

Experiment 10: Install and Explore Selenium for Automated Testing

Objective

To install and explore Selenium WebDriver — an automation tool for testing web applications — on both macOS and Windows platforms using JavaScript (Node.js).

Software / Tools Required

- Node.js (with npm)
- Google Chrome browser
- ChromeDriver
- Selenium WebDriver (JavaScript client library)
- macOS (Intel / Apple Silicon) or Windows 10/11

Theory

Selenium is an open-source framework for automating browsers. It allows developers and testers to simulate user interactions — clicking, typing, submitting forms, and verifying page behavior — for web-based functional testing.

Key Components:

1. **Selenium WebDriver** – Controls browsers through code.
2. **Selenium IDE** – A simple record-and-playback tool.
3. **Selenium Grid** – Executes tests on multiple machines and browsers in parallel.

Procedure

A. Install Prerequisites

macOS (Intel / Apple Silicon)

1. **Install Homebrew** (if not installed):

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2. Install Node.js (npm included):

```
brew install node
```

```
node -v
```

```
npm -v
```

Windows 10 / 11

1. Install Node.js:

- Download the LTS installer from <https://nodejs.org> or

Use Chocolatey (Run PowerShell as Admin):

```
choco install nodejs-lts -y
```

```
node -v
```

```
npm -v
```

B. Install Selenium Client Library (JavaScript)

Same for both OS:

```
mkdir selenium-lab && cd selenium-lab
```

```
npm init -y
```

```
npm install selenium-webdriver
```

C. Install Chrome and ChromeDriver

macOS

1. Install Google Chrome:

```
brew install --cask google-chrome
```

2. Install ChromeDriver:

```
brew install chromedriver  
chromedriver --version
```

For Apple Silicon, driver may be located at `/opt/homebrew/bin/chromedriver`.

If macOS blocks execution:

```
xattr -d com.apple.quarantine /opt/homebrew/bin/chromedriver
```

Alternative (project-local install):

```
npm install chromedriver
```

Windows

1. Install Google Chrome:

```
choco install googlechrome -y
```

2. Install ChromeDriver:

- **Option 1:** Download matching `chromedriver.exe` from <https://chromedriver.chromium.org/downloads>, place it in `C:\WebDriver\`, and add that folder to PATH.

- **Option 2:**

```
npm install chromedriver
```

3. *Ensure ChromeDriver major version matches Chrome.*

D. Verify Selenium Setup (Exploration Test)

Create a file named **testSetup.js**:

```
// testSetup.js
const { Builder } = require('selenium-webdriver');

(async function () {
  let driver = await new
Builder().forBrowser('chrome').build();
  try {
    await driver.get('https://example.com');
    console.log('Title:', await driver.getTitle());
  } finally {
    await driver.quit();
  }
})();
```

Run the script:

```
node testSetup.js
```

Expected Behavior:

- A Chrome window launches, navigates to <https://example.com>, and closes.
- Terminal prints the page title, e.g.:

Title: Example Domain

Output

- Selenium WebDriver successfully launches Chrome and retrieves the webpage title.
- Installation verified on both macOS and Windows.

Result

Selenium WebDriver was successfully installed and explored on macOS and Windows. The test script executed correctly, confirming browser automation functionality.

Conclusion

Selenium WebDriver provides a cross-platform automation interface for browser testing. Installing and verifying Selenium on both macOS and Windows ensures readiness for developing automated test scripts in later experiments.

Experiment 11: Write a Simple JavaScript Program and Test it using Selenium

Objective

To write a simple Selenium WebDriver test script in **JavaScript** that automates a Google search and validates the result page.

Software / Tools Required

- Node.js (with npm)
- Selenium WebDriver (JavaScript client)
- Google Chrome Browser
- ChromeDriver
- macOS / Windows

Theory

Selenium WebDriver allows programmatic control of a browser — navigating to URLs, filling forms, clicking elements, and verifying web content.

