End-to-End Test Automation with Playwright & TypeScript

*By Inder P Singh for **SDETs**, **QA** and **Testers**, Basics to Framework Development, Interview Questions and Answers and Reference (Cheat) Sheet*

Download for reference ⬇️ Like 👍 Share 🔗

YouTube Channel: Software and Testing Training (341 Tutorials, 82,000 Subscribers)

Blog: Software Testing Space (1.8 Million Views)

1 Playwright with TypeScript Overview

Playwright is an open-source end-to-end testing framework from Microsoft for modern web apps. It supports all major browser engines (Chromium, Firefox, WebKit) and works on Windows, macOS, and Linux. Unlike Selenium, Playwright uses a fast WebSocket-based protocol for communication, making tests faster and more reliable. It natively supports TypeScript and JavaScript (as well as Java, Python, .NET). Using TypeScript adds static typing and IDE autocomplete, catching errors at compile time and improving maintainability.

Key Playwright features include mobile device emulation, headless and headed modes, and built-in video recording for debugging. It provides cross-browser testing (Chromium, Firefox, WebKit) and multi-language support. It has a comprehensive API for navigation and DOM interactions, integrations for CI/CD pipelines, visual testing (screenshots diff), and a code generator for recording tests. Playwright's architecture enables writing stable, fast tests with auto-waiting locators and fixture-based isolation.

2 End-to-End Test Setup in Playwright & TypeScript

To set up an E2E project with Playwright and TypeScript, install Node.js and run `npm init playwright@latest`. This scaffolds a TS-based project, adding playwright.config.ts, tests folder and example spec. Install dependencies with `npm install --save-dev @playwright/test` and download browsers via `npx playwright install`. In playwright.config.ts, you can set options like `testDir`, `timeout`, `retries`, and `projects` for browsers.

Write tests using the Playwright Test runner. For example, see below. This test navigates to a URL and asserts the page title. Run tests with "npx playwright test". The runner auto-detects ".ts" files, compiles TS, and launches Chromium, Firefox, and WebKit by default. You can use test hooks ("test.beforeEach") for common setup, and Playwright handles browser contexts per test for isolation. Use the VS Code extension or "npx playwright codegen <url>" to generate example tests and selectors.

```
import { test, expect } from '@playwright/test';

test('homepage has expected title', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  await expect(page).toHaveTitle(/Playwright/);
});
```

3 Core Playwright Test Automation Concepts

Playwright Test uses fixtures and the "test" function for structuring tests. Key concepts include:

- Locators: Playwright's locator API (e.g. "page.getByRole(...)", "page.locator(...)") auto-waits and retries actions until elements are stable. Prefer user-facing selectors ("getByRole", "getByLabel", "getByText") over brittle CSS/XPath.
- Actions & Auto-waiting: Commands like "click()", "fill()", "hover()" automatically wait for element visibility and readiness. No need for manual sleeps; Playwright polls internally.
- Assertions: Use "expect()" for asynchronous checks (e.g. "await expect(page).toHaveURL('/home')"). These "web-first" assertions wait for conditions (e.g. element to appear) before timing out.
- Test Isolation: By default, each test runs in a fresh browser context with empty state (cookies, storage) for reproducibility. Avoid sharing state between tests. Use "test.use({ storageState: 'state.json' })" or fixtures to reuse auth state if needed.
- Fixtures & Hooks: Playwright provides fixtures (e.g. "page", "browser") via test parameters. Use "test.beforeEach", "test.afterAll", or custom fixtures for setup/teardown. For example, "test.beforeEach(async ({ page }) => { await page.goto('/login'); })" logs in before every test.
- Configuration: Define "playwright.config.ts" to set global options (e.g. "baseURL", "headless", "screenshot", "retries"). This centralizes settings like timeouts, browsers, and CI behavior.

