Playwright Automation Framework - Complete Learning Guide

> **A comprehensive guide covering Register and Login Test Automation with Playwright + TypeScript**

Table of Contents

1. Introduction

What is Playwright?

Playwright is a modern, open-source automation library developed by Microsoft for end-to-end testing of web applications. It provides a single API to automate Chromium, Firefox, and WebKit browsers.

Key Features of Playwright

Feature	Description
Cross-browser	Single API for Chromium, Firefox, and WebKit
Auto-wait	Automatically waits for elements before performing actions
Web-first assertions	Built-in assertions that retry automatically
Isolation	Each test runs in a fresh browser context
Tracing	Capture screenshots, videos, and traces
Parallelism	Run tests in parallel out of the box

What is TypeScript?

TypeScript is a strongly-typed superset of JavaScript that compiles to plain JavaScript. It adds optional static typing and class-based object-oriented programming to the language.

Why Playwright + TypeScript?

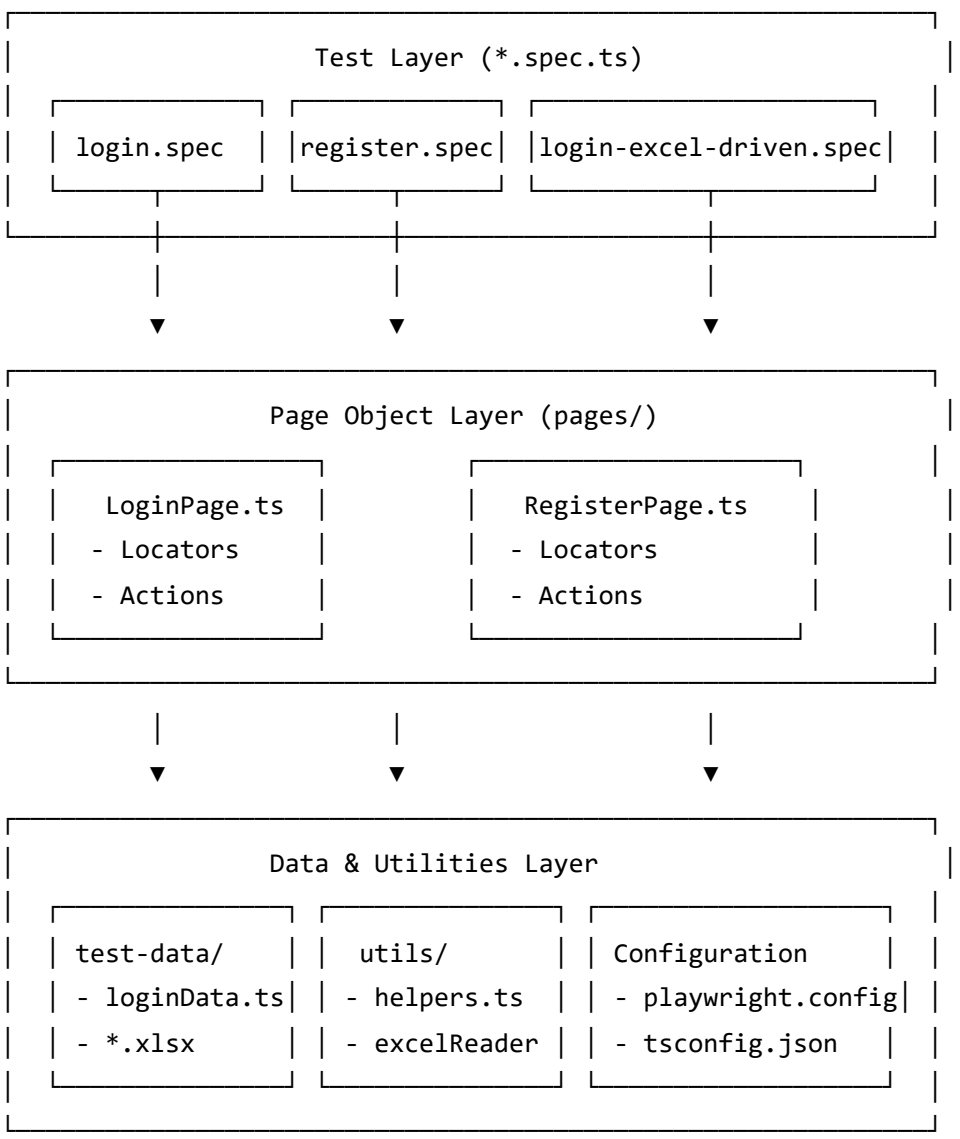
1. **Type Safety**: Catch errors at compile-time
2. **IntelliSense**: Better IDE autocomplete support
3. **Maintainability**: Self-documenting code with interfaces
4. **Refactoring**: Safer code refactoring
5. **Modern Features**: Access to ES6+ features

