Effective Web Test Automation

Selenium WebDriver with JUnit

Courtney Zhan

Effective Web Test Automation
with Selenium WebDriver and JUnit

Courtney Zhan

Contents

Foreword

In 2025, I celebrate 20 years in the field of End-to-End (E2E) Test Automation and Continuous Testing.

Over the past two decades, I have witnessed numerous failures in E2E automation efforts. A recurring issue I have observed is that many principal software engineers and architects have only a superficial understanding of E2E test automation, often missing the bigger picture: running the test suite for real. For example, in one meeting, I heard a manager ask the Agile Transformation Team Lead, "*When can you show us the execution of E2E tests in the CI server? It's been nine months!*" Ironically, the Agile transformation team, consisting of several Agile coaches and DevOps engineers, had been responsible for selecting both the E2E test automation tool and the CI server.

Senior software testing engineers are frequently misled by overhyped commercial tools, drawn to flashy features that look impressive at first but deliver little practical value. Common examples include:

- Record-and-Playback functionality
- GUI Utilities for object identification (then drag-n-drop)
- Test replay
- Headless browser testing
- Framework's Built-in auto-retry mechanisms
- Sharding and parallelization gimmicks in the framework
- ...

Over the past three decades, waves of test automation hype, such as HP QuickTest Pro (commerical) and ProtractorJS (free & open-source), have come and gone. Yet, in most so-called 'Agile' projects, manual end-to-end testing remains the norm, just as it was 30 years ago.

I have had the opportunity to rescue several failed E2E test automation efforts. In most cases, within just a day or two, I was able to get a dozen reliable, real-world automated tests running on a continuous testing server. People were often amazed: not just by the visible (and objective) results, but by the clear, logical, and practical approach that made sense.

In my opinion, here are the core principles for effective End-to-End Test Automation:

- Focus on high-quality, well-structured test scripts rather than over-relying on tools.
- Design E2E test scripts with maintainability as a top priority.
- Write automated tests in a clear, readable format that all team members, including non-technical business analysts, can understand (and willing to run).
- Ensure each test script is reliable, stable, and resilient to changes.
- Develop tests with high efficiency, which is required for E2E test automation.
- Execute the complete test suite frequently, as regression testing, on a dedicated Continuous Testing server.
- Make test execution and results highly visible, enabling rapid feedback and shared accountability across the team.

When my daughter Courtney decided to pursue a career in software development, I began teaching her my approach and best practices from the very start of her university journey. Early on, she contributed to several of my applications by taking on E2E test automation and Continuous Testing tasks. This hands-on experience

not only helped her excel academically but also played a key role in landing a job at a FAANG company.

This book is a pioneering effort to bring together E2E test automation and CI/CD, two essential yet often disconnected aspects of modern software development, offering a cohesive, practical perspective that will serve as a valuable resource for software teams striving to elevate their software development practices. Courtney did an excellent job of explaining the principles and practices mentioned above, with easy-to-follow and hands-on exercises. Readers who complete these exercises will gain a deeper understanding of real E2E Test Automation, Continuous Testing and real CI/CD. I believe motivated software engineers and testing professionals will find it highly valuable.

Zhimin Zhan

Brisbane, Australia

Preface

Why this book?

I began blogging on Medium in December 2021, mainly writing about end-to-end (E2E) test automation, continuous testing, and programming, one article per week. To date, I've published over 190 articles.

Admittedly, there were times when I struggled to come up with new ideas. One day, my father suggested, "Why not write a series about how you approach E2E test automation in a new project, and how you set up a continuous testing server to run the suite frequentlly?" I replied, "That is quite straightforward." But he reminded me, "Others might find it difficult. Think back to your internship, the senior test automation engineer, the head of the testing excellence center at that major telecom company, who promoted Playwright but couldn't keep even a dozen Playwright tests valid."

That conversation inspired me to write a few articles on the topic, which ended up being more popular than most of my previous posts. Along the way, I began to feel that a book format might serve the material better. So, here it is.

This is the first book in a series, with each volume focusing on a different scripting language or test automation framework:

- Volume 1 (this book): Selenium WebDriver with JUnit (Java)
- Volume 2: Selenium WebDriver with RSpec (Ruby)
- Volume 3: Playwright Test with TypeScript (JavaScript)
- Volume 4: Selenium WebDriver with Pytest (Python)

Some readers may ask, "What about C#?". I've previously authored an Apress book, "Selenium Recipes in C#". I am comfortable with both Java and C#, two of the most-popular compiled languages. I've also built Continuous Testing pipelines with small suites of automated tests in C#, and the process is very similar to Java. In fact, readers who go through two or more books in this series will notice that the practical E2E test automation and Continuous Testing workflows are fundamentally alike, regardless of framework or language. Unless there is strong demand, I believe the four books in this series are sufficient to demonstrate the broad applicability of this approach.

You might also wonder: Does this approach apply to mobile test automation? The answer is yes, absolutely. I'm currently working on a separate book titled "Practical Mobile Test Automation", which will cover that topic in depth.

What's unique about this book series?

1. **Bridging E2E Test Automation and Continuous Testing**

   Typically, E2E test automation and Continuous Testing are treated as separate topics, but I disagree. Often, capable test automation engineers can develop high-quality automated E2E tests, yet without knowledge of Continuous Testing (or access to the right tools), they struggle to maintain the suite, leading it to eventually fail. Pity!

   This book is a pioneering attempt to combine two closely related topics, E2E test automation and Continuous Testing, into a single, cohesive guide.