4 Playwright Test Automation Best Practices

- Test the User's View: Interact with visible UI elements and avoid implementation details. Verify user-visible behaviors (text, attributes) rather than internal state.
- Isolate Tests: Each test should run independently. Do not rely on the outcome of another test – use fresh state or recreate preconditions via fixtures.
- Robust Locators: Use Playwright's built-in locators with roles or labels. For example, "page.getByRole('button', { name: 'Submit' })" is preferred over CSS/XPath selectors that may break if DOM structure changes.
- Avoid External Dependencies: Stub third-party calls instead of hitting live endpoints. Use "page.route" to mock network responses for external APIs. For example, to make tests are deterministic and fast:

```
await page.route('**/external-api/**', route =>
  route.fulfill({ status: 200, body: JSON.stringify({ data: [] }) }));
await page.goto('https://example.com');
```

- Use Web-first Assertions: Always await on "expect()". E.g. "await expect(page.locator('text=Welcome')).toBeVisible()" will retry until timeout, avoiding flakiness.
- Maintainability: Follow Don't Repeat Yourself (DRY) principles with Page Objects or helper functions to encapsulate page structure and repeated flows. Keep tests readable and name them clearly.
- Parallel Safe: Configure "workers" and avoid shared resources. For CI, set "workers: 1" to avoid flaky parallel issues, or use test-specific fixtures to isolate data.

5 Building a Playwright Test Automation Framework with TypeScript

A robust framework structure separates tests, pages (POM), configs, and utilities. For example:

```
project/
├─ tests/               # Test spec files (e.g. login.spec.ts)
├─ pages/               # Page Object classes (e.g. LoginPage.ts)
├─ fixtures/            # Custom fixtures or shared test hooks
├─ config/              # Environment-specific configs (e.g. staging.config.ts)
├─ utils/               # Helper functions or test data
├─ playwright.config.ts # Global Playwright settings
├─ package.json
└─ README.md
```

Use "playwright.config.ts" (or multiple) for settings like "testDir", timeouts, reporter, "use" options, and "projects". For example, to runs tests against all browsers and configures artifacts:

```
import { defineConfig } from '@playwright/test';
export default defineConfig({
  testDir: './tests',
  timeout: 30_000,
  retries: process.env.CI ? 2 : 0,
  use: {
    baseURL: 'https://example.com',
    browserName: 'chromium',
    headless: true,
    screenshot: 'only-on-failure',
    video: 'retain-on-failure',
  },
  projects: [
    { name: 'Chromium', use: { browserName: 'chromium' } },
    { name: 'Firefox',  use: { browserName: 'firefox'  } },
    { name: 'WebKit',   use: { browserName: 'webkit'   } }
  ]
});
```

Follow Inder P Singh (18 years' experience in Test Automation) to get the new AI-based Testing, Test Automation and QA documents.

Implement Page Object Models for reusable page interactions. For example:

```
// pages/LoginPage.ts
import { Page, Locator } from '@playwright/test';
export class LoginPage {
  readonly page: Page;
  readonly username: Locator;
  readonly password: Locator;
  readonly submitBtn: Locator;
  constructor(page: Page) {
    this.page = page;
    this.username = page.getByLabel('Username');
    this.password = page.getByLabel('Password');
    this.submitBtn = page.getByRole('button', { name: 'Sign in' });
  }
  async goto() { await this.page.goto('/login'); }
  async login(user: string, pass: string) {
    await this.username.fill(user);
    await this.password.fill(pass);
    await this.submitBtn.click();
  }
}
```

In tests, instantiate Page Object Model such as below to encapsulate selectors and flows:

```
import { test } from '@playwright/test';
import { LoginPage } from '../pages/LoginPage';

test('valid login', async ({ page }) => {
  const loginPage = new LoginPage(page);
  await loginPage.goto();
  await loginPage.login('user1', 'Password123');
  // Assert post-login behavior...
});
```

Use custom fixtures and command-line or environment parameters for multi-environment runs. For instance, have separate configs (e.g. "staging.config.ts") and run with "npx

playwright test --config=config/staging.config.ts". Implement utilities for common tasks (test data setup, login, etc.) under "utils/".

6 Playwright with TypeScript API Testing Guide

Playwright Test includes an HTTP client via the "request" fixture for API testing. You can write tests to call REST endpoints directly, without a browser page. First, configure the base URL and headers (e.g. auth token) in "playwright.config.ts":

```
export default defineConfig({
  use: {
    baseURL: 'https://api.example.com',
    extraHTTPHeaders: {
      'Accept': 'application/json',
      'Authorization': "Bearer ${process.env.API_TOKEN}"
    },
  },
});
```

Then in a test:

```
test('create and fetch resource', async ({ request }) => {
  const createRes = await request.post('/items', { data: { name:
'foo' } });
  expect(createRes.ok()).toBeTruthy();
  const getRes = await request.get('/items');
  expect(await getRes.json()).toEqual(
```

```
    expect.arrayContaining([expect.objectContaining({ name: 'foo' })])
  );
});
```

This uses "request.get()" and "request.post()", and checks "response.ok()" and response data. You can also integrate this with page tests (prepare state via API before visiting a page) or perform full API CRUD sequences. The APIRequestContext follows the same "baseURL" and headers, making API tests concise and type-safe in TypeScript.