When integrated with JavaScript (Node.js), it enables developers to create **automated browser tests** that mimic real user actions.

Selenium interacts with browsers using a driver interface — in this case, **ChromeDriver** for the Chrome browser.

Procedure

A. Setup Environment

(Prerequisites should already be completed from **Experiment 10**.)

Create a new folder:

```
mkdir selenium-test && cd selenium-test
```

```
npm init -y
```

```
npm install selenium-webdriver
```

Ensure Chrome and ChromeDriver are properly installed and on PATH.

B. Write a Simple Test Script

Create a file named **googleTest.js** and add the following code:

```
// googleTest.js
const { Builder, By, Key, until } = require('selenium-
webdriver');

(async function googleSearchTest() {
  let driver = await new
Builder().forBrowser('chrome').build();
  try {
    await driver.get('https://www.google.com');

    // Handle consent popup (if visible, for EU users)
    try {
      let accept = await
driver.findElements(By.css('button[aria-label="Accept
all"]'));
      if (accept.length) await accept[0].click();
    } catch (e) {}

    // Find the search box and perform a query
    const q = await driver.findElement(By.name('q'));
    await q.sendKeys('DevOps', Key.RETURN);

    // Wait until the page title contains "DevOps"
    await driver.wait(until.titleContains('DevOps'), 7000);
    console.log('✅ Page title:', await driver.getTitle());
  } catch (err) {
    console.error('❌ Test failed:', err);
  } finally {
    await driver.quit();
  }
})();
```

C. Run the Test

In the terminal:

```
node googleTest.js
```

Expected Behavior:

- Chrome launches.
- Navigates to Google.

- Searches for “DevOps”.
- Prints the page title (e.g., “DevOps - Google Search”).
- Closes the browser automatically.

D. Headless Mode (for CI/CD Environments)

In continuous integration environments (e.g., Jenkins, GitHub Actions), browsers cannot open visually.

We can run Chrome in **headless mode** — without displaying a window.

Modify your script to include Chrome options:

```
const { Builder, By, Key, until } = require('selenium-
webdriver');
const chrome = require('selenium-webdriver/chrome');

const options = new chrome.Options().addArguments(
  '--headless=new',          // New headless mode (Chrome 109+)
  '--no-sandbox',
  '--disable-dev-shm-usage',
  '--window-size=1280,800'
);

(async function googleSearchHeadless() {
  let driver = await new Builder()
    .forBrowser('chrome')
    .setChromeOptions(options)
    .build();

  try {
    await driver.get('https://www.google.com');
    const q = await driver.findElement(By.name('q'));
    await q.sendKeys('DevOps', Key.RETURN);
    await driver.wait(until.titleContains('DevOps'), 7000);
    console.log('✅ Headless test passed. Title:', await
driver.getTitle());
  } finally {
    await driver.quit();
  }
})();
Run:
```

```
node googleTest.js
```


E. Explanation of Code

Line / Section	Description
<code>Builder()</code>	Creates a new WebDriver instance for the chosen browser (Chrome).
<code>driver.get()</code>	Opens a URL in the browser.
<code>driver.findElement(By.name('q'))</code>	Locates the Google search input field.
<code>sendKeys('DevOps', Key.RETURN)</code>	Types a search query and presses Enter.
<code>until.titleContains('DevOps')</code>	Waits until the page title contains “DevOps”.
<code>driver.quit()</code>	Closes the browser and ends the session.

Output



Page title: DevOps – Google Search

Browser closes automatically after the test.

Result

A JavaScript Selenium test script was successfully created and executed.

The script automated a Google search, verified the page title, and demonstrated browser automation using Selenium WebDriver.

Conclusion

This experiment shows how Selenium WebDriver can be used with JavaScript to automate real browser actions.

Such scripts can be expanded into test suites for validating web applications, integrated with CI/CD pipelines, and executed in headless mode for continuous testing.

