← Playwright Basics

Playwright Advanced Scenarios →

Advanced Playwright - Web Automation

Table of Contents

Page Object Model (POM) Design Pattern

What is Page Object Model?

Page Object Model (POM) is a design pattern that creates an object repository for web UI elements. It helps make test code more maintainable, readable, and reusable.

```
// ❌ BAD: Locators scattered across tests
test('login test 1', async ({ page }) => {
  await page.fill('#username', 'user1');
  await page.fill('#password', 'pass1');
  await page.click('button[type="submit"]');
});

test('login test 2', async ({ page }) => {
  await page.fill('#username', 'user2');  // Duplicate locator
  await page.fill('#password', 'pass2');  // Duplicate locator
  await page.click('button[type="submit"]');  // Duplicate locator
});

// If locators change, you need to update ALL tests!
```

Benefits of POM:

- ✅ Single source of truth for locators
- ✅ Easy maintenance (change once, effect everywhere)
- ✅ Reusable methods across tests
- ✅ Better readability
- ✅ Separation of test logic and page logic

Basic POM Structure

File Structure:

```
project/
├── pages/
|   ├── LoginPage.ts
|   ├── DashboardPage.ts
|   └── BasePage.ts
├── tests/
|   └── login.spec.ts
└── playwright.config.ts
```

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

export class LoginPage {
  readonly page: Page;
  readonly usernameInput: Locator;
  readonly passwordInput: Locator;
  readonly loginButton: Locator;
  readonly errorMessage: Locator;

  constructor(page: Page) {
    this.page = page;
    this.usernameInput = page.getByLabel('Username');
    this.passwordInput = page.getByLabel('Password');
    this.loginButton = page.getByRole('button', { name: 'Login' });
    this.errorMessage = page.locator('.error-message');
  }

  async goto() {
    await this.page.goto('/login');
  }

  async login(username: string, password: string) {
    await this.usernameInput.fill(username);
    await this.passwordInput.fill(password);
    await this.loginButton.click();
  }

  async getErrorMessage(): Promise<string> {
    return await this.errorMessage.textContent() || '';
  }

  async isErrorVisible(): Promise<boolean> {
    return await this.errorMessage.isVisible();
  }
}
```

Using Page Object in Tests:

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

```
  test.beforeEach(async ({ page }) => {
    loginPage = new LoginPage(page);
    await loginPage.goto();
  });

  test('successful login', async ({ page }) => {
    await loginPage.login('validuser', 'validpass');
    await expect(page).toHaveURL('/dashboard');
  });

  test('login with invalid credentials', async () => {
    await loginPage.login('invalid', 'invalid');
    await expect(await loginPage.isErrorVisible()).toBe(true);
    const error = await loginPage.getErrorMessage();
    expect(error).toContain('Invalid credentials');
  });

  test('login with empty fields', async () => {
    await loginPage.login('', '');
    await expect(await loginPage.isErrorVisible()).toBe(true);
  });
});
```

Advanced POM Patterns

1. Base Page Class:

```
 // pages/BasePage.ts
import { Page, Locator } from '@playwright/test';

export class BasePage {
  readonly page: Page;

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

  async goto(path: string) {
    await this.page.goto(path);
  }

  async getTitle(): Promise<string> {
    return await this.page.title();
```

```
    return this.page.url();
  }

  async waitForPageLoad() {
    await this.page.waitForLoadState('networkidle');
  }

  async takeScreenshot(name: string) {
    await this.page.screenshot({ path: `screenshots/${name}.png` });
  }

  async clickElement(locator: Locator) {
    await locator.click();
  }

  async fillInput(locator: Locator, text: string) {
    await locator.fill(text);
  }

  async getText(locator: Locator): Promise<string> {
    return await locator.textContent() || '';
  }

  async isVisible(locator: Locator): Promise<boolean> {
    return await locator.isVisible();
  }

  async waitForElement(locator: Locator) {
    await locator.waitFor({ state: 'visible' });
  }
}
```

2. Inheriting from BasePage:

```
 // pages/DashboardPage.ts
import { Page, Locator } from '@playwright/test';
import { BasePage } from './BasePage';

export class DashboardPage extends BasePage {
  readonly welcomeMessage: Locator;
  readonly logoutButton: Locator;
  readonly profileLink: Locator;
```

```
    this.logoutButton = page.getByRole('button', { name: 'Logout' });
    this.profileLink = page.getByRole('link', { name: 'Profile' });
  }

  async goto() {
    await super.goto('/dashboard');
  }

  async getWelcomeMessage(): Promise<string> {
    return await this.getText(this.welcomeMessage);
  }

  async logout() {
    await this.clickElement(this.logoutButton);
  }

  async goToProfile() {
    await this.clickElement(this.profileLink);
  }

  async isDashboardLoaded(): Promise<boolean> {
    return await this.isVisible(this.welcomeMessage);
  }
}
```

3. Component Objects (for reusable UI components):