7 Advanced Playwright Test Techniques & Tricks

- Executing In-Page Scripts: Use "page.evaluate()" to run JavaScript in the page context, e.g. setting localStorage or reading the clipboard. For example:

```
// Set localStorage item await page.evaluate(() =>
localStorage.setItem('token', 'ABC123'));
```

This can prepare app state when no UI flow is provided.

- Test Steps: Use "test.step()" to group actions with descriptive names for clearer reports:

```
await test.step('Fill form and submit', async () => {
    await page.fill('#email', 'user@example.com');
    await page.click('#submit');
  });
```

Each step appears in the trace and report, aiding debug.

- Device & Geolocation Emulation: Playwright can emulate devices and geolocation. E.g., "browser.newContext({ ...devices['Pixel 5'], geolocation: { latitude: ..., longitude: ... } })".

- Advanced Waiting: Beyond simple waits, you can use "page.waitForResponse(urlOrPredicate)" or "locator.waitFor()" for custom sync logic.
- Handling Complex UI: For drag-and-drop: use "await page.locator('#drag').dragTo(page.locator('#drop'))". For mouse/keyboard events: "await page.mouse.move(x, y)", "await page.keyboard.press('Enter')".
- File Upload/Download: Upload with "await page.setInputFiles(selector, 'path/to/file.png')". For downloads, use "const [download] = await Promise.all([page.waitForEvent('download'), page.click('#download')]); await download.path();".
- Dialogs & Alerts: Listen for "'dialog'" events to accept or dismiss alerts/prompts:

```
page.on('dialog', dialog => dialog.accept());

  // or dialog.dismiss() to cancel
```

- Evaluate with Locators: You can call ".evaluate()" on a Locator to transform the element, e.g. "await page.locator('.input').evaluate(el => el.setAttribute('placeholder', 'Foo'))".
- Debugging: Use "page.on('console', msg => console.log(msg.text()))" to stream browser console logs. Start Playwright with "DEBUG=pw:api" to see verbose protocol logs.

8 Playwright Network Interception & Mocking in TypeScript

Playwright allows intercepting and modifying network requests via "page.route()" or "context.route()". You can abort, continue, or fulfill requests. For example:

Modifying requests lets you inject auth tokens or block ads:

```
await page.route('**/*', (route, request) => {
  // override headers
  const headers = { ...request.headers(), 'X-Test': 'true' };
  route.continue({ headers });
});
```

Mocking responses: Use "route.fulfill()" to stub a response. For example, to simulate a network error or serve test data:

```
// Return 404 for all requests
await page.route('**/*', route =>
  route.fulfill({ status: 404, contentType: 'text/plain', body: 'Not Found' })
);

// Serve static JSON for a specific endpoint
await page.route('**/api/data', route =>
  route.fulfill({ json: { items: [1,2,3] } })
);
```

Logging or waiting on requests: Use "page.on('request')" and "page.on('response')" to log or wait for specific network traffic:

```
page.on('request', request => console.log('>>', request.method(), request.url()));
page.on('response', response => console.log('<<', response.status(), response.url()));
await page.goto('https://example.com');
```

If you're interested in learning advanced automation testing and AI/ML in test automation, please follow or better, connect with me in LinkedIn at https://www.linkedin.com/in/inderpsingh/

9 Debugging & Reporting: Trace Viewer, Screenshots, and Logs

Playwright provides rich debugging tools. To troubleshoot failures:

- Tracing: Enable tracing to record snapshots of test execution. In config or CLI, set "trace: 'on-first-retry'" (or "'on'" in development). Run tests with "--trace on" or "--trace on-first-retry". After a test run, open the trace with:

```
npx playwright show-trace path/to/trace.zip
```

This opens the Trace Viewer GUI (or use https://trace.playwright.dev), allowing you to step through each action, see DOM snapshots, network, logs, and screenshots. Traces are invaluable for investigating flaky or CI failures.