2. **Develop 20+ automated E2E tests for a real web app**

   Unlike many tutorial-style technical books that use oversimplified examples, this book guides readers through developing real and practically useful test scripts (which you can applly directly to work), against one real website. Through each carefully chosen exercise, readers are guided to master essential automation scripting techniques.

   Another important aspect is the size of the test suite. Only after a test suite reaches a certain size, test automation engineers begin to appreciate the

importance of script maintenance, the benefits of maintainable test design and refactoring, and the necessity of Continuous Testing.

3. **Short, Focused Sessions to Master Test Automation Gradually**

   Each session is designed with a specific focus and typically takes 15–20 minutes to complete. Test automation concepts and best practices are introduced gradually, with key knowledge points reinforced through repetition in subsequent sessions.

4. **Cover the top two leading web automation frameworks: Selenium WebDriver and Playwright**

   Test automation is a cornerstone of Agile development. In 2003, Watir (Web Application Testing in Ruby) pioneered browser-based, end-to-end (functional) testing. Since then, numerous web automation frameworks have emerged and faded. Two decades later, Selenium WebDriver and Playwright remain the dominant frameworks[1] in the field.

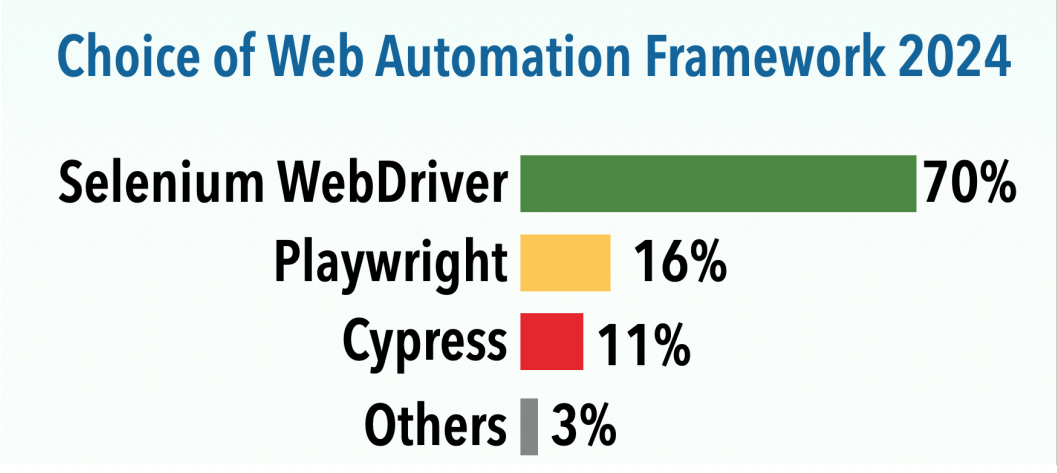


Figure 1. Two independent surveys conducted in late 2023 yielded similar results

The solutions for all the exercises in this book series are in both Selenium WebDriver and Playwright, so readers can choose to learn either one or both.

[1] https://agileway.substack.com/p/selenium-webdriver-is-still-the-best

Who should read this book?

Software professionals, from testers, programmers, software architects and agile coaches, who want to learn hands-on web test automation, Continuous Testing and real CI/CD.

Prior experience with test automation and programming is not necessary. Basic scripting knowledge will help, but again, not necessary.

How to read this book?

I strongly recommend readers to do the exercises through chapters in order. The solutions are provided on the book site[2]. If you get stuck, refer to those resources.

Send me feedback

I would like to hear from you. Comments, suggestions, errors in the book and test/build scripts are all welcome. You can submit your feedback via the book website.

Courtney Zhan

Brisbane, Australia

[2] http://courtneyzhan.com/books/ewta-selenium-junit

1: Introduction

Let's begin with the goal.

1.1: Goal

The goal of this book is to help you set up and run a suite of Selenium WebDriver tests in parallel on a continuous testing server – within a matter of hours. Some readers might think this sounds a bit ambitious for beginners. But I believe every end-to-end (E2E) test automation engineer should aim for this from the start. If not, they're not taking the craft seriously.

You might wonder: Is this just another guide promising quick results but falls apart in real-world scenarios – something that looks good in a demo (for just a few simple tests) but fails miserably at work (for a sizable suite of complex tests)? Absolutely not.

My father has successfully used this exact setup and practices across many applications—including two that run a 600+ test suite of E2E (UI) tests regularly. I've also had my own share of successes using the same approach.

By the way, every software and tool presented in this book is free, in the software sense.

> "Free software is a matter of liberty, not price." - GNU Home page[1]

Some might wonder, "what about the price then?". Oh well, it is mostly free as well. Simply put, money will not be the barrier that holds people back from mastering

[1] https://www.gnu.org/home.en.html

the E2E test automation knowledge in this book, but the lack of willingness to do hands-on practice will.

1.2: A Story

A few years ago, when I started my internship at a large telecom company (*assigned to testing, with many other new IT interns*), I set up continuous testing on my very first day by running a few business automated E2E tests using Selenium WebDriver with RSpec. My team leader was deeply impressed to see real E2E test automation running and regularly detecting regression defects, for the first time. For me, it felt completely natural—I had been watching my father do this daily for over a decade, and this was exactly how he had always trained me: "*End-to-end test automation must be visible and useful, from day 1, and every day onwards*". Later, a senior test automation engineer from the Sydney office, watched my video presentation to the entire division (*recommended by my team leader and supported by the division director*) and asked me to switch to Playwright.

