**Playwright Setup and Testing for Blazor Web Applications**

Contents

[1. Installation and Setup 2](#_Toc178865600)

[Node.js Requirement 2](#_Toc178865601)

[**Initializing Playwright** 2](#_Toc178865602)

[**Project Structure** 2](#_Toc178865603)

[**2. Playwright Management and Updates** 2](#_Toc178865604)

[**Starting Playwright (New Project Setup)** 2](#_Toc178865605)

[**Updating Playwright** 3](#_Toc178865606)

[**Installing Browser Dependencies** 3](#_Toc178865607)

[**Checking Playwright Version** 3](#_Toc178865608)

[**4.** **Running Tests** 3](#_Toc178865609)

[**Running All Tests** 4](#_Toc178865610)

[**Running a Specific Test** 4](#_Toc178865611)

[**Viewing HTML Reports** 4](#_Toc178865612)

[**Interactive Debugging with UI** 4](#_Toc178865613)

[**4. Running Tests for Specific Browsers** 4](#_Toc178865614)

[**WebKit (Safari Engine)** 4](#_Toc178865615)

[**Firefox** 5](#_Toc178865616)

[**Chromium (Chrome/Edge)** 5](#_Toc178865617)

[**5. Additional Notes and Useful Commands** 5](#_Toc178865618)

[**Running Tests in Headed Mode** 5](#_Toc178865619)

[**Using Test Filters** 5](#_Toc178865620)

[**Debugging with Breakpoints** 5](#_Toc178865621)

[**Taking Screenshots for Visual Testing** 5](#_Toc178865622)

# 1. Installation and Setup

### Node.js Requirement

* Ensure you have the latest version of **Node.js** installed before starting with Playwright.
* Download it from [Node.js official website](https://nodejs.org/).
* To check if Node.js is installed or verify the version, run the following command in your terminal:

node -v

**Initializing Playwright**

1. Go to your Blazor project’s main folder.
2. Run the following command to initialize Playwright:

npm init playwright@latest

1. This command will:
   * Install Playwright dependencies.
   * Set up a **tests** folder containing .spec.js/.spec.ts files (test scripts).
   * Download necessary browser binaries (Chromium, Firefox, WebKit).

**Project Structure**

* Once Playwright is initialized, the default folder structure will look like this:

tests/

├── example.spec.js (\*Sample test file generated by Playwright\*)

└── <your custom test files>

**2. Playwright Management and Updates**

**Starting Playwright (New Project Setup)**

* When setting up a new project or integrating Playwright into an existing Blazor project, initialize Playwright with:

npm init playwright@latest

**Updating Playwright**

* To update Playwright to the latest version, run:

npm install -D @playwright/test@latest

**Installing Browser Dependencies**

* If Playwright requires additional browser dependencies (e.g., for CI environments), install them using:

npx playwright install --with-deps

**Checking Playwright Version**

* To verify the current version of Playwright installed, run:

npx playwright --version

1. **Running Tests**

A Playwright test script typically follows this structure:

* **Launching a browser**.
* **Navigating to a webpage**.
* **Interacting with UI elements**.
* **Verifying expected behaviour** (using assertions).

**Running Recorder**

* PS C:\Users\Boyd\Documents\GitHub\playwrite\BlazorApp> cd C:\Users\Boyd\Documents\GitHub\playwrite\BlazorApp
* PS C:\Users\Boyd\Documents\GitHub\playwrite\BlazorApp> npm init -y
* PS C:\Users\Boyd\Documents\GitHub\playwrite\BlazorApp> npm install playwright --save-dev
* PS C:\Users\Boyd\Documents\GitHub\playwrite\BlazorApp> npx playwright install
* **Run application in different terminal**(dotnet run)
* PS C:\Users\Boyd\Documents\GitHub\playwrite\BlazorApp> npx playwright codegen <http://localhost:5128> (running url)
* Make sure its recording, once finished copy the code make a spec.js((only difference to typescript should be Run**(**npm install ts-node @types/node --save-dev) and the file being saved as spec.ts) file paste the code into it fix the structure. Run the code using “npx playwright test –ui”**(better ui),** **or** “npx playwright test”**(answers)**

