

EXPERIMENT-12

AIM: Develop test cases for the above containerized application using selenium.

Objective: To validate the functional, UI, authentication, and integration behavior of the web application running inside Docker by automating end-to-end tests using Selenium (remote WebDriver / Selenium container).

Software Requirements

- Code editor (VS Code or any)
 - Docker Desktop
 - Python (for running scripts locally if needed)
 - Selenium (pip install selenium)
 - Web browser (Chrome, Firefox, etc.)
-

Lab Procedure

Clone this repository to your local repository

<https://github.com/Srivaishnavi08/tests>

Or follow the below steps

Step-1:

Create a directory named selenium-test and navigate to the current directory path.

tests/

```
|── Dockerfile  
|── index.html  
|── SeleniumTest.py  
|── docker-compose.yml
```

Step 2: Create the index.html file

This is your sample web page that Selenium will interact with.

index.html

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Selenium Login Example</title>
</head>
<body>
    <!-- Homepage -->
    <div id="homepage">
        <h1>Welcome!!!!!!</h1>
        <!-- Get Started Free Button -->
        <a href="#loginPage" id="get-started" onclick="navigateToLogin()">Get
        started</a>
    </div>

    <!-- Login Page -->
    <div id="loginPage" style="display: none;">
        <h2>Login to the Page</h2>
        <form onsubmit="return validateLogin()">
            <label for="user_email_login">Email:</label>
            <input type="email" id="user_email_login" name="user_email_login"
            required>
            <br><br>
            <label for="user_password">Password:</label>
            <input type="password" id="user_password" name="user_password"
            required>
            <br><br>
            <button type="submit" name="commit">Login</button>
        </form>
        <p id="error-message" style="color: red; display: none;">Invalid
        credentials, please try again.</p>
    </div>
</body>
</html>
```

```
</div>

<!-- Dashboard Section (only shown after successful login) -->
<div id="dashboard" style="display: none;">
    <h2>Welcome to Your Dashboard!</h2>
    <p>This is the dashboard area you see after a successful login.</p>
</div>

<script>
    // Function to navigate to the login page
    function navigateToLogin() {
        document.getElementById('homepage').style.display = 'none';
        document.getElementById('loginPage').style.display = 'block';
    }

    // Function to validate login credentials
    function validateLogin() {
        const email = document.getElementById('user_email_login').value;
        const password = document.getElementById('user_password').value;

        if (email === "abc@gmail.com" && password === "password") {
            // Hide login page and display dashboard
            document.getElementById('loginPage').style.display = 'none';
            document.getElementById('dashboard').style.display = 'block';
            return false; // Prevent actual form submission
        } else {
            // Show error message if credentials are incorrect
            document.getElementById('error-message').style.display = 'block';
            return false; // Prevent actual form submission
        }
    }

```

```
    }
</script>
</body>
</html>
```

Step 3: Create the dockerfile

This dockerfile sets up a simple HTTP server to serve your index.html file.

Dockerfile

```
# Use the official Python image as the base image
```

```
FROM python:3.9
```

```
# Set the working directory in the container
```

```
WORKDIR /app
```

```
# Copy the index.html file to the container
```

```
COPY index.html .
```

```
# Expose port 8000 for the HTTP server
```

```
EXPOSE 8000
```

```
# Start a simple HTTP server to serve the index.html file
```

```
CMD ["python", "-m", "http.server", "8000"]
```

Step-4: Create the Selenium Test Script (seleniumTest.py)

This script will automate testing of your HTML page using Selenium.

```
from selenium import webdriver
```

```
from selenium.webdriver.common.by import By
```

```
from selenium.webdriver.support.ui import WebDriverWait
```

```
from selenium.webdriver.support import expected_conditions as EC
```

```
import time
```

```
print("Test Execution Started")
options = webdriver.ChromeOptions()
options.add_argument('--ignore-ssl-errors=yes')
options.add_argument('--ignore-certificate-errors')

# Start the Selenium WebDriver
driver = webdriver.Remote(
    command_executor='http://localhost:4444/wd/hub',
    options=options
)

# Maximize the window size
driver.maximize_window()
time.sleep(10)
driver.get("http://host.docker.internal:8000") # Access the local server
time.sleep(10)

try:
    # Wait for the "Get started free" link to be clickable
    link = WebDriverWait(driver, 30).until(
        EC.element_to_be_clickable((By.LINK_TEXT, "Get started"))
    )
    link.click() # Click the link
    time.sleep(10) # Wait for any resulting page to load

    WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, "user_email_login"))
    )
    WebDriverWait(driver, 10).until(
```

```
        EC.presence_of_element_located((By.ID, "user_password"))
    )

# Enter login credentials
username = driver.find_element(By.ID, "user_email_login")
password = driver.find_element(By.ID, "user_password")
login_button = driver.find_element(By.NAME, "commit")

username.send_keys("abc@gmail.com") # Replace with actual username
password.send_keys("password") # Replace with actual password
login_button.click()

# Check for a post-login element (adjust to your page's unique element for
logged-in users)
try:
    error_message = WebDriverWait(driver, 10).until(
        EC.visibility_of_element_located((By.ID, "error-message"))
    )
    time.sleep(10)
    print("Login failed: Incorrect credentials")
except:
    # No error message found, proceed with checking for dashboard
    WebDriverWait(driver, 10).until(
        EC.visibility_of_element_located((By.ID, "dashboard")) # Replace with
actual post-login element ID
    )
    print("Login Successful!")

except Exception as e:
    print(f"An error occurred while trying to click the link: {e}")
```

```
finally:  
    # Ensure the browser quits after execution  
    driver.quit()  
    print("Test Execution Completed!")
```

Step 5: Create the docker-compose.yml File

This file defines two services: your HTML server and the Selenium Chrome container.

(docker-compose.yml)

services:

```
app:  
  build:  
    context: .  
    dockerfile: Dockerfile  
  container_name: html-server  
  ports:  
    - "8000:8000"
```

```
selenium:  
  image: selenium/standalone-chrome  
  container_name: selenium-chrome  
  ports:  
    - "4444:4444"  
  depends_on:  
    - app
```

Step-6:

- Build and Run Your Docker Containers
- In terminal use the command

docker compose -f docker-compose.yml up --build

This command will:

- Build the Docker image for your HTML server.
- Pull the Selenium standalone Chrome image.