By the next day, I had over 20 Playwright tests running on the BuildWise CT server. Later, I learned that this same senior engineer, the head of the "Test Excellence Center", had struggled to keep even a dozen of his own Playwright tests running. He only did individual test automation demos in presentations, with no regular suite executions and no one requesting test execution statistics. His fake test automation went unnoticed—until I, a young intern, arrived. People, at least my team members, had witnessed **effective** web test automation in action.

1.3: E2E Test Automation and Continuous Testing: Universal Skills for Rapid Results

Readers of my other books and blog articles know that I've worked on test automation across multiple frameworks and languages. The key points are:

- **End-to-End Test Automation is a transferable skill**

In other words, the principles and best practices apply regardless of the framework or programming language.

- **Continuous Testing works similarly across frameworks and languages**

  The setup and practices remain largely the same. For example, on my father's Continuous Testing server (running since 2012), there are projects using over 10 different combinations of automation frameworks, syntax frameworks, and programming languages, including: Selenium RSpec (Ruby), Selenium Pytest (Python), Selenium JUnit (Java), Selenium MSTest (C#), Selenium Cucumber (Ruby), Playwright Test (TypeScript), Appium RSpec (Ruby) and even performance and load testing projects, all within the same CT server instance.

- **Rapid results are possible**

  With the right approach, you can achieve a sizeable End-to-End Test Automation suite and Continuous Testing solution in days, not weeks or months.

1.4: Book Structure

This book is designed for beginners and takes a **hands-on** approach. The fastest way to master end-to-end test automation is through practical experience, which is exactly what this book provides.

You'll learn to:

- Set up your test development and execution environment.
- Write your first test script.
- Organize your test project and leverage productivity tools.
- Develop additional test scripts with various script development and debugging techniques.

  Against a real webapp.

- Refactor tests into a maintainable form.

  Go beyond "just working" by applying solid design principles and best practices. Unmaintainable automated tests are useless.

- Expand your suite to around 22 test cases while learning more automated testing skills.

- Run the complete test suite.

  With just a 22-test E2E suite, you'll quickly experience the two main challenges of running good-quality test scripts through tools or the command line: long feedback loop and execution reliability. You need a Continuous Testing server.

- Execute the whole suite on a Continuous Testing (CT) server.

- Scale up with parallel execution on the CT server, with multiple build agents.

  Quicker feedback and more reliable test execution.

By the end of this book, you'll have the skills, knowledge, tools, and confidence to build a sizeable test suite at work—and run it daily on a continuous testing server.

2: Set up Selenium WebDriver

In this book, we'll use Selenium WebDriver's Java bindings to build automated test scripts for web applications.

A few software tools need to be installed, but setup is straightforward, some may already be on your computer:

- Java
- Testing Libraries (Selenium WebDriver and JUnit)
- Google Chrome Browser and its matching driver
- (Optional but recommended) A Java IDE such as Intellij IDEA

2.1: Installation

Web test automation is platform-independent, and so are Selenium WebDriver and Java. The same software, scripts, and practices apply across all operating systems, with only minor setup differences (such as file path conventions). In this book, we'll use macOS as the primary platform.

> All the automated test scripts in this book were developed and executed on a Continuous Testing Server running on a $599 M4 Mac Mini.

2.1.1: Install Java

Java has been one of the most widely used programming languages for over two decades, primarily for building web applications. It is also a can by used writing end-to-end (E2E) automated tests.

When installing Java, you'll find two options: the JDK (Java Development Kit) and the JRE (Java Runtime Environment). The JDK includes everything needed to both compile and run Java programs, while the JRE only allows you to run them. For development work, whether building applications or writing test scripts, ensure that the JDK is installed.

I recommend installing an LTS version of Java, so I'll be installing Java 21, but you can use whichever version you prefer.

2.1.1.1: macOS

To install Java for macOS, visit the Java downloads page[1], click the `macOS` tab, then "ARM64 DMG Installer" to download. Open the downloaded file and run the installer.

To verify installation, run `java -version` in a terminal.

```
$ java -version
java version "21.0.7" 2025-04-15 LTS
Java(TM) SE Runtime Environment (build 21.0.7+8-LTS-245)
Java HotSpot(TM) 64-Bit Server VM (build 21.0.7+8-LTS-245, mixed mode, sharing)
```

2.1.1.2: Windows

To install Java for Windows, visit the Java downloads page[2] to download and run the EXE installer file.

[1] https://www.oracle.com/au/java/technologies/downloads/#java21
[2] https://www.oracle.com/au/java/technologies/downloads/#jdk21-windows

2.1.1.3: Linux

Install Java with the appropriate binary/installer for your Linux distribution from the Java downloads page[3].

2.1.2: Download and Organize Testing Libraries

Install the test libraries (known as 'JARs' (Java ARchives) in Java).

The test libraries you will need are:

- Selenium WebDriver (automation framework); and
- JUnit (test syntax framework)

Download the latest stable Selenium WebDriver release from the Selenium Download page[4]. This will be a zip file containing, multiple jars, unzip it and move it to a central location (e.g. `/Users/me/java-lib-selenium-webdriver`).

JUnit currently has two main versions in use: JUnit 4 and JUnit 5. While JUnit 5 is the newer release and introduces significant changes, I prefer JUnit 4 for its simplicity and ease of setup. For end-to-end test automation, JUnit 4 is sufficient.

To install JUnit 4, download two JARs, `junit.jar` and `hamcrest-core.jar` from the downloads page[5]. Again, move these to the same repository as the Selenium JARs.

