← Async JavaScript & Node.js

Advanced Playwright →

Playwright Basics

Table of Contents

Playwright Overview and Key Advantages

What is Playwright?

Playwright is a modern, open-source automation framework developed by Microsoft for testing web applications. It enables reliable end-to-end testing across all modern browsers.

Key Features

1. Cross-Browser Testing

- Chromium (Chrome, Edge)
- Firefox
- WebKit (Safari)
- Single API for all browsers

2. Auto-Waiting

- Automatically waits for elements to be ready
- No need for manual waits (`sleep()`)
- Reduces flaky tests

3. Fast Execution

- Runs tests in parallel by default
- Browser contexts for isolation
- Significantly faster than Selenium

4. Powerful Selectors

- CSS selectors
- XPath selectors
- Text selectors
- Role-based selectors (accessibility)
- Custom test ID selectors

5. Network Control

- Mock API responses
- Intercept network requests
- Modify responses
- Test offline scenarios

- Touch events support
- Geolocation testing

7. Screenshots & Videos

- Automatic failure screenshots
- Video recording of test execution
- Visual comparison testing

8. Debugging Tools

- Playwright Inspector
- Trace Viewer
- Time-travel debugging
- Visual step-by-step execution

Playwright vs Other Tools

Feature	Playwright	Selenium	Cypress	Puppeteer
Auto-waiting	✅ Yes	❌ No	✅ Yes	❌ No
Cross-browser	✅ All major	✅ All	⚠️ Limited	❌ Chrome only
Speed	⚡ Very Fast	Medium	⚡ Fast	⚡ Fast
API Testing	✅ Built-in	❌ No	✅ Yes	❌ No
Network Mocking	✅ Built-in	❌ Manual	✅ Built-in	⚠️ Basic

Parallel Execution	✅ Default	⚠️ Manual	⚠️ Paid	❌ Manual
Mobile Testing	✅ Emulation	✅ Real devices	❌ Limited	✅ Emulation
Learning Curve	Easy	Medium	Easy	Medium
Company Support	Microsoft	SeleniumHQ	Cypress.io	Google

