

# Playwright Automation with TypeScript

The Enterprise Edition

PART I & II: Complete Detailed Guide

*Chapters 1-8 with Advanced Patterns*

## Table of Contents

### PART I: FOUNDATIONS (Chapters 1-4)

- Chapter 1: Introduction to Playwright
- Chapter 2: Setting Up Your Environment
- Chapter 3: Writing Your First Test
- Chapter 4: Understanding Playwright Architecture

### PART II: CORE CONCEPTS (Chapters 5-8)

- Chapter 5: Locators and Selectors
- Chapter 6: Actions and Interactions
- Chapter 7: Assertions and Validations
- Chapter 8: Page Object Model & Component Object Model

## PART I: FOUNDATIONS

## CHAPTER 1: INTRODUCTION TO PLAYWRIGHT

### 1.1 What is Playwright & The Testing Trophy Strategy

Playwright is a modern, open-source automation framework developed by Microsoft that enables reliable end-to-end testing for web applications. Released in 2020, it was created by the same team that built Puppeteer at Google.

#### The Testing Trophy vs Testing Pyramid

Traditional software testing followed the "Testing Pyramid" approach: 70% unit tests, 20% integration tests, and 10% E2E tests. This was designed when E2E tests were slow and unreliable.

With modern tools like Playwright, the Testing Trophy represents a better approach:

- 10% Static Analysis (TypeScript, ESLint)
- 20% Unit Tests (Pure business logic)
- 50% Integration Tests (Components + API together)
- 20% E2E Tests (Critical user journeys)



**ENTERPRISE:** The Trophy approach works because Playwright makes E2E tests as fast as unit tests but with 10x the confidence. A single E2E test validates frontend, backend, database, and APIs together.

#### Why the Trophy Strategy Wins

Integration and E2E tests provide more value because they:

- Test how components actually work together
- Catch integration bugs that unit tests miss
- Verify real user workflows
- Test database interactions
- Validate API contracts

#### Banking Login Example - Trophy in Action

```
test('complete banking login flow', async ({ page }) => {
```

```
await page.goto('/login');
await page.getByLabel('Account Number').fill('123456789');
await page.getByLabel('PIN').fill('1234');
await page.getByRole('button', { name: 'Login' }).click();

// This ONE test validates:
// ✓ UI rendering ✓ Form validation ✓ Auth API
// ✓ Database lookup ✓ Session creation ✓ Routing

await expect(page).toHaveURL('/dashboard');
await expect(page.getText('Account Balance')).toBeVisible();
});
```

 **BEST PRACTICE:** Focus 70% of your testing effort on integration and E2E tests. They provide the highest return on investment for catching real bugs.

## 1.2 Why Playwright for Enterprise Applications

### Cross-Browser Support

Playwright natively supports ALL major browser engines:

- Chromium - Chrome, Edge, Opera, Brave
- Firefox - Mozilla Firefox
- WebKit - Safari (desktop and iOS)

 **NOTE:** Playwright bundles specific browser versions, ensuring consistent behavior across all environments - no "works on my machine" issues.

### Auto-Wait Mechanism

Playwright automatically waits for elements to be ready before acting:

- Attached to DOM
- Visible on screen
- Stable (not animating)
- Receives events (not covered)

- Enabled (not disabled)

```
// Playwright - auto-waits built in
await page.getByRole('button', { name: 'Submit' }).click();
// vs Selenium - manual waiting required
WebDriverWait wait = new WebDriverWait(driver, 10);
wait.until(ExpectedConditions.elementToBeClickable(...));
```

 **BEST PRACTICE:** Auto-waiting eliminates 80% of flaky test issues.  
Trust it - resist adding sleep() or arbitrary timeouts.

## 1.3 Playwright vs Selenium, Cypress, Puppeteer

### vs Selenium

- Speed: Playwright 20-50% faster
- Setup: Playwright auto-downloads browsers
- API: Playwright modern async/await
- Reliability: Playwright built-in auto-wait

### vs Cypress

- Browsers: Playwright supports Safari/WebKit
- Multi-tab: Playwright full support
- Mobile: Playwright native mobile browsers
- Languages: Playwright supports Python, .NET, Java

### vs Puppeteer

- Browsers: Playwright adds Firefox + WebKit
- Testing: Playwright built for testing
- Selectors: Playwright user-facing locators

 **ENTERPRISE:** For enterprise applications requiring cross-browser support, mobile testing, and maximum reliability, Playwright is the clear choice.

## CHAPTER 2: SETTING UP YOUR ENVIRONMENT

### 2.1 Prerequisites

- Node.js 16+ (LTS version recommended)
- Visual Studio Code (recommended)
- Basic TypeScript knowledge
- Command line familiarity

### 2.2 Installation Steps

#### Step 1: Create Project

```
mkdir playwright-automation
cd playwright-automation
npm init -y
```

#### Step 2: Install Playwright

```
npm init playwright@latest
```

Answer the prompts:

- TypeScript? → Yes
- Test directory? → tests
- GitHub Actions? → Yes
- Install browsers? → Yes

#### Step 3: Project Structure

```
playwright-automation/
├── node_modules/
├── tests/
│   └── example.spec.ts
└── playwright.config.ts
└── package.json
└── package-lock.json
```

## 2.3 Configuration

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

export default defineConfig({
  testDir: './tests',
  timeout: 30 * 1000,
  fullyParallel: true,
  retries: process.env.CI ? 2 : 0,
  reporter: 'html',
  use: [
    baseURL: 'http://localhost:3000',
    trace: 'on-first-retry',
    screenshot: 'only-on-failure',
  ],
  projects: [
    { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    { name: 'firefox', use: { ...devices['Desktop Firefox'] } },
    { name: 'webkit', use: { ...devices['Desktop Safari'] } },
  ],
});
```

## 2.4 Verify Installation

```
npx playwright test
```

 **BEST PRACTICE:** Use VS Code Playwright extension for debugging, test runner UI, and visual locator picker.

## CHAPTER 3: WRITING YOUR FIRST TEST

### 3.1 Basic Test Structure

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

test('basic navigation', async ({ page }) => {
  await page.goto('https://playwright.dev');
  await expect(page).toHaveTitle(/Playwright/);
  await page.getByRole('link', { name: 'Get started' }).click();
  await expect(page).toHaveURL(/.*intro/);
});
```

### 3.2 Finding Elements

#### Role-Based (Recommended)

```
await page.getByRole('button', { name: 'Submit' })
await page.getByRole('link', { name: 'Contact' })
await page.getByRole('textbox', { name: 'Email' })
```

#### Text-Based

```
await page.getByText('Sign up')
await page.getByText(/sign up/i) // regex
```

#### Label-Based

```
await page.getByLabel('Email address')
await page.getByLabel('Password')
```

#### Test ID

```
await page.getByTestId('submit-btn')
```

 **BEST PRACTICE:** Prefer `getByRole()` and `getByLabel()` - they're most resilient and promote accessibility.

### 3.3 Common Actions

```
// Click
await page.getByRole('button').click();

// Type text
await page.getLabel('Username').fill('john');

// Check/uncheck
await page.getRole('checkbox').check();

// Select dropdown
await page.getLabel('Country').selectOption('USA');
```