- Screenshots & Videos: Configure screenshots in tests or config: "await page.screenshot({ path: 'image.png' })" captures the page. Use "fullPage: true" or element screenshots ("await page.locator('header').screenshot({ path: 'header.png' })"). In "playwright.config.ts", you can set "screenshot: 'only-on-failure'" and "video: 'retain-on-failure'" to automatically save artifacts for failed tests. These images/videos appear in the HTML report.
- Reporters: Playwright has built-in reporters ("list", "dot", "json", "html", etc.). Run with "npx playwright test --reporter=html" to generate an HTML report folder. The default HTML reporter collects screenshots, traces, and step logs, and opens if failures occur. Configure multiple reporters in "playwright.config.ts" (e.g. "reporter: [['list'], ['json', { outputFile: 'results.json' }]]"). On CI, upload the "playwright-report/" folder (or artifacts) to review these results.
- Console Logs: Use "page.on('console', msg => ...)" to capture browser console output. Alternatively, use "DEBUG=pw:api" env var to see Playwright's debug logs in the terminal during execution.

10 Playwright with TypeScript CI/CD Pipeline Integration

To integrate Playwright into CI/CD (GitHub Actions, Jenkins, GitLab, etc.), confirm that the browsers can run on the agent (use Playwright's Docker images or install dependencies). Typical steps:

1. Install dependencies: "npm ci" and "npx playwright install --with-deps" to install browser binaries.
2. Run tests: "npx playwright test". Use "workers: 1" in CI to avoid resource contention (config or env).
3. Artifacts and reports: Configure the workflow to upload "playwright-report/" (HTML report and trace logs) as build artifacts. For example, a GitHub Actions job may use:

```
- name: Install dependencies
  run: npm ci
- name: Install Playwright Browsers
  run: npx playwright install --with-deps
- name: Run Playwright tests
  run: npx playwright test
- name: Upload Report
  uses: actions/upload-artifact@v4
  with:
    name: playwright-report
    path: playwright-report/
```

Playwright's documentation provides sample CI configs for many platforms. In pipelines, set environment variables for secrets (e.g. "API_TOKEN"). Use the HTML reporter for review and consider using GitHub's or other systems to display test status. Use "cache: npm" and browser caches to speed up pipelines. For example, GitLab CI and Jenkins can similarly checkout the repo, setup Node, install Playwright, run tests, and archive results.

I'm happy to answer your questions on Software and Testing Training YouTube channel or in LinkedIn at https://www.linkedin.com/in/inderpsingh/

11 Playwright Parallel & Cross-Browser Test Automation

Playwright Test is designed for parallel execution and cross-browser coverage:

- Parallelism: By default, tests in different files run in parallel across multiple workers. Enable even more parallelism with "fullyParallel: true" in config to run tests in the same file concurrently. Control concurrency with the "workers" setting or "--workers" CLI flag (set workers to CPU count or a percentage). On powerful machines you can scale up workers, but on CI it's safer to use one worker for predictability. For very large suites, consider sharding tests across multiple jobs.

- Cross-Browser: Define "projects" in playwright.config.ts for each browser/device. For example:

```
projects: [
    { name: 'Desktop Chrome', use: { browserName: 'chromium' } },
    { name: 'Desktop Firefox', use: { browserName: 'firefox' } },
    { name: 'Mobile Safari', use: { ...devices['iPhone 13'] } },
    // etc.
  ]
```

Running "npx playwright test" will execute all tests under each project, giving coverage across browsers and devices. You can also override via CLI: "npx playwright test --project=firefox".

- Headed vs Headless: For debugging or video capture, run browsers in headed mode ("headless: false"). In CI, use headless. Set this in config or via "use: { headless: true/false }".

- Isolated Contexts: Each worker launches its own browser context to avoid interference. You can manually spawn additional contexts ("browser.newContext()") for even more isolated parallel flows.

12 Playwright Interview Questions: TypeScript & Automation

Common interview questions in Playwright & TS include:

- Language Support: Q: What languages does Playwright support? A: Playwright has official libraries for JavaScript/TypeScript, Python, .NET, and Java. TypeScript is widely used for its type safety.
- Playwright vs Selenium: Q: How is Playwright different from Selenium? A: Playwright does not use the WebDriver protocol and instead uses direct browser connections

(WebSocket), which makes it faster and less flaky. It also natively supports multiple browsers (via separate engine binaries) and has auto-waiting and an assertion library built-in.