```
E:\CNRCECT20240105\III Yr\Devops Lab Exp Final\Exp-12\tests>docker compose -f docker-compose.yml up --build
[+] Building 59.4s (11/11) FINISHED
  => [internal] load local bake definitions
  => => reading from stdin 564B
  => [internal] load build definition from Dockerfile
  => => transferring dockerfile: 396B
  => [internal] load metadata for docker.io/library/python:3.9
  => [auth] library/python:pull token for registry-1.docker.io
  => [internal] load .dockerignore
  => => transferring context: 2B
  => [1/3] FROM docker.io/library/python:3.9@sha256:61c518a24fa2c5e6c2ead2b29bc0c81ff7691d6f36459d2e399d4e25ddc0db38
  => => resolve docker.io/library/python:3.9@sha256:61c518a24fa2c5e6c2ead2b29bc0c81ff7691d6f36459d2e399d4e25ddc0db38
  => [internal] load build context
  => => transferring context: 32B
  => CACHED [2/3] WORKDIR /app
  => CACHED [3/3] COPY index.html .
  => exporting to image
  => exporting layers
  => => exporting manifest sha256:c5ae8632caeab87ebe9338ad9fcfc1f5204f3fa1404eed6c1738767c57e01
  => => exporting config sha256:dc57edbd6d681abffed02b7e93fd48a497c1521f714365279fa4a7fefef671b7
  => => exporting attestation manifest sha256:373fd5b14253682773f5e08447643e08f672521bc267f8535de458976421113
  => => exporting manifest list sha256:fb42a91e0f2e9efb02fef913c58fd9352079209ebc7e37ddfe148673c189887d
  => => naming to docker.io/library/tests-app:latest
  => => unpacking to docker.io/library/tests-app:latest
  => => resolving provenance for metadata file
[+] Running 2/2
  tests-app           Built
  Container html-server Recreated
Attaching to html-server, selenium-chrome
selenium-chrome    Virtual environment detected at /opt/venv, activating...
selenium-chrome    Python 3.12.3
selenium-chrome    2025-10-28 16:40:58,088 INFO Included extra file "/etc/supervisor/conf.d/chrome-setup.conf" during parsing
selenium-chrome    2025-10-28 16:40:58,095 INFO Included extra file "/etc/supervisor/conf.d/recorder.conf" during parsing
selenium-chrome    2025-10-28 16:40:58,096 INFO Included extra file "/etc/supervisor/conf.d/selenium.conf" during parsing
selenium-chrome    2025-10-28 16:40:58,099 INFO Included extra file "/etc/supervisor/conf.d/uploader.conf" during parsing
selenium-chrome    Unlinking stale socket /tmp/supervisor.sock
selenium-chrome    2025-10-28 16:40:59,539 INFO RPC interface 'supervisor' initialized
selenium-chrome    2025-10-28 16:40:59,541 INFO supervisord started with pid 10
selenium-chrome    2025-10-28 16:41:00,551 INFO spawned: 'xvfb' with pid 11
selenium-chrome    2025-10-28 16:41:00,598 INFO spawned: 'vnc' with pid 12
selenium-chrome    2025-10-28 16:41:01,087 INFO spawned: 'novnc' with pid 13
selenium-chrome    2025-10-28 16:41:01,293 INFO spawned: 'selenium-standalone' with pid 15
selenium-chrome    2025-10-28 16:41:01,611 INFO success: xvfb entered RUNNING state, process has stayed up for > than 0 seconds (startsecs)

```

- Start both services.
- Press v to navigate to docker.

●	tests													
●	selenium-chr	85ad964df47d		selenium/s_ 4444:4444			3.15%	181.7MB / 1.81Gi	9.78%	142MB				
●	html-server	f641b54bf357		tests-app 8000:8000			0.03%	13.54MB / 1.81Gi	0.73%	18.5MB				

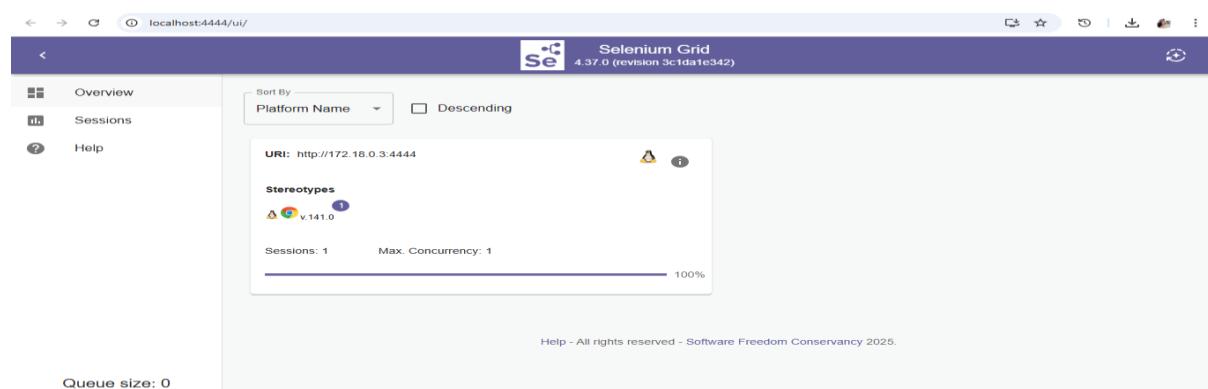
- Click on the links

8000:8000



Welcome!!!!!!
[Get started](#)

4444:4444



Step 7:

Run the Selenium Test Script

While your containers are running, open a new terminal and run:

python seleniumTest.py

Go to sessions, click on video icon.

The screenshot shows the Selenium Grid UI at `localhost:4444/ui/#/sessions`. On the left, there's a sidebar with 'Overview', 'Sessions' (selected), and 'Help'. The main area has a title bar 'Selenium Grid 4.26.0 (revision 69fb65e)'. Below it, a table lists a single session:

Session	Capabilities	Start time	Duration ↑	Node URI
de87d0970c00d6f3f8159b0aa785096e	Chrome v.130.0.6723.91	13/11/2024 17:26:37	2.5s	http://172.18.0.3:4444

Below the table, it says 'Queue size: 0' and 'Concurrency' with a 100% circle. A video camera icon next to the session row indicates a video stream is available.

Enter the password “secret” to view the video of automatic testing.

The screenshot shows the session details for the session `de87d0970c00d6f3f8159b0aa785096e`. It displays the capabilities and start time. A modal dialog titled 'LiveView (VNC) Password' is in the foreground, asking for a password. The word 'secret' is typed into the password field. There are 'CANCEL' and 'ACCEPT' buttons at the bottom of the dialog.

Step 8:

Stop the Docker Containers Once you are done testing.

Stop the containers using: **docker compose -f docker-compose down**

This command stops and removes the containers, networks, and any volumes associated with the services.

After running the test and clicking the video icon, the video will show how the test interacts with the web page (clicking buttons, entering passwords, etc.)