### 3.4 Assertions

```
// Page
await expect(page).toHaveTitle('Dashboard');
await expect(page).toHaveURL(/dashboard/);

// Element
await expect(page.getText('Success')).toBeVisible();
await expect(page.getRole('heading')).toHaveText('Welcome');
await
expect(page.getLabel('Email')).toHaveValue('test@example.com');
```

### 3.5 Real Login Example

```
test.describe('Login', () => {
  test.beforeEach(async ({ page }) => {
    await page.goto('/login');
  });

  test('successful login', async ({ page }) => {
    await page.getLabel('Email').fill('user@example.com');
    await page.getLabel('Password').fill('pass123');
    await page.getRole('button', { name: 'Log in' }).click();
  });
});
```

Author-Vaibhav Sahu  
**PLAYWRIGHT HANDBOOK**

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

## CHAPTER 4: UNDERSTANDING PLAYWRIGHT ARCHITECTURE

### 4.1 Three-Tier Architecture

- Your test code runs in Node.js
- Playwright controls browsers via DevTools Protocol
- Browsers run in separate processes

### 4.2 Browser, Context, and Page

#### Browser

```
const browser = await chromium.launch({ headless: false });
```

#### Context (Isolated Session)

```
const context = await browser.newContext({  
    viewport: { width: 1280, height: 720 },  
    locale: 'en-US'  
});
```

#### Page (Single Tab)

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

 **ENTERPRISE:** Browser contexts allow testing multiple users simultaneously in one browser - perfect for enterprise applications.

### 4.3 Auto-Waiting Details

Before every action, Playwright waits for:

- Element attached to DOM
- Element visible
- Element stable (not animating)

- Element receives events
- Element enabled

## 4.4 Network Interception

```
await page.route('**/api/users', route => {
  route.fulfill({
    status: 200,
    body: JSON.stringify([{ id: 1, name: 'Mock' }])
  });
}) ;
```

## PART II: CORE CONCEPTS

## CHAPTER 5: LOCATORS AND SELECTORS

### 5.1 Locator Strategy Priority

#### 1. Role-based (BEST)

```
page.getByRole('button', { name: 'Submit' })
```

#### 2. Label-based (Great for forms)

```
page.getLabel('Email address')
```

#### 3. Text-based

```
page.getText('Welcome')
```

#### 4. Test ID (Stable)

```
page.getByTestId('submit-btn')
```

### 5.2 Chaining Locators

```
await page
  .locator('.user-section')
    .getByRole('button', { name: 'Edit' })
      .click();
```

### 5.3 Filtering

```
await page
  .getByRole('listitem')
    .filter({ hasText: 'Product 1' })
      .click();
```

### 5.4 Multiple Elements

```
const count = await page.getByRole('listitem').count();
await page.getByRole('button').first().click();
await page.getByRole('button').nth(2).click();
```

 **BEST PRACTICE:** Use role-based locators - they reflect how users interact with your app and promote accessibility.

## CHAPTER 6: ACTIONS AND INTERACTIONS

### 6.1 Clicking

```
// Simple click
await page.getByRole('button').click();

// Double click
await page.getText('File').dblclick();

// Right click
await page.getText('Item').click({ button: 'right' });

// Click with modifiers
await page.getText('Link').click({ modifiers: ['Control'] });
```

### 6.2 Typing

```
// Fill (clears first)
await page.getLabel('Username').fill('john.doe');

// Type (doesn't clear)
await page.getLabel('Search').type('playwright');

// Press keys
await page.keyboard.press('Enter');
await page.keyboard.press('Control+A');
```

### 6.3 Form Interactions

```
// Checkbox
await page.getByRole('checkbox', { name: 'Terms' }).check();
await page.getByRole('checkbox').uncheck();

// Radio button
await page.getByRole('radio', { name: 'Option 1' }).check();

// Dropdown
```

```
await page.getLabel('Country').selectOption('USA');
await page.getLabel('Country').selectOption({ label: 'United States' });
```

## 6.4 File Uploads

```
await page.getLabel('Upload').setInputFiles('file.pdf');
await page.getLabel('Upload').setInputFiles(['file1.jpg',
'file2.jpg']);
```

## 6.5 Mouse Actions

```
// Hover
await page.getRole('link', { name: 'Products' }).hover();

// Drag and drop
await page.getText('Drag').dragTo(page.getText('Drop here'));
```

## 6.6 Complete Form Example

```
await page.getLabel('First Name').fill('John');
await page.getLabel('Email').fill('john@example.com');
await page.getLabel('Country').selectOption('USA');
await page.getLabel('I agree').check();
await page.getRole('button', { name: 'Register' }).click();
```

## CHAPTER 7: ASSERTIONS AND VALIDATIONS

### 7.1 Auto-Retrying Assertions

Playwright assertions automatically retry until timeout:

```
// Will retry for 5 seconds by default
await expect(page.getText('Success')).toBeVisible();
```

### 7.2 Page Assertions

```
await expect(page).toHaveURL('https://example.com/dashboard');
await expect(page).toHaveURL('/dashboard/');
await expect(page).toHaveTitle('My Dashboard');
await expect(page).toHaveTitle('/Dashboard/');
```

### 7.3 Element Visibility

```
await expect(page.getText('Welcome')).toBeVisible();
await expect(page.getText('Loading')).toBeHidden();
await expect(page.getText('Error')).not.toBeVisible();
```

### 7.4 Element State

```
await expect(page.getRole('button')).toBeEnabled();
await expect(page.getRole('button')).toBeDisabled();
await expect(page.getLabel('Terms')).toBeChecked();
await expect(page.getLabel('Terms')).not.toBeChecked();
```

### 7.5 Text Content

```
// Exact match
await expect(page.getRole('heading')).toHaveText('Welcome');

// Contains
await
expect(page.getRole('paragraph')).toContainText('Playwright');
```

```
// Array match
await expect(page.getByRole('listitem')).toHaveText(['Item 1', 'Item 2']);
```

## 7.6 Input Values

```
await
expect(page.getLabel('Email')).toHaveValue('user@example.com');
await expect(page.getLabel('Email')).toHaveValue(/.*@example.com/);
```

## 7.7 Attributes

```
await expect(page.getByRole('button')).toHaveAttribute('type',
'submit');
await expect(page.getByRole('button')).toHaveClass('btn-primary');
await expect(page.getByRole('button')).toHaveClass(/btn-/);
```

## 7.8 Count

```
await expect(page.getByRole('listitem')).toHaveLength(5);
await expect(page.getByRole('listitem')).toHaveLength(0);
```

## 7.9 Screenshots

```
await expect(page).toHaveScreenshot('homepage.png');
await
expect(page.getByRole('button')).toHaveScreenshot('button.png');
```

## 7.10 Custom Timeout

```
await expect(page.getText('Slow')).toBeVisible({ timeout: 30000 });
```

 **BEST PRACTICE:** Use soft assertions for non-critical checks that shouldn't stop test execution: `await expect.soft(element).toBeVisible();`