- Locators & Selectors: Q: How do you select elements in Playwright? A: Use Playwright's locator API such as "page.getByRole('button', { name: 'OK' })" or "page.locator('css=div.my-class')". Playwright also allows XPath via "page.locator('//div')", but CSS or role-based selectors are preferred.
- File Upload/Download: Q: How to upload a file? A: Use "setInputFiles()". E.g. "await page.getByLabel('Upload').setInputFiles('myfile.pdf');". Q: How to download a file? A: Use "page.waitForEvent('download')" while performing the download action, then call "download.path()" to get the file path.
- Drag and Drop: Q: How to perform drag-and-drop? A: Use "await sourceLocator.dragTo(targetLocator)", which internally handles the mouse events. Alternatively, use "page.mouse" methods for custom actions.
- Dialogs/Alerts: Q: How to handle alerts or dialogs? A: Listen to the "'dialog'" event:

```
page.on('dialog', dialog => dialog.accept()); // clicks OK
 // or dialog.dismiss() to cancel, or dialog.accept('text') for
prompt.
```

This must be set before the action that triggers the dialog.

- Network Logs: Q: How to capture network traffic? A: Use event listeners:

```
page.on('request', req => console.log('>>', req.method(), req.url()));
  page.on('response', res => console.log('<<', res.status(),
res.url()));
  await page.goto('https://example.com');
```

This logs all requests and responses during the test.

- Assertions: Q: How to assert page state? A: Use the "expect" API. E.g., "await expect(page).toHaveURL('/dashboard')" or "await expect(page.locator('text=Success')).toBeVisible()". Avoid manual checks like "isVisible()" without "await" to ensure auto-waiting.

- Parallel Execution: Q: How to run tests in parallel? A: By default Playwright Test runs files in parallel. Use config "fullyParallel: true" to parallelize even within a file. Control concurrency with "workers".
- Fixtures & Hooks: Q: What are fixtures and hooks? A: Fixtures (e.g. "page") are built-in test context objects injected into tests. Hooks like "test.beforeEach()" allow code to run before every test (e.g. login).
- Debugging Tools: Q: How do you debug Playwright tests? A: Use Playwright Inspector ("npx playwright test --debug"), VS Code debugging (with the Playwright extension), tracing ("--trace on"), and logs/screenshots. Use "page.pause()" or "test.step()" for interactive debugging.
- API Testing: Q: Does Playwright support API testing? A: Yes. You can call REST APIs using the built-in "request" fixture and perform assertions on the response status and JSON (see Chapter 6). For example: "const res = await request.get('/api/users'); expect(res.ok()).toBeTruthy();".
- TypeScript Benefits: Q: Why use TypeScript with Playwright? A: TypeScript adds static types to tests, preventing many runtime errors and providing better autocomplete and refactoring support. Playwright itself is written in TS, so the typings are first-class.

13 Playwright & TypeScript Cheat Sheet

- CLI Shortcuts:

"npm init playwright@latest" – scaffold project (choose TypeScript).

"npx playwright test --headed" – run tests in a headed browser.

"npx playwright test --project=Firefox" – run tests only in Firefox.

"npx playwright show-report" – open the latest HTML report.

- Common Selectors:

"page.getByRole('button', { name: 'Submit' })" – accessible name role.

"page.getByLabel('Username')" – form label.

"page.locator('css=#login')" – CSS.

Avoid XPath unless necessary.

- Code Snippets:

  await page.goto('https://example.com');

  await expect(page).toHaveURL('/home');

  await page.fill('#input', 'text');

  await page.click('button.submit');

  Take full-page screenshot: "await page.screenshot({ path: 'screen.png', fullPage: true });".

- Assertions:

  "await expect(page.locator('text=Hello')).toBeVisible();"

  "await expect(response.status()).toBe(200);" (API).

- Playwright Config Keys:

  "projects": define browsers (e.g. "{ name: 'chromium', use: { browserName: 'chromium' } }").

  "use": e.g. "{ headless: true, baseURL: '...', screenshot: 'only-on-failure' }".

  "retries", "workers", "timeout".

- Test Structure:

  "test.describe('Feature', () => { ... });" – group tests.

  "test.beforeEach(async ({ page }) => {...});" – hook.

  "test('does X', async ({ page }) => { ... });" – a test case.

- Browser Context:

  Create new context: "const context = await browser.newContext({ permissions: ['clipboard-read'] });"

  Tracing: "await context.tracing.start({ screenshots: true, snapshots: true }); ...; await context.tracing.stop({ path: 'trace.zip' });".