2. Framework Architecture Overview

Project Structure

```
playwright-typescript-framework/
├── pages/                    # Page Object Model classes
│   ├── LoginPage.ts          # Login page actions & locators
│   └── RegisterPage.ts       # Registration page actions & locators
├── tests/                    # Test specification files
│   ├── login.spec.ts         # Login test cases
│   ├── login-excel-driven.spec.ts  # Excel data-driven tests
│   ├── login-data-driven.spec.ts   # Array data-driven tests
│   └── register.spec.ts      # Registration test cases
├── test-data/                # Test data files
│   ├── loginData.ts          # Login test data
│   ├── registerData.ts       # Registration test data
│   └── loginTestData.xlsx    # Excel test data
├── utils/                    # Utility functions
│   ├── helpers.ts            # Common helper functions
│   └── excelReader.ts        # Excel file reader utility
├── playwright.config.ts      # Playwright configuration
├── tsconfig.json             # TypeScript configuration
└── package.json              # Project dependencies
```

Architecture Diagram



3. Step-by-Step Setup Guide

Prerequisites

Before setting up the framework, ensure you have:

1. **Node.js** (version 18 or higher) - Download
2. **Visual Studio Code** (recommended IDE) - Download
3. **Git** (optional, for version control) - Download

Step 1: Create Project Directory

```
# Create a new directory for your project
mkdir playwright-typescript-framework
cd playwright-typescript-framework
```

Step 2: Initialize Node.js Project

```
# Initialize with default settings
npm init -y
```

This creates a `package.json` file with default configuration.

Step 3: Install Playwright

```
# Install Playwright Test as a dev dependency
npm install -D @playwright/test

# Install Playwright browsers
npx playwright install
```

Step 4: Install TypeScript

```
# Install TypeScript and Node.js types
npm install -D typescript @types/node
```

Step 5: Install Additional Dependencies

```
# For Excel file reading (data-driven testing)
npm install -D xlsx
```

Step 6: Configure TypeScript (tsconfig.json)

Create `tsconfig.json` in the project root:

```
{
    "compilerOptions": {
        "target": "ES2020",
        "module": "commonjs",
        "lib": ["ES2020", "DOM"],
        "strict": true,
        "esModuleInterop": true,
        "skipLibCheck": true,
        "forceConsistentCasingInFileNames": true,
        "outDir": "./dist",
        "rootDir": "./",
        "moduleResolution": "node",
        "resolveJsonModule": true,
        "declaration": true,
        "declarationMap": true,
        "sourceMap": true,
        "baseUrl": ".",
        "paths": {
            "@pages/*": ["pages/*"],
            "@test-data/*": ["test-data/*"],
            "@utils/*": ["utils/*"]
        }
    },
    "include": [
        "tests/**/*.ts",
        "pages/**/*.ts",
        "test-data/**/*.ts",
        "utils/**/*.ts",
        "playwright.config.ts"
    ],
    "exclude": ["node_modules", "dist"]
}
```

Step 7: Configure Playwright (playwright.config.ts)

Create `playwright.config.ts` :

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

export default defineConfig({
    // Test directory
    testDir: './tests',

    // Run tests in parallel
    fullyParallel: true,

    // Fail on CI if test.only is left
    forbidOnly: !!process.env.CI,

    // Retry configuration
    retries: process.env.CI ? 2 : 1,

    // Workers configuration
    workers: process.env.CI ? 1 : undefined,

    // Reporter configuration
    reporter: [
        ['html', { open: 'never' }],
        ['list']
    ],

    // Global settings
    use: {
        baseURL: 'https://your-app-url.com',
        trace: 'on-first-retry',
        screenshot: 'on',
        video: 'on',
        actionTimeout: 10000,
        navigationTimeout: 30000,
    },

    // Global timeout
    timeout: 60000,

    // Assertion timeout
    expect: {
        timeout: 10000,
    },

    // Browser projects
```

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

Step 8: Create Folder Structure

```
# Create required directories
mkdir pages tests test-data utils
```

Step 9: Update package.json Scripts

Add the following scripts to `package.json` :

```
{
  "scripts": {
    "test": "npx playwright test",
    "test:headed": "npx playwright test --headed",
    "test:debug": "npx playwright test --debug",
    "test:ui": "npx playwright test --ui",
    "report": "npx playwright show-report"
  }
}
```

Step 10: Verify Setup

```
# Run a simple test to verify setup
npx playwright test --version
```

4. TypeScript Concepts Used

4.1 Interfaces

Definition: An interface defines a contract for objects, specifying what properties and methods they must have.