## CHAPTER 8: PAGE OBJECT MODEL & COMPONENT OBJECT MODEL

### 8.1 Traditional Page Object Pattern

#### Basic Page Object

```
// pages/LoginPage.ts

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

export class LoginPage {
    readonly page: Page;
    readonly emailInput: Locator;
    readonly passwordInput: Locator;
    readonly submitButton: Locator;

    constructor(page: Page) {
        this.page = page;
        this.emailInput = page.getByLabel('Email');
        this.passwordInput = page.getByLabel('Password');
        this.submitButton = page.getByRole('button', { name: 'Log in' });
    }

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

    async login(email: string, password: string) {
        await this.emailInput.fill(email);
        await this.passwordInput.fill(password);
        await this.submitButton.click();
    }
}
```

#### Using Page Objects

```
test('login test', async ({ page }) => {
    const loginPage = new LoginPage(page);
```

```
await loginPage.goto();
await loginPage.login('user@example.com', 'pass123');
await expect(page).toHaveURL('/dashboard');
});
```

## 8.2 Component Object Model (COM) - Enterprise Approach

 **ENTERPRISE:** Modern apps are component-based. If your Header appears on 15 pages and changes, you have to update 15 Page Objects. Component Objects solve this.

### Component Object Example

```
// components/HeaderComponent.ts
export class HeaderComponent {
    readonly page: Page;
    readonly logo: Locator;
    readonly searchInput: Locator;
    readonly userMenu: Locator;
    readonly logoutBtn: Locator;

    constructor(page: Page) {
        this.page = page;
        this.logo = page.getByTestId('logo');
        this.searchInput = page.getByTestId('search');
        this.userMenu = page.getByTestId('user-menu');
        this.logoutBtn = page.getByTestId('logout');
    }

    async clickLogo() {
        await this.logo.click();
    }

    async search(query: string) {
        await this.searchInput.fill(query);
        await this.page.keyboard.press('Enter');
    }

    async logout() {
```

```
        await this.userMenu.click();
        await this.logoutBtn.click();
    }
}
```

## Using Component Objects in Page Objects

```
// pages/DashboardPage.ts
import { HeaderComponent } from '../components/HeaderComponent';

export class DashboardPage {
    header: HeaderComponent;

    constructor(private page: Page) {
        this.header = new HeaderComponent(page);
    }

    async navigateToTransactions() {
        await this.header.search('transactions');
    }
}
```

## 8.3 Reusable Component Architecture

### Base Page Pattern

```
// pages/BasePage.ts
export class BasePage {
    readonly page: Page;
    readonly header: HeaderComponent;
    readonly footer: FooterComponent;

    constructor(page: Page) {
        this.page = page;
        this.header = new HeaderComponent(page);
        this.footer = new FooterComponent(page);
    }
}
```

## Extending Base Page

```
export class ProductPage extends BasePage {
    readonly addToCartBtn: Locator;

    constructor(page: Page) {
        super(page);
        this.addToCartBtn = page.getByRole('button', { name: 'Add to Cart' });
    }

    async addToCart() {
        await this.addToCartBtn.click();
    }
}
```

 **ENTERPRISE:** Component Object Model reduces maintenance by 80%. Update HeaderComponent once, and all 15 pages automatically get the update.

 **BEST PRACTICE:** Use COM for Header, Footer, Navigation, Modals, and any UI component that appears across multiple pages.

## 8.4 Best Practices

- Use descriptive class and method names
- Keep page objects focused - one page, one class
- Use `readonly` for locators
- Return page objects for method chaining
- Extract common components (header, footer)
- Use base page for shared functionality

# SUMMARY

## What You've Learned

### PART I - Foundations:

- Testing Trophy Strategy for modern applications
- Why Playwright for enterprise applications
- Environment setup and configuration
- Writing your first tests
- Playwright architecture and auto-waiting

### PART II - Core Concepts:

- Modern locator strategies (role-based)
- Actions and interactions
- Auto-retrying assertions
- Page Object Model
- Component Object Model for enterprise apps



**ENTERPRISE:** You now have the foundation to build enterprise-grade test automation. Parts III & IV will cover advanced testing, performance, CI/CD, and AI-powered patterns.

### Next Steps:

- Practice writing tests with role-based locators
- Implement Component Objects in your project
- Set up CI/CD with GitHub Actions
- Move to Part III for advanced patterns

# Playwright Automation with TypeScript

## The Enterprise Edition

PARTS III, IV, V: Advanced & Enterprise Patterns

*Chapters 9-17 • AI-Driven Testing • Banking-Ready*

## Table of Contents

### PART III: ADVANCED TESTING (Chapters 9-12)

- Chapter 9: Handling Complex Scenarios
- Chapter 10: API Testing - REST, GraphQL, Hybrid UI+API
- Chapter 11: Visual Testing & Accessibility
- Chapter 12: Performance Testing

### PART IV: ENTERPRISE PATTERNS (Chapters 13-16)

- Chapter 13: Test Organization and Structure
- Chapter 14: CI/CD Integration & Test Sharding
- Chapter 15: Debugging and Troubleshooting
- Chapter 16: Advanced Patterns and Techniques

### PART V: THE FUTURE (Chapter 17)

- Chapter 17: AI-Driven Testing

---

## PART III: ADVANCED TESTING

---

## CHAPTER 9: HANDLING COMPLEX SCENARIOS

### 9.1 Authentication State Management

#### Saving Authentication State

Save login state once, reuse across all tests - massive time savings.

```
// auth.setup.ts - Run once before all tests
import { test as setup } from '@playwright/test';

const authFile = 'playwright/.auth/user.json';

setup('authenticate', async ({ page }) => {
    await page.goto('/login');
    await page.getByLabel('Email').fill('user@example.com');
    await page.getByLabel('Password').fill('SecurePass123');
    await page.getByRole('button', { name: 'Log in' }).click();

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

    // Save authentication state
    await page.context().storageState({ path: authFile });
});

});
```

#### Reusing Authentication State

```
// playwright.config.ts
export default defineConfig({
  projects: [
    { name: 'setup', testMatch: './.*\\.setup\\.ts/' },
    {
      name: 'chromium',
      use: {
        ...devices['Desktop Chrome'],
        storageState: 'playwright/.auth/user.json'
      }
    }
  ]
});
```

```
    },
    dependencies: ['setup']
}
]
});
```

 **ENTERPRISE:** This pattern saves 5-10 seconds per test. For 1000 tests, that's 1.5-3 hours saved!

## 9.2 File Downloads

```
test('download report', async ({ page }) => {
  // Wait for download event
  const [download] = await Promise.all([
    page.waitForEvent('download'),
    page.getText('Download Report').click()
  ]);

  // Get download path
  const path = await download.path();
  console.log('Downloaded to:', path);

  // Save with custom name
  await download.saveAs('/downloads/' + download.suggestedFilename());

  // Verify file downloaded
  expect(download.suggestedFilename()).toContain('report.pdf');
});
```

## 9.3 File Uploads

```
// Single file
await page.getLabel('Upload document').setInputFiles('contract.pdf');
```

```
// Multiple files
await page.getLabel('Upload photos').setInputFiles([
  'photo1.jpg',
  'photo2.jpg',
  'photo3.jpg'
]);

// Remove files
await page.getLabel('Upload').setInputFiles([]);
```

