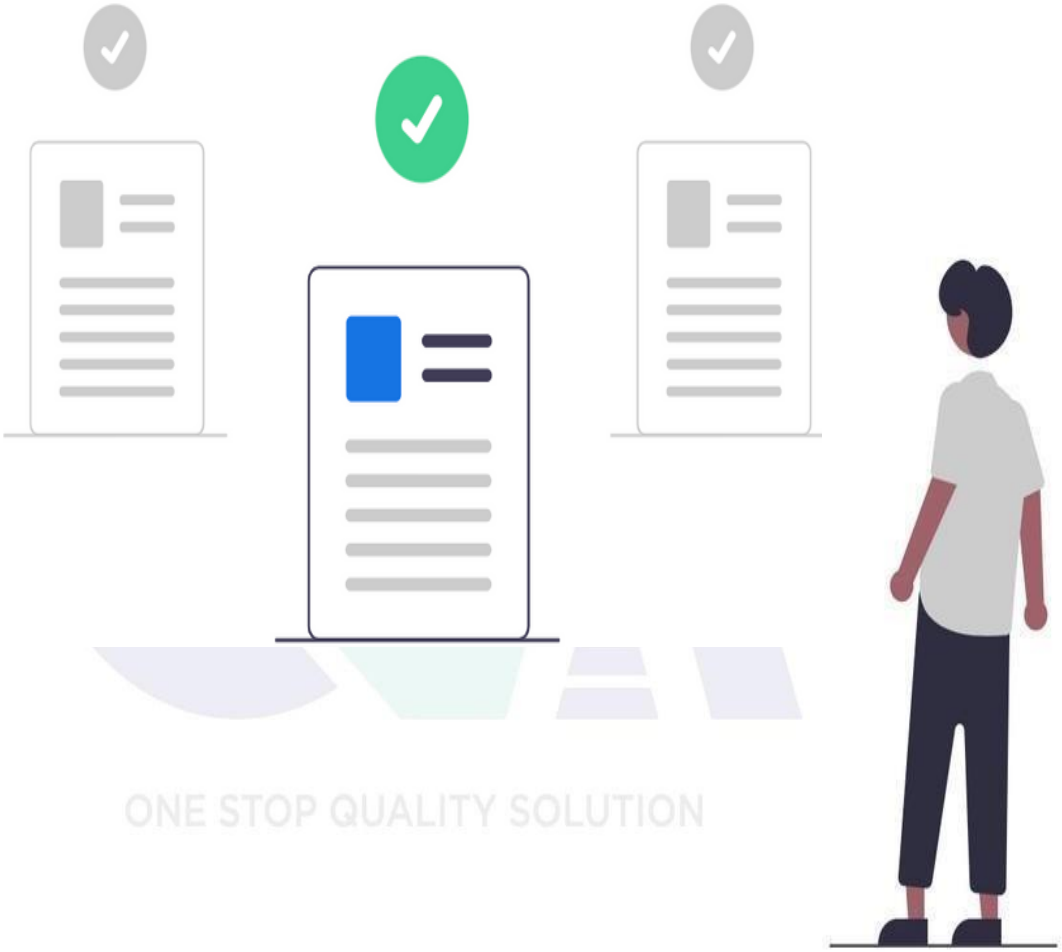

ONE STOP QUALITY SOLUTION

Playwright Automation in Java: A Beginner's Guide

Automating Web Testing with Playwright and Java

Chapter 1

Introduction to Playwright and Java

Overview of Playwright

Playwright is a robust, open-source automation framework developed by Microsoft for end-to-end web testing. It supports multiple programming languages, including Java, and provides cross-browser testing capabilities for Chromium, Firefox, and WebKit. Playwright's powerful features include automated interaction with web elements, handling asynchronous operations, network interception, and parallel test execution. Its built-in wait mechanisms enhance test reliability, and it supports headless and headful browser modes. Ideal for modern web applications, Playwright integrates seamlessly with CI/CD pipelines, making it a versatile choice for developers and testers aiming to ensure comprehensive web application testing and quality assurance.