2.1.3: Browser and Driver

While Selenium works for all major browsers, this book will focus on the most dominant browser - Google Chrome. If you already have Google Chrome installed, you're ready to start.

[3] https://www.oracle.com/au/java/technologies/downloads/#jdk21-linux
[4] https://www.selenium.dev/downloads/
[5] https://github.com/junit-team/junit4/wiki/Download-and-Install

Web browsers, such as Google Chrome, require a separate component—called a driver—to enable automation with tools like Selenium. For Chrome, this component is ChromeDriver, and its version typically needs to match your installed Chrome browser closely (within one version). Previously, this meant manually updating ChromeDriver every time Chrome updated, which happens roughly every two months.

Starting with Selenium v4.11 and Chrome v115, manual management of ChromeDriver is no longer necessary. Selenium Manager now handles it automatically, making setup and maintenance much simpler.

Manually installing ChromeDriver

Here's a quick guide to manually install a specific version of ChromeDriver (for older Chromebrowser versions), in three simple steps:

1. Download the ChromeDriver zip file from the Chrome for Testing availability site[6]. Make sure to select the version that matches your Chrome browser and OS.
2. Unzip the file to extract the chromedriver executable.
3. Place the chromedriver executable in a directory that's included in your system's PATH.

2.1.4: Install Apache Ant

Apache Ant is a Java-based build tool used to automate tasks, which we could use run Selenium tests with flexibility.

Some older versions of macOS come with Ant pre-installed (you can verify this by running `ant -version`). If it is not there, you can install Ant va Homebrew:

[6] https://googlechromelabs.github.io/chrome-for-testing/

```
brew install ant
```

For Windows and Linux users, follow the official Ant installation guide[7].

2.2: Run a sample automation script

No setup is complete without verification—that's the tester's mindset. Let's run a Selenium WebDriver script.

Create a new text file (in any text editor) and name it '`Sample.java`'.

Inside `Sample.java`, paste or type in the following contents:

```
package com.sample;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class Sample {
  public static void main(String[] args) throws InterruptedException {
    WebDriver driver = new ChromeDriver();

    // And now use this to visit Google
    driver.get("http://www.google.com");

    // Check the title of the page
    System.out.println("Page title is: " + driver.getTitle());

    Thread.sleep(5000);
    driver.quit();
  }
}
```

Then, in a terminal window, navigate to the folder where `Sample.java` is, and run the command:

[7] https://ant.apache.org/manual/install.html

```
% javac -d target -classpath "target:/Users/me/java-lib-selenium-webdriver/seleni\
um-java-4.34.0/*" Sample.java
```

This command compiles the `Sample.java` and outputs the compiled classes into a folder named `target`. The `classpath` flag specifies the classpath, which should be the directory where you saved the Selenium Java testing libraries.

You should now see a folder named `target` containing the compiled file `com/sample/Sample.class`. With the Java file compiled, you can run it using the following command (from the same directory where you ran the compile command):

```
% java -classpath 'target:/Users/me/java-lib-selenium-webdriver/selenium-java-4.3\
4.0/*' com.sample.Sample
```

A Chrome browser window should launch, navigate to the Google homepage, and then close automatically. The console output will look like this:

```
courtney@Courtneys-Mac-mini ch02-sample-test % javac -d target -classpath "target:/Users/Shared/java-lib-selenium
-webdriver/selenium-java-4.34.0/*" Sample.java

courtney@Courtneys-Mac-mini ch02-sample-test % java -classpath 'target:/Users/Shared/java-lib-selenium-webdriver/
selenium-java-4.34.0/*' com.sample.Sample

Aug 12, 2025 8:57:29 PM org.openqa.selenium.devtools.CdpVersionFinder findNearestMatch
WARNING: Unable to find CDP implementation matching 139
Aug 12, 2025 8:57:29 PM org.openqa.selenium.chromium.ChromiumDriver lambda$new$4
WARNING: Unable to find version of CDP to use for 139.0.7258.67. You may need to include a dependency on a specif
ic version of the CDP using something similar to `org.seleniumhq.selenium:selenium-devtools-v86:4.34.0` where the
 version ("v86") matches the version of the chromium-based browser you're using and the version number of the art
ifact is the same as Selenium's.
Page title is: Google
courtney@Courtneys-Mac-mini ch02-sample-test %
```

Congratulations, you've just executed your first automation script!

2.3: Run a sample automated test script

We've just run an automation script that interacts with a browser. Now, let's take it a step further by converting it into a JUnit test. Automated tests not only perform

actions but also include assertions to verify results. JUnit is a widely used Java testing framework that provides both the structure for writing tests and built-in assertion methods.

The following is an example JUnit test script that opens the Google homepage and checks that the page title is correct.

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

import static org.junit.Assert.assertEquals;

public class SampleTest {
  WebDriver driver = new ChromeDriver();

  @Before
  public void setUp() throws Exception {
    driver.get("http://www.google.com");
  }

  @Test
  public void testCanLoadGoogle() throws Exception {
    System.out.println("Page title is: " + driver.getTitle());
    assertEquals("Google", driver.getTitle());
  }

  @After
  public void tearDown() throws Exception {
    driver.quit();
  }
}
```

Save the script above to a file named `SampleTest.java`.

Similar to the sample script, we need to compile the test file first. In the classpath, keep the Selenium libraries, but also add the JUnit dependency (`junit.jar`):

```
% javac -d target -classpath 'target:/Users/Shared/java-lib-selenium-webdriver/se\
lenium-java-4.34.0/*:/Users/Shared/java-lib-selenium-webdriver/junit4/junit-4.13.\
2.jar' SampleTest.java
```

To run the file, we don't want to use the regular Java runner, instead, use the JUnit runner by using `org.junit.runner.JUnitCore` before you specify the class to be run. This will run the test and return the test result. The runner is in the Hamcrest dependency (`hamcrest-core.jar`), so remember to add that to the classpath as well.