```
 // components/NavigationComponent.ts
import { Page, Locator } from '@playwright/test';

export class NavigationComponent {
  readonly page: Page;
  readonly homeLink: Locator;
  readonly productsLink: Locator;
  readonly cartLink: Locator;
  readonly userMenu: Locator;

  constructor(page: Page) {
    this.page = page;
    this.homeLink = page.getByRole('link', { name: 'Home' });
    this.productsLink = page.getByRole('link', { name: 'Products' });
    this.cartLink = page.getByRole('link', { name: 'Cart' });
    this.userMenu = page.locator('.user-menu');
```

```
    await this.homeLink.click();
  }

  async goToProducts() {
    await this.productsLink.click();
  }

  async goToCart() {
    await this.cartLink.click();
  }

  async openUserMenu() {
    await this.userMenu.click();
  }
}
```

Using Components in Page Objects:

```
 // pages/ProductsPage.ts
import { Page, Locator } from '@playwright/test';
import { BasePage } from './BasePage';
import { NavigationComponent } from '../components/NavigationComponent';

export class ProductsPage extends BasePage {
  readonly navigation: NavigationComponent;
  readonly productCards: Locator;
  readonly addToCartButtons: Locator;

  constructor(page: Page) {
    super(page);
    this.navigation = new NavigationComponent(page);
    this.productCards = page.locator('.product-card');
    this.addToCartButtons = page.getByRole('button', { name: 'Add to Cart'
  }

  async goto() {
    await super.goto('/products');
  }

  async getProductCount(): Promise<number> {
    return await this.productCards.count();
  }
```

```
  async goToCart() {
    await this.navigation.goToCart();
  }
}
```

POM with Methods Returning Page Objects

Method Chaining Pattern:

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

export class LoginPage {
  // ... locators ...

  async login(username: string, password: string): Promise<DashboardPage>
    await this.usernameInput.fill(username);
    await this.passwordInput.fill(password);
    await this.loginButton.click();

    // Return the next page object
    return new DashboardPage(this.page);
  }
}

// Usage in test
test('user flow', async ({ page }) => {
  const loginPage = new LoginPage(page);
  await loginPage.goto();

  // Method returns next page object
  const dashboardPage = await loginPage.login('user', 'pass');

  // Continue with dashboard page
  expect(await dashboardPage.isDashboardLoaded()).toBe(true);
  await dashboardPage.goToProfile();
});
```

POM Best Practices

```
// ✅ Good - Simple, focused methods
class LoginPage {
  async login(username: string, password: string) {
    await this.usernameInput.fill(username);
    await this.passwordInput.fill(password);
    await this.loginButton.click();
  }
}

// ❌ Bad - Too much logic in page object
class LoginPage {
  async loginAndVerifyDashboard(username: string, password: string) {
    await this.usernameInput.fill(username);
    await this.passwordInput.fill(password);
    await this.loginButton.click();
    await this.page.waitForURL('/dashboard');  // Don't verify in POM
    expect(await this.page.title()).toBe('Dashboard');  // Don't assert in
  }
}
```

2. Use Meaningful Method Names:

```
// ✅ Good
async login(username: string, password: string)
async clickAddToCartButton()
async searchForProduct(productName: string)

// ❌ Bad
async doLogin(u: string, p: string)
async click1()
async search(s: string)
```

3. Keep Assertions in Tests, Not Page Objects:

```
// ✅ Good - Getter method, assertion in test
class LoginPage {
  async getErrorMessage(): Promise<string> {
    return await this.errorMessage.textContent() || '';
  }
}

test('test', async ({ page }) => {
```

```
});

// ❌ Bad - Assertion in page object
class LoginPage {
  async verifyErrorMessage(expectedText: string) {
    const actual = await this.errorMessage.textContent();
    expect(actual).toContain(expectedText);  // Don't do this
  }
}
```

Playwright Fixtures (Built-in and Custom)

What are Fixtures?

Fixtures are objects or services that tests depend on. Playwright provides built-in fixtures and allows you to create custom ones.

Built-in Fixtures

Common Built-in Fixtures:

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

test('using built-in fixtures', async ({
  page,           // Page object
  context,        // Browser context
  browser,        // Browser instance
  browserName,    // Browser name string
  request         // API request context
}) => {
  console.log('Browser:', browserName);
  await page.goto('https://example.com');
});
```

Built-in Fixture List:

- `browser` - Browser instance
- `browserName` - Browser name ('chromium', 'firefox', 'webkit')
- `request` - APIRequestContext for API testing

Creating Custom Fixtures

Basic Custom Fixture:

```
 // fixtures.ts
import { test as base } from '@playwright/test';

// Define custom fixture type
type MyFixtures = {
  testUser: {
    username: string;
    password: string;
  };
};

// Extend base test with custom fixture
export const test = base.extend<MyFixtures>({
  testUser: async ({}, use) => {
    // Setup
    const user = {
      username: 'testuser@example.com',
      password: 'password123'
    };

    // Provide fixture to test
    await use(user);

    // Teardown (if needed)
    // Cleanup code here
  },
});

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

Using Custom Fixture:

```
  await page.goto('/login');
  await page.fill('#username', testUser.username);
  await page.fill('#password', testUser.password);
  await page.click('button[type="submit"]');

  await expect(page).toHaveURL('/dashboard');
});
```