```
// From loginData.ts
export interface LoginCredentials {
    username: string;
    password: string;
}
```

Why use interfaces?

- Type checking at compile time
- IntelliSense support in IDEs
- Self-documenting code
- Contract enforcement

4.2 Classes

Definition: A class is a blueprint for creating objects with predefined properties and methods.

```
// From LoginPage.ts
export class LoginPage {
    readonly page: Page;
    readonly usernameInput: Locator;

    constructor(page: Page) {
        this.page = page;
        this.usernameInput = page.locator('#username');
    }
}
```

4.3 Access Modifiers

Modifier	Description	Example
`readonly`	Property can only be assigned in constructor	`readonly page: Page`

Modifier	Description	Example
`public`	Accessible from anywhere (default)	`public login()`
`private`	Only accessible within the class	`private validate()`
`protected`	Accessible within class and subclasses	`protected helper()`

4.4 Type Annotations

Definition: Explicitly specifying the type of variables, parameters, and return values.

```
// Variable type annotation
const username: string = 'testuser';
const count: number = 5;
const isValid: boolean = true;

// Function parameter and return type annotations
async function login(username: string, password: string): Promise<void> {
    // implementation
}

// Array type annotation
const errors: string[] = [];
```

4.5 Union Types

Definition: A type that can be one of several types.

```
// From excelReader.ts
export interface ExcelRow {
    [key: string]: string | number | boolean | undefined;
}

// Literal union type
expectedOutcome: 'success' | 'error';
```

4.6 Optional Properties

Definition: Properties that may or may not be present, denoted by `?`.

```typescript
// From loginData.ts
export interface PartialLoginCredentials {
    username?: string;  // Optional
    password?: string;  // Optional
}
```

4.7 Export/Import

Definition: Module system for sharing code between files.

```typescript
// Exporting (from LoginPage.ts)
export class LoginPage { }

// Importing (in test file)
import { LoginPage } from '../pages/LoginPage';
import { test, expect } from '@playwright/test';
```

4.8 Async/Await

Definition: Syntax for handling asynchronous operations in a synchronous-looking manner.

```typescript
// Async function declaration
async function navigate(): Promise<void> {
    await this.page.goto('/');
    await this.page.waitForLoadState('networkidle');
}

// Using async/await in tests
test('login test', async ({ page }) => {
    await page.goto('/login');
    await page.fill('#username', 'user');
});
```

4.9 Generics

Definition: Allows creating reusable components that work with multiple types.

```
// From excelReader.ts
const data = XLSX.utils.sheet_to_json<ExcelRow>(sheet);

// Array with generic type
const messages: string[] = [];
for (let i = 0; i < count; i++) {
    // Process each item
}
```

4.10 Arrow Functions

Definition: Concise syntax for writing functions.

```
// Arrow function syntax
const generateRandomString = (length: number): string => {
    // implementation
};

// Arrow function in callback
errors.some(error => error.includes('username'));

// Arrow function with async
const wait = async (ms: number): Promise<void> => {
    return new Promise(resolve => setTimeout(resolve, ms));
};
```

4.11 Destructuring

Definition: Extract values from objects or arrays into distinct variables.

```
// Object destructuring
const { username, password } = validLoginCredentials;

// Object destructuring in parameters
test.beforeEach(async ({ page }) => {
    // 'page' is destructured from the test fixtures
});

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

4.12 Template Literals

Definition: String literals allowing embedded expressions.

```
// Using template literals
const email = `test_${randomPart}_${timestamp}@example.com`;
const username = `user_${Date.now()}`;
console.log(`Testing with: username="${data.username}"`);
```

4.13 Type Assertions

Definition: Override TypeScript's inferred type.

```
// Type assertion with 'as'
expectedOutcome: (String(row.expectedOutcome || 'success').toLowerCase()) as 'success' | 'error
```

4.14 Spread Operator

Definition: Expand elements of an array or object.

```
// Spread in objects
use: { ...devices['Desktop Chrome'] }

// Shuffle array using spread
return password.split('').sort(() => Math.random() - 0.5).join('');
```

5. Playwright Concepts Used

5.1 Page Object

Definition: Represents a browser page/tab. Central object for interacting with web pages.