```
% java -classpath 'target:/Users/Shared/java-lib-selenium-webdriver/selenium-java\
-4.34.0/*:/Users/Shared/java-lib-selenium-webdriver/junit4/junit-4.13.2.jar:/User\
s/Shared/java-lib-selenium-webdriver/selenium-dependent/hamcrest-core-1.3.jar' or\
g.junit.runner.JUnitCore SampleTest
```

A Chrome browser window should launch, navigate to the Google homepage, and then close automatically. The console output will look like this:

```
JUnit version 4.13.2
.
Page title is: Google

Time: 2.407

OK (1 test)
```

The "OK (1 test)" in the JUnit console output means that 1 test was run and the result was successful. If it had failed, you would have seen a line like "`FAILURES!!! Tests run: 1, Failures: 1`".

At this stage, all the necessary tools are installed, and their setup has been verified using the two scripts above. In Chapters 4 and 7, I will show how to run tests more easily (replacing the manually crated `javac` and `java` commands) using an IDE and Ant, respectively.

3: Run your first Selenium WebDriver Test

In the previous chapter, we successfully ran a simple Selenium JUnit test to confirm that our setup was working correctly. Now, let's dive into the fundamentals of Selenium WebDriver and JUnit syntax, and then move on to developing a more realistic (and useful) test case.

3.1: Selenium JUnit Test Structure

Let's start by learning from the sample test script from before - `SampleTest.java`.

> Explanation of the file name:
>
> - `Test` indicates it's a test file — it is a test script to verify a business behaviour
> - `.java` means it is a Java file.

The contents of `SampleTest.java`:

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

import static org.junit.Assert.assertEquals;

public class SampleTest {
  WebDriver driver = new ChromeDriver();

  @BeforeClass
  public static void beforeAll() throws Exception {
  }

  @Before
  public void setUp() throws Exception {
      driver.get("http://www.google.com");
  }

  @Test
  public void testCanLoadGoogle() throws Exception {
      System.out.println("Page title is: " + driver.getTitle());
      assertEquals("Google", driver.getTitle());
  }

  @After
  public void tearDown() throws Exception {
      driver.quit();
  }

  @AfterClass
  public static void afterAll() throws Exception {
  }
}
```

The first few lines, starting with `import`, load in Selenium WebDriver (the automation framework) and JUnit (the test syntax framework). These (and any other libraries you use in your test scripts) are always required at the beginning of any Selenium JUnit test script. To keep the scripts concise, I'll leave them out in later example scripts.

`@Test`, `@BeforeClass`, `@Before`, `@After` and `@AfterClass` are JUnit structure lines that

form the test structure. If you're familiar with xUnit-style frameworks, this should feel familiar. For now, just remember that the `@Test` block defines a single JUnit test case. I'll go into more detail about JUnit in a later chapter.

Putting the JUnit test structure aside, we're left with four core test steps:

```
WebDriver driver = new ChromeDriver();
driver.get("http://www.google.com");
assertEquals("Google", driver.getTitle());
driver.quit();
```

Meaning:

1. Launch a Chrome browser
2. Navigate to the Google homepage
3. Check that the page title is 'Google'
4. Close the browser

These four steps mirror what you would do in a manual test, right?

My book "Web Test Automation in Action: Volume 1" includes a set of Selenium and Playwright tests for a sample test site: Agile Travel (travel.agileway.net). In this book, we will develop dozens of tests against a real webapp: WhenWise (whenwise.agileway.net).

3.2: First Test: User Login

The first automated test case for web apps is almost always user login, so let's start there. Test data:

- Site URL: https://whenwise.agileway.net
  (a sandbox site)
- User Login: `driving@biz.com`; Password: `test01`
  (a list of test users are shown on the sandbox site)

3.3: Rename the test script file and add the first step

I recommend using a more meaningful test script file name, i.e, rename `SampleTest.java` to `LoginClientOkTest.java`. Remember to update the class name to match.

We're also going to update the script so that it visits the target website ('https://whenwise.agileway.net') instead of the Google home page in the `@Before` block. Also, to keep our test passing, update the website title we expect to be 'WhenWise – Booking Made Easy'.

```
public class LoginClientOkTest {
  WebDriver driver = new ChromeDriver();

  @BeforeClass
  public static void beforeAll() throws Exception {
  }

  @Before
  public void before() throws Exception {
    driver.get("https://whenwise.agileway.net");
  }

  @Test
  public void testLoginOk() throws Exception {
    System.out.println("Page title is: " + driver.getTitle());
    assertEquals("WhenWise – Booking Made Easy", driver.getTitle());
  }

  @After
  public void tearDown() throws Exception {
    driver.quit();
  }

  @AfterClass
  public static void afterAll() throws Exception {
  }
}
```

To perform the above, no Selenium or Java knowledge is required. Really, just a

simple string replacement.