## 9.4 Geolocation Testing

```
test('location-based features', async ({ context, page }) => {
  // Grant geolocation permission
  await context.grantPermissions(['geolocation']);

  // Set location to New York
  await context.setGeolocation({
    latitude: 40.7128,
    longitude: -74.0060
  });

  await page.goto('/stores');

  // Verify location-based content
  await expect(page.getText('Stores near you')).toBeVisible();
  await expect(page.getText('New York, NY')).toBeVisible();
});
```

## 9.5 Network Mocking

```
test('mock API response', async ({ page }) => {
  // Intercept API call
  await page.route('**/api/users', route => {
    route.fulfill({
      status: 200,
      contentType: 'application/json',
    });
  });
});
```

```
body: JSON.stringify([
  { id: 1, name: 'Alice', role: 'Admin' },
  { id: 2, name: 'Bob', role: 'User' }
])
});

});

await page.goto('/users');

// Verify mocked data appears
await expect(page.getText('Alice')).toBeVisible();
await expect(page.getText('Bob')).toBeVisible();
});
```

## 9.6 Multiple Browser Contexts

```
test('multi-user scenario', async ({ browser }) => {
  // User 1: Admin
  const adminContext = await browser.newContext({
    storageState: 'admin-auth.json'
  });
  const adminPage = await adminContext.newPage();

  // User 2: Regular user
  const userContext = await browser.newContext({
    storageState: 'user-auth.json'
  });
  const userPage = await userContext.newPage();

  // Admin creates a resource
  await adminPage.goto('/admin/resources');
  await adminPage.getRole('button', { name: 'Create' }).click();

  // User sees the new resource
  await userPage.goto('/resources');
  await expect(userPage.getText('New Resource')).toBeVisible();

  // Cleanup
  await adminContext.close();
```

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

 **ENTERPRISE:** Multiple contexts enable testing collaborative features, permissions, and multi-user workflows - essential for enterprise applications.

## 9.7 Handling Popups and New Tabs

```
test('popup window', async ({ context, page }) => {
  // Wait for popup
  const [popup] = await Promise.all([
    context.waitForEvent('page'),
    page.getByText('Open Terms').click()
  ]);

  // Wait for popup to load
  await popup.waitForLoadState();

  // Interact with popup
  await expect(popup.getByRole('heading')).toHaveText('Terms of Service');
  await popup.getByRole('button', { name: 'Accept' }).click();

  // Popup closes, continue on main page
  await expect(page.getByText('Terms Accepted')).toBeVisible();
});
```

## 9.8 Working with iFrames

```
test('iframe interaction', async ({ page }) => {
  await page.goto('/payment');

  // Get frame locator
  const paymentFrame = page.frameLocator('iframe[name="payment"]');
```

```
// Interact with elements inside iframe
await paymentFrame.getByLabel('Card
Number').fill('4242424242424242');
await paymentFrame.getByLabel('Expiry').fill('12/25');
await paymentFrame.getByLabel('CVC').fill('123');
await paymentFrame.getByRole('button', { name: 'Pay' }).click();

// Back to main page
await expect(page.getText('Payment Successful')).toBeVisible();
});
```

 **BANKING:** Payment gateways often use iframes. This pattern is essential for banking and e-commerce testing.

## CHAPTER 10: API TESTING WITH PLAYWRIGHT

### 10.1 REST API Testing

#### Basic GET Request

```
test('GET users', async ({ request }) => {
  const response = await
request.get('https://api.example.com/users');

  expect(response.status()).toBe(200);

  const users = await response.json();
  expect(users).toBeInstanceOf(Array);
  expect(users[0]).toHaveProperty('id');
  expect(users[0]).toHaveProperty('email');
});
```

#### POST Request

```
test('POST create user', async ({ request }) => {
  const response = await
request.post('https://api.example.com/users', {
    data: {
      name: 'John Doe',
      email: 'john@example.com',
      role: 'user'
    }
  });

  expect(response.status()).toBe(201);

  const user = await response.json();
  expect(user.name).toBe('John Doe');
  expect(user).toHaveProperty('id');
});
```

#### Authentication Headers

```
test('authenticated request', async ({ request }) => {
  const token = 'Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...';

  const response = await request.get('/api/admin/users', {
    headers: {
      'Authorization': token,
      'Content-Type': 'application/json'
    }
  });

  expect(response.status()).toBe(200);
});
```

## 10.2 GraphQL API Testing

```
test('GraphQL query', async ({ request }) => {
  const query = `
    query GetUser($id: ID!) {
      user(id: $id) {
        id
        name
        email
        posts {
          title
        }
      }
    }
  `;

  const response = await request.post('/graphql', {
    data: {
      query,
      variables: { id: '123' }
    }
  });

  const result = await response.json();
  expect(result.data.user.name).toBe('Alice');
  expect(result.data.user.posts).toHaveLength(5);
```

```
|});
```

## 10.3 Hybrid UI + API Testing (The 80/20 Rule)

 **ENTERPRISE:** The 80/20 Rule: Use APIs for 80% of test setup (creating users, adding data, setting state). Use UI for the critical 20% you actually want to test.

### Why Hybrid Testing?

- Speed: API calls take 200ms vs UI login taking 5 seconds
- Reliability: APIs don't have animation delays or loading spinners
- Focus: Test the UI behavior, not the setup
- Efficiency: Run 1000 tests in minutes instead of hours

### Banking Purchase Flow Example

```
test('purchase with API setup', async ({ request, page }) => {
    // Step 1: Create user via API (200ms)
    const userResponse = await request.post('/api/users', {
        data: {
            name: 'Test User',
            accountNumber: '123456789',
            balance: 10000
        }
    });
    const user = await userResponse.json();

    // Step 2: Get auth token via API (100ms)
    const authResponse = await request.post('/api/auth/login', {
        data: {
            accountNumber: '123456789',
            pin: '1234'
        }
    });
    const { token } = await authResponse.json();

    // Step 3: Set auth token in browser (instant)
    await page.context().addCookies([
        {
            name: 'authToken',
            value: token,
            path: '/',
            domain: 'http://localhost:3001'
        }
    ]);
});
```

```
name: 'auth_token',
value: token,
domain: 'localhost',
path: '/'
}]);  
  
// Step 4: UI Test - Focus on purchase flow (5s)
await page.goto('/shop');
await page.getByRole('button', { name: 'Buy Now' }).click();
await page.getByRole('button', { name: 'Confirm Purchase' }).click();  
  
// Verify
await expect(page.getText('Purchase Successful')).toBeVisible();  
  
// Verify via API
const balanceResponse = await
request.get(`/api/users/${user.id}/balance`, {
  headers: { Authorization: `Bearer ${token}` }
});
const { balance } = await balanceResponse.json();
expect(balance).toBeLessThan(10000);
});
```

 **BEST PRACTICE:** This test runs in 6 seconds instead of 15 seconds.  
Multiply by 1000 tests = 2.5 hours saved!

## 10.4 API-First Data Setup Strategy

```
// test.beforeEach() - Set up test data via API
test.beforeEach(async ({ request }) => {
  // Create test products
  await request.post('/api/products', {
    data: [
      { name: 'Laptop', price: 999 },
      { name: 'Mouse', price: 29 }
    ]
});
```

```
// Create test categories
await request.post('/api/categories', {
  data: [
    { name: 'Electronics' },
    { name: 'Accessories' }
  ]
});
});

// test.afterEach() - Clean up via API
test.afterEach(async ({ request }) => {
  await request.delete('/api/test-data');
});
```

 **BANKING:** For banking applications, use APIs to set up account balances, transaction history, and user preferences. Then use UI to test the actual user journey.