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

export class LoginPage {
    readonly page: Page;

    constructor(page: Page) {
        this.page = page;
    }
}
```

Key Methods:

Method	Description
`page.goto(url)`	Navigate to URL
`page.url()`	Get current URL
`page.waitForURL(pattern)`	Wait for URL to match pattern
`page.waitForLoadState(state)`	Wait for load state
`page.waitForTimeout(ms)`	Explicit wait (use sparingly)
`page.textContent(selector)`	Get text content
`page.screenshot()`	Take screenshot

5.2 Locator

Definition: A way to find and interact with elements. Auto-waits and auto-retries.

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

// Creating locators
readonly usernameInput: Locator;
this.usernameInput = page.locator('#username');
```

Locator Strategies:

```
// CSS Selectors
page.locator('#username')                    // By ID
page.locator('.login-form')                  // By class
page.locator('input[type="password"]')       // By attribute
page.locator('button:has-text("Login")')     // With text

// Data Test ID (recommended)
page.locator('[data-testid="login-button"]')

// Combining locators
page.locator('#password + button')           // Adjacent sibling
page.locator('#password ~ button').first()   // General sibling, first match
```

Key Locator Methods:

Method	Description
`locator.click()`	Click element
`locator.fill(value)`	Fill input with value
`locator.clear()`	Clear input field
`locator.getAttribute(name)`	Get attribute value
`locator.inputValue()`	Get input value
`locator.textContent()`	Get text content
`locator.count()`	Count matching elements
`locator.nth(index)`	Get nth matching element
`locator.first()`	Get first matching element
`locator.waitFor()`	Wait for element
`locator.scrollIntoViewIfNeeded()`	Scroll to element

5.3 Test Runner (test)

Definition: Playwright's test runner provides test organization and execution.

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

// Basic test
test('should login successfully', async ({ page }) => {
    // test code
});

// Test with description
test.describe('Login Page Tests', () => {
    test('test case 1', async ({ page }) => {});
    test('test case 2', async ({ page }) => {});
});
```

5.4 Test Hooks

Definition: Functions that run at specific points in the test lifecycle.

```
test.describe('Test Suite', () => {
    // Runs before each test
    test.beforeEach(async ({ page }) => {
        await page.goto('/login');
    });

    // Runs after each test
    test.afterEach(async ({ page }) => {
        await page.close();
    });

    // Runs before all tests in this describe
    test.beforeAll(async () => {
        // setup
    });

    // Runs after all tests in this describe
    test.afterAll(async () => {
        // cleanup
    });
});
```

5.5 Assertions (expect)

Definition: Built-in assertions for validating test conditions.

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

// Page assertions
await expect(page).toHaveURL(/.*\/account/);
await expect(page).toHaveTitle('Dashboard');

// Locator assertions
await expect(locator).toBeVisible();
await expect(locator).toBeEnabled();
await expect(locator).toHaveValue('expected');
await expect(locator).toHaveAttribute('type', 'password');
await expect(locator).toContainText('Welcome');
await expect(locator).toBeInViewport();

// Soft assertions (don't stop test)
expect.soft(value).toBe(expected);

// Negation
await expect(locator).not.toBeVisible();
```

5.6 Fixtures

Definition: Test fixtures provide setup/teardown and are passed to tests.

```
// Built-in fixtures
test('example', async ({ page, context, browser }) => {
    // page - isolated page instance
    // context - browser context
    // browser - browser instance
});
```

5.7 Configuration (defineConfig)

Definition: Function to create typed Playwright configuration.

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

export default defineConfig({
    testDir: './tests',
    fullyParallel: true,
    retries: 2,
    workers: 4,
    reporter: [['html'], ['list']],
    use: {
        baseURL: 'https://example.com',
        trace: 'on-first-retry',
        screenshot: 'on',
        video: 'on',
    },
    projects: [
        { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    ],
});
```

5.8 Browser Devices

Definition: Pre-configured device emulation settings.

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

// Desktop devices
devices['Desktop Chrome']
devices['Desktop Firefox']
devices['Desktop Safari']

// Mobile devices
devices['iPhone 14']
devices['Pixel 7']
```

5.9 Wait States

Definition: Different page load states to wait for.

```
// Wait for network idle (no requests for 500ms)
await page.waitForLoadState('networkidle');

// Wait for DOM content loaded
await page.waitForLoadState('domcontentloaded');

// Wait for full load
await page.waitForLoadState('load');
```

5.10 Actions Auto-Waiting

Definition: Playwright automatically waits for elements to be actionable.

Auto-wait includes:

- Element is attached to DOM
- Element is visible
- Element is stable (not animating)
- Element receives events (not obscured)
- Element is enabled (for inputs)

```
// These automatically wait
await locator.click();    // Waits until clickable
await locator.fill('x');  // Waits until fillable
await locator.check();    // Waits until checkable
```

6. Page Object Model (POM) Deep Dive

What is Page Object Model?

