ID, "user_input")
    password_input = driver.find_element(By.ID, "password")
    login_button = driver.find_element(By.ID, "loginButton")
    email_input.clear()
    email_input.send_keys("   admin@gmail.com   ")
    password_input.clear()
    password_input.send_keys("password123")
    login_button.click()
    self.assertEqual(driver.current_url, "http://127.0.0.1:5000/")      #should redirect to home
```

Figure 22: Leading and Trailing Spaces in Email/Phone Field (Test Case 5)

The app also ignores leading and trailing spaces in the email/phone number field, this is tested by this code which it passes

Evaluation

Our Testing Process

Our experience with Selenium automation testing was both challenging and rewarding. At first, setting up the test environment took more time than we expected. We had to learn how Selenium works, figure out how to select elements on our web page, and handle things like waiting for page loads.

Sometimes our tests would work perfectly, but other times they would fail randomly because of timing issues. This was frustrating, especially when tests passed on one team member's computer but failed on another's. The running time of our tests varied depending on local network speed and computer performance. The tests also still encounter an issue with tabbing out which we still correlate with network speeds

Contribution to the Development Cycle
Test automation significantly contributes to the software development lifecycle (SDLC) in terms of both velocity and quality. Selenium automated tests provide immediate feedback to developers, allowing them to identify and fix issues early in the development cycle when they're less expensive to address. Manual regression testing of the login functionality would take so much time, maybe an hour every time, but our automated suite executes much quicker, allowing for more frequent testing. Testers can develop new test cases while automated tests run, increasing overall team productivity. With confidence in automated test coverage, release cycles can be shortened without sacrificing quality.

Regarding quality enhancements, every test run executes the exact same steps, ensuring nothing is overlooked due to human error or time constraints. Automated tests detect subtle issues that might be missed during manual testing, such as specific error messages or state transitions. When defects are found, they can be consistently reproduced using the automated test, making them easier to fix. Our test scripts serve as living documentation of expected application behavior, improving knowledge sharing among team members.

The long-term benefits include reduced technical debt as regular automated testing prevents the accumulation of undetected bugs that can become costly to fix later. Knowing that automated tests will catch regressions, developers can make changes more confidently. Frequent automated testing results in a more stable product with fewer defects making it to production.

In conclusion, our Selenium automation for the login functionality has demonstrated significant value in both accelerating development velocity and enhancing product quality. While requiring initial investment in setup and learning, the long-term benefits in terms of efficiency, coverage, and reliability have proven worthwhile for our team's software development process.