



Playwright Automation – Step-by-Step Hands-On Guide

A comprehensive training programme designed for manual testers transitioning to automation, freshers, and early-career QA/QE professionals. This hands-on session will guide you through the complete process of setting up, writing, executing, and reporting Playwright tests.

What is Playwright?



Playwright is a modern, open-source test automation framework developed by Microsoft. It enables reliable end-to-end testing for web applications across multiple browsers including Chromium, Firefox, and WebKit.

Unlike traditional automation tools, Playwright offers auto-wait capabilities, powerful debugging features, and native support for modern web applications. It's designed to handle dynamic content, complex user interactions, and cross-browser testing scenarios efficiently.

Prerequisites & Environment Requirements

Node.js Runtime

Version 14 or higher required for running
JavaScript-based automation

Visual Studio Code

Recommended IDE with excellent
JavaScript support and extensions

Internet Connection

Required for downloading dependencies
and browser binaries

- Trainer Note: Ensure all participants have administrative access to install software on their machines before proceeding.

Verify Node.js & NPM Installation

Why Node.js is Required

Playwright is built on Node.js, which provides the runtime environment for executing JavaScript code outside the browser. NPM (Node Package Manager) is the tool we use to install Playwright and manage project dependencies.

Before creating any Playwright project, we must verify that both Node.js and NPM are properly installed and accessible from the command line.

Verification Commands

```
node -v
```

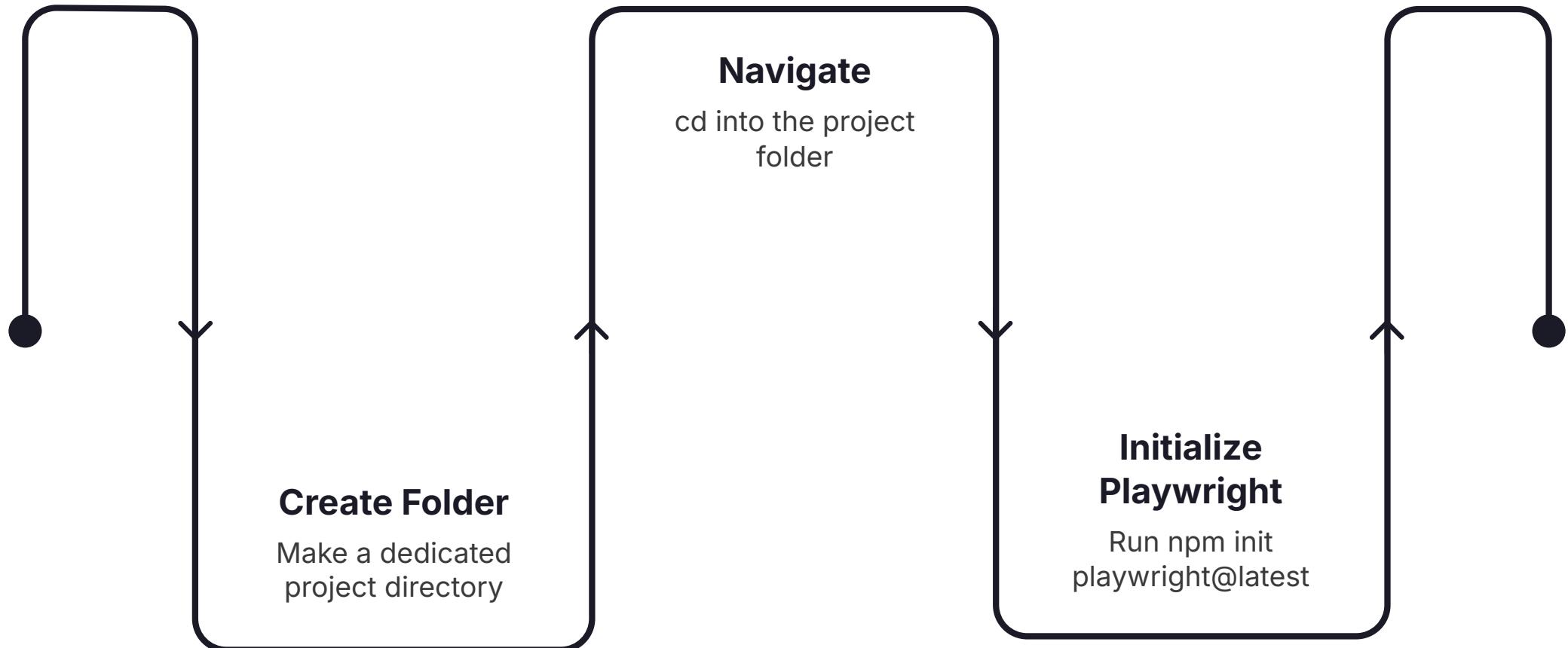
Expected output: v14.0.0 or higher

```
npm -v
```