Page Object Fixtures

Create fixture for Page Objects:

```
 // fixtures.ts
import { test as base } from '@playwright/test';
import { LoginPage } from './pages/LoginPage';
import { DashboardPage } from './pages/DashboardPage';
import { ProductsPage } from './pages/ProductsPage';

type PageFixtures = {
  loginPage: LoginPage;
  dashboardPage: DashboardPage;
  productsPage: ProductsPage;
};

export const test = base.extend<PageFixtures>({
  loginPage: async ({ page }, use) => {
    await use(new LoginPage(page));
  },

  dashboardPage: async ({ page }, use) => {
    await use(new DashboardPage(page));
  },

  productsPage: async ({ page }, use) => {
    await use(new ProductsPage(page));
  },
});

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

Using Page Object Fixtures:

```
  await loginPage.goto();
  await loginPage.login('user@example.com', 'password123');

  expect(await dashboardPage.isDashboardLoaded()).toBe(true);
});

test('user can view products', async ({ productsPage }) => {
  await productsPage.goto();
  const count = await productsPage.getProductCount();
  expect(count).toBeGreaterThan(0);
});
```

Authenticated User Fixture

Auto-login fixture:

```
 // fixtures.ts
import { test as base } from '@playwright/test';
import { LoginPage } from './pages/LoginPage';

type AuthFixtures = {
  authenticatedPage: Page;
};

export const test = base.extend<AuthFixtures>({
  authenticatedPage: async ({ page }, use) => {
    // Login before each test
    const loginPage = new LoginPage(page);
    await loginPage.goto();
    await loginPage.login('user@example.com', 'password123');

    // Wait for successful login
    await page.waitForURL('**/dashboard');

    // Provide logged-in page to test
    await use(page);

    // Logout after test (optional)
    // await page.click('#logout');
  },
});
```

Using Authenticated Fixture:

```
 import { test, expect } from './fixtures';

// This test starts with user already logged in
test('logged in user can view profile', async ({ authenticatedPage }) => {
  await authenticatedPage.goto('/profile');
  await expect(authenticatedPage.getByRole('heading', { name: 'My Profile'
});

test('logged in user can change settings', async ({ authenticatedPage }) =
  await authenticatedPage.goto('/settings');
  // User is already logged in, no need to login again
  await authenticatedPage.click('#save-settings');
});
```

API Context Fixture

Setup API request context:

```
 // fixtures.ts
import { test as base } from '@playwright/test';

type APIFixtures = {
  apiContext: APIRequestContext;
  apiBaseURL: string;
};

export const test = base.extend<APIFixtures>({
  apiBaseURL: async ({}, use) => {
    await use('https://api.example.com');
  },

  apiContext: async ({ playwright, apiBaseURL }, use) => {
    const context = await playwright.request.newContext({
      baseURL: apiBaseURL,
      extraHTTPHeaders: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
      },
    });
```

```
});

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

Test Data Fixture

Provide test data:

```
 // fixtures.ts
import { test as base } from '@playwright/test';

type TestData = {
  validUser: { username: string; password: string };
  invalidUser: { username: string; password: string };
  testProducts: Array<{ id: string; name: string; price: number }>;
};

export const test = base.extend<TestData>({
  validUser: async ({}, use) => {
    await use({
      username: 'valid@example.com',
      password: 'ValidPass123!'
    });
  },

  invalidUser: async ({}, use) => {
    await use({
      username: 'invalid@example.com',
      password: 'wrongpassword'
    });
  },

  testProducts: async ({}, use) => {
    const products = [
      { id: 'prod1', name: 'Laptop', price: 999 },
      { id: 'prod2', name: 'Mouse', price: 25 },
      { id: 'prod3', name: 'Keyboard', price: 75 },
    ];
    await use(products);
  },
});
```

Using Test Data Fixture:

```
import { test, expect } from './fixtures';

test('login with valid user', async ({ page, validUser }) => {
  await page.goto('/login');
  await page.fill('#username', validUser.username);
  await page.fill('#password', validUser.password);
  await page.click('button[type="submit"]');

  await expect(page).toHaveURL('/dashboard');
});

test('verify products', async ({ page, testProducts }) => {
  await page.goto('/products');

  for (const product of testProducts) {
    await expect(page.getByText(product.name)).toBeVisible();
  }
});
```

Combining Multiple Fixtures

Complex fixture setup:

```
// fixtures.ts
import { test as base } from '@playwright/test';
import { LoginPage } from './pages/LoginPage';
import { DashboardPage } from './pages/DashboardPage';

type AllFixtures = {
  loginPage: LoginPage;
  dashboardPage: DashboardPage;
  authenticatedContext: BrowserContext;
  testUser: { username: string; password: string };
};

export const test = base.extend<AllFixtures>({
  testUser: async ({}, use) => {
    await use({
      username: 'test@example.com',
      password: 'Test123!'
```

```
  loginPage: async ({ page }, use) => {
    await use(new LoginPage(page));
  },

  dashboardPage: async ({ page }, use) => {
    await use(new DashboardPage(page));
  },

  authenticatedContext: async ({ browser, testUser }, use) => {
    const context = await browser.newContext();
    const page = await context.newPage();

    // Login
    const loginPage = new LoginPage(page);
    await loginPage.goto();
    await loginPage.login(testUser.username, testUser.password);
    await page.waitForURL('**/dashboard');

    await use(context);
    await context.close();
  },
});

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

Data-Driven Testing

What is Data-Driven Testing?

Data-Driven Testing is a methodology where test data is separated from test logic, allowing the same test to run with different sets of data.

Test.describe.parallel with Multiple Data Sets

Basic data-driven test:

```
  { username: 'user1@example.com', password: 'Pass1234!' },
  { username: 'user2@example.com', password: 'Pass5678!' },
  { username: 'user3@example.com', password: 'Pass9012!' },
];

for (const cred of loginCredentials) {
  test(`login with ${cred.username}`, async ({ page }) => {
    await page.goto('/login');
    await page.fill('#username', cred.username);
    await page.fill('#password', cred.password);
    await page.click('button[type="submit"]');

    await expect(page).toHaveURL('/dashboard');
  });
}
```

Reading Data from JSON Files

Create test data file:

```
// data/users.json
{
  "validUsers": [
    {
      "username": "john@example.com",
      "password": "John1234!",
      "firstName": "John",
      "lastName": "Doe"
    },
    {
      "username": "jane@example.com",
      "password": "Jane5678!",
      "firstName": "Jane",
      "lastName": "Smith"
    }
  ],
  "invalidUsers": [
    {
      "username": "invalid@example.com",
      "password": "wrongpass",
      "expectedError": "Invalid credentials"
    }
```

Read JSON in tests:

```
 import { test, expect } from '@playwright/test';
import testData from './data/users.json';

test.describe('Data-driven login tests', () => {
  for (const user of testData.validUsers) {
    test(`valid login for ${user.username}`, async ({ page }) => {
      await page.goto('/login');
      await page.fill('#username', user.username);
      await page.fill('#password', user.password);
      await page.click('button[type="submit"]');

      await expect(page).toHaveURL('/dashboard');
      await expect(page.getByText(`${user.firstName} ${user.lastName}`)).t
    });
  }

  for (const user of testData.invalidUsers) {
    test(`invalid login for ${user.username}`, async ({ page }) => {
      await page.goto('/login');
      await page.fill('#username', user.username);
      await page.fill('#password', user.password);
      await page.click('button[type="submit"]');

      await expect(page.locator('.error-message')).toContainText(user.expe
    });
  }
});
```

Reading Data from CSV Files

Install CSV parser:

```
npm install -D csv-parse
```

Create CSV file:

```
2,Mouse,25,Electronics
3,Desk,300,Furniture
4,Chair,150,Furniture
```

Read CSV in tests:

```
 import { test, expect } from '@playwright/test';
import fs from 'fs';
import { parse } from 'csv-parse/sync';

interface Product {
  id: string;
  name: string;
  price: string;
  category: string;
}

const csvContent = fs.readFileSync('./data/products.csv', 'utf-8');
const products = parse(csvContent, {
  columns: true,
  skip_empty_lines: true,
}) as Product[];

test.describe('Product tests', () => {
  for (const product of products) {
    test(`verify product ${product.name}`, async ({ page }) => {
      await page.goto(`/products/${product.id}`);

      await expect(page.getByRole('heading', { name: product.name })).toBe
      await expect(page.getByText(`$${product.price}`)).toBeVisible();
      await expect(page.getByText(product.category)).toBeVisible();
    });
  }
});
```

Reading Data from Excel Files

Install xlsx library:

```
 npm install -D xlsx
```

```
 import { test, expect } from '@playwright/test';
import * as XLSX from 'xlsx';

interface TestData {
  username: string;
  password: string;
  expectedResult: string;
}

function readExcelFile(filePath: string, sheetName: string): TestData[] {
  const workbook = XLSX.readFile(filePath);
  const worksheet = workbook.Sheets[sheetName];
  const data = XLSX.utils.sheet_to_json<TestData>(worksheet);
  return data;
}

const testData = readExcelFile('./data/test-data.xlsx', 'LoginTests');

test.describe('Excel data-driven tests', () => {
  for (const data of testData) {
    test(`test with ${data.username}`, async ({ page }) => {
      await page.goto('/login');
      await page.fill('#username', data.username);
      await page.fill('#password', data.password);
      await page.click('button[type="submit"]');

      if (data.expectedResult === 'success') {
        await expect(page).toHaveURL('/dashboard');
      } else {
        await expect(page.locator('.error-message')).toBeVisible();
      }
    });
  }
});
```

Environment-Based Test Data

Different data per environment:

```
 // data/config.ts
interface EnvironmentConfig {
```

```
    admin: { username: string; password: string };
    user: { username: string; password: string };
  };
}

const configs: Record<string, EnvironmentConfig> = {
  development: {
    baseURL: 'http://localhost:3000',
    apiURL: 'http://localhost:3001/api',
    users: {
      admin: { username: 'admin@dev.com', password: 'DevAdmin123!' },
      user: { username: 'user@dev.com', password: 'DevUser123!' },
    },
  },
  staging: {
    baseURL: 'https://staging.example.com',
    apiURL: 'https://api-staging.example.com',
    users: {
      admin: { username: 'admin@staging.com', password: 'StagingAdmin123!'
      user: { username: 'user@staging.com', password: 'StagingUser123!' },
    },
  },
  production: {
    baseURL: 'https://example.com',
    apiURL: 'https://api.example.com',
    users: {
      admin: { username: 'admin@example.com', password: process.env.ADMIN_
      user: { username: 'user@example.com', password: process.env.USER_PAS
    },
  },
};

export function getConfig(): EnvironmentConfig {
  const env = process.env.TEST_ENV || 'development';
  return configs[env];
}
```

Using environment config:

```
 import { test, expect } from '@playwright/test';
import { getConfig } from './data/config';

const config = getConfig();
```

```
  await page.fill('#username', config.users.user.username);
  await page.fill('#password', config.users.user.password);
  await page.click('button[type="submit"]');

  await expect(page).toHaveURL(config.baseURL + '/dashboard');
});
```

Parameterized Tests with test.describe

Group tests with shared data:

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

const searchQueries = [
  { query: 'Playwright', expectedResults: 10 },
  { query: 'Automation', expectedResults: 15 },
  { query: 'Testing', expectedResults: 20 },
];

test.describe('Search functionality', () => {
  for (const data of searchQueries) {
    test.describe(`search for "${data.query}"`, () => {
      test('returns correct number of results', async ({ page }) => {
        await page.goto('/search');
        await page.fill('#search-input', data.query);
        await page.click('button[type="submit"]');

        const results = await page.locator('.search-result').count();
        expect(results).toBeGreaterThanOrEqual(data.expectedResults);
      });

      test('displays correct heading', async ({ page }) => {
        await page.goto('/search');
        await page.fill('#search-input', data.query);
        await page.click('button[type="submit"]');

        await expect(page.getByRole('heading', { name: `Results for "${dat
      });
    });
  }
});
```

Text Inputs

Basic text input:

```
// Type text
await page.fill('#username', 'testuser');

// Alternative: type with delay
await page.locator('#username').type('testuser', { delay: 100 });

// Clear and fill
await page.fill('#username', '');
await page.fill('#username', 'newuser');

// Press keys
await page.fill('#username', 'test');
await page.press('#username', 'Enter');
```

Textareas

```
// Multi-line text
await page.fill('textarea#comments', 'Line 1\nLine 2\nLine 3');

// Get textarea value
const text = await page.inputValue('textarea#comments');
console.log(text);
```

Checkboxes

```
// Check checkbox
await page.check('#agree-terms');

// Uncheck checkbox
await page.uncheck('#agree-terms');

// Toggle checkbox
const isChecked = await page.isChecked('#agree-terms');
```

```
// Check with locator
await page.getByRole('checkbox', { name: 'I agree' }).check();

// Verify checked state
await expect(page.locator('#agree-terms')).toBeChecked();
await expect(page.locator('#agree-terms')).not.toBeChecked();
```

Radio Buttons

```
 // Select radio button
await page.check('#gender-male');

// Select by label
await page.getByLabel('Male').check();

// Select by value
await page.check('input[name="gender"][value="male"]');

// Verify selection
await expect(page.locator('#gender-male')).toBeChecked();

// Get selected value
const selectedValue = await page.$eval(
  'input[name="gender"]:checked',
  (el) => (el as HTMLInputElement).value
);
console.log('Selected:', selectedValue);
```

Dropdowns (Select Elements)

```
 // Select by value
await page.selectOption('#country', 'US');

// Select by label
await page.selectOption('#country', { label: 'United States' });

// Select by index
await page.selectOption('#country', { index: 1 });
```

```
// Get selected value
const value = await page.inputValue('#country');
console.log('Selected:', value);

// Get all options
const options = await page.$$eval('#country option', (elements) =>
  elements.map((el) => ({
    value: (el as HTMLOptionElement).value,
    text: el.textContent,
  }))
);
console.log('Options:', options);
```

Custom Dropdowns (Non-select)

```
 // Open custom dropdown
await page.click('.dropdown-toggle');

// Wait for dropdown menu
await page.waitForSelector('.dropdown-menu', { state: 'visible' });

// Select option by text
await page.click('.dropdown-menu >> text="Option 1"');

// Or using getByText
await page.getByText('Option 1').click();

// Complete example
async function selectFromCustomDropdown(
  page: Page,
  dropdownSelector: string,
  optionText: string
) {
  await page.click(dropdownSelector);
  await page.waitForSelector('.dropdown-menu', { state: 'visible' });
  await page.click(`.dropdown-menu >> text="${optionText}"`);
}

await selectFromCustomDropdown(page, '#custom-dropdown', 'United States');
```