Run it.

```
% javac -d target -classpath 'target:/Users/me/java-lib-selenium-webdriver/seleni\
um-java-4.34.0/*:/Users/me/java-lib-selenium-webdriver/junit4/junit-4.13.2.jar' L\
oginClientOkTest.java

% java -classpath 'target:/Users/me/java-lib-selenium-webdriver/selenium-java-4.3\
4.0/*:/Users/me/java-lib-selenium-webdriver/junit4/junit-4.13.2.jar:/Users/me/jav\
a-lib-selenium-webdriver/selenium-dependent/hamcrest-core-1.3.jar' org.junit.runn\
er.JUnitCore LoginClientOkTest
```

You'll see a Chrome browser launch and navigate to the WhenWise sandbox site, but it will close very quickly (maybe even before the page content loads).

To keep the browser open for three seconds so you can actually see something, add the following line *(to pause the execution for three seconds)* to the end of the test (`@Test` block):

```
Thread.sleep(3000);
```

Run the test again, you should be able to see the WhenWise home page content load.

3.4: Add more user login steps and finish the login test case complete

Let's develop this login test script. First, we should create a test design by doing the test manually and noting down the steps:

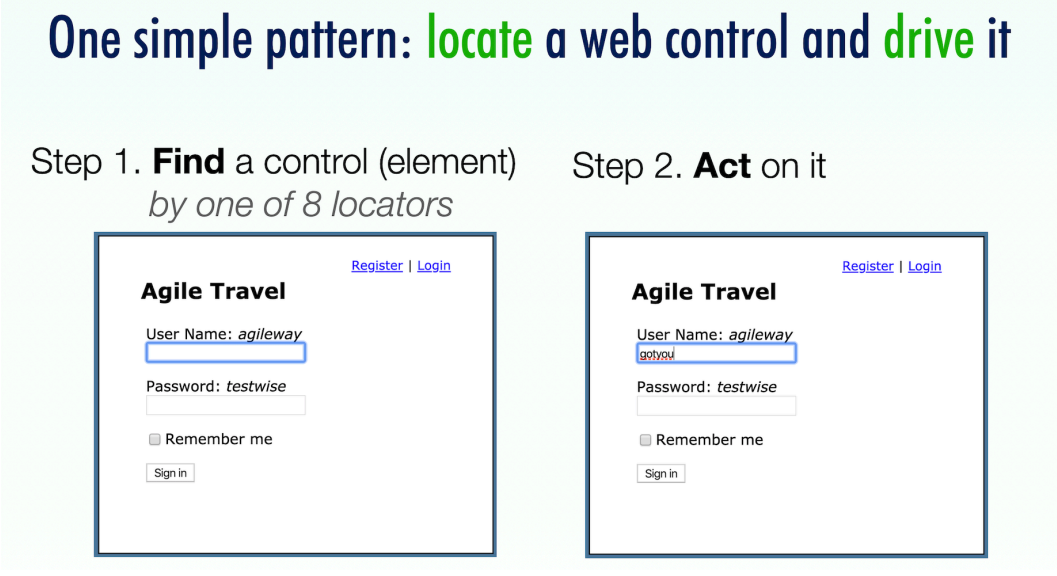
1. Open 'https://whenwise.agileway.net' in a browser.
2. Click the 'Sign in' button.

3. Enter a valid user name or email.
4. Enter the correct password.
5. Click the “SIGN IN” button.
6. Verify signed in successfully.

The first step (opening the website) is already completed in the above script. Next, we will work out the remaining steps in Selenium WebDriver.

3.4.1: Selenium WebDriver’s intuitive syntax pattern

Selenium WebDriver is the easiest-to-learn automation framework because its syntax follows such a simple and intuitive pattern.



3.4.2: Working out Selenium Step 1: Locate the control

I don't use record-n-playback utility. I prefer to manually inspect the element (or control) in the browser, and come up with the best locator, then type the test step in the testing IDE.

> With a good testing IDE, such as TestWise or a customised IntelliJ, manually entering test steps can be quite efficient (with auto complete and snippets, see in Chapter 5). The real importance here is the recorded test steps are usually brittle and not good quality, it might work but is unmaintainable over time.

Let me illustrate with the Step 2 of the test: "Click the 'Sign in' button".

In Chrome (manually), right-click the 'Sign in' button on the WhenWise home page and select the 'Inspect' option.

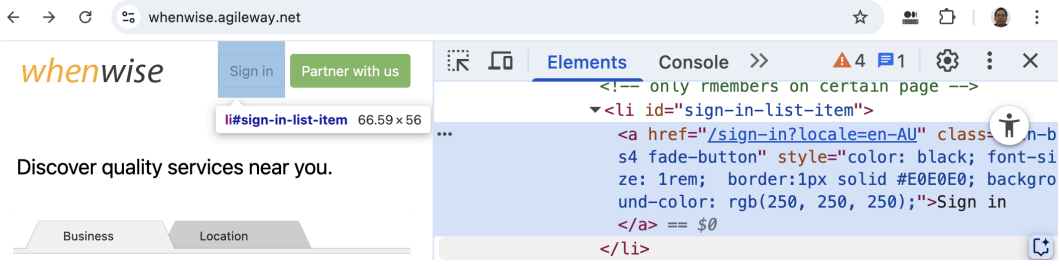


Figure 2. Right-click to inspect a control in Chrome

In Chrome's Inspect pane, look for the HTML fragment of the control you are interested in.

```
<a href="/sign-in?locale=en-AU" class="btn-bs4" style="...">Sign in</a>
```

It turns out that despite looking like a button, the 'Sign in' control is actually a hyperlink.