Definition: A design pattern that creates an object repository for web UI elements. Each page in the application has a corresponding Page Class.

Benefits of POM

1. **Maintainability**: Locators in one place
2. **Reusability**: Methods can be reused across tests

3. **Readability**: Tests read like user actions
4. **Single Responsibility**: Each page handles its own elements
5. **Easy Updates**: Change locator once, not in every test

LoginPage Implementation Analysis

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

export class LoginPage {
    // ========== PROPERTIES ==========
    readonly page: Page;              // Page instance
    readonly usernameInput: Locator;  // Locators
    readonly passwordInput: Locator;
    readonly loginButton: Locator;
    readonly resetButton: Locator;
    readonly rememberMeCheckbox: Locator;

    // ========== CONSTRUCTOR ==========
    constructor(page: Page) {
        this.page = page;

        // Initialize locators once
        this.usernameInput = page.locator('#username');
        this.passwordInput = page.locator('#password');
        this.rememberMeCheckbox = page.locator('#remember-me');
        this.loginButton = page.locator('[data-testid="login-button"]');
        this.resetButton = page.locator('button:has-text("Reset")');
    }

    // ========== NAVIGATION ==========
    async navigate(): Promise<void> {
        await this.page.goto('/');
        await this.page.waitForLoadState('networkidle');
        await this.page.locator('button:has-text("Login")').first().click();
        await this.page.waitForURL('**/login');
    }

    // ========== FORM ACTIONS ==========
    async fillLoginForm(username: string, password: string): Promise<void> {
        await this.usernameInput.fill(username);
        await this.passwordInput.fill(password);
    }

    async submitForm(): Promise<void> {
        await this.loginButton.click();
    }
```

```
    async login(username: string, password: string): Promise<void> {
        await this.fillLoginForm(username, password);
        await this.submitForm();
    }

    // ========== VERIFICATION ==========
    async verifyPageLoaded(): Promise<void> {
        await expect(this.usernameInput).toBeVisible();
        await expect(this.passwordInput).toBeVisible();
        await expect(this.loginButton).toBeVisible();
    }

    async isLoginSuccessful(): Promise<boolean> {
        try {
            await this.page.waitForURL('**/account', { timeout: 5000 });
            return true;
        } catch {
            return false;
        }
    }
}
```

Page Object Best Practices

1. **Use readonly for locators**: Prevents accidental reassignment
2. **Return Promise** for action methods
3. **Separate concerns**: Navigation, actions, verifications
4. **Use meaningful method names**: `login()` not `doStuff()`
5. **Don't expose locators directly**: Provide methods instead
6. **Add JSDoc comments**: Document complex methods

7. Test Data Management

7.1 TypeScript Interfaces for Data

```
// Define structure for test data
export interface LoginCredentials {
    username: string;
    password: string;
}

export interface LoginTestScenario {
    testId: string;
    description: string;
    username: string;
    password: string;
    expectedOutcome: 'success' | 'error';
    expectedError?: string;
}
```

7.2 Static Test Data

```
// Simple credentials
export const validLoginCredentials: LoginCredentials = {
    username: 'testuser',
    password: 'SecurePass123!'
};

// Empty credentials for validation testing
export const emptyLoginCredentials: LoginCredentials = {
    username: '',
    password: ''
};
```

7.3 Dynamic Test Data Generation

```
// Generate unique credentials for each test run
export function generateUniqueLoginCredentials(): LoginCredentials {
    const timestamp = Date.now();
    return {
        username: `user_${timestamp}`,
        password: 'SecurePass123!'
    };
}

// Generate random email
export function generateRandomEmail(): string {
    const timestamp = Date.now();
    const randomPart = generateRandomString(5);
    return `test_${randomPart}_${timestamp}@example.com`;
}
```

7.4 Test Scenario Arrays for Data-Driven Testing

```
export const validLoginScenarios: LoginTestScenario[] = [
    {
        testId: 'VL001',
        description: 'Standard alphanumeric username',
        username: 'testuser123',
        password: 'Password123!',
        expectedOutcome: 'success'
    },
    {
        testId: 'VL002',
        description: 'Email format username',
        username: 'user@example.com',
        password: 'SecurePass@2024',
        expectedOutcome: 'success'
    }
];
```

8. Writing Tests - Basic to Advanced

8.1 Basic Test Structure

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

test.describe('Login Page Tests', () => {
    let loginPage: LoginPage;

    // Setup - runs before each test
    test.beforeEach(async ({ page }) => {
        loginPage = new LoginPage(page);
        await loginPage.navigate();
    });

    // Simple test case
    test('TC001 - Should login successfully', async ({ page }) => {
        // Arrange
        const username = 'testuser';
        const password = 'SecurePass123!';

        // Act
        await loginPage.login(username, password);

        // Assert
        await expect(page).toHaveURL(/.*\/account/);
    });
});
```