Expected output: 6.0.0 or higher

- If these commands return version numbers, you're ready to proceed. If not, install Node.js from nodejs.org first.

Creating Your Playwright Project



We'll begin by creating a dedicated folder for our automation project, then use NPM to initialise Playwright with all necessary configurations and dependencies.

Project Creation Commands

Step-by-Step Setup

These three commands form the foundation of your Playwright project. First, we create a directory to house our tests. Then we navigate into that directory. Finally, we initialise Playwright, which downloads all required packages and sets up the project structure.

The `npm init playwright@latest` command is an interactive installer that will prompt you with several configuration questions.

Terminal Commands

```
mkdir playwright-training
```

Creates a new directory named 'playwright-training'

```
cd playwright-training
```

Changes into the newly created directory

```
npm init playwright@latest
```

Initialises Playwright project with guided setup

Installation Prompts & Recommended Selections

1 Choose Language

Select: JavaScript

Recommended for beginners due to simpler syntax and wider community support

2 Test Folder Location

Select: tests

Default folder structure keeps test files organised and separate from configuration

3 GitHub Actions Workflow

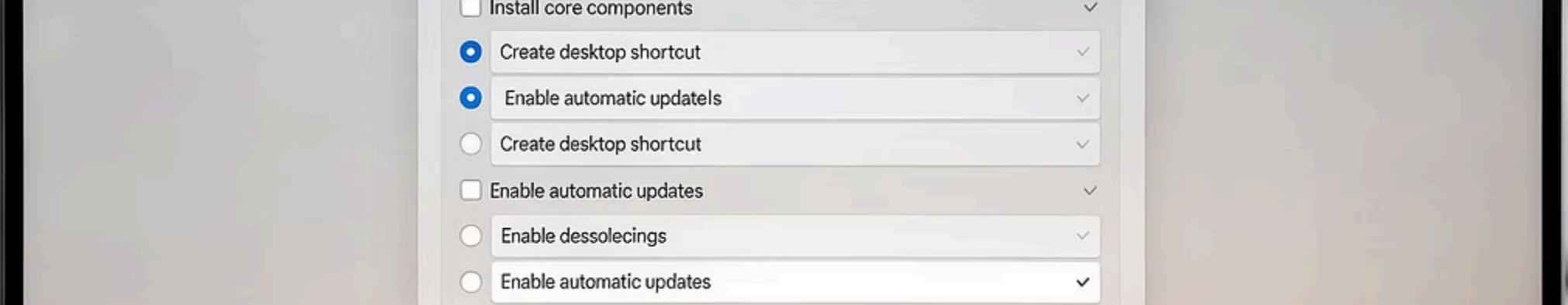
Select: No

We'll focus on local execution first; CI/CD integration can be added later

4 Install Browsers

Select: Yes

Downloads Chromium, Firefox, and WebKit browsers required for test execution



Understanding Installation Choices

Each selection during the Playwright installation process serves a specific purpose. Choosing JavaScript over TypeScript reduces the initial learning curve, allowing you to focus on automation concepts rather than type systems.

The 'tests' folder becomes your central repository for all test files. Declining GitHub Actions keeps the setup simple for training purposes. Installing browsers ensures you have everything needed to run tests immediately without additional downloads.

- Trainer Note: Pause here to ensure all participants have successfully completed the installation. Address any error messages or installation issues before moving forward.

Playwright Project Structure

Default Folder Organisation

After initialisation, Playwright creates a standardised project structure. Understanding this structure is essential for navigating your automation project and knowing where to place different types of files.