When to Use Playwright

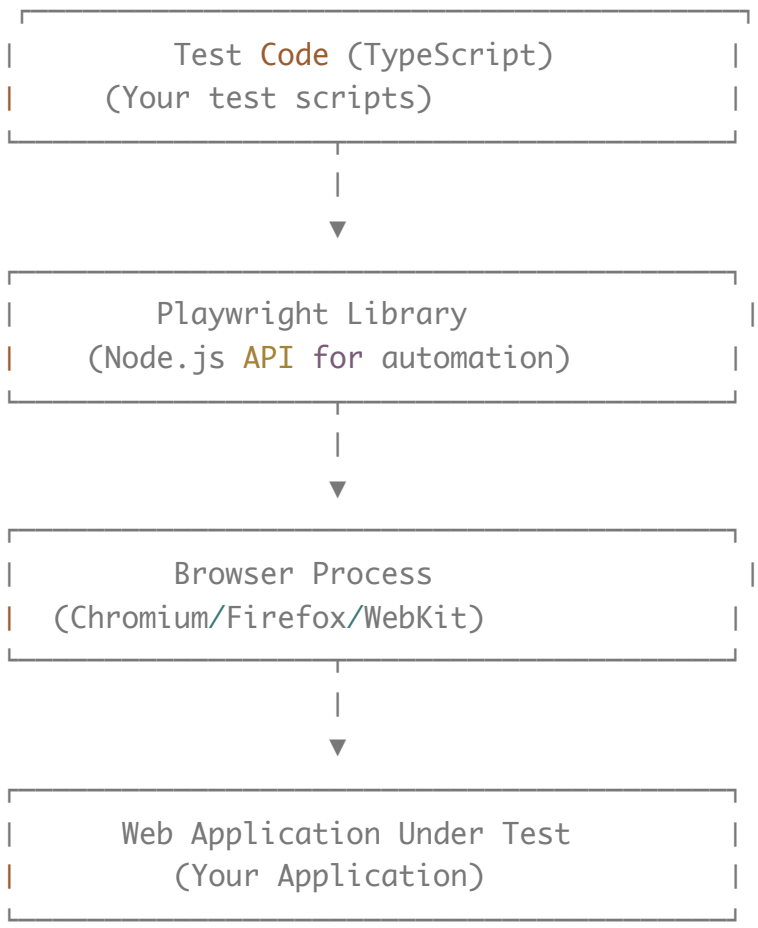
✅ **Best For:**

- Modern web applications (SPAs)
- Cross-browser testing requirements
- API + UI testing in same framework
- Fast, reliable test execution
- Teams using TypeScript/JavaScript
- Projects requiring network control
- CI/CD integration

⚠️ **Consider Alternatives For:**

- Legacy applications (use Selenium)
- Mobile native apps (use Appium)
- Non-JavaScript teams (consider Java + Selenium)

Playwright Architecture



Key Architectural Concepts:

1. Browser Context

- Isolated browser session (like incognito mode)
- Independent cookies, storage, and cache
- Multiple contexts = parallel execution
- Automatically cleaned up after test

2. Page

- Single tab in a browser
- Multiple pages per context
- Auto-waits for navigation

3. Frame

- Can interact with frame content

Playwright's Out-of-Process Architecture

Advantage: Playwright runs outside the browser process, communicating via WebSocket/CDP:

Benefits:

- More stable than in-process tools
- Better error recovery
- Can control browser lifecycle
- Network interception capabilities

Installation and Setup

Prerequisites

Required:

- Node.js 18+ installed
- npm or yarn package manager
- Code editor (VS Code recommended)

Verify Installation:

```
 node --version  # Should be v18 or higher
npm --version   # Should be 9 or higher
```

Create New Playwright Project

Option 1: Using npm init (Recommended)

```
cd playwright-automation

# Initialize Playwright
npm init playwright@latest

# Follow the prompts:
# - TypeScript or JavaScript? → TypeScript
# - Where to put tests? → tests
# - Add GitHub Actions? → No (we'll add later)
# - Install browsers? → Yes
```

This creates:

```
playwright-automation/
├── node_modules/
├── tests/
|   └── example.spec.ts
├── tests-examples/
|   └── demo-todo-app.spec.ts
├── .gitignore
├── package.json
├── package-lock.json
├── playwright.config.ts
└── README.md
```

Option 2: Manual Installation

```
# Initialize npm project
npm init -y

# Install Playwright
npm install -D @playwright/test

# Install browsers
npx playwright install
```

Project Structure Explained

playwright.config.ts - Main configuration file:

```
  testDir: './tests',
  fullyParallel: true,
  forbidOnly: !!process.env.CI,
  retries: process.env.CI ? 2 : 0,
  workers: process.env.CI ? 1 : undefined,
  reporter: 'html',

  use: {
    baseURL: 'http://localhost:3000',
    trace: 'on-first-retry',
    screenshot: 'only-on-failure',
    video: 'retain-on-failure',
  },

  projects: [
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'] },
    },
    {
      name: 'firefox',
      use: { ...devices['Desktop Firefox'] },
    },
    {
      name: 'webkit',
      use: { ...devices['Desktop Safari'] },
    },
  ],
});
```

Key Configuration Options:

Option	Description	Default
`testDir`	Directory containing tests	`./tests`
`timeout`	Maximum test execution time	30000ms
`retries`	Number of retries on failure	0

`workers`	Number of parallel workers	CPU cores
`reporter`	Test result reporter	`list`
`use.baseURL`	Base URL for tests	undefined
`use.trace`	When to collect trace	`on-first-retry`
`use.screenshot`	Screenshot capture	`only-on-failure`
`use.video`	Video recording	`retain-on-failure`

Installing Browsers

Install all browsers:

```
npx playwright install
```

Install specific browser:

```
npx playwright install chromium
npx playwright install firefox
npx playwright install webkit
```

Browser locations:

- Windows: `%USERPROFILE%\AppData\Local\ms-playwright`
- macOS: `~/Library/Caches/ms-playwright`
- Linux: `~/.cache/ms-playwright`

VS Code Extension

Install Playwright Test for VS Code:

1. Open VS Code

4. Install

Features:

- Run tests from editor
- Set breakpoints
- View test results inline
- Record new tests
- Generate locators

Understanding async/await in Playwright Context

Why async/await in Playwright?