In Selenium, 'locating the control' means finding the element using a locator. For this 'Sign in' hyperlink, the optimal locator is `linkText`:

```
driver.findElement(By.linkText("Sign in"));
```

`linkText` is one of the eight core locators in Selenium WebDriver:

Locator	Example
ID	`findElement(By.id("user"))`
Name	`findElement(By.name("username"))`
Link Text	`findElement(By.linkText("Login"))`
Partial Link Text	`findElement(By.partialLinkText("Next")`
XPath	`findElement(By.xpath("//div[@id="login"]/input"))`
Tag Name	`findElement(By.tagName("body")`
Class Name	`findElement(By.className("table")`
CSS	`findElement(By.cssSelector("#login > input[type="text"]")`

You may use any one of them to locate the control you are looking for.

3.4.3: Work out Selenium Step 2: Perform action on the located control

Once a control is located in Selenium, we append the operation we want to perform. In this case, since it's a hyperlink, the operation is '`.click()`'.

```
driver.findElement(By.linkText("Sign in")).click();
```

Quite simple, right?

Some readers might wonder, 'What about entering text?' The next step, entering an email on the login page, demonstrates this.

```
<input id="email" type="text" name="session[email]" class="active">
```

We begin by locating the element. There are two suitable locator options: `id: email` or `name: session[email]`. Either option is acceptable.

```
driver.findElement(By.name("session[email]"));
```

For a text box, the primary operation is to input text. In Selenium, this is done using '`.sendKeys(...)`'.

```
driver.findElement(By.name("session[email]")).sendKeys("james@client.com");
```

With the above knowledge, you shall be able to work out the Selenium statements up to Step 5.

One other kind of action you can take on a locator is reading the element text, you can do this with `.getText()`.

```
driver.findElement(By.id("username")).getText();
```

Commonly, you might want to get all the text on the page, which is useful for verification steps. One easy way to do this is to get all the text on the HTML page. You can easily do this via the HTML '`body`' tag.

```
driver.findElement(By.tagName("body")).getText();
```

3.5: Run your test

By now, you should have a test script that goes all the way to clicking the sign in button (Step 5). We will verify the sign-in was successful (Step 6) a little bit later. Give your test script a run with the `javac` and `java` commands to see if the sign in steps worked.

```
% javac -d target -classpath 'target:/Users/me/java-lib-selenium-webdriver/seleni\
um-java-4.34.0/*:/Users/me/java-lib-selenium-webdriver/junit4/junit-4.13.2.jar' L\
oginClientOkTest.java

% java -classpath 'target:/Users/me/java-lib-selenium-webdriver/selenium-java-4.3\
4.0/*:/Users/me/java-lib-selenium-webdriver/junit4/junit-4.13.2.jar:/Users/me/jav\
a-lib-selenium-webdriver/selenium-dependent/hamcrest-core-1.3.jar' org.junit.runn\
er.JUnitCore LoginClientOkTest
```

Hopefully you should see a browser open, enter login details then successfully login! Feel free to add a pause at the end if it went by too fast to see the successful login screen (`Thread.sleep(500);`).

3.5.1: Common Errors

Before we go further, let's discuss a common error message you might encounter. Here is the initial draft test script I created.

```
@Test
public void testLoginOk() throws Exception {
  driver.findElement(By.linkText("Sign in")).click();
  driver.findElement(By.id("username")).sendKeys("james@client.com");
  driver.findElement(By.id("password")).sendKeys("test01");
  driver.findElement(By.id("login-btn")).click();
}
```

If you run it with `java`, you will see the following error:

```
There was 1 failure:
1) testLoginOk(LoginClientOkTest)
org.openqa.selenium.NoSuchElementException: no such element: Unable to locate ele\
ment: {"method":"css selector","selector":"#username"}
  (Session info: chrome=139.0.7258.139)
For documentation on this error, please visit: https://www.selenium.dev/documenta\
tion/webdriver/troubleshooting/errors#no-such-element-exception
Build info: version: '4.34.0', revision: '2a4c61c498'
System info: os.name: 'Mac OS X', os.arch: 'aarch64', os.version: '15.6.1', java.\
version: '21.0.7'
Driver info: org.openqa.selenium.chrome.ChromeDriver
Command: [81c26fbda44557d213cb543aae37b207, findElement {value=username, using=id\
}]
Capabilities {acceptInsecureCerts: false, browserName: chrome, browserVersion: 13\
9.0.7258.139, ...}
Session ID: 81c26fbda44557d213cb543aae37b207
 at java.base/jdk.internal.reflect.DirectConstructorHandleAccessor.newInstance(Di\
rectConstructorHandleAccessor.java:62)
 ...
 at org.openqa.selenium.remote.RemoteWebDriver.execute(RemoteWebDriver.java:544)
 at org.openqa.selenium.remote.ElementLocation$ElementFinder$2.findElement(Elemen\
tLocation.java:165)
 ...
 at org.openqa.selenium.remote.RemoteWebDriver.findElement(RemoteWebDriver.java:3\
61)
 at LoginClientOkTest.testLoginOk(LoginClientOkTest.java:21)
```

The above error message is a `NoSuchElementException`, for line 21 of the `LoginClientOkTest` file (bottom of the stack trace).

This script is intentionally set up to fail - I had used the wrong ID for the email field. The ID should be `email`, not `username`.

Why am I showing you this failing test? It's common for beginners to make typos, which can lead to execution failures like the one above. When you see this error, go back to the script and check the locator that failed is spelt correctly or using the correct locator type.

3.6: Add Verification

Tests require verification (or assertion) to be useful, otherwise it just drives the browser without checking for intended behaviour.