File Upload

```
// Set file input
await page.setInputFiles('#file-upload', 'path/to/file.pdf');

// Multiple files
await page.setInputFiles('#file-upload', [
  'path/to/file1.pdf',
  'path/to/file2.pdf',
]);

// From buffer
const buffer = Buffer.from('file content');
await page.setInputFiles('#file-upload', {
  name: 'test.txt',
  mimeType: 'text/plain',
  buffer: buffer,
});

// Clear file input
await page.setInputFiles('#file-upload', []);
```

Drag and drop file upload:

```
// Using file chooser
const [fileChooser] = await Promise.all([
  page.waitForEvent('filechooser'),
  page.click('#upload-button'),
]);
await fileChooser.setFiles('path/to/file.pdf');
```

Date Pickers

Native HTML5 date input:

```
// Set date (YYYY-MM-DD format)
await page.fill('input[type="date"]', '2024-12-31');

// Get date value
const date = await page.inputValue('input[type="date"]');
console.log('Selected date:', date);
```

```
async function selectDate(page: Page, day: number, month: string, year: nu
  // Open date picker
  await page.click('.date-picker-input');

  // Select year
  await page.selectOption('.year-selector', year.toString());

  // Select month
  await page.selectOption('.month-selector', month);

  // Select day
  await page.click(`.calendar-day >> text="${day}"`);
}

await selectDate(page, 15, 'December', 2024);
```

Sliders and Range Inputs

```
// Set slider value
await page.fill('input[type="range"]', '75');

// Or using evaluate
await page.$eval('input[type="range"]', (el, value) => {
  (el as HTMLInputElement).value = value;
  el.dispatchEvent(new Event('input', { bubbles: true }));
  el.dispatchEvent(new Event('change', { bubbles: true }));
}, '75');

// Get slider value
const value = await page.inputValue('input[type="range"]');
console.log('Slider value:', value);
```

Color Pickers

```
// Set color
await page.fill('input[type="color"]', '#ff0000');

// Get color
const color = await page.inputValue('input[type="color"]');
```

Autocomplete/Search Suggestions

```
 // Type in search box
await page.fill('#search-input', 'play');

// Wait for suggestions
await page.waitForSelector('.suggestions-list', { state: 'visible' });

// Click on suggestion
await page.click('.suggestion-item >> text="Playwright"');

// Or select by index
await page.click('.suggestion-item >> nth=0');
```

Browser Contexts and Page Management

Understanding Browser Contexts

Browser Context is an isolated browser session, similar to an incognito window. Each context has:

- Independent cookies
- Independent local storage
- Independent session storage
- Independent cache

```
Browser
  ├── Context 1 (Isolated)
  |    ├── Page 1
  |    └── Page 2
  └── Context 2 (Isolated)
       ├── Page 1
       └── Page 2
```

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

test('multiple contexts', async ({ browser }) => {
  // Create first context (user 1)
  const context1 = await browser.newContext();
  const page1 = await context1.newPage();

  // Create second context (user 2)
  const context2 = await browser.newContext();
  const page2 = await context2.newPage();

  // User 1 logs in
  await page1.goto('/login');
  await page1.fill('#username', 'user1@example.com');
  await page1.fill('#password', 'pass1');
  await page1.click('button[type="submit"]');

  // User 2 logs in
  await page2.goto('/login');
  await page2.fill('#username', 'user2@example.com');
  await page2.fill('#password', 'pass2');
  await page2.click('button[type="submit"]');

  // Both users are logged in independently
  await expect(page1.getByText('user1@example.com')).toBeVisible();
  await expect(page2.getByText('user2@example.com')).toBeVisible();

  // Cleanup
  await context1.close();
  await context2.close();
});
```

Context Options

Configure context:

```
test('context with options', async ({ browser }) => {
  const context = await browser.newContext({
    // Viewport size
    viewport: { width: 1920, height: 1080 },
```

```
    // Geolocation
    geolocation: { longitude: -122.4194, latitude: 37.7749 },
    permissions: ['geolocation'],

    // Timezone
    timezoneId: 'America/Los_Angeles',

    // Locale
    locale: 'en-US',

    // Color scheme
    colorScheme: 'dark',

    // Extra HTTP headers
    extraHTTPHeaders: {
      'X-Custom-Header': 'value',
    },

    // Accept downloads
    acceptDownloads: true,

    // Ignore HTTPS errors
    ignoreHTTPSErrors: true,
  });

  const page = await context.newPage();
  await page.goto('https://example.com');

  // Test with configured context

  await context.close();
});
```

Managing Multiple Pages

Open multiple tabs:

```
test('multiple tabs', async ({ context }) => {
  // Open first page
  const page1 = await context.newPage();
  await page1.goto('https://example.com/page1');
```

```
  // Open third page
  const page3 = await context.newPage();
  await page3.goto('https://example.com/page3');

  // Work with different pages
  await page1.click('button');
  await page2.fill('input', 'text');
  await page3.check('checkbox');

  // Close specific page
  await page2.close();