All Playwright actions are asynchronous because:

- Browser operations take time
- Network requests are async
- Page navigation is async
- Element queries are async

Example:

```
 // ❌ WRONG - Without await
test('wrong way', async ({ page }) => {
  page.goto('https://example.com');  // Doesn't wait!
  page.click('button');              // May click before page loads!
});

// ✅ CORRECT - With await
test('correct way', async ({ page }) => {
  await page.goto('https://example.com');  // Waits for navigation
  await page.click('button');              // Waits for click to complete
});
```

1. Element to be attached to DOM
2. Element to be visible
3. Element to be stable (not animating)
4. Element to receive events (not obscured)
5. Element to be enabled (for inputs)

Example:

```
test('auto-waiting example', async ({ page }) => {
  await page.goto('https://example.com');

  // Playwright automatically waits for:
  // - Element to exist
  // - Element to be visible
  // - Element to be enabled
  // - Element to be stable
  await page.click('button');

  // No manual waits needed! 🎉
});
```

Common Async Patterns in Playwright

Pattern 1: Sequential Actions

```
test('sequential actions', async ({ page }) => {
  await page.goto('https://example.com');
  await page.fill('#username', 'testuser');
  await page.fill('#password', 'password123');
  await page.click('button[type="submit"]');
  await page.waitForURL('**/dashboard');
});
```

Pattern 2: Parallel Actions

```
  // Wait for multiple elements in parallel
  const [button, input, heading] = await Promise.all([
    page.waitForSelector('button'),
    page.waitForSelector('input'),
    page.waitForSelector('h1')
  ]);
});
```

Pattern 3: Getting Values

```
test('getting values', async ({ page }) => {
  await page.goto('https://example.com');

  const title = await page.title();
  const url = await page.url();
  const text = await page.textContent('h1');

  console.log({ title, url, text });
});
```

Writing Your First Automated Test

Test Structure

Basic Test Template:

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

test('test description', async ({ page }) => {
  // 1. Navigate to page
  await page.goto('https://example.com');

  // 2. Perform actions
  await page.click('button');

  // 3. Verify results
```

Your First Test

Create tests/first-test.spec.ts:

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

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

  // Expect page to have a title containing "Playwright"
  await expect(page).toHaveTitle(/Playwright/);
});

test('get started link', async ({ page }) => {
  // Navigate to Playwright homepage
  await page.goto('https://playwright.dev/');

  // Click the "Get started" link
  await page.getByRole('link', { name: 'Get started' }).click();

  // Verify we're on the installation page
  await expect(page.getByRole('heading', { name: 'Installation' })).toBeVi
});
```

Run the test:

```
 # Run all tests
npx playwright test

# Run specific test file
npx playwright test first-test.spec.ts

# Run in headed mode (see browser)
npx playwright test --headed

# Run in debug mode
npx playwright test --debug
```

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

test('basic test', async ({ page }) => {
  // Test code
});

test.only('only this test runs', async ({ page }) => {
  // Only this test will execute
});

test.skip('skip this test', async ({ page }) => {
  // This test will be skipped
});

test.fixme('test needs fixing', async ({ page }) => {
  // Marked as needing attention
});

test.slow('slow test', async ({ page }) => {
  // Triples the timeout
});

// Conditional skip
test('conditional skip', async ({ page, browserName }) => {
  test.skip(browserName === 'firefox', 'Not ready for Firefox yet');
  // Test code
});
```

Test Hooks

beforeEach/afterEach - Run before/after each test:

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

test.beforeEach(async ({ page }) => {
  // Runs before each test
  await page.goto('https://example.com');
  console.log('Test starting...');
});
```

```
});

test('test 1', async ({ page }) => {
  // Test code
});

test('test 2', async ({ page }) => {
  // Test code
});
```

beforeAll/afterAll - Run once for all tests:

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

test.beforeAll(async () => {
  // Runs once before all tests
  console.log('Starting test suite');
});

test.afterAll(async () => {
  // Runs once after all tests
  console.log('Test suite finished');
});

test('test 1', async ({ page }) => {
  // Test code
});

test('test 2', async ({ page }) => {
  // Test code
});
```

Test Organization with describe

Group related tests:

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

test.describe('Login Page Tests', () => {
  test.beforeEach(async ({ page }) => {
    await page.goto('https://example.com/login');
```

```
    await page.fill('#username', 'validuser');
    await page.fill('#password', 'validpass');
    await page.click('button[type="submit"]');
    await expect(page).toHaveURL('**/dashboard');
  });

  test('invalid credentials', async ({ page }) => {
    await page.fill('#username', 'invalid');
    await page.fill('#password', 'invalid');
    await page.click('button[type="submit"]');
    await expect(page.locator('.error')).toBeVisible();
  });

  test('empty fields validation', async ({ page }) => {
    await page.click('button[type="submit"]');
    await expect(page.locator('.validation-error')).toHaveCount(2);
  });
});

test.describe('Registration Page Tests', () => {
  test.beforeEach(async ({ page }) => {
    await page.goto('https://example.com/register');
  });

  test('successful registration', async ({ page }) => {
    // Test code
  });
});
```

Locator Strategies

What are Locators?

Locators are Playwright's way of finding elements on a page. They are:

- **Auto-waiting**: Wait for element to be actionable
- **Strict**: Throw error if multiple elements match
- **Resilient**: Re-query on each action

```
// By button role
await page.getByRole('button', { name: 'Submit' }).click();

// By link role
await page.getByRole('link', { name: 'Sign up' }).click();

// By textbox role
await page.getByRole('textbox', { name: 'Email' }).fill('user@example.com'

// By heading role
await expect(page.getByRole('heading', { name: 'Welcome' })).toBeVisible()

// By checkbox role
await page.getByRole('checkbox', { name: 'I agree' }).check();
```

Common ARIA Roles:

- `button` - Buttons
- `link` - Links
- `textbox` - Text inputs
- `checkbox` - Checkboxes
- `radio` - Radio buttons
- `heading` - Headings (h1-h6)
- `listitem` - List items
- `table` - Tables
- `row` - Table rows
- `cell` - Table cells

2. getByText() - By visible text

```
// Exact text match
await page.getByText('Submit').click();
await page.getByText('Welcome to Playwright').click();

// Substring match
await page.getByText('Submit', { exact: false }).click();
```

3. getByLabel() - Form fields by label

```
// Find input by associated label
await page.getByLabel('Email').fill('user@example.com');
await page.getByLabel('Password').fill('password123');
await page.getByLabel('Remember me').check();
```

4. getByPlaceholder() - By placeholder text

```
await page.getByPlaceholder('Enter your email').fill('user@example.com');
await page.getByPlaceholder('Search...').fill('Playwright');
```

5. getByAltText() - Images by alt text

```
await page.getByAltText('Company logo').click();
await expect(page.getByAltText('Product image')).toBeVisible();
```

6. getByTitle() - By title attribute

```
await page.getByTitle('Close').click();
await page.getByTitle('Settings').click();
```

7. getByTestId() - By data-testid attribute

```
await page.getByTestId('submit-button').click();
await page.getByTestId('user-name').fill('John Doe');

// Custom test id attribute (configure in playwright.config.ts)
// testIdAttribute: 'data-test-id'
```

CSS Selectors

Using page.locator() with CSS:

```
// By class
await page.locator('.submit-button').click();

// By attribute
await page.locator('[name="email"]').fill('user@example.com');
await page.locator('[type="submit"]').click();

// Descendant selector
await page.locator('.form input[type="text"]').fill('value');

// Complex selectors
await page.locator('button.primary:not(:disabled)').click();
```

XPath Selectors

Using XPath expressions:

```
 // Start with // for XPath
await page.locator('//button[text()="Submit"]').click();

// By ID
await page.locator('//input[@id="username"]').fill('testuser');

// By text content
await page.locator('//div[contains(text(), "Welcome")]').isVisible();

// Complex XPath
await page.locator('//form[@id="login"]//button[@type="submit"]').click();
```