Each component serves a distinct purpose in the testing framework, from storing test cases to managing configuration and dependencies.

Folder Tree

```
playwright-training/
├── tests/
│   └── example.spec.js
├── playwright.config.js
└── package.json
└── node_modules/
```

Key Project Components Explained



tests/ Directory

Contains all your test specification files. Each file typically holds related test cases grouped by functionality or feature



playwright.config.js

Central configuration file controlling browser settings, test timeouts, reporters, and execution behaviour



package.json

Defines project metadata, dependencies, and script commands. Manages version control for installed packages



node_modules/

Contains all downloaded dependencies and libraries. Generated automatically; never edit manually



MODULE 4

Writing Your First Test Case

A Playwright test file uses the `.spec.js` extension and contains one or more test cases. Each test is defined using the `test()` function, which takes a description and an asynchronous function containing your test steps.

Test files must import the necessary Playwright modules at the top, specifically `test` and `expect`, which provide the core testing functionality.

Basic Playwright Test Syntax

Understanding Test Structure

Every Playwright test follows a consistent pattern: import required modules, define test cases with descriptive names, and write test steps using the `page` object. The `async/await` syntax ensures operations complete before moving to the next step.

The `page` object represents the browser page and provides methods for interacting with web elements. The `expect` function creates assertions to validate expected behaviour.

Core Concepts

- **`async/await`:** Handles asynchronous operations sequentially
- **`page object`:** Represents browser context for interactions
- **`test()`:** Defines individual test case
- **`expect()`:** Creates assertions for validation

Example: Open Google and Verify Title

```
const { test, expect } = require('@playwright/test');

test('Open Google and verify title', async ({ page }) => {
  await page.goto('https://www.google.com');
  await expect(page).toHaveTitle(/Google/);
});
```

This test demonstrates the fundamental structure of a Playwright test case. The `page.goto()` method navigates to the specified URL. The `expect(page).toHaveTitle()` assertion verifies that the page title matches the expected pattern using a regular expression.

- ❑ Trainer Note: Explain that the forward slashes in `/Google/` denote a regular expression, which allows flexible pattern matching rather than exact string comparison.

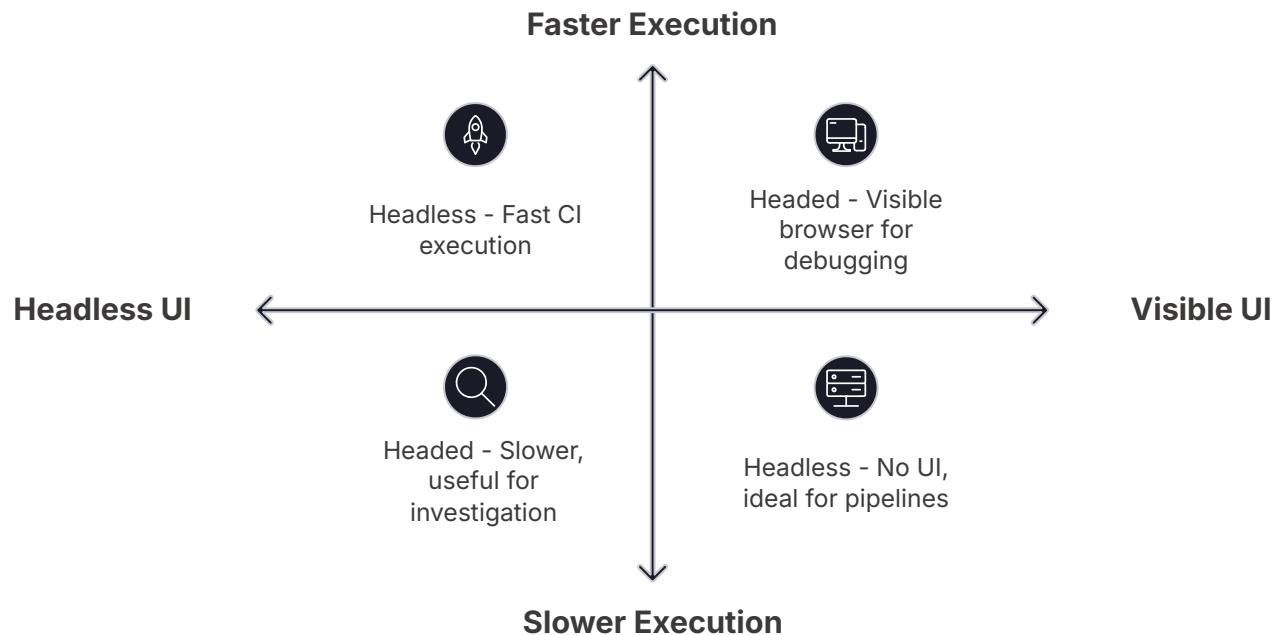
Example: Google Search Test

```
test('Search Playwright in Google', async ({ page }) => {
  await page.goto('https://www.google.com');
  await page.fill('textarea[name="q"]', 'Playwright automation');
  await page.keyboard.press('Enter');
  await expect(page).toHaveURL(/search/);
});
```