Feature of Playwright:

- **Cross-Browser Testing:** Automate tests across different browsers to ensure compatibility.
- **Headless and Headful Modes:** Run tests without a GUI (headless) or with a visible browser (headful).
- **Automatic Waiting:** Built-in mechanisms for waiting until elements are ready, reducing flaky tests.
- **Parallel Execution:** Execute tests in parallel to speed up the testing process.
- **Network Interception:** Monitor and modify network requests and responses.
- **Integration with CI/CD:** Seamlessly integrate Playwright tests into continuous integration and delivery pipelines.

Why choose Playwright for automation?

Choosing Playwright for automation comes with several advantages, making it a compelling choice for modern web testing needs:

1. Cross-Browser Support
    - Multiple Browsers: Playwright supports Chromium, Firefox, and WebKit, enabling comprehensive cross-browser testing from a single codebase.
    - Consistency: Ensures consistent behavior across different browser engines, which is crucial for web applications that need to function seamlessly on various platforms.
2. Reliable and Fast
    - Automatic Waiting: Built-in mechanisms automatically wait for elements to be ready before interacting with them, reducing flakiness and making tests more reliable.
    - Fast Execution: Leveraging browser-specific DevTools protocols, Playwright often executes tests faster than traditional WebDriver-based tools.
3. Modern Features
    - Network Interception: Intercept and modify network requests and responses, useful for mocking backends and testing various network conditions.
    - Multiple Contexts: Run multiple browser contexts in a single instance, simulating different users and improving test isolation.
    - Headless and Headful Modes: Support for both headless (without a GUI) and headful (with a GUI) browser modes, suitable for different testing and debugging scenarios.
4. Comprehensive API
    - Rich API: Extensive and easy-to-use API covering a wide range of interactions, including handling iframes, shadow DOMs, and complex user gestures.
    - Language Support: While this guide focuses on Java, Playwright also supports JavaScript/TypeScript, Python, and C#, offering flexibility in language choice.
5. Ease of Setup and Use
    - Quick Setup: Straightforward installation and setup process, getting you up and running quickly.

- Integrated Test Runner: Comes with an integrated test runner that simplifies test execution and reporting.

6. Direct Browser Control

Unlike relying on an intermediary translation layer, Playwright enables direct control and insight into the browser. This approach facilitates the simulation of more meaningful and realistic user scenarios, contributing to the accuracy of your tests.

7. Robust Debugging Tools

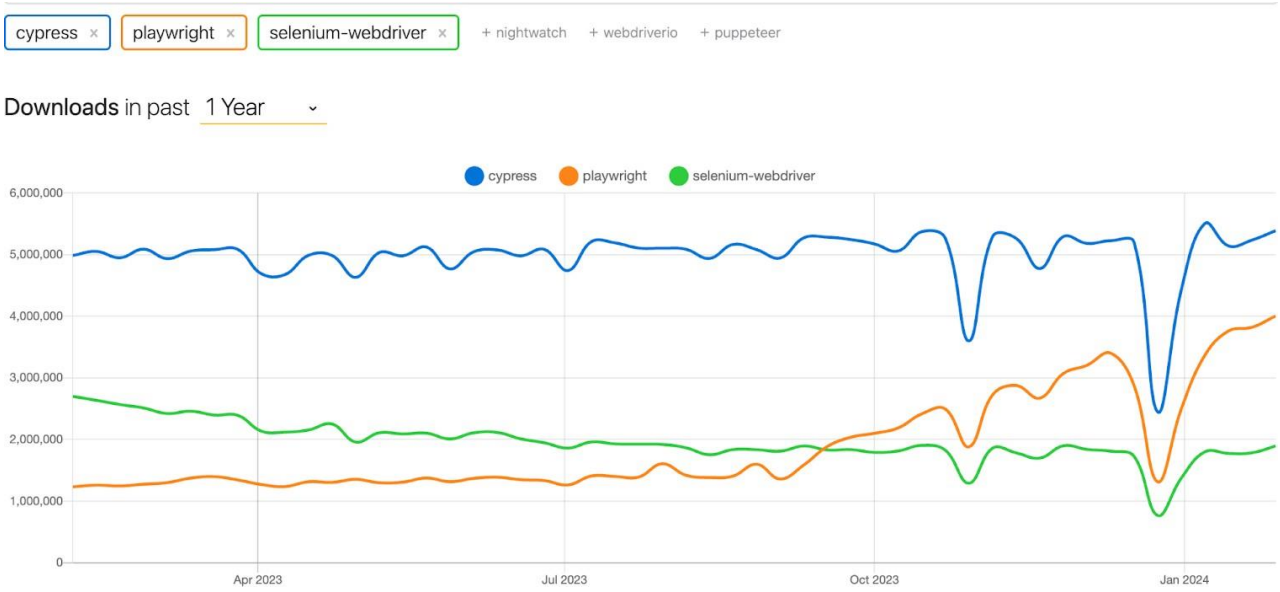
- Inspector: Built-in inspector tool helps debug and develop tests interactively.
- Trace Viewer: Visualize test execution and diagnose failures with detailed trace logs.