Chaining Locators

Narrow down selection by chaining:

```
 // Find button inside a specific div
await page
  .locator('.modal')
  .getByRole('button', { name: 'Confirm' })
  .click();
```

```
  .filter({ hasText: 'Playwright' })
  .getByRole('button', { name: 'Read more' })
  .click();
```

Filtering Locators

Filter by text or nested elements:

```
 // Filter by text
await page
  .getByRole('listitem')
  .filter({ hasText: 'Active' })
  .click();

// Filter by nested element
await page
  .getByRole('listitem')
  .filter({ has: page.getByRole('button', { name: 'Delete' }) })
  .click();

// Exclude by text
await page
  .getByRole('listitem')
  .filter({ hasNotText: 'Disabled' })
  .count();
```

Locator Best Practices

Priority Order (Recommended by Playwright):

1. `getByRole()` - Best for accessibility
2. `getByLabel()` - Good for forms
3. `getByPlaceholder()` - Inputs without labels
4. `getByText()` - Non-interactive elements
5. `getByTestId()` - When other methods don't work
6. CSS/XPath - Last resort

Example of good locator strategy:

```
  // ✅ Good - Using semantic locators
  await page.getByLabel('Email').fill('user@example.com');
  await page.getByLabel('Password').fill('password123');
  await page.getByRole('button', { name: 'Sign in' }).click();

  // ❌ Bad - Using CSS selectors that might break
  // await page.locator('#input-1234').fill('user@example.com');
  // await page.locator('.btn-primary').click();
});
```

Basic Assertions and Test Structure

Playwright Assertions

Import expect:

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

Page Assertions

Title and URL:

```
 // Page title
await expect(page).toHaveTitle('Playwright');
await expect(page).toHaveTitle(/Playwright/);

// Page URL
await expect(page).toHaveURL('https://playwright.dev/');
await expect(page).toHaveURL(/.*playwright.*/);
await expect(page).toHaveURL(/\/docs\/intro/);
```

Element Assertions

Visibility:

```
// Element is hidden
await expect(page.getByText('Loading')).toBeHidden();

// Element is not visible
await expect(page.getByText('Error')).not.toBeVisible();
```

Text Content:

```
 // Exact text match
await expect(page.locator('h1')).toHaveText('Welcome');

// Contains text
await expect(page.locator('h1')).toContainText('Welcome');

// Regex match
await expect(page.locator('h1')).toHaveText(/welcome/i);

// Array of texts
await expect(page.locator('li')).toHaveText(['Item 1', 'Item 2', 'Item 3']
```

Element State:

```
 // Element is enabled
await expect(page.locator('button')).toBeEnabled();

// Element is disabled
await expect(page.locator('button')).toBeDisabled();

// Checkbox is checked
await expect(page.getByRole('checkbox')).toBeChecked();

// Element is editable
await expect(page.locator('input')).toBeEditable();

// Element is focused
await expect(page.locator('input')).toBeFocused();
```

Attributes:

```
// Has class
await expect(page.locator('div')).toHaveClass('active');
await expect(page.locator('div')).toHaveClass(/btn-/);

// Has ID
await expect(page.locator('div')).toHaveId('header');
```

Values:

```
// Input value
await expect(page.locator('input')).toHaveValue('test');
await expect(page.getByLabel('Email')).toHaveValue('user@example.com');

// Has values (for select multiple)
await expect(page.locator('select')).toHaveValues(['option1', 'option2']);
```

Count:

```
// Element count
await expect(page.getByRole('listitem')).toHaveCount(5);
await expect(page.locator('.product')).toHaveCount(10);

// At least/at most
const count = await page.locator('.item').count();
expect(count).toBeGreaterThan(0);
expect(count).toBeLessThanOrEqual(20);
```

Advanced Assertions

Custom timeout:

```
// Wait up to 10 seconds
await expect(page.getByText('Loaded')).toBeVisible({ timeout: 10000 });
```

Soft assertions (continue on failure):

```
  // These won't stop test execution
  await expect.soft(page.locator('h1')).toHaveText('Title');
  await expect.soft(page.locator('h2')).toHaveText('Subtitle');

  // Test continues even if above fail
  await page.click('button');
});
```