8.2 AAA Pattern (Arrange, Act, Assert)

```
test('TC002 - Should show error for empty username', async () => {
    // ARRANGE - Set up preconditions
    const credentials = {
        username: '',
        password: 'SecurePass123!'
    };

    // ACT - Perform the action
    await loginPage.fillFields(credentials);
    await loginPage.submitForm();

    // ASSERT - Verify results
    const errors = await loginPage.getErrorMessages();
    expect(errors.some(e => e.includes('Username'))).toBeTruthy();
    expect(loginPage.getCurrentUrl()).toContain('/login');
});
```

8.3 Testing Form Validation

```
test('TC003 - Should show errors when submitting empty form', async () => {
    // Act
    await loginPage.submitForm();

    // Assert
    const errors = await loginPage.getErrorMessages();
    expect(errors.length).toBeGreaterThanOrEqual(2);
    expect(errors.some(e => e.includes('username'))).toBeTruthy();
    expect(errors.some(e => e.includes('password'))).toBeTruthy();
});
```

8.4 Testing UI Interactions

```
test('TC005 - Should toggle password visibility', async () => {
    // Arrange
    await loginPage.fillFields({ password: 'SecurePass123!' });

    // Initial state - password hidden
    let inputType = await loginPage.getPasswordInputType();
    expect(inputType).toBe('password');

    // Act - Toggle visibility ON
    await loginPage.togglePasswordVisibility();
    inputType = await loginPage.getPasswordInputType();
    expect(inputType).toBe('text');

    // Act - Toggle visibility OFF
    await loginPage.togglePasswordVisibility();
    inputType = await loginPage.getPasswordInputType();
    expect(inputType).toBe('password');
});
```

8.5 Testing Navigation

```
test('TC008 - Should navigate to Register page', async ({ page }) => {
    // Act
    await loginPage.goToRegister();

    // Assert
    await expect(page).toHaveURL(/.*\/register/);
});
```

9. Data-Driven Testing

9.1 Array-Based Data-Driven Testing

```
import { validLoginScenarios } from '../test-data/loginData';

test.describe('Valid Login Scenarios', () => {
    for (const scenario of validLoginScenarios) {
        test(`${scenario.testId} - ${scenario.description}`, async ({ page }) => {
            const loginPage = new LoginPage(page);
            await loginPage.navigate();

            await loginPage.login(scenario.username, scenario.password);

            if (scenario.expectedOutcome === 'success') {
                await expect(page).toHaveURL(/.*\/account/);
            } else {
                expect(loginPage.getCurrentUrl()).toContain('/login');
            }
        });
    }
});
```

9.2 Excel-Based Data-Driven Testing

Excel Reader Utility:

```typescript
import * as XLSX from 'xlsx';
import * as path from 'path';

export interface LoginExcelData {
    testId: string;
    description: string;
    username: string;
    password: string;
    expectedOutcome: 'success' | 'error';
    expectedError?: string;
}

export function readLoginTestData(
    filePath: string,
    sheetName?: string
): LoginExcelData[] {
    const absolutePath = path.resolve(filePath);
    const workbook = XLSX.readFile(absolutePath);

    const sheet = sheetName
        ? workbook.Sheets[sheetName]
        : workbook.Sheets[workbook.SheetNames[0]];

    const rawData = XLSX.utils.sheet_to_json(sheet);

    return rawData.map(row => ({
        testId: String(row.testId || ''),
        description: String(row.description || ''),
        username: String(row.username || ''),
        password: String(row.password || ''),
        expectedOutcome: String(row.expectedOutcome || 'success') as 'success' | 'error',
        expectedError: row.expectedError ? String(row.expectedError) : undefined
    }));
}
```

Using Excel Data in Tests:

```
import { readLoginTestData } from '../utils/excelReader';
import * as path from 'path';

const EXCEL_FILE = path.join(__dirname, '../test-data/loginTestData.xlsx');
const testData = readLoginTestData(EXCEL_FILE, 'ValidLogins');

test.describe('Excel Data-Driven Tests', () => {
    for (const data of testData) {
        test(`${data.testId} - ${data.description}`, async ({ page }) => {
            const loginPage = new LoginPage(page);
            await loginPage.navigate();

            await loginPage.login(data.username, data.password);

            if (data.expectedOutcome === 'success') {
                await expect(page).toHaveURL(/.*\/account/);
            }
        });
    }
});
```