## CHAPTER 11: VISUAL TESTING & ACCESSIBILITY

### 11.1 Visual Regression Testing

#### Basic Screenshot Comparison

```
test('homepage visual test', async ({ page }) => {
  await page.goto('/');
  await expect(page).toHaveScreenshot('homepage.png');
});
```

#### Element Screenshot

```
test('header visual test', async ({ page }) => {
  await page.goto('/');
  const header = page.locator('header');
  await expect(header).toHaveScreenshot('header.png');
});
```

#### Full Page Screenshot

```
test('full page screenshot', async ({ page }) => {
  await page.goto('/pricing');
  await expect(page).toHaveScreenshot('pricing-page.png', {
    fullPage: true
  });
});
```

#### Advanced Configuration

```
test('dashboard with masking', async ({ page }) => {
  await page.goto('/dashboard');

  await expect(page).toHaveScreenshot('dashboard.png', {
    maxDiffPixels: 100,           // Allow 100 pixels difference
    threshold: 0.2,             // 20% difference threshold
    mask: [                     // Hide dynamic content
      page.locator('.timestamp'),
      page.locator('.balance')
    ]
});
```

```
  ],
  animations: 'disabled'          // Disable animations
});
});
```

## Responsive Visual Testing

```
const viewports = [
  { name: 'mobile', width: 375, height: 667 },
  { name: 'tablet', width: 768, height: 1024 },
  { name: 'desktop', width: 1920, height: 1080 }
];

for (const viewport of viewports) {
  test(`visual test - ${viewport.name}`, async ({ page }) => {
    await page.setViewportSize(viewport);
    await page.goto('/');
    await expect(page).toHaveScreenshot(`page-${viewport.name}.png`);
  });
}
```

## Updating Baselines

```
# Update all screenshots
npx playwright test --update-snapshots

# Update specific test
npx playwright test homepage.spec.ts --update-snapshots
```

## 11.2 Accessibility Testing with axe-core

 **BANKING:** Banks like JPMorgan and Mastercard must comply with ADA and WCAG 2.1 Level AA standards. Failing accessibility audits can result in lawsuits and regulatory fines.

## Installation

```
| npm install -D @axe-core/playwright
```

## Basic Accessibility Test

```
import { test, expect } from '@playwright/test';
import AxeBuilder from '@axe-core/playwright';

test('homepage accessibility', async ({ page }) => {
  await page.goto('/');

  const accessibilityScanResults = await new AxeBuilder({ page })
    .analyze();

  expect(accessibilityScanResults.violations).toEqual([]);
});
```

## Testing Specific WCAG Criteria

```
test('banking form accessibility', async ({ page }) => {
  await page.goto('/transfer');

  const results = await new AxeBuilder({ page })
    .withTags(['wcag2a', 'wcag2aa', 'wcag21aa'])
    .analyze();

  if (results.violations.length > 0) {
    console.log('Accessibility Violations:');
    results.violations.forEach(violation => {
      console.log(`- ${violation.description}`);
      console.log(` Impact: ${violation.impact}`);
      console.log(` Elements: ${violation.nodes.length}`);
    });
  }

  expect(results.violations).toEqual([]);
});
```

## 11.3 Banking Compliance & WCAG Standards

## Critical WCAG Success Criteria

- 1.1.1: Text Alternatives (Level A)
- 1.3.1: Info and Relationships (Level A)
- 1.4.3: Contrast (Minimum) (Level AA)
- 2.1.1: Keyboard (Level A)
- 2.4.7: Focus Visible (Level AA)
- 3.3.2: Labels or Instructions (Level A)

## Exclude Known Issues

```
test('accessibility with exclusions', async ({ page }) => {
  await page.goto('/');

  const results = await new AxeBuilder({ page })
    .exclude('.third-party-widget') // Exclude third-party
    .disableRules(['color-contrast']) // Temporary exclusion
    .analyze();

  expect(results.violations).toEqual([]);
});
```

 **BEST PRACTICE:** Run accessibility tests in CI/CD to catch violations before production. A single lawsuit can cost millions.

## CHAPTER 12: PERFORMANCE TESTING

### 12.1 Core Web Vitals

```
test('measure Core Web Vitals', async ({ page }) => {
  await page.goto('/');

  const metrics = await page.evaluate(() => {
    const perfEntries = performance.getEntriesByType('paint');
    const navigation = performance.getEntriesByType('navigation')[0];

    return {
      FCP: perfEntries.find(e => e.name === 'first-contentful-paint')?.startTime,
      LCP: performance.getEntriesByType('largest-contentful-paint')[0]?.startTime,
      domContentLoaded: navigation?.domContentLoadedEventEnd - navigation?.domContentLoadedEventStart,
      loadComplete: navigation?.loadEventEnd - navigation?.loadEventStart
    };
  });
});

// Assert thresholds
expect(metrics.FCP).toBeLessThan(1800); // < 1.8s
expect(metrics.LCP).toBeLessThan(2500); // < 2.5s
});
```

### 12.2 Page Load Time

```
test('page load performance', async ({ page }) => {
  const startTime = Date.now();
  await page.goto('https://example.com');
  const loadTime = Date.now() - startTime;

  console.log(`Page loaded in: ${loadTime}ms`);
  expect(loadTime).toBeLessThan(3000); // < 3 seconds
});
```

```
|});
```

## 12.3 Resource Timing

```
test('analyze resource loading', async ({ page }) => {
  await page.goto('/');

  const resources = await page.evaluate(() => {
    return performance.getEntriesByType('resource').map(r => ({
      name: r.name,
      duration: r.duration,
      size: r.transferSize,
      type: r.initiatorType
    }));
  });

  // Find slow resources
  const slowResources = resources.filter(r => r.duration > 1000);
  console.log('Slow resources:', slowResources);

  expect(slowResources.length).toBe(0);
});
```

## 12.4 API Performance

```
test('API response time', async ({ request }) => {
  const startTime = Date.now();

  const response = await request.get('/api/users');

  const responseTime = Date.now() - startTime;

  console.log(`API responded in: ${responseTime}ms`);
  expect(response.status()).toBe(200);
  expect(responseTime).toBeLessThan(500); // < 500ms
});
```