Negation:

```
 // Not visible
await expect(page.getByText('Error')).not.toBeVisible();

// Not have text
await expect(page.locator('h1')).not.toHaveText('Wrong title');

// Not checked
await expect(page.getByRole('checkbox')).not.toBeChecked();
```

Test Structure Best Practices

AAA Pattern (Arrange-Act-Assert):

```
test('user can login', async ({ page }) => {
  // Arrange - Set up test data and navigate
  const email = 'user@example.com';
  const password = 'password123';
  await page.goto('https://example.com/login');

  // Act - Perform actions
  await page.getByLabel('Email').fill(email);
  await page.getByLabel('Password').fill(password);
  await page.getByRole('button', { name: 'Sign in' }).click();

  // Assert - Verify results
  await expect(page).toHaveURL('**/dashboard');
  await expect(page.getByText('Welcome')).toBeVisible();
});
```

```
test('user can add item to cart', async ({ page }) => {
  // Given user is on product page
  await page.goto('https://example.com/products/123');
  await expect(page.getByRole('heading', { name: 'iPhone 15' })).toBeVisib

  // When user adds product to cart
  await page.getByRole('button', { name: 'Add to Cart' }).click();

  // Then product is added successfully
  await expect(page.getByText('Added to cart')).toBeVisible();
  await expect(page.locator('.cart-count')).toHaveText('1');
});
```

Debugging with Playwright Inspector and Trace Viewer

Playwright Inspector

What is Playwright Inspector?

- Interactive debugging tool
- Step through test execution
- Inspect locators
- Generate code
- View console logs

Launch Inspector:

```
# Run test in debug mode
npx playwright test --debug

# Debug specific test
npx playwright test example.spec.ts --debug

# Debug from specific line
```

Using Inspector:

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

test('debug example', async ({ page }) => {
  await page.goto('https://example.com');

  // Pause execution here
  await page.pause();

  // Continue with test
  await page.click('button');
});
```

Inspector Features:

- **Pick Locator**: Click to generate selector
- **Step Over**: Execute current action
- **Step Into**: Step into action details
- **Resume**: Continue execution
- **Console**: View console output

Debugging in VS Code

Using VS Code Extension:

1. Install "Playwright Test for VSCode"
2. Set breakpoints in test code
3. Click "Debug Test" in editor
4. Use VS Code debug controls

Debug Configuration (.vscode/launch.json):

```
{
  "version": "0.2.0",
  "configurations": [
    {
```

```
      "program": "${workspaceFolder}/node_modules/@playwright/test/cli.js",
      "args": ["test", "--debug"],
      "console": "integratedTerminal"
    }
  ]
}
```

Trace Viewer

What is Trace Viewer?

- Time-travel debugger
- View full test execution
- Network activity
- Console logs
- Screenshots at each step
- DOM snapshots

Enable Tracing:

In playwright.config.ts:

```
export default defineConfig({
  use: {
    trace: 'on',  // Always record trace
    // OR
    trace: 'on-first-retry',  // Only on retry
    // OR
    trace: 'retain-on-failure',  // Only on failure
  },
});
```

In Test:

```
test('with tracing', async ({ page, context }) => {
  // Start tracing
  await context.tracing.start({ screenshots: true, snapshots: true });
```

```
  // Stop tracing and save
  await context.tracing.stop({ path: 'trace.zip' });
});
```

View Trace:

```
 # After test runs, trace is saved in test-results folder
npx playwright show-trace test-results/trace.zip

# Or open HTML report and click on trace
npx playwright show-report
```

Trace Viewer Features:

- **Timeline**: See all actions
- **Screenshots**: Visual state at each step
- **Console**: All console messages
- **Network**: All network requests
- **Source**: Test source code
- **Call**: Action details
- **Errors**: Error messages and stack traces

Common Debugging Techniques

1. Headed Mode (See Browser):

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

2. Slow Motion:

```
 test.use({ launchOptions: { slowMo: 1000 } }); // 1 second delay

test('slow test', async ({ page }) => {
  await page.goto('https://example.com');
```

3. Screenshots:

```
test('with screenshots', async ({ page }) => {
  await page.goto('https://example.com');

  // Take screenshot
  await page.screenshot({ path: 'screenshot.png' });

  // Full page screenshot
  await page.screenshot({ path: 'full-page.png', fullPage: true });

  // Element screenshot
  await page.locator('.header').screenshot({ path: 'header.png' });
});
```

4. Console Logs:

```
test('console logs', async ({ page }) => {
  // Listen to console messages
  page.on('console', msg => console.log('Browser log:', msg.text()));

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

5. Network Monitoring:

```
test('network monitoring', async ({ page }) => {
  // Log all network requests
  page.on('request', request => {
    console.log('Request:', request.url());
  });

  // Log all network responses
  page.on('response', response => {
    console.log('Response:', response.url(), response.status());
  });

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

```
test('page errors', async ({ page }) => {
  // Listen to page errors
  page.on('pageerror', error => {
    console.error('Page error:', error.message);
  });

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

Practice Exercises

Exercise 1: Basic Navigation and Assertions

Task: Test a Public Website

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

test.describe('Wikipedia Tests', () => {
  test('should load Wikipedia homepage', async ({ page }) => {
    // Navigate to Wikipedia
    // Your code: goto 'https://www.wikipedia.org/'

    // Assert title contains "Wikipedia"
    // Your code: expect page to have title

    // Assert logo is visible
    // Your code: expect logo to be visible
  });

  test('should search for "Playwright"', async ({ page }) => {
    // Navigate to Wikipedia

    // Enter "Playwright" in search box
    // Your code: find search input and type

    // Click search button or press Enter

    // Assert we're on Playwright page
    // Your code: expect heading to contain "Playwright"
```

```
    await page.goto('https://www.wikipedia.org/');

    // Click on a language (e.g., Español)
    // Your code: find and click language link

    // Assert URL changed
    // Your code: expect URL to contain "es.wikipedia"
  });
});
```

Exercise 2: Form Interaction

Task: Test a Demo Form

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

test.describe('Form Tests', () => {
  test.beforeEach(async ({ page }) => {
    // Navigate to: https://www.saucedemo.com/
    await page.goto('https://www.saucedemo.com/');
  });

  test('successful login', async ({ page }) => {
    // Enter username: standard_user
    // Your code

    // Enter password: secret_sauce
    // Your code

    // Click login button
    // Your code

    // Assert successful login
    // Your code: check for products page
  });

  test('login with empty credentials', async ({ page }) => {
    // Click login without entering credentials
    // Your code

    // Assert error message is displayed
    // Your code: find error message
  });
```

```
    // Your code

    // Click login
    // Your code

    // Assert error message
    // Your code
  });
});
```

Exercise 3: Multiple Locator Strategies

Task: Use Different Locators

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

test('locator strategies', async ({ page }) => {
  await page.goto('https://playwright.dev/');

  // 1. Use getByRole to find "Get started" link
  // Your code: click the link

  // 2. Use getByText to find installation heading
  // Your code: assert heading is visible

  // 3. Use getByPlaceholder to find search box
  // Your code: type "locators"

  // 4. Use CSS selector to find navigation menu
  // Your code: assert menu exists

  // 5. Use XPath to find a specific element
  // Your code: create XPath locator
});
```

Exercise 4: Shopping Cart Workflow

Task: Automate E-commerce Flow

```
  test('add product to cart', async ({ page }) => {
    // 1. Login to https://www.saucedemo.com/
    await page.goto('https://www.saucedemo.com/');
    await page.getByTestId('username').fill('standard_user');
    await page.getByTestId('password').fill('secret_sauce');
    await page.getByTestId('login-button').click();

    // 2. Add first product to cart
    // Your code: click "Add to cart" button

    // 3. Verify cart badge shows "1"
    // Your code: assert cart count

    // 4. Go to cart
    // Your code: click cart icon

    // 5. Verify product is in cart
    // Your code: assert product name is visible

    // 6. Remove product from cart
    // Your code: click remove button

    // 7. Verify cart is empty
    // Your code: assert no products in cart
  });

  test('checkout flow', async ({ page }) => {
    // Login and add product to cart (reuse from above)