8. CI/CD Integration

- Seamless Integration: Easily integrates with popular CI/CD pipelines, enhancing automated testing workflows and ensuring continuous quality.

9. Rapid Adoption and Popularity

since its initial release in January 2020, Playwright has gained significant popularity. Its quick rise in the industry attests to its effectiveness and appeal among developers.



Popularity comparison: Playwright, Cypress & Selenium

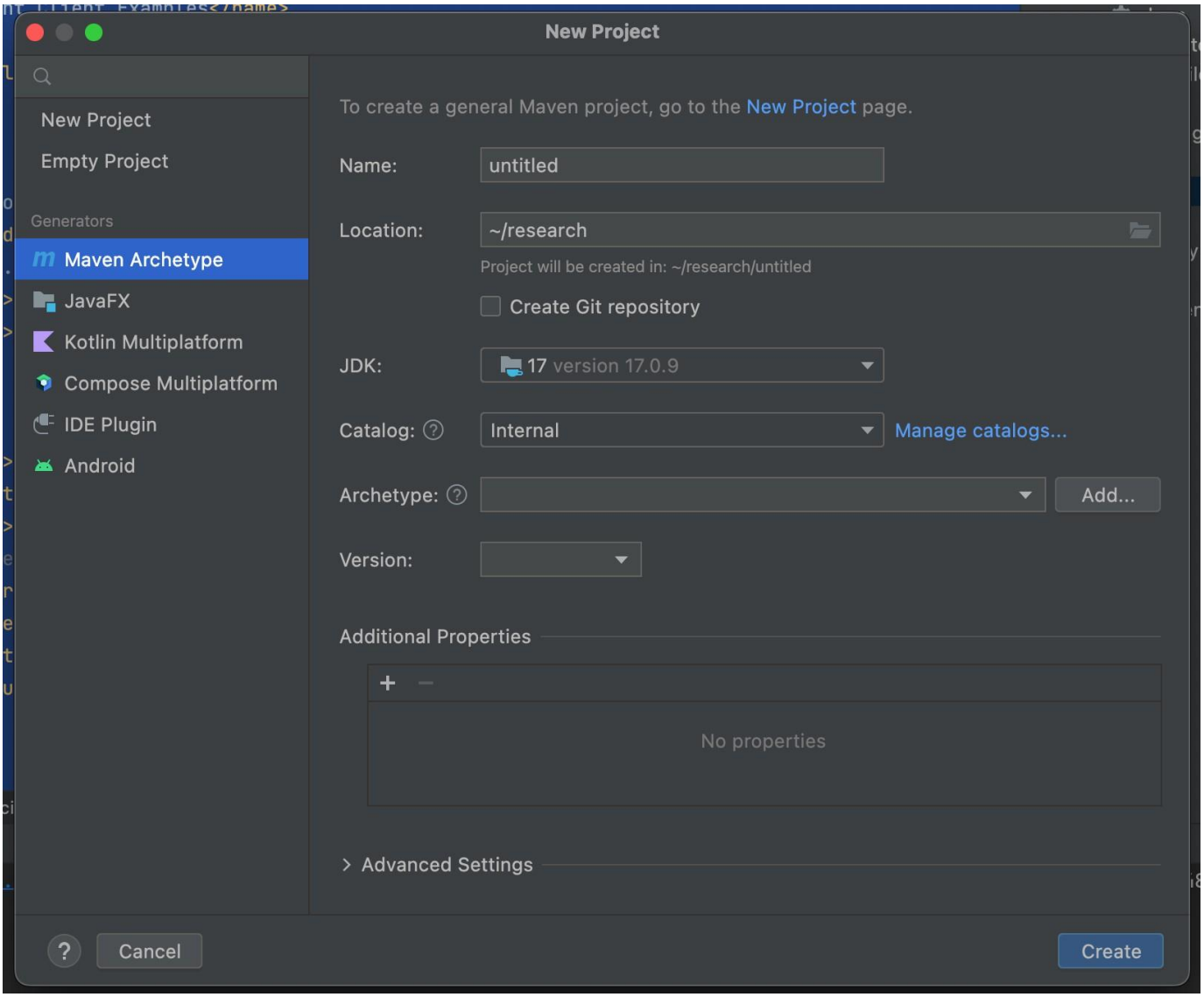
Chapter 2

Installing and Configuring Playwright

Creating a Maven Project in IntelliJ IDEA

Step 1: Launch IntelliJ IDEA

1. Open IntelliJ IDEA.
2. On the welcome screen, click on "New Project".



Step 2: Select Maven as the Project Type

1. In the "New Project" wizard, select "Maven" from the list on the left side.
2. Ensure the "Project SDK" is set to your preferred JDK version. If no SDK is specified, you can add one by clicking "New..." and navigating to the JDK installation directory.

Step 3: Configure Project Details

1. Click "Next" to proceed.
2. Fill in the project details:
   - GroupId: Typically, your domain name in reverse (e.g., `com.example`).
   - ArtifactId: The name of your project (e.g., `my-playwright-project`).
   - Version: Default is `1.0-SNAPSHOT`.
3. Click "Next".

Step 4: Specify Project Location

1. Choose the project location by clicking "..." and selecting the desired directory.
2. Click "Finish".

Step 5: Maven Project Structure

1. IntelliJ will create the Maven project structure for you, including the `pom.xml` file, which manages your project's dependencies.