## 12.5 Performance Budget

```
test('performance budget', async ({ page }) => {
  await page.goto('/');

  const metrics = await page.evaluate(() => {
    const nav = performance.getEntriesByType('navigation')[0];
    const resources = performance.getEntriesByType('resource');

    return {
      pageLoadTime: nav?.loadEventEnd - nav?.fetchStart,
      domContentLoaded: nav?.domContentLoadedEventEnd - nav?.fetchStart,
      totalPageSize: resources.reduce((sum, r) => sum + r.transferSize, 0),
      numberOfRequests: resources.length
    };
  });

  // Performance budget
  const budget = {
    pageLoadTime: 3000,           // 3 seconds
    domContentLoaded: 2000,        // 2 seconds
    totalPageSize: 2 * 1024 * 1024, // 2 MB
    numberOfRequests: 50          // 50 requests
  };

  expect(metrics.pageLoadTime).toBeLessThan(budget.pageLoadTime);

  expect(metrics.domContentLoaded).toBeLessThan(budget.domContentLoaded);
  expect(metrics.totalPageSize).toBeLessThan(budget.totalPageSize);

  expect(metrics.numberOfRequests).toBeLessThan(budget.numberOfRequests);
});
```



**BANKING:** Performance is critical for banking applications. Slow pages lead to abandoned transactions and lost revenue.

Author-Vaibhav Sahu  
**PLAYWRIGHT HANDBOOK**

---

## PART IV: ENTERPRISE PATTERNS

---

## CHAPTER 13: TEST ORGANIZATION AND STRUCTURE

### 13.1 Enterprise Directory Structure

```
playwright-project/
├── tests/
│   ├── e2e/                      # End-to-end tests
│   │   ├── auth/
│   │   ├── transactions/
│   │   └── admin/
│   ├── api/                       # API tests
│   ├── visual/                    # Visual regression
│   └── performance/              # Performance tests
├── pages/
│   ├── LoginPage.ts
│   └── DashboardPage.ts
├── components/                  # Component Objects
│   ├── HeaderComponent.ts
│   └── FooterComponent.ts
├── fixtures/                    # Custom fixtures
│   └── customFixtures.ts
├── helpers/                     # Utility functions
│   ├── apiHelpers.ts
│   └── testData.ts
├── data/                         # Test data
│   └── users.json
├── config/                       # Environment configs
│   ├── staging.config.ts
│   └── prod.config.ts
└── playwright.config.ts
```

### 13.2 Custom Fixtures

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

```
type CustomFixtures = {
    loginPage: LoginPage;
    authenticatedPage: Page;
};

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

    authenticatedPage: async ({ page }, use) => {
        // Auto-login before test
        await page.goto('/login');
        await page.getByLabel('Email').fill('user@example.com');
        await page.getByLabel('Password').fill('pass123');
        await page.getByRole('button', { name: 'Log in' }).click();
        await page.waitForURL('/dashboard');

        await use(page);
    }
});
```

### 13.3 Test Hooks

```
test.beforeEach(async ({ page }) => {
    await page.goto('/');
});

test.afterEach(async ({ page }, testInfo) => {
    if (testInfo.status !== 'passed') {
        // Capture screenshot on failure
        await page.screenshot({ path: `failure-${testInfo.title}.png` });
    }
});
```

### 13.4 Tagging Tests

```
test('critical flow @smoke @critical', async ({ page }) => {
    // Test implementation
});

test('admin feature @admin', async ({ page }) => {
    // Test implementation
});

# Run only smoke tests
npx playwright test --grep @smoke

# Run everything except admin
npx playwright test --grep-invert @admin
```

## 13.5 Data-Driven Testing

```
const users = [
    { role: 'admin', email: 'admin@example.com', expectedUrl: '/admin' },
    { role: 'user', email: 'user@example.com', expectedUrl: '/dashboard' },
    { role: 'guest', email: 'guest@example.com', expectedUrl: '/welcome' }
];

for (const userData of users) {
    test(`#${userData.role} login flow`, async ({ page }) => {
        await page.goto('/login');
        await page.getByLabel('Email').fill(userData.email);
        await page.getByRole('button', { name: 'Log in' }).click();
        await expect(page).toHaveURL(userData.expectedUrl);
    });
}
```

## 13.6 Environment Configuration

```
// config/environments.ts
export const environments = {
  local: {
    baseUrl: 'http://localhost:3000',
    apiUrl: 'http://localhost:8000'
  },
  staging: {
    baseUrl: 'https://staging.example.com',
    apiUrl: 'https://api.staging.example.com'
  },
  production: {
    baseUrl: 'https://example.com',
    apiUrl: 'https://api.example.com'
  }
};

const env = process.env.ENV || 'local';
export const config = environments[env];
```

 **BEST PRACTICE:** Organize tests by feature/module, not by type. Keep related tests together for easier maintenance.

## CHAPTER 14: CI/CD INTEGRATION & TEST SHARDING

### 14.1 GitHub Actions Advanced Workflows

#### Basic Workflow

```
# .github/workflows/playwright.yml
name: Playwright Tests
on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 18

      - name: Install dependencies
        run: npm ci

      - name: Install Playwright
        run: npx playwright install --with-deps

      - name: Run tests
        run: npx playwright test

      - uses: actions/upload-artifact@v3
        if: always()
        with:
          name: playwright-report
          path: playwright-report/
          retention-days: 30
```

### 14.2 Test Sharding: 1000 Tests in 5 Minutes

 **ENTERPRISE:** Sharding splits your test suite across 10-20 parallel virtual machines. Each machine runs a subset of tests simultaneously.

## The Math

- $1000 \text{ tests} \times 5 \text{ seconds each} = 5000 \text{ seconds (83 minutes)}$
- With 10 shards:  $5000 \div 10 = 500 \text{ seconds (8.3 minutes)}$
- With 20 shards:  $5000 \div 20 = 250 \text{ seconds (4.2 minutes)}$

## GitHub Actions Sharding

```
name: Playwright Tests (Sharded)
on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
      matrix:
        shardIndex: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        shardTotal: [10]

    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3

      - name: Install dependencies
        run: npm ci

      - name: Install Playwright
        run: npx playwright install --with-deps

      - name: Run tests (Shard ${{ matrix.shardIndex }}/${{
matrix.shardTotal }})
        run: npx playwright test --shard=${{ matrix.shardIndex }}/${{
matrix.shardTotal }}

      - uses: actions/upload-artifact@v3
        if: always()
        with:
          name: playwright-report-${{ matrix.shardIndex }}
```

```
| path: playwright-report/
```

## 14.3 Parallel Execution Strategies

### Configuration

```
// playwright.config.ts
export default defineConfig({
  workers: process.env.CI ? 2 : undefined, // 2 in CI, max locally
  fullyParallel: true, // Parallel within files
  retries: process.env.CI ? 2 : 0, // Retry failures in CI
});
```

### Docker Integration

```
# Dockerfile
FROM mcr.microsoft.com/playwright:v1.40.0-focal
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
CMD ["npx", "playwright", "test"]
```

```
# docker-compose.yml
version: '3'
services:
  playwright:
    build: .
    environment:
      - CI=true
    volumes:
      - ./playwright-report:/app/playwright-report
```

 **BEST PRACTICE:** Use 10 shards for most projects. Use 20+ shards for massive test suites (2000+ tests). GitHub Actions allows up to 256 parallel jobs.