  // Get all pages
  const pages = context.pages();
  console.log(`Number of open pages: ${pages.length}`);
});
```

Handling Popups and New Windows

Wait for popup:

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

  // Wait for popup to open
  const [popup] = await Promise.all([
    page.waitForEvent('popup'),
    page.click('a[target="_blank"]'),  // Click link that opens popup
  ]);

  // Work with popup
  await popup.waitForLoadState();
  await expect(popup).toHaveTitle(/New Window/);
  await popup.click('button');

  // Close popup
  await popup.close();

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

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

  const popups: Page[] = [];

  // Listen for all popups
  page.on('popup', async (popup) => {
    await popup.waitForLoadState();
    popups.push(popup);
  });

  // Trigger actions that open popups
  await page.click('#open-popup-1');
  await page.click('#open-popup-2');

  // Wait for popups to be registered
  await page.waitForTimeout(1000);

  // Work with each popup
  for (const popup of popups) {
    console.log('Popup URL:', popup.url());
    await popup.close();
  }
});
```

Storage State (Session Persistence)

Save storage state:

```
test('save login session', async ({ page, context }) => {
  await page.goto('/login');
  await page.fill('#username', 'user@example.com');
  await page.fill('#password', 'password123');
  await page.click('button[type="submit"]');

  // Wait for login to complete
  await page.waitForURL('**/dashboard');

  // Save storage state (cookies, localStorage)
  await context.storageState({ path: 'auth.json' });
});
```

```
test('reuse login session', async ({ browser }) => {
  // Create context with saved state
  const context = await browser.newContext({
    storageState: 'auth.json',
  });

  const page = await context.newPage();

  // Already logged in!
  await page.goto('/dashboard');
  await expect(page.getByText('Welcome')).toBeVisible();

  await context.close();
});
```

In playwright.config.ts:

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

export default defineConfig({
  use: {
    // Load storage state for all tests
    storageState: 'auth.json',
  },
});
```

Mobile Emulation

Emulate mobile devices:

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

test('mobile emulation', async ({ browser }) => {
  // iPhone 12 emulation
  const iphone12 = devices['iPhone 12'];
  const context = await browser.newContext({
    ...iphone12,
  });

  const page = await context.newPage();
  await page.goto('https://example.com');
```

```
  await context.close();
});

// Or use project configuration
test.use({ ...devices['Pixel 5'] });

test('test on Pixel 5', async ({ page }) => {
  await page.goto('https://example.com');
  // Test runs with Pixel 5 specs
});
```

Available devices:

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

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

// Mobile
devices['iPhone 12']
devices['iPhone 13']
devices['iPhone 14']
devices['Pixel 5']
devices['Galaxy S9+']

// Tablet
devices['iPad Pro']
devices['iPad Mini']
```

Debugging, Screenshots, and Video Recording

Screenshots

Take screenshots:

```
  // Full page screenshot
  await page.screenshot({ path: 'screenshots/fullpage.png' });

  // Viewport screenshot
  await page.screenshot({
    path: 'screenshots/viewport.png',
    fullPage: false
  });

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

  // Screenshot with specific options
  await page.screenshot({
    path: 'screenshots/custom.png',
    fullPage: true,
    type: 'jpeg',
    quality: 80,
  });
});
```

Automatic screenshots on failure:

```
 // playwright.config.ts
export default defineConfig({
  use: {
    screenshot: 'only-on-failure',  // or 'on', 'off'
  },
});
```

Screenshot in test hooks:

```
 test.afterEach(async ({ page }, testInfo) => {
  if (testInfo.status !== testInfo.expectedStatus) {
    // Test failed, take screenshot
    await page.screenshot({
      path: `screenshots/${testInfo.title}-failure.png`
    });
  }
```

Video Recording

Enable video recording:

```
// playwright.config.ts
export default defineConfig({
  use: {
    video: 'on',  // or 'off', 'retain-on-failure', 'on-first-retry'
    videoSize: { width: 1280, height: 720 },
  },
});
```

Video in specific test:

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

  // Test actions...

  // Get video path after test
  const path = await page.video()?.path();
  console.log('Video saved to:', path);
});
```

Trace Files

Enable tracing:

```
// playwright.config.ts
export default defineConfig({
  use: {
    trace: 'on-first-retry',  // or 'on', 'off', 'retain-on-failure'
  },
});
```

Manual trace control:

```
    screenshots: true,
    snapshots: true,
    sources: true
  });

  // Test actions
  await page.goto('https://example.com');
  await page.click('button');

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

View trace:

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

Console and Network Logs

Capture console logs:

```
test('console logs', async ({ page }) => {
  const logs: string[] = [];

  page.on('console', (msg) => {
    logs.push(`${msg.type()}: ${msg.text()}`);
  });

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

  // Print collected logs
  console.log('Browser console logs:');
  logs.forEach(log => console.log(log));
});
```

Network activity:

```
  page.on('request', (request) => {
    requests.push(request.url());
  });

  page.on('response', (response) => {
    responses.push({
      url: response.url(),
      status: response.status(),
    });
  });

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

  console.log('Total requests:', requests.length);
  console.log('Failed responses:',
    responses.filter(r => r.status >= 400).length
  );
});
```

HAR (HTTP Archive) Files

Record network traffic:

```
test('record HAR', async ({ browser }) => {
  const context = await browser.newContext({
    recordHar: { path: 'network.har' },
  });

  const page = await context.newPage();
  await page.goto('https://example.com');