Step 6: Add Playwright Dependency

1. Open the `pom.xml` file.
2. Add the Playwright dependency inside the `<dependencies>` section:

```
<?xml version="1.0" encoding="UTF-8"?>
<project                                    xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                              xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>org.example</groupId>
 <artifactId>examples</artifactId>
 <version>0.1-SNAPSHOT</version>
```

```
 <name>Playwright Client Examples</name>
 <properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 </properties>
 <dependencies>
   <dependency>
     <groupId>com.microsoft.playwright</groupId>
     <artifactId>playwright</artifactId>
     <version>1.44.0</version>
   </dependency>
 </dependencies>
 <build>
   <plugins>
     <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-compiler-plugin</artifactId>
       <version>3.10.1</version>
         <!-- References to interface static methods are allowed only at source
level 1.8 or above -->
       <configuration>
         <source>1.8</source>
         <target>1.8</target>
       </configuration>
     </plugin>
   </plugins>
 </build>
</project>
```

3. Click the "Reload Maven Projects" button in the Maven tool window or right-click on the `pom.xml` file and select "Maven" > "Reload Project".

Basic Playwright Concepts

Understanding Playwright Architecture

Modern Architecture Alignment: Playwright's architecture is designed to closely align with modern browsers, operating out-of-process. This setup avoids the constraints of in-process test runners like Cypress, providing more flexibility.