## CHAPTER 15: DEBUGGING AND TROUBLESHOOTING

### 15.1 Debug Mode

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

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

# Pause on test
await page.pause(); // Add this line in your test
```

### 15.2 Headed Mode

```
# See browser while testing
npx playwright test --headed

# Slow down execution
npx playwright test --headed --slow-mo=1000
```

### 15.3 Trace Viewer Analysis for Flaky Tests

#### Configuration

```
// playwright.config.ts
use: {
  trace: 'on-first-retry',      // Capture on failure
  screenshot: 'only-on-failure',
  video: 'retain-on-failure'
}
```

#### [View Trace](#)

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

 **ENTERPRISE:** Trace Viewer lets you see exactly what happened during test execution - screenshots, DOM snapshots, network activity, console logs - perfect for debugging flaky tests without re-running them.

## 15.4 Production Debugging Strategies

### Network Debugging

```
page.on('request', request => {
  console.log('→', request.method(), request.url());
});

page.on('response', response => {
  console.log('←', response.status(), response.url());
});

page.on('requestfailed', request => {
  console.log('✗', request.url(), request.failure()?.errorText());
});
```

### Console Debugging

```
page.on('console', msg => {
  console.log('Browser console:', msg.text());
});

page.on('pageerror', error => {
  console.log('Page error:', error.message());
});
```

### Element Debugging

```
// Check if element exists
const count = await button.count();
console.log(`Found ${count} elements`);
```

```
// Highlight element
await button.highlight();

// Get element info
const box = await button.boundingBox();
console.log('Element position:', box);
```

## 15.5 Common Issues & Solutions

### Timeout Issues

```
// Increase timeout for specific test
test.setTimeout(120000); // 2 minutes

// Increase timeout for specific action
await page.getByRole('button').click({ timeout: 60000 });
```

### Flaky Tests

**⚠️ WARNING:** Avoid using `page.waitForTimeout()` - it makes tests slower and more flaky. Use proper waits instead.

```
// ✗ Bad - arbitrary wait
await page.waitForTimeout(3000);

// ✓ Good - wait for specific condition
await page.waitForLoadState('networkidle');
await page.waitForSelector('.loaded');
await page.getText('Success').waitFor();
```

### Screenshots for Debugging

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

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

 **BEST PRACTICE:** Enable trace on first retry to get detailed information about failures without slowing down successful tests.

## CHAPTER 16: ADVANCED PATTERNS AND TECHNIQUES

### 16.1 Network Mocking for Resilience Testing

 **ENTERPRISE:** Test how your application handles server failures, timeouts, and network errors. This is critical for banking applications where uptime is essential.

#### Simulating Server Downtime

```
test('handles banking server down', async ({ page }) => {
    // Mock API to return 503 Service Unavailable
    await page.route('**/api/transactions', route => {
        route.fulfill({
            status: 503,
            contentType: 'application/json',
            body: JSON.stringify({
                error: 'Service temporarily unavailable'
            })
        });
    });

    await page.goto('/transactions');

    // Verify graceful error handling
    await expect(page.getText('Service temporarily unavailable')).toBeVisible();
    await expect(page.getRole('button', { name: 'Retry' })).toBeVisible();
    await expect(page.getText('Please try again later')).toBeVisible();
});
```

#### Simulating Slow Network

```
test('handles slow network', async ({ page }) => {
    await page.route('**/api/**', route => {
        // Delay response by 5 seconds
        setTimeout(() => route.continue(), 5000);
    });
});
```

```
};

await page.goto('/dashboard');

// Verify loading state shows
await expect(page.getText('Loading...')).toBeVisible();
});
```

## 16.2 Global Teardown & Data Cleanup

### Global Teardown Script

```
// global-teardown.ts
import { FullConfig } from '@playwright/test';

async function globalTeardown(config: FullConfig) {
    console.log('Cleaning up test data...');

    // Clean up database
    const response = await fetch('http://localhost:8000/api/cleanup', {
        method: 'DELETE',
        headers: { 'X-Test-Cleanup': 'true' }
    });

    if (response.ok) {
        console.log('✓ Test data cleaned up successfully');
    }
}

export default globalTeardown;
```

### Configuration

```
// playwright.config.ts
export default defineConfig({
    globalTeardown: require.resolve('./global-teardown'),
});
```

 **ENTERPRISE:** After 1000 tests, ensure all test data in the database is cleaned up via API. This prevents data pollution and ensures tests remain independent.

## 16.3 Self-Healing Test Patterns

### Smart Locator with Fallbacks

```
// utils/selfHealingLocator.ts
export async function smartLocator(
  page: Page,
  primarySelector: string,
  fallbackSelectors: string[]
): Promise<Locator> {
  // Try primary selector
  let element = page.locator(primarySelector);

  if (await element.count() > 0) {
    return element;
  }

  // Try fallback selectors
  for (const selector of fallbackSelectors) {
    element = page.locator(selector);
    if (await element.count() > 0) {
      console.log(`Self-healed: Using ${selector} instead of ${primarySelector}`);
      return element;
    }
  }

  throw new Error(`Could not locate element: ${primarySelector}`);
}
```

### Usage

```
const loginButton = await smartLocator(
  page,
  'button.login-btn', // Primary
  [ // Fallbacks
```

```
'button#login',
  '[data-testid="login-button"]',
  'button:has-text("Log in")'

];
);

await loginButton.click();
```

 **BEST PRACTICE:** Self-healing patterns reduce test maintenance by 60%. When selectors break, tests auto-repair instead of failing.

---

## PART V: THE FUTURE

---

## CHAPTER 17: AI-DRIVEN TESTING

 **AI-POWERED:** The future of QA automation lies in AI-assisted testing. This chapter explores how AI is transforming Playwright automation in 2025 and beyond.

### 17.1 Prompt Engineering for Playwright Scripts

#### Using AI to Generate Test Scripts

Large Language Models like GPT-4 can generate Playwright test scripts from natural language descriptions.

#### *Example Prompt*

Prompt: "Generate a Playwright test for a banking login flow with multi-factor authentication. The test should:  
1. Navigate to /login  
2. Enter account number and PIN  
3. Handle OTP verification  
4. Verify successful login to dashboard"

#### *AI-Generated Output*

```
test('banking login with MFA', async ({ page }) => {
  await page.goto('/login');

  await page.getByLabel('Account Number').fill('123456789');
  await page.getByLabel('PIN').fill('1234');
  await page.getByRole('button', { name: 'Continue' }).click();

  // Wait for OTP screen
  await expect(page.getText('Enter OTP')).toBeVisible();

  // Enter OTP (in real scenario, fetch from email/SMS)
  await page.getByLabel('OTP').fill('123456');
  await page.getByRole('button', { name: 'Verify' }).click();
```

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

 **BEST PRACTICE:** Use AI to generate 70% of boilerplate code, then refine manually for edge cases and business logic.

## 17.2 Self-Healing Selectors with AI

### The Problem: Brittle Selectors

Traditional selectors break when developers change class names, IDs, or DOM structure. This causes test maintenance nightmares.

### The Solution: AI-Powered Self-Healing

AI can analyze the page context and automatically find the correct element even when selectors change.

#### *Self-Healing Implementation*