**Running All Tests**

* To run all tests across multiple browsers (Chromium, Firefox, WebKit), use:

npx playwright test

* + This will open up a tab where you will see all test with the available browsers and you can view the logs

**Running a Specific Test**

* To run a specific test file (e.g., weather.spec.js), run:

npx playwright test weather.spec.js

**Viewing HTML Reports**

* After running tests, Playwright generates a detailed **HTML report**. To view the report, use:

npx playwright show-report

**Interactive Debugging with UI**

* For an more interactive test/running experience with real-time updates, use:

npx playwright test --ui

* + This will open a UI where you can browse test results, inspect errors, and re-run tests.
  + Upon running the tests in here you will be able to see the timeline of the application running
  + If an error occurs then you will be able to see what line in the spec.js/ts the error is within the source tab
  + There is also the before; after buttons so you can see what happens when activating an action

**4. Running Tests for Specific Browsers**

Playwright supports testing across multiple browsers, which is essential for cross-browser testing. Below are commands to run tests on specific browsers.

**WebKit (Safari Engine)**

* To run tests on the **WebKit** engine (Safari), use:

npx playwright test --project=webkit

**Firefox**

* To run tests in **Firefox**, use:

npx playwright test --project=firefox

**Chromium (Chrome/Edge)**

* To run tests on the **Chromium** engine (Chrome, Edge), use:

npx playwright test --project=chromium

**5. Additional Notes and Useful Commands**

**Running Tests in Headed Mode**

* Playwright runs in headless mode (no browser UI) by default. If you want to run it with the browser UI visible for debugging purposes, use:

npx playwright test --headed

**Using Test Filters**

* Playwright allows you to filter and run specific tests based on keywords. To run tests that match a particular keyword, use:

npx playwright test --grep "Login"

**Debugging with Breakpoints**

* You can pause test execution at any point using the following command in your test script:

await page.pause();

* + This opens the browser and pauses the test, allowing you to manually inspect the state.

**Taking Screenshots for Visual Testing**

* You can capture screenshots of specific elements or entire pages during your tests:

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

\*Screenshots are useful for visual regression testing, ensuring that the UI does not change unexpectedly between releases.\*

**Cs Running**

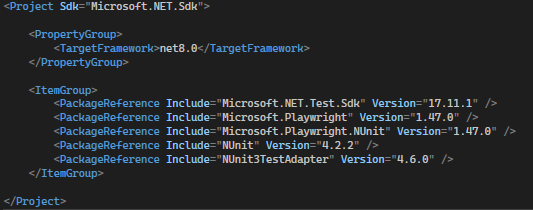
Install-Package NUnit

Install-Package NUnit3TestAdapter

Install-Package Microsoft.Playwright.NUnit

Create a new NUnit Test project (name it something like BlazorApp.PlaywrightTests)

Make sure the .csproj file looks something like this:



Change Test Runner to NUnit (in the record window with the test code)

Recording is the same

Copy the code, Save into .cs file

Build ; Rebuild Solution

Right click and run Debug Tests (Test; Test Explorer; Run(top left))

\*Cs Running only provides the test explorer for the visual, you don’t get a ui test experience like with .js / .ts \*

**Useful links**

* <https://playwright.dev/docs/test-cli> } Command line links
* <https://playwright.dev/docs/test-ui-mode> } UI mode

**Automation**

* make sure that the yml file that your editing is the on the .github folder in the base repository, not the in the application.
* Once that is sorted the testing will run after every commit,

### 1. ****Actions****

These are methods that simulate user interactions with the page (like clicking, filling forms, etc.):