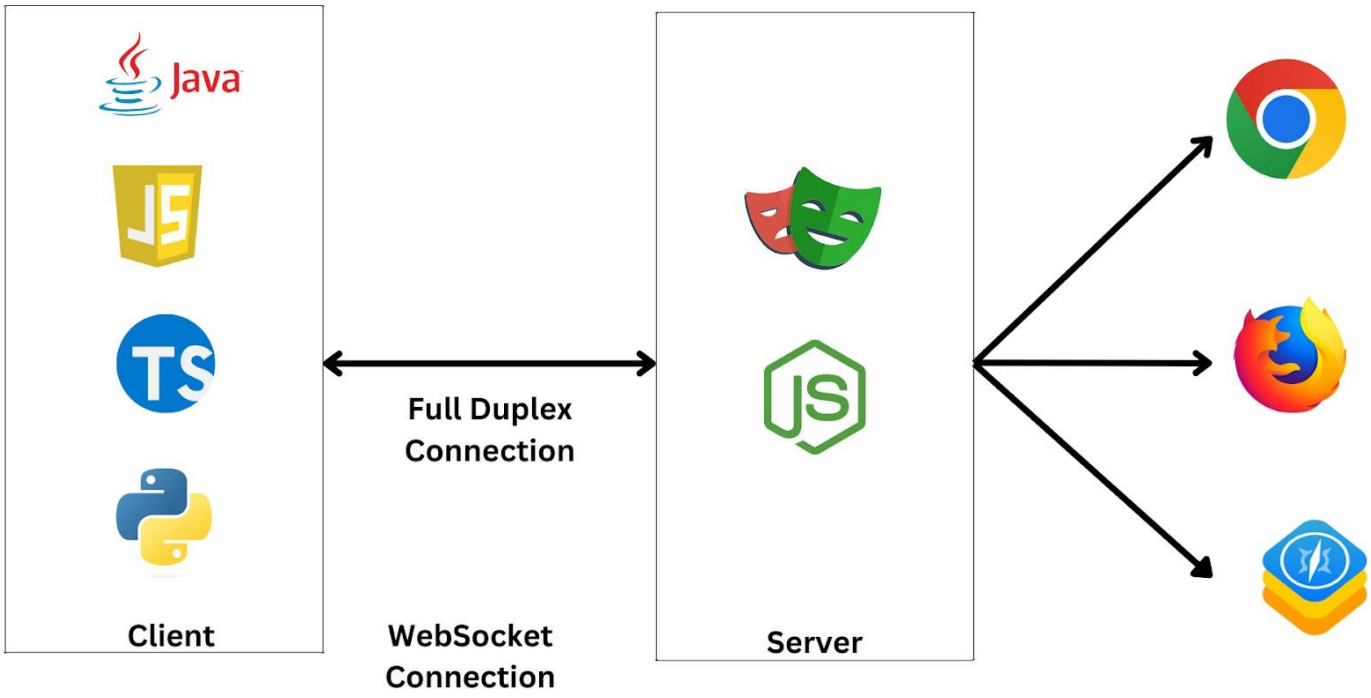
Efficient Communication: Playwright opts for a WebSocket connection for bi-directional client-server communication. This method is faster and more efficient compared to traditional HTTP communication.

Concise Breakdown:

Client: Playwright offers API bindings in various programming languages, allowing seamless interaction with the framework. These APIs translate high-level commands into concise browser actions for developers.

Server: Facilitating communication between client scripts and supported browser engines, the Playwright Node.js server plays a pivotal role in the framework's functionality.

Client-Server Communication: Each script in Playwright establishes a low-latency WebSocket-based communication channel over a single TCP connection. This connection persists until the completion of the test, reducing the likelihood of test failure or flakiness. This efficient setup ensures that commands are executed quickly, outperforming alternative tools using HTTP connections.



Introduction to browser contexts and pages

In Playwright, browser contexts and pages are fundamental concepts that allow developers to manage isolated browsing sessions and interact with web pages. Let's explore these concepts in more detail:

1. Browser Contexts

- Definition: A browser context represents an isolated browsing session within a browser instance.
- Purpose: Browser contexts allow for scenarios such as incognito mode, separate user profiles, or parallel testing.
- Isolation: Each browser context operates independently, with its own cookies, local storage, and other browsing data.
- Creation: Browser contexts are created using the browser.newContext() method, specifying options such as viewport size, user agent, and permissions.

2. Pages

- Definition: A page is a single tab or window within a browser context, representing a specific web page.
- Interactions: Pages are where most interactions with web elements and navigation take place.

- Creation: Pages are created using the browser.newPage() method, which opens a new tab within the specified browser context.
- Navigation: Pages can navigate to URLs using the page.navigate() method, reload using page.reload(), go back and forward in history, and more.
- Element Interaction: Pages provide methods for interacting with web elements, such as clicking, typing, and evaluating JavaScript.
- Lifecycle Events: Pages emit lifecycle events such as domcontentloaded, load, and close, which can be useful for monitoring page state.

Working with selectors and locators

In Playwright Java, working with selectors and locators allows you to identify and interact with specific elements on a web page. Here's how you can do it:

1. Using Selectors

Selectors help you target specific elements on a web page using CSS selectors, XPath, or other strategies.