This test showcases user interaction automation. The `page.fill()` method locates the search input field using a CSS selector and enters text. The `page.keyboard.press()` method simulates pressing the Enter key. Finally, we verify the URL changes to include 'search', confirming the search was executed.

Note how each action waits for completion before proceeding to the next step, thanks to the `await` keyword.

Executing Your Tests



Execution Modes

Playwright offers two primary execution modes. Headless mode runs tests in the background without displaying the browser window, making it faster and ideal for automated pipelines. Headed mode shows the browser window, allowing you to observe test execution visually.

For learning and debugging, headed mode is invaluable. For continuous integration and faster execution, headless mode is preferred.

Test Execution Commands

Headless Mode

```
npx playwright test
```

Executes all tests in the background without opening browser windows. Default mode for faster execution and CI/CD pipelines

Headed Mode

```
npx playwright test --headed
```

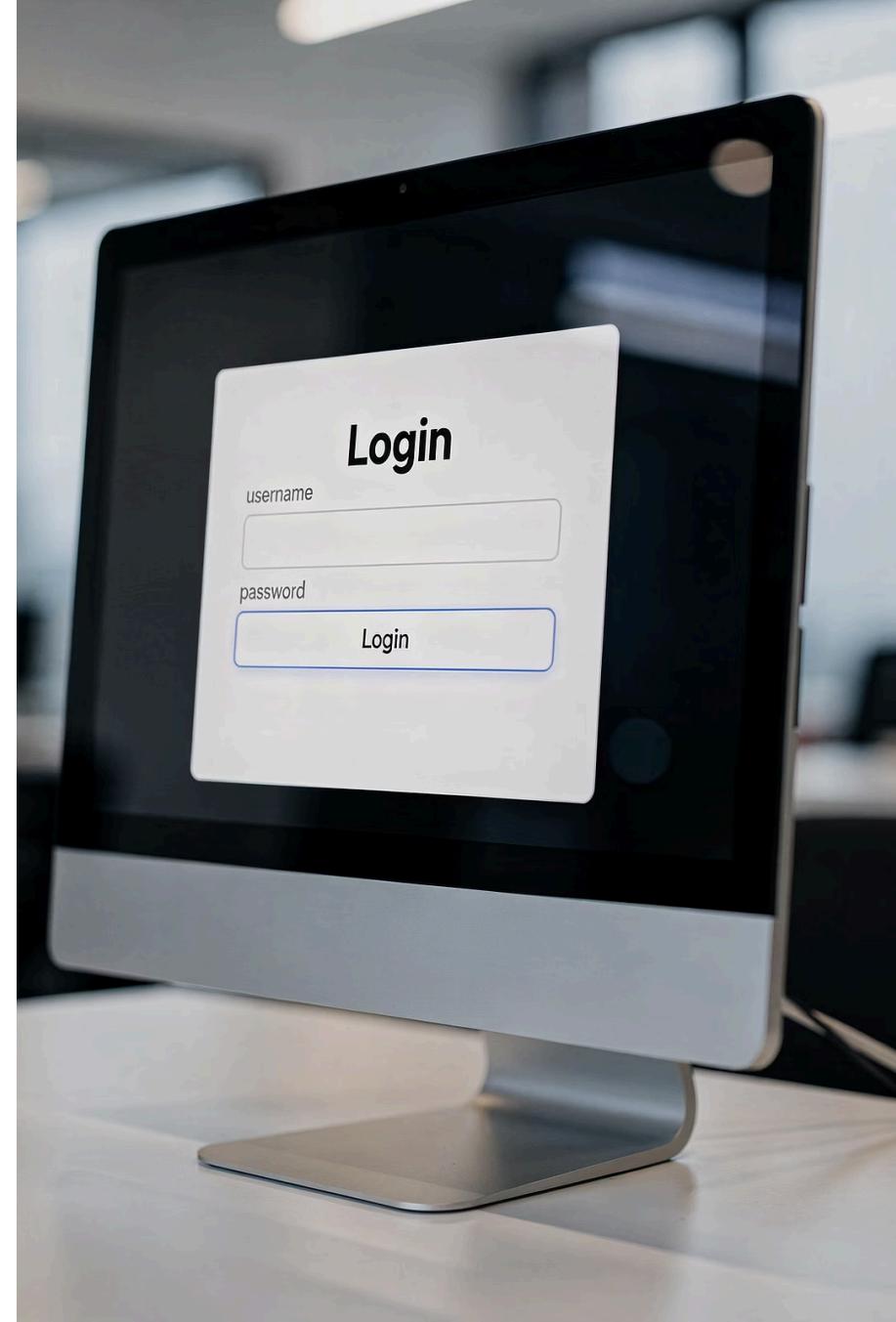
Runs tests with visible browser windows. Useful for debugging, demonstrations, and understanding test behaviour visually