    // Go to cart and click checkout
    // Your code

    // Fill checkout form
    // Your code: first name, last name, zip code

    // Continue to next step
    // Your code

    // Verify order summary
    // Your code: assert product details

    // Complete order
    // Your code: click finish
```

```
});
```

Exercise 5: Debugging Practice

Task: Debug Failing Tests

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

test('buggy test - fix me!', async ({ page }) => {
  await page.goto('https://playwright.dev/');

  // This test has bugs - use debugger to fix
  await page.pause(); // Start debugging here

  // Bug 1: Wrong locator
  await page.click('button#get-started'); // This selector is wrong

  // Bug 2: Wrong assertion
  await expect(page).toHaveURL('https://playwright.dev/docs/installation')

  // Bug 3: Element not visible
  await page.click('.hidden-element');

  // Use Playwright Inspector to:
  // 1. Pick the correct locators
  // 2. Verify the URL
  // 3. Check element visibility
});

// Run with: npx playwright test --debug
```

Exercise 6: Comprehensive Test Suite

Task: Build Complete Test Suite

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

test.describe('Complete Test Suite', () => {
  let page;
```

```
  test.afterAll(async () => {
    // Cleanup: Close page
  });

  test('test 1: Homepage loads', async () => {
    // Your code
  });

  test('test 2: Navigation works', async () => {
    // Your code
  });

  test('test 3: Search functionality', async () => {
    // Your code
  });

  test('test 4: Form submission', async () => {
    // Your code
  });

  test('test 5: Responsive design', async ({ page }) => {
    // Test on mobile viewport
    await page.setViewportSize({ width: 375, height: 667 });
    // Your code
  });
});
```

Exercise 7: Create Your Own Test

Task: Test Your Favorite Website

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

// Choose a website and create a test suite
// Examples:
// - GitHub: search repos, view profile
// - Amazon: search products, add to cart
// - Netflix: browse content
// - YouTube: search videos, play video

test.describe('My Website Tests', () => {
  // Implement at least 3 test scenarios
```

```
  test('scenario 1', async ({ page }) => {
    // Your creative test here
  });

  test('scenario 2', async ({ page }) => {
    // Your creative test here
  });

  test('scenario 3', async ({ page }) => {
    // Your creative test here
  });
});
```

Summary

In Day 3, you learned:

✅ Playwright Overview

- Modern automation framework
- Cross-browser support
- Auto-waiting and reliability
- Comparison with other tools

✅ Architecture

- Browser contexts and pages
- Out-of-process architecture
- WebSocket communication

✅ Installation and Setup

- Project initialization
- Configuration options
- Browser installation

- Why all actions are async
- Auto-waiting behavior
- Sequential vs parallel execution

✅ First Tests

- Test structure
- Test annotations
- Hooks and lifecycle
- Test organization

✅ Locator Strategies

- getByRole (recommended)
- getByLabel, getByText
- CSS and XPath selectors
- Chaining and filtering
- Best practices

✅ Assertions

- Page assertions
- Element assertions
- State checks
- Custom timeouts
- Soft assertions

✅ Debugging

- Playwright Inspector
- Trace Viewer
- VS Code debugging
- Screenshots and videos

Key Takeaways

- **Always use await** with Playwright actions
- **Prefer semantic locators** (getByRole) over CSS/XPath
- **Let Playwright auto-wait** - no manual waits needed
- **Use trace viewer** for debugging failures
- **Test in multiple browsers** for compatibility

Next Steps

- Practice writing tests for different websites
- Experiment with all locator strategies
- Get comfortable with debugging tools
- Try testing in different browsers
- Complete all practice exercises

Useful Commands

```
 # Run tests
npx playwright test

# Run specific file
npx playwright test example.spec.ts

# Run in headed mode
npx playwright test --headed

# Debug mode
npx playwright test --debug

# Run in specific browser
npx playwright test --project=chromium

# Generate test
npx playwright codegen

# Show report
```

```
npx playwright show-trace trace.zip
```

End of Day 3 Documentation

← Async JavaScript & Node.js

Advanced Playwright →