Example:

```
package org.example;

import com.microsoft.playwright.*;

public class SelectorExample {
    public static void main(String[] args) {
        try (Playwright playwright = Playwright.create()) {
            Browser browser = playwright.chromium().launch();
            Page page = browser.newPage();

            // Using CSS selector
            ElementHandle element = page.querySelector("#myElement");

            // Using XPath
            ElementHandle element = page.querySelector("//input[@name='username']");

            // Interact with the element
            element.click();
        }
    }
}
```

2. Using Locators

Playwright provides a set of built-in locator strategies to find elements based on various criteria.

Example:

```
import com.microsoft.playwright.*;

public class LocatorExample {
   public static void main(String[] args) {
       try (Playwright playwright = Playwright.create()) {
           Browser browser = playwright.chromium().launch();
           Page page = browser.newPage();

           // Locating elements
           ElementHandle elementById = page.locator("#myElement");
           ElementHandle elementByXPath = page.locator("//input[@name='username']");
           ElementHandle elementByText = page.locator("text=Submit");

           // Interact with the elements
           elementById.click();
           elementByXPath.type("username");
           elementByText.hover();
       }
   }
}
```

Writing Your First Test

Creating a simple test case

1. Import Playwright Libraries: You import the necessary Playwright classes.
2. Create a Playwright Instance: Use Playwright.create() to initialize Playwright.
3. Launch a Browser: Use playwright.chromium().launch() to launch a Chromium browser. You can set the browser to run headless (without a GUI) by setting .setHeadless(true), or visible (with a GUI) by setting .setHeadless(false).
4. Create a Page: Use browser.newPage() to open a new browser page.
5. Navigate to a URL: Use page.navigate("https://example.com") to go to the desired webpage.
6. Take a Screenshot: Use page.screenshot() to take a screenshot of the page and save it to a file.
7. Close the Browser: Finally, close the browser with browser.close().

This example demonstrates the basics of using Playwright with Java to automate browser actions. You can expand this to include more complex interactions with web pages, such as filling forms, clicking buttons, and extracting information.

```
package org.example;

import com.microsoft.playwright.*;
import java.nio.file.Paths;

public class App {
 public static void main(String[] args) {
   // Create a Playwright instance
   try (Playwright playwright = Playwright.create()) {
     // Create a browser instance (Chromium in this case)
                        Browser     browser     =     playwright.chromium().launch(new
BrowserType.LaunchOptions().setHeadless(false));

     // Create a new page
     Page page = browser.newPage();

     // Navigate to a URL
     page.navigate("https://example.com");

     // Take a screenshot
     page.screenshot(new Page.ScreenshotOptions().setPath(Paths.get("example.png")));

     // Close the browser
     browser.close();
   }
 }
}
```

Run The Test

To run the test just type, **mvn test** in the console.

After running the tests, Maven generates test reports in the target/surefire-reports directory. You can view the test results in HTML format by opening the index.html file in a web browser.

Navigating web pages

To navigate web pages in Playwright Java, you can use the navigate() method of the Page class. Here's how you can do it:

```
import com.microsoft.playwright.*;

public class NavigationExample {
 public static void main(String[] args) {
   try (Playwright playwright = Playwright.create()) {
     Browser browser = playwright.chromium().launch();
     Page page = browser.newPage();

     // Navigate to a URL
     page.navigate("https://example.com");

     // Wait for navigation to complete
     page.waitForNavigation();

     // Perform further actions on the loaded page
     System.out.println("Page title: " + page.title());

     // Close the browser
     browser.close();
   }
 }
}
```

In this example:

- We create a new Playwright instance and launch a Chromium browser.
- We create a new page using browser.newPage().
- We use the navigate() method to load a URL ("https://example.com" in this case).
- We wait for the navigation to complete using page.waitForNavigation().
- We perform further actions on the loaded page, such as retrieving its title.
- Finally, we close the browser using browser.close().

This code demonstrates the basic navigation capabilities of Playwright Java. You can extend it to perform more complex navigation tasks and interact with the loaded pages as needed.

Interacting with web elements (clicking, typing, etc.)