- Trainer Note: Run both commands during the live demo. Show participants how headed mode displays browser actions, then demonstrate the speed of headless execution.

Adding a Realistic Login Test

Login scenarios are amongst the most common automation requirements in real projects. They validate authentication mechanisms, security measures, and error handling. Testing both successful and unsuccessful login attempts ensures your application responds correctly to various user inputs.

We'll create a negative test case that intentionally uses invalid credentials to verify proper error message display.



Login Validation Test Example

```
test('Example login form validation', async ({ page }) => {
  await page.goto('https://the-internet.herokuapp.com/login');
  await page.fill('#username', 'wronguser');
  await page.fill('#password', 'wrongpass');
  await page.click('button[type="submit"]');
  await expect(page.locator('#flash')).toContainText('Your username is invalid');
});
```

This test navigates to a login page, fills in incorrect credentials, submits the form, and validates that an appropriate error message appears. The `page.locator()` method identifies elements, and `toContainText()` verifies the error message content.

- Trainer Note: Emphasise that negative testing is crucial. Applications must handle incorrect inputs gracefully and provide clear feedback to users.

Understanding Test Reporting

Why Reporting Matters

Test reports provide essential visibility into test execution results. They document which tests passed, which failed, execution duration, and error details. In professional environments, reports serve as evidence of testing activities and help teams identify issues quickly.

Playwright supports multiple reporter formats, from simple console output to detailed HTML reports with screenshots and traces.

Reporter Types

- **Console (list):** Real-time terminal output during execution
- **HTML:** Interactive web-based report with detailed results
- **JSON:** Machine-readable format for integration
- **JUnit:** XML format for CI/CD tools

Configuring Playwright Reporter

Open the `playwright.config.js` file and locate the reporter configuration section. Add or modify the reporter array to include both console and HTML reporting:

```
reporter: [
  ['list'],
  ['html', { open: 'never' }]
]
```

The 'list' reporter provides real-time console output during test execution. The 'html' reporter generates a comprehensive web-based report. Setting `open: 'never'` prevents the report from automatically opening after each test run, giving you control over when to view it.

- This configuration appears in the `module.exports` object within `playwright.config.js`

Viewing the HTML Report

Generating & Opening Reports

After executing tests, Playwright automatically generates the HTML report in the `playwright-report` directory. The report includes detailed information about each test, including execution time, browser used, and any captured screenshots or videos.

The `show-report` command launches a local web server and opens the report in your default browser, providing an interactive interface to explore test results.

Report Commands

```
npx playwright test
```

Executes tests and generates the report

```
npx playwright show-report
```

Opens the generated HTML report in your browser

- ☐ Trainer Note: Navigate through the report interface, showing test status, execution timeline, and any failure details.