```
// utils/aiSelfHealing.ts
import { Page, Locator } from '@playwright/test';
import OpenAI from 'openai';

export async function aiSmartLocator(
  page: Page,
  primarySelector: string,
  fallbackSelectors: string[],
  aiContext?: string
): Promise<Locator> {
  // Try primary selector
  let element = page.locator(primarySelector);

  if (await element.count() > 0) {
    return element;
  }

  // Try fallback selectors
  for (const selector of fallbackSelectors) {
    element = page.locator(selector);
    if (await element.count() > 0) {
      return element;
    }
  }

  // If none found, use AI
  const response = await OpenAI.createCompletion({
    model: 'text-davinci-003',
    prompt: `Given the current page context, identify the correct element for the selector ${primarySelector}. The page has the following structure: ${page.innerHTML}.`,
  });

  const aiContext = response.choices[0].text;
  const aiResponse = await page.evaluate((context) => {
    const script = `window.__aiContext__ = ${JSON.stringify(context)};`;
    return new Promise((resolve) => {
      document.head.innerHTML += script;
      resolve();
    });
  }, { aiContext });

  const aiElement = page.locator(`[data-testid="ai-element"]`);
  if (await aiElement.count() > 0) {
    return aiElement;
  }
}
```

```
}

// Try fallback selectors
for (const selector of fallbackSelectors) {
    element = page.locator(selector);
    if (await element.count() > 0) {
        console.log(`Self-healed: ${selector}`);
        return element;
    }
}

// Use AI to find element
if (aiContext) {
    const aiSelector = await findElementWithAI(page, aiContext);
    return page.locator(aiSelector);
}

throw new Error(`Could not locate: ${primarySelector}`);
}

async function findElementWithAI(page: Page, context: string) {
    const openai = new OpenAI({ apiKey: process.env.OPENAI_API_KEY });

    // Get page HTML
    const html = await page.content();

    // Ask AI to find selector
    const response = await openai.chat.completions.create({
        model: 'gpt-4',
        messages: [
            {
                role: 'user',
                content: `Find the best selector for:
${context}\n\nHTML:\n${html}`
            }
        ]
    });

    return response.choices[0].message.content || '';
}
```

## Usage

```
const loginButton = await aiSmartLocator(
```

```
page,  
'button.login-btn',  
['button#login', '[data-testid="login"]'],  
'The main login button on the login page'  
) ;  
  
await loginButton.click();
```

 **AI-POWERED:** Self-healing reduces test maintenance by 60%. When selectors break, tests auto-repair instead of failing.

## 17.3 Using LLMs to Generate Edge-Case Test Data

### The Challenge

Testing edge cases (special characters, unicode, SQL injection, XSS) is tedious and often incomplete.

### *AI-Generated Test Data*

```
// utils/aiTestData.ts  
import OpenAI from 'openai';  
  
export async function generateEdgeCases(fieldType: string):  
Promise<string[]> {  
    const openai = new OpenAI({ apiKey: process.env.OPENAI_API_KEY });  
  
    const prompt = `Generate 20 edge case test values for a  
${fieldType}  
field in a banking application. Include:  
- Special characters  
- Unicode  
- Very long strings  
- SQL injection attempts  
- XSS attempts  
- Empty strings  
- Whitespace  
- Null bytes
```

```
Return as JSON array.';

const response = await openai.chat.completions.create({
  model: 'gpt-4',
  messages: [{ role: 'user', content: prompt }]
});

const content = response.choices[0].message.content || '[]';
return JSON.parse(content);
}
```

### Using AI-Generated Data

```
test('form handles edge cases', async ({ page }) => {
  const edgeCases = await generateEdgeCases('account_name');

  for (const testValue of edgeCases) {
    await page.goto('/account/create');
    await page.getByLabel('Account Name').fill(testValue);
    await page.getByRole('button', { name: 'Submit' }).click();

    // Should either accept valid or show error
    const hasError = await page.getText('Invalid').isVisible();
    const hasSuccess = await page.getText('Success').isVisible();

    expect(hasError || hasSuccess).toBeTruthy();
  }
});
```

 **BEST PRACTICE:** AI can generate 1000+ edge cases in seconds. This level of coverage is impossible to achieve manually.

## 17.4 AI-Powered Test Maintenance

### Automated Flaky Test Detection

AI can analyze test execution patterns and identify flaky tests before they become problematic.

```
// AI analyzes 100 test runs and identifies patterns:  
// - Test "login" passes 95/100 times (flaky)  
// - Test "checkout" passes 100/100 times (stable)  
//  
// AI suggests fixes:  
// 1. Add explicit wait for login button  
// 2. Increase timeout from 5s to 10s  
// 3. Add retry logic for network calls
```

## Intelligent Test Generation

```
// AI watches manual testing sessions  
// Generates automated tests automatically  
// Suggests new test cases based on code changes
```

## 17.5 The Future of QA Automation

 **AI-POWERED:** AI will not replace QA engineers. Instead, it will elevate them from script writers to test strategists.

### The AI-Augmented QA Engineer (2025)

- AI generates 80% of test code
- Engineers focus on test strategy and business logic
- Self-healing tests reduce maintenance by 70%
- AI-generated edge cases increase coverage by 10x
- Predictive analytics identify bugs before they happen

### Skills for the Future

 **ENTERPRISE:** To thrive in AI-powered QA, master: Prompt engineering, AI tool integration, test strategy, domain knowledge, and Playwright architecture.

- Prompt Engineering: Write effective prompts for AI
- AI Integration: Integrate LLMs into test frameworks
- Test Strategy: Design comprehensive test approaches

- Domain Knowledge: Understand business requirements
- Playwright Expertise: Deep framework knowledge



**BANKING:** JPMorgan and Mastercard are already using AI for test generation, maintenance, and analysis. Stay ahead of the curve.

# CONCLUSION

## You Are Now Enterprise-Ready

Congratulations! You have completed the Playwright Enterprise Edition handbook.

You now possess advanced skills in:

### PART III - Advanced Testing:

- Complex scenarios (auth, files, iframes, popups)
- API testing (REST, GraphQL, Hybrid UI+API)
- Visual regression and accessibility testing
- Performance testing and optimization

### PART IV - Enterprise Patterns:

- Professional test organization
- CI/CD integration and test sharding
- Debugging with Trace Viewer
- Network resilience and self-healing tests

### PART V - The Future:

- AI-powered test generation
- Self-healing selectors
- LLM-generated edge cases
- AI-powered test maintenance



**ENTERPRISE:** This knowledge positions you for senior roles at JPMorgan, Mastercard, and other Fortune 500 companies.

### Next Steps:

- Implement test sharding in your CI/CD pipeline
- Add accessibility tests with axe-core

- Experiment with AI-powered test generation
- Build a portfolio showcasing enterprise patterns
- Apply for senior QA/SDET roles

 **AI-POWERED:** The future of QA is here. Lead the transformation. 

*Thank you for learning with Playwright Enterprise Edition!*