Experiment 12. Develop Test Cases for the Containerized Application using Selenium

Objective

To develop and execute automated test cases using **Selenium WebDriver** for a containerized web application, ensuring the correctness of its functionality and deployment consistency.

Software / Tools Required

- **Docker** – for containerization
- **Node.js** – for running Selenium scripts
- **Selenium WebDriver** – for browser automation
- **Chrome & ChromeDriver** – for browser-based testing
- **Nginx** – to serve the HTML application inside a container

Description

The web application (`Event_registration.html`) is a simple event registration form hosted inside a Docker container using **Nginx**.

We use **Selenium WebDriver** (JavaScript) to automate the form submission and validate the success message.

This ensures the containerized application behaves correctly under automated testing.

Test Environment Setup

Create Dockerfile

```
FROM nginx:alpine
```

```
COPY Event_registration.html /usr/share/nginx/html/index.html
```

```
EXPOSE 80
```

```
CMD ["nginx", "-g", "daemon off;"]
```

Build and Run the Container

```
docker build -t event-app:1.0 .
```

```
docker run -d -p 8000:80 --name event-container event-app:1.0
```

Install Selenium Dependencies

```
npm install selenium-webdriver chromedriver
```

Run Selenium Test

```
node 3_registrationTest.js
```

Selenium Test Script (3_registrationTest.js)

```
const { Builder, By, until } = require('selenium-webdriver');

(async function registrationFormTest() {
  const driver = await new
  Builder().forBrowser('chrome').build();
  try {
    await driver.get('http://localhost:8000');
    await
    driver.findElement(By.id('fullname')).sendKeys('John Doe');
    await
    driver.findElement(By.id('email')).sendKeys('john@example.com
  ');
    await
    driver.findElement(By.id('phno')).sendKeys('9876543210');
    await
    driver.findElement(By.css('input[value="Male"]')).click();
    await
    driver.findElement(By.id('event')).sendKeys('Workshop');
    await driver.findElement(By.id('address')).sendKeys('123
    Test Street');
    await
    driver.findElement(By.css('input[type="submit"]')).click();

    await driver.wait(until.alertIsPresent(), 5000);
    const alert = await driver.switchTo().alert();
    const msg = await alert.getText();
    console.log('Alert message:', msg);
    await alert.accept();







    if (msg.includes('Successful')) console.log('✅ Test
    Passed');
    else console.log('❌ Unexpected message');
  }
  catch (err) {
```

```

        console.error('❌ Error:', err);
    }
    finally {
        await driver.quit();
    }
}
})();

```

Test Cases

Test Case ID	Test Scenario	Test Steps	Expected Result	Actual Result	Status
TC01	Verify page loads successfully	Open <code>http://localhost:8000</code>	Registration form should load properly	Form loaded	 Pass
TC02	Validate all form fields	Enter valid data in all input fields	Data should be accepted	Accepted	 Pass
TC03	Check form submission	Click on “Submit” after filling the form	Alert message “Registration Successful!” should appear	Alert displayed	 Pass
TC04	Check invalid phone number	Enter fewer than 10 digits	Alert: “Please enter a valid 10-digit Phone Number.”	Alert displayed	 Pass
TC05	Check email validation	Enter invalid email (no “@”)	Alert: “Please enter a valid Email ID.”	Alert displayed	 Pass
TC06	Reset functionality	Click on “Reset”	All input fields should clear	Fields cleared	 Pass


Output

Docker container started successfully.

Selenium automated the form filling process.

Console output:

Alert message: Registration Successful!

 Test Passed

Result

The containerized web application was successfully tested using Selenium WebDriver. All functional test cases passed, confirming that the application behaves correctly within the containerized environment.

Conclusion

Selenium provides an effective way to automate browser-based testing for applications deployed in Docker containers.

This approach ensures consistency, reliability, and repeatability of test execution in DevOps workflows.