What to Observe in the HTML Report



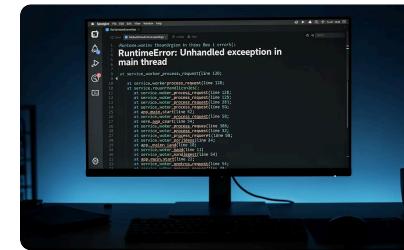
Test Summary

Overall pass/fail statistics, execution duration, and browser coverage



Individual Test Details

Step-by-step actions, timing information, and assertion results for each test



Failure Analysis

Error messages, stack traces, and screenshots captured at the point of failure



MODULE 9

Evidence Collection: Screenshots, Video & Trace

In professional testing environments, collecting evidence of test execution is crucial for debugging failures, documenting issues, and providing proof of testing activities. Playwright offers three types of evidence: screenshots, videos, and traces.

These artifacts are particularly valuable when tests run in CI/CD pipelines where you cannot observe execution directly. They help developers reproduce and fix issues quickly.

Configuring Evidence Collection

Evidence Collection Strategy

Configure evidence collection in playwright.config.js within the 'use' section. The settings control when and how evidence is captured. Capturing only on failure conserves storage space whilst ensuring critical debugging information is available when needed.

Traces provide the most detailed information, including network activity, console logs, and DOM snapshots at each step.

Configuration Code

```
use: {  
  screenshot: 'only-on-failure',  
  video: 'retain-on-failure',  
  trace: 'on-first-retry'  
}
```

screenshot: Captures image on test failure

video: Saves recording when test fails

trace: Generates detailed trace on retry

Demonstrating Test Failure

Why Show Failures?

Observing failing tests is an essential part of learning test automation. It helps you understand how failures are reported, what information is captured, and how to interpret error messages. Deliberately creating failures also validates that your assertions are working correctly.

We'll modify a working test to make it fail, then examine the resulting report, screenshots, and traces.

Forcing a Failure

```
test('Failing test example', async ({ page }) => {
  await page.goto('https://www.google.com');
  await expect(page).toHaveTitle('Wrong Title');
});
```

This assertion will fail because Google's title doesn't match 'Wrong Title', triggering evidence collection.

Analysing Failure Output

When you run the failing test and open the HTML report, observe the detailed failure information:

Error Message

Clear description of what was expected versus what was actually found

Screenshot

Visual capture of the page state at the moment of failure

Stack Trace

Exact line number and file location where the test failed

Video Recording

Playback of the entire test execution leading up to the failure

- Trainer Note: After showing the failure, fix the test by correcting the assertion, then re-run to demonstrate the green passing state.

Playwright Command Cheat Sheet

Essential commands for daily Playwright automation work:

1

Run All Tests

```
npx playwright test
```

Executes all test files in headless mode

2

Run in Headed Mode

```
npx playwright test --headed
```

Shows browser during execution

3

Run Specific Test

```
npx playwright test example.spec.js
```

Executes only the specified test file

4

View HTML Report

```
npx playwright show-report
```

Opens the test results report

Mapping Manual Testing to Automation

Understanding how manual testing concepts translate to Playwright code:

| Manual Testing Concept | Playwright Equivalent |
|------------------------|--|
| Test Case | <code>test()</code> function |
| Test Steps | Sequential <code>await</code> statements |
| Test Data Input | <code>page.fill()</code> , <code>page.click()</code> |
| Expected Result | <code>expect()</code> assertions |
| Test Execution | <code>npx playwright test</code> |
| Test Report | HTML reporter output |
| Evidence/Screenshots | Automatic screenshot/video capture |

Key Takeaways & Next Steps

What You've Learnt Today

- Installing and configuring Playwright projects
- Understanding project structure and key files
- Writing basic test cases with assertions
- Executing tests in different modes
- Generating and interpreting HTML reports
- Collecting evidence through screenshots and videos

Continue Your Journey

Intermediate Topics:

- Page Object Model (POM) design pattern
- Data-driven testing with external files
- API testing with Playwright
- CI/CD integration with GitHub Actions
- Advanced locator strategies
- Parallel test execution