  // Perform actions...

  await context.close();
  // HAR file saved to network.har
});
```

Analyze HAR file:

Debugging Slow Tests

Add timing information:

```
test('performance tracking', async ({ page }) => {
  console.time('Page Load');
  await page.goto('https://example.com');
  console.timeEnd('Page Load');

  console.time('Form Fill');
  await page.fill('#username', 'user');
  await page.fill('#password', 'pass');
  console.timeEnd('Form Fill');

  console.time('Form Submit');
  await page.click('button[type="submit"]');
  await page.waitForURL('**/dashboard');
  console.timeEnd('Form Submit');
});
```

Measure API response time:

```
test('API timing', async ({ page }) => {
  const startTime = Date.now();

  const [response] = await Promise.all([
    page.waitForResponse('**/api/users'),
    page.goto('/users'),
  ]);

  const endTime = Date.now();
  const duration = endTime - startTime;

  console.log(`API response time: ${duration}ms`);
  console.log(`Status: ${response.status()}`);
});
```

Exercise 1: Complete POM Implementation

Task: Build a complete Page Object Model for an e-commerce site

Create the following page objects:

1. `BasePage.ts` - Common functionality
2. `LoginPage.ts` - Login functionality
3. `ProductsPage.ts` - Product listing
4. `ProductDetailPage.ts` - Single product view
5. `CartPage.ts` - Shopping cart
6. `CheckoutPage.ts` - Checkout process

```
 // Your implementation here
// Include all necessary methods and locators
// Use proper TypeScript types
// Follow POM best practices
```

Exercise 2: Custom Fixtures

Task: Create a comprehensive fixture file

Create fixtures for:

1. Authenticated user (already logged in)
2. Test data (users, products, orders)
3. Page objects
4. API context

```
 // fixtures.ts
import { test as base } from '@playwright/test';

// Your fixture implementations
```

1. Create `data/test-users.json` with 5 users
2. Create `data/products.csv` with 10 products
3. Write tests that use this data
4. Handle both valid and invalid scenarios

```
// Your data-driven test implementation
```

Exercise 4: Multi-Page Workflow

Task: Test complete user journey

Implement a test for:

1. User registers a new account
2. Logs in
3. Browses products
4. Adds items to cart
5. Completes checkout
6. Verifies order confirmation

Use POM and proper assertions throughout.

Exercise 5: Complex Form Handling

Task: Automate complex form

Create a test that fills out a registration form with:

- Text inputs (name, email, phone)
- Password with validation
- Date of birth (date picker)
- Gender (radio buttons)
- Interests (multiple checkboxes)

- Terms and conditions (checkbox)

Verify all validations work correctly.

Exercise 6: Browser Context Management

Task: Multi-user scenario

Test a chat application where:

1. Create 2 browser contexts (2 users)
2. User 1 sends a message
3. Verify User 2 receives the message
4. User 2 replies
5. Verify User 1 receives the reply

Exercise 7: Debug and Fix

Task: Debug failing tests

Given the following failing test, use debugging tools to:

1. Identify the issues
2. Fix the problems
3. Add proper waits
4. Implement error handling

```
test('buggy test', async ({ page }) => {
  await page.goto('https://example.com');
  await page.click('#submit-button');
  const text = await page.textContent('.result');
  expect(text).toBe('Success');
});

// Use Playwright Inspector, trace viewer, and screenshots
// to debug and fix this test
```

In Day 4, you learned:

✅ Page Object Model (POM)

- Design pattern for maintainable tests
- Base page and inheritance
- Component objects
- Method chaining
- POM best practices

✅ Playwright Fixtures

- Built-in fixtures (page, context, browser)
- Creating custom fixtures
- Page object fixtures
- Authenticated user fixtures
- Test data fixtures
- API context fixtures

✅ Data-Driven Testing

- Parameterized tests
- Reading from JSON files
- Reading from CSV files
- Reading from Excel files
- Environment-based data
- Data-driven test organization

✅ Form Handling

- Text inputs and textareas
- Checkboxes and radio buttons
- Dropdowns (select and custom)
- File uploads

- Autocomplete fields

✅ Browser Contexts

- Multiple isolated contexts
- Context configuration
- Multiple pages/tabs
- Popup handling
- Storage state persistence
- Mobile emulation

✅ Debugging & Recording

- Screenshots (full page, viewport, element)
- Video recording
- Trace files
- Console and network logs
- HAR files
- Performance tracking

Key Takeaways

- **Use POM** for maintainable test code
- **Leverage fixtures** for test setup and data
- **Separate test data** from test logic
- **Browser contexts** enable isolated testing
- **Always debug** with proper tools

Best Practices Learned

1. Keep page objects simple and focused
2. Assertions belong in tests, not page objects
3. Use custom fixtures for common setups

6. Enable screenshots and videos for CI/CD
7. Use trace viewer for debugging failures

Next Steps

- Practice building POM for real applications
- Create reusable fixtures library
- Implement data-driven tests for your project
- Experiment with browser contexts
- Master debugging tools

End of Day 4 Documentation

← Playwright Basics

Playwright Advanced Scenarios →