10. Configuration Deep Dive

10.1 Playwright Configuration Options

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

export default defineConfig({
    // ========== TEST DISCOVERY ==========
    testDir: './tests',              // Directory containing tests
    testMatch: '**/*.spec.ts',       // Pattern to match test files
    testIgnore: '**/fixtures/**',    // Pattern to ignore

    // ========== EXECUTION ==========
    fullyParallel: true,             // Run all tests in parallel
    workers: 4,                       // Number of parallel workers
    retries: 2,                       // Retry failed tests
    timeout: 60000,                   // Test timeout (60 seconds)

    // ========== CI/CD ==========
    forbidOnly: !!process.env.CI,    // Fail if test.only left in code

    // ========== REPORTING ==========
    reporter: [
        ['html', { open: 'never' }], // HTML report
        ['list'],                     // Console list output
        ['json', { outputFile: 'results.json' }],
        ['junit', { outputFile: 'results.xml' }]
    ],

    // ========== GLOBAL USE OPTIONS ==========
    use: {
        baseURL: 'https://example.com',

        // Screenshots
        screenshot: 'only-on-failure', // or 'on', 'off'

        // Video recording
        video: 'retain-on-failure',   // or 'on', 'off'

        // Tracing
        trace: 'on-first-retry',      // or 'on', 'off', 'retain-on-failure'
```

```
        // Timeouts
        actionTimeout: 10000,
        navigationTimeout: 30000,

        // Browser options
        headless: true,
        viewport: { width: 1280, height: 720 },
    },

    // ========== ASSERTIONS ==========
    expect: {
        timeout: 10000,              // Assertion timeout
        toHaveScreenshot: {
            maxDiffPixels: 100,      // Visual comparison threshold
        },
    },

    // ========== PROJECTS (Browsers) ==========
    projects: [
        {
            name: 'chromium',
            use: { ...devices['Desktop Chrome'] },
        },
        {
            name: 'firefox',
            use: { ...devices['Desktop Firefox'] },
        },
        {
            name: 'mobile',
            use: { ...devices['iPhone 14'] },
        },
    ],
});
```

10.2 TypeScript Configuration (tsconfig.json)

```jsonc
{
    "compilerOptions": {
        // ========== LANGUAGE VERSION ==========
        "target": "ES2020",           // Output JavaScript version
        "module": "commonjs",         // Module system
        "lib": ["ES2020", "DOM"],     // Available type definitions

        // ========== STRICT MODE ==========
        "strict": true,               // Enable all strict checks
        "esModuleInterop": true,      // CommonJS import compatibility
        "skipLibCheck": true,         // Skip .d.ts file checking

        // ========== OUTPUT ==========
        "outDir": "./dist",           // Compiled output directory
        "rootDir": "./",              // Source root
        "sourceMap": true,            // Enable source maps

        // ========== MODULE RESOLUTION ==========
        "moduleResolution": "node",   // How to resolve modules
        "resolveJsonModule": true,    // Allow importing JSON

        // ========== PATH ALIASES ==========
        "baseUrl": ".",
        "paths": {
            "@pages/*": ["pages/*"],
            "@test-data/*": ["test-data/*"],
            "@utils/*": ["utils/*"]
        }
    },
    "include": ["tests/**/*.ts", "pages/**/*.ts"],
    "exclude": ["node_modules"]
}
```

11. Best Practices & Patterns

11.1 Locator Best Practices

```
// ✅ GOOD - Using data-testid (most reliable)
page.locator('[data-testid="login-button"]')

// ✅ GOOD - Using ID (unique identifier)
page.locator('#username')

// ✅ GOOD - Using specific attributes
page.locator('button[type="submit"]')

// ⚠️ OKAY - Using text (may break with translations)
page.locator('button:has-text("Login")')

// ❌ BAD - Using fragile selectors
page.locator('div > div > button:nth-child(2)')
page.locator('.btn-primary.mt-4.px-8')
```

11.2 Wait Strategies

```
// ✅ GOOD - Use built-in auto-waiting
await loginButton.click();  // Playwright auto-waits

// ✅ GOOD - Use explicit waits when needed
await page.waitForURL('**/account');
await page.waitForLoadState('networkidle');

// ⚠️ USE SPARINGLY - Hard waits (only when absolutely necessary)
await page.waitForTimeout(500);
```

11.3 Assertion Best Practices

```
// ✅ GOOD - Use web-first assertions (auto-retry)
await expect(page).toHaveURL(/.*\/account/);
await expect(loginButton).toBeVisible();