A typical JUnit assertion is in the format '`assertTrue(...)`' or '`assertFalse(...)`' or '`assertEquals(..., ...)`'. For example '`assertEquals(5, 2 + 3);`', which is an exact match.

In addition to checking for exact matches, you can also verify substring containment using `contains`. For example, `assertTrue("scare".contains("care"));`.

Now, back to the login test, the final step (Step 6) is to verify the sign in was successful.

If you try the test steps manually, you will see that after you sign in, there will be an alert that says 'You have signed in successfully'.

So the verification step in this test would look like:

```
assertTrue(driver.findElement(By.tagName("body")).getText().contains("You have si\
gned in successfully"));
```

This verifies that the text is shown anywhere on the page contents (under the HTML tag '`body`').

3.7: Add a user login failed test case in a separate test script

We should now have a successful login test scenario. Usually, we need at least verify the user login failed test scenario as well.

Create another file: `LoginClientFailedTest.java` with the below contents:

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

import static org.junit.Assert.assertTrue;

public class LoginClientFailedTest {
  WebDriver driver = new ChromeDriver();

  @Before
  public void setUp() throws Exception {
    driver.get("https://whenwise.agileway.net");
  }

  @Test
  public void testLoginFailed() throws Exception {
    driver.findElement(By.linkText("Sign in")).click();
    driver.findElement(By.name("session[email]")).sendKeys("james@client.com");
    driver.findElement(By.id("password")).sendKeys("badpass");
    driver.findElement(By.id("login-btn")).click();

    Thread.sleep(500);
    assertTrue(driver.findElement(By.tagName("body")).getText().contains(
      "Password/email is invalid"));
  }

  @After
  public void tearDown() throws Exception {
    driver.quit();
  }
}
```

Run this test script.

Note: The only changes here from the “Login Ok” test were the password (“badpass” to make it incorrect), and the verification (invalid password message instead of the login success message).

3.8: Refactoring: Merge two user login test cases into one test script file

In the above, we defined two test cases in two separate files. In this case, it is actually better to merge into one, because they are related; they're both login tests. Furthermore, we might be able to reuse some test steps in `@BeforeClass`, `@Before`, `@After`, and `@AfterClass` blocks.

Create a new `LoginClientTest.java` file including the two previous test cases. It is not a simple concat, rather, a merge to put them into a test syntax structure.

```
public class LoginClientTest {
  static WebDriver driver = new ChromeDriver();

  @BeforeClass
  public static void beforeAll() throws Exception {
      driver.get("https://whenwise.agileway.net");
  }

  @Before
  public void before() throws Exception {
  }

  @Test
  public void testLoginOk() throws Exception {
    driver.findElement(By.linkText("Sign in")).click();
    driver.findElement(By.name("session[email]")).sendKeys("james@client.com");
    driver.findElement(By.id("password")).sendKeys("test01");
    driver.findElement(By.id("login-btn")).click();

    Thread.sleep(500);
    assertTrue(driver.findElement(By.tagName("body")).getText().contains(
      "You have signed in successfully"));
  }

  @Test
  public void testLoginFailed() throws Exception {
    driver.findElement(By.linkText("Sign in")).click();
    driver.findElement(By.name("session[email]")).sendKeys("james@client.com");
    driver.findElement(By.id("password")).sendKeys("badpass");
```

```
    driver.findElement(By.id("login-btn")).click();

    Thread.sleep(500);
    assertTrue(driver.findElement(By.tagName("body")).getText().contains(
      "Password/email is invalid"));
  }

  @After
  public void after() throws Exception {
  }

  @AfterClass
  public static void afterAll() throws Exception {
    driver.quit();
  }
}
```

3.9: Run the test script

For the test script above (which includes two test cases), try running it in the following ways:

1. Run only `testLoginOk` test case. (Comment out the `testLoginFailed` test)
2. Run only `testLoginFailed` test case. (Comment out the `testLoginOk` test)
3. Run the entire test script with both test cases.

You'll likely observe:

1. `testLoginOk` passes.
2. `testLoginFailed` passes.
3. When running both together, `testLoginOk` passes but `testLoginFailed` fails.

Why does the second test fail only when both are executed together? Clearly, the execution of the first test case is affecting the outcome of the second. In test automation, this issue is known as "**inter-test dependency within the test script**".

What's causing this behavior? Try running the entire test script several times and watch the browser closely as it executes.

Here's the issue: both test cases assume that the browser session in a logged-out state. When run individually, that assumption holds. But when run together, the first test logs in successfully – leaving the session authenticated. The second test then tries to log in again, which causes it to fail.

The solution is simple: log out at the end of the first test case. Specifically, at the end of `testLogin0k`, add the following statement:

```
driver.get("https://whenwise.agileway.net/sign-out");
```

Run the entire test script again with `java` – both test cases should now pass. Then, run each test individually to confirm they still work independently.

Just as we saw with the case where each test passed on its own but failed when run together, the opposite scenario is also common: the full test script passes, but certain individual tests in it fail. This usually happens when tests are written with intentional dependencies – for example, "*Test 1: create a client*" followed by "*Test 2: edit that client*". While convenient, this approach makes debugging more difficult and should be avoided.

The goal of this exercise is to recognize both execution modes and ensure that every test case can pass both individually and as part of the full script.

About the Cover Photo

Featured on the cover is the Hiroshima Castle (Japan), photographed by Zhimin Zhan in May 2025.

Book Cover Designed by the author.