5 Web layer test automation

v2025-10-21

Learning objectives

- Write web-layer tests using Selenium API or Playwright Java API and automate their execution in JUnit.
- Understand best practices for elements location and possible causes of flaky tests
- Apply the Page Object Pattern to enhance tests maintainability.

Key Points

- Acceptance tests exercise the user interface of the system as if a real user was using the application. The system is treated as a black box.
- Browser automation (control the browser interaction from a script) is an essential step to implement acceptance tests on web applications. There are several frameworks for browser automation (e.g.: Puppeteer); for Java, the most used frameworks are the WebDriver API, provided by Selenium (that can be used with JUnit engine) and the more recent Playwright Java API.
- The test script can easily get "messy" and hard to read. To improve the code (and its maintainability) we could apply the Page Objects Pattern.
- Web browser automation is also helpful to implement “smoke tests”.

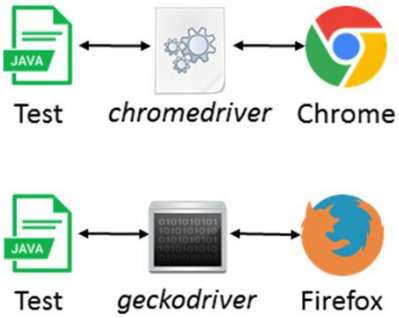
Useful resources

- B. Garcia’s Book “Hands-On Selenium WebDriver with Java”
- Criticism on the Page Object Pattern for modern web apps (and alternatives).
- Playwright online documentation, namely the Playwright Java API.

5.1 WebDriver “hello world”

Selenium WebDriver offers a programming interface (i.e., API) to drive/control a web browser, as if a real user was operating the browser.

You are required to install the driver implementation (binding) for the specific browser you want to use. This step can be simplified with the help of a Web Driver management framework (such as included in the Selenium-Jupiter extension).



a) Implement the example discussed in the “hello world” section of B. Garcia’s book (search “Example 2-1”).

   Adapt to a custom scenario (choose another web site,…).

   Main dependencies for POM.xml:

   - `JUnit 5: org.junit.jupiter:`**`junit-jupiter`**
   - `Selenium: org.seleniumhq.selenium:`**`selenium-java`**
   - `Selenium-JUnit extension: io.github.bonigarcia:` **`selenium-jupiter`**

b) Enrich your test script to consider the following actions:
   - Navigate to the “Slow calculator” link present in the initial page from previous exercise (e.g.*: findElement(By.linkText("Slow calculator")*)
   - Assert that you arrived to the correct page (e.g.: assess *driver.getCurrentUrl()*)

c) Refactor (or create an additional test class) to use the approach in “Example 2-2”, based on Selenium-Jupiter extension (which uses **parameter-level dependency injection** to get the WebDriver instance). Note the changes: (1) @ExtendWith(SeleniumJupiter.class); (2) use a dependency injection to get a “browser” instance; (3) no explicit need for “quit”.

5.2 Interactive test recording

Often you can record your tests interactively to explore the “locators” (e.g.: id for a given web element) speeding up the test preparation.

Record the test interactively

a) Install the Selenium IDE plug-in/add-on for your browser or the alternative Katalon Recorder plugin.

   Using the https://blazedemo.com/ dummy travel agency web app, record a test in which you select a and buy a trip. Be sure to add relevant “asserts” or verifications to your test.

   Replay the test and confirm the success. Also experiment to break the test (e.g.: by explicitly editing the parameters of some test step in the test script).

b) Add a new step, at the end, to assert that the confirmation page contains the title “BlazeDemo Confirmation”. Enter this assertion “manually” (in the editor panel, **not** by recording).

c) Be sure to save you Selenium IDE test project (it creates a *.side file, to be included in your git).

Export and run the test using the WebDriver API

d) Export the test from Selenium IDE into a Java test class and include it in the previous project.

Important: you need to refactor the generated code to be compliant with JUnit 5! Select only the relevant parts from the generated output.

Run the test programmatically (as a JUnit 5 test).

Refactor to use Selenium extensions for JUnit 5

Be sure to use the Selenium-Jupiter extension. Note that this library will ensure several tasks:

- enable dependency injection with respect to the WebDriver implementation (i.e., hides the use of WebDriverManager to resolve the specific browser implementation). You do not need to pre-install the WebDriver binaries; they are retrieved on demand.
- if using dependency injection, it will also ensure that the WebDriver is initialized and closed.
- You may check this related example (consider only for the parts related to the WebDriver…).