Interacting with web elements such as clicking buttons, typing into input fields, and other actions is essential in web automation. In Playwright Java, you can achieve this using the ElementHandle class. Here's how to interact with web elements:

```java
import com.microsoft.playwright.*;

public class ElementInteractionExample {
 public static void main(String[] args) {
   try (Playwright playwright = Playwright.create()) {
     Browser browser = playwright.chromium().launch();
     Page page = browser.newPage();

     // Navigate to a URL
     page.navigate("https://example.com");

     // Wait for a specific element to be ready
     ElementHandle button = page.waitForSelector("button");

     // Click on the button
     button.click();

     // Type into an input field
     ElementHandle inputField = page.querySelector("input[type='text']");
     inputField.type("Hello, Playwright!");

     // Wait for a specific element to appear
     ElementHandle resultElement = page.waitForSelector("#result");

     // Get text content of the element
     String resultText = resultElement.textContent();
     System.out.println("Result: " + resultText);

     // Close the browser
     browser.close();
   }
 }
}
```

Debugging and Troubleshooting

Debugging tools and techniques

Debugging in Playwright Java involves using built-in debugging tools and techniques to diagnose and troubleshoot issues in your automation scripts. Here are some tools and techniques you can use:

1. Inspector

- Description: Playwright comes with a built-in inspector tool that allows you to inspect the state of browser instances, pages, and elements.
- Usage:

- Launch the Playwright Inspector by setting the environment variable PLAYWRIGHT_DEBUG=1 before running your script.
- Open a browser and navigate to http://localhost:8543 to access the inspector UI.
- Use the inspector to view page structure, inspect element properties, and debug script execution.

2. Logging

- Description: Adding logging statements to your code can help track the flow of execution and identify potential issues.
- Usage:
  - Use System.out.println() statements to print debug information to the console.
  - Log relevant details such as page titles, element text, and any errors encountered during script execution.

3. Debugging Breakpoints

- Description: Setting breakpoints in your code allows you to pause script execution at specific points and inspect variables and state.
- Usage:
  - Use your IDE's debugging features to set breakpoints in your code.
  - When the breakpoint is hit during script execution, you can inspect variables, step through code, and analyze the script's behavior.

4. Error Handling

- Description: Implementing error handling mechanisms can help catch and handle exceptions gracefully, providing more informative error messages.
- Usage:
  - Use try-catch blocks to catch exceptions and handle errors appropriately.
  - Include meaningful error messages and context information to aid in troubleshooting.

5. Logging and Reporting

- Description: Implement logging and reporting mechanisms to record test execution details and track test results over time.
- Usage:

- Use logging frameworks such as Log4j or SLF4J to log test execution details to files.
- Generate test reports with information about test outcomes, including passed, failed, and skipped tests.

Capturing Screenshots and Videos

Capturing screenshots and videos during test execution can be invaluable for debugging, analysis, and documentation purposes. In Playwright Java, you can easily capture screenshots and videos using built-in methods provided by the Page class. Here's how:

```
import com.microsoft.playwright.*;

public class ScreenshotExample {
 public static void main(String[] args) {
   try (Playwright playwright = Playwright.create()) {
     Browser browser = playwright.chromium().launch();
     Page page = browser.newPage();

     // Navigate to a URL
     page.navigate("https://example.com");

     // Capture a screenshot of the page
     page.screenshot(new Page.ScreenshotOptions().setPath("screenshot.png"));

     // Close the browser
     browser.close();
   }
 }
}
```

Videos are saved upon browser context closure at the end of a test. If you create a browser context manually, make sure to await BrowserContext.close().

```
context = browser.newContext(new
Browser.NewContextOptions().setRecordVideoDir(Paths.get("videos/")));
// Make sure to close, so that videos are saved.
context.close();
```

You can also specify video size. The video size defaults to the viewport size scaled down to fit 800x800. The video of the viewport is placed in the top-left corner of the output video, scaled down to fit if necessary. You may need to set the viewport size to match your desired video size.

```
BrowserContext context = browser.newContext(new Browser.NewContextOptions()
 .setRecordVideoDir(Paths.get("videos/"))
 .setRecordVideoSize(640, 480));
```

Saved video files will appear in the specified folder. They all have generated unique names. For the multi-page scenarios, you can access the video file associated with the page via the Page.video().

```
path = page.video().path();
```