// ⚠️ OKAY - Regular assertions (no retry)
expect(errors.length).toBeGreaterThan(0);

// ✅ GOOD - Check multiple conditions
await expect(loginPage.usernameInput).toBeVisible();
await expect(loginPage.passwordInput).toBeVisible();
await expect(loginPage.loginButton).toBeEnabled();
```

11.4 Test Organization

```
test.describe('Feature: User Login', () => {

    test.describe('Positive Tests', () => {
        test('should login with valid credentials', async () => {});
        test('should remember user when checked', async () => {});
    });

    test.describe('Negative Tests', () => {
        test('should reject empty username', async () => {});
        test('should reject empty password', async () => {});
    });

    test.describe('Edge Cases', () => {
        test('should handle special characters', async () => {});
        test('should handle very long inputs', async () => {});
    });
});
```

12. Interview Questions

Basic Level

Q1: What is Playwright and how does it differ from Selenium?

A: Playwright is a modern automation library by Microsoft with:

- Single API for all browsers (Chromium, Firefox, WebKit)
- Built-in auto-waiting (no explicit waits needed)
- Better parallelization support
- Native TypeScript support
- Faster execution due to CDP protocol

Q2: What is Page Object Model (POM)?

A: POM is a design pattern that creates an object repository for web elements. Each page has a corresponding class containing:

- Locators as properties
- Actions as methods
- Benefits: maintainability, reusability, readability

Q3: What is the difference between `locator.click()` and `page.$().click()`?

A:

- `locator.click()` - Auto-waits for element to be actionable, recommended approach
- `page.$().click()` - Returns ElementHandle, doesn't auto-wait, legacy approach

Intermediate Level

Q4: How does Playwright handle auto-waiting?

A: Playwright automatically waits for:

1. Element to be attached to DOM
2. Element to be visible
3. Element to be stable (no animations)
4. Element to receive events (not obscured)
5. Element to be enabled (for inputs)

Q5: Explain different reporter types in Playwright.

A:

- `html` - Interactive HTML report with screenshots/videos
- `list` - Console output with test names
- `json` - JSON format for parsing
- `junit` - XML format for CI/CD integration
- `dot` - Minimal dot output

Q6: What is the purpose of `test.beforeEach` hook?

A: Runs before each test in a describe block for:

- Setting up test prerequisites
- Navigating to test page
- Initializing Page Objects
- Setting up test data

Advanced Level

Q7: How would you implement data-driven testing with Excel?

A:

1. Install xlsx package: `npm install xlsx`
2. Create utility to read Excel file
3. Map rows to typed interface
4. Use for loop to generate tests dynamically
5. Each row becomes a separate test case

Q8: Explain parallel test execution in Playwright.

A:

- `fullyParallel: true` - All tests run in parallel
- `workers: n` - Number of parallel workers
- Each test gets isolated browser context
- Use `test.describe.serial()` for sequential tests

Q9: How do you handle flaky tests?

A:

1. Use proper auto-waiting (avoid `waitForTimeout`)
2. Configure retries in config
3. Use stable locators (data-testid)
4. Wait for network idle when needed
5. Use `toPass()` for polling assertions
6. Add trace and video for debugging

Q10: What's the difference between `expect(locator).toBeVisible()` and `locator.isVisible()`?

A:

- `expect(locator).toBeVisible()` - Assertion that auto-retries
- `locator.isVisible()` - Returns boolean immediately, no retry
- Use assertions for verifications, use methods for conditional logic

Quick Reference Commands

```
# Run all tests
npx playwright test

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

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

# Run in debug mode
npx playwright test --debug

# Run in UI mode (interactive)
npx playwright test --ui

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

# Generate HTML report
npx playwright show-report

# Run with trace on
npx playwright test --trace on

# Run specific test by title
npx playwright test -g "should login successfully"
```

Conclusion

This guide covered the complete Playwright + TypeScript automation framework including:

- ✅ Project setup and configuration
- ✅ TypeScript concepts (interfaces, classes, generics, async/await)
- ✅ Playwright concepts (Page, Locator, assertions, hooks)
- ✅ Page Object Model implementation
- ✅ Test data management strategies

✅ Data-driven testing with arrays and Excel
✅ Best practices and patterns
✅ Interview preparation questions

Next Steps:

1. Practice implementing Page Objects for different pages
2. Create more complex test scenarios
3. Explore visual testing with `toHaveScreenshot()`
4. Implement API testing with Playwright
5. Set up CI/CD with GitHub Actions

Document created: December 31, 2024
Framework version: Playwright 1.40.0, TypeScript 5.3.0