5.3 Selecting elements with locators

Consider the test case of searching for a book in this (fictitious) online store.

a) Implement a simple test that, for a search for “Harry Potter”, finds a result that corresponds to the book “Harry Potter and the Sorcerer's Stone” by J. K. Rowlling.

b) Check the locators being used. Are there any instances of “xpath”? What about identifier-based locators? Which locator strategies are more robust?

c) Create a new version of the previous test, considering a refactoring that (1) favors well defined selectors[1], and (2) increases robustness by introducing explicit waits that tolerate search response latency.

By.cssSelector("[data-testid=book-search-item]”

5.4 Page Object pattern

Consider the example discussed here (or, for a more in depth discussion, here).

Note: the target web site implementation may have changed from the time the article was written, and the example may require some adaptations to pass the tests…

“Page Object model is an **object design pattern** where webpages are represented as classes, and the various elements [of interest] on the page are defined as variables on the class. All possible user interactions [on a page] can then be implemented as methods on the class.”

a) Implement the “Page object” design pattern for a cleaner and more readable test using the same application problem from exercise 5.2, in a new project.

Suggestion: use the Page Factory approach to bind class attributes to the web page elements.

[1] You may need the inspect the page elements/HTML to find identifiers.

5.5 Playwright: “hello world”

Playwright uses a totally different approach to automate the target browsers, by using CDP (Chrome DevTools Protocol) and provides auto-waiting capabilities avoid many of the flakiness usually found in Selenium tests. Whenever using Playwright it will install local versions of Chromium, Firefox, WebKit for the sole purpose of testing. Playwright uses 3 core concepts: browsers, contexts, and pages.

A Browser is an instance of a real browser (e.g., Chromium). BrowserContexts provide a way to operate multiple independent, non-persistent browser sessions; we can think of a BrowserContext as a opening a new incognito window – it’s quite fast to create one. A Page refers to a single tab or a popup window within a browser context - it’s also quite fast to create one/many.

Like the previous exercise (#5.1), implement a basic test using Playwright and its Java bindings.

a) You can reimplement the same “hello world” example, using Playwright instead of Selenium.

   Main dependencies for POM.xml (you need to adapt them!):

   - `JUnit 5: org.junit.jupiter:junit-jupiter`
   - `Playwright: com.microsoft.playwright:playwright`

   Note: by default, Playwright runs the browsers in headless mode. To see the browser UI, setHeadless option to false. You can also use setSlowMo to slow down execution.

b) Enrich your test script to consider the following actions:

- Navigate to the “Slow calculator” link present in the initial page from previous exercise (e.g.*: page.navigate(), page.click("text=Slow calculator")*)
- Assert that you arrived to the correct page (e.g.: assess *page.url()*)

c) Refactor (or create an additional test class) to use the built-in JUnit Extension using @UsePlaywright and the fixtures it provides. Note the changes: (1) @UsePlaywright, (2) use a dependency injection to get a “page” instance; (3) no explicit need for creating the browser.

5.6 Playwright: Interactive test recording

Use Playwright’s Test Generator (codegen) to record tests.

You need to have a basic Maven project, with a pom.xml having `com.microsoft.playwright:playwright` dependency. Then execute it.

```
mvn exec:java -e -D exec.mainClass=com.microsoft.playwright.CLI -D
exec.args=”codegen blazedemo.com”
```

a) Similarly to what you’ve done for 5.2, record a test in which you select a and buy a trip on the

https://blazedemo.com/ dummy travel agency web app, making the proper assertions along the way.

b) Export it either using the “Library” or “Junit” options; check the differences.

c) The preferred locators used by Playwright’s codegen are based on what? Are some of the locators being fragile and why?

d) Refactor to clean-up the code and adjust the locators, if needed.

5.7 Playwright: Selecting elements with locators

Similarly to 5.3, consider the test case of searching for a book in this (fictitious) online store.

a) Implement a simple test that, for a search for “Harry Potter”, finds a result that corresponds to the book “Harry Potter and the Sorcerer's Stone” by J. K. Rowlling. Try to use the best locators as possible (if not, you may need to revert to CSS/XPath). Note that you can use locators within locators…

5.8 [Optional] Playwright: Page Object pattern

Refactor and evolve the previous exercise (5.7) to apply the Page Object design pattern.