* **click()**: Simulates a user clicking an element.
  + await page.getByRole('button', { name: 'Submit' }).click();
* **fill()**: Fills an input field with text.
  + await page.fill('input[name="username"]', 'myUsername');
* **type()**: Types text into an input field, character by character.
  + await page.type('input[name="password"]', 'myPassword');
* **check()**: Checks a checkbox or radio button.
  + await page.check('input[type="checkbox"]');
* **uncheck()**: Unchecks a checkbox.
  + await page.uncheck('input[type="checkbox"]');
* **selectOption()**: Selects an option in a <select> dropdown.
  + await page.selectOption('select[name="country"]', 'US');
* **hover()**: Simulates hovering over an element.
  + await page.hover('button#tooltip');
* **dblclick()**: Simulates a double-click on an element.
  + await page.dblclick('button');
* **press()**: Simulates pressing a key on the keyboard.
  + await page.press('input[name="username"]', 'Enter');
* **dragAndDrop()**: Drags an element and drops it onto another element.
  + await page.dragAndDrop('#source', '#target');

### 2. ****Assertions****

These are used to verify the state of elements or the page (visibility, content, attributes, etc.):

* **toBeVisible()**: Asserts that an element is visible on the page.
  + await expect(page.getByText('Welcome')).toBeVisible();
* **toBeHidden()**: Asserts that an element is hidden (not visible).
  + await expect(page.getByText('Loading...')).toBeHidden();
* **toHaveText()**: Asserts that an element contains specific text.
  + await expect(page.locator('h1')).toHaveText('Welcome to Playwright');
* **toHaveValue()**: Asserts that an input field contains a specific value.
  + await expect(page.locator('input[name="username"]')).toHaveValue('myUsername');
* **toHaveAttribute()**: Asserts that an element has a specific attribute with a given value.
  + await expect(page.locator('img#logo')).toHaveAttribute('src', '/images/logo.png');
* **toHaveClass()**: Asserts that an element has a specific CSS class.
  + await expect(page.locator('div#alert')).toHaveClass('error');
* **toHaveCount()**: Asserts that a locator (like a list or table rows) has a specific number of elements.
  + await expect(page.locator('ul li')).toHaveCount(3);
* **toBeChecked()**: Asserts that a checkbox or radio button is checked.
  + await expect(page.locator('input[type="checkbox"]')).toBeChecked();
* **toBeDisabled()**: Asserts that an element (like a button) is disabled.
  + await expect(page.locator('button#submit')).toBeDisabled();
* **toBeEnabled()**: Asserts that an element is enabled.
  + await expect(page.locator('button#submit')).toBeEnabled();
* **toBeFocused()**: Asserts that an element is focused.
  + await expect(page.locator('input[name="username"]')).toBeFocused();

### 3. ****Waiting for Conditions****

Playwright also offers utilities for waiting for specific conditions to be met:

* **waitForSelector()**: Waits for an element matching a selector to appear in the DOM.
  + await page.waitForSelector('div#loading-spinner');
* **waitForTimeout()**: Waits for a specified number of milliseconds.
  + await page.waitForTimeout(3000); // waits for 3 seconds
* **waitForResponse()**: Waits for a network request to complete and returns the response.
  + await page.waitForResponse(response => response.url().includes('/api/data') && response.status() === 200);
* **waitForEvent()**: Waits for a specific event to be triggered on the page (e.g., navigation, click, or network events).
  + await page.waitForEvent('load');
* **waitForLoadState()**: Waits for the page to reach a specific load state (like "load", "domcontentloaded", or "networkidle").
  + await page.waitForLoadState('networkidle');

### 4. ****Other Utilities****

* **locator()**: Locates an element or group of elements.
  + const element = page.locator('div.item');
* **getByRole()**: Locates an element by its ARIA role (useful for accessible testing).
  + await page.getByRole('button', { name: 'Submit' });
* **screenshot()**: Captures a screenshot of the page or a specific element.
  + await page.screenshot({ path: 'screenshot.png' });
* **evaluate()**: Executes code in the page's context.
  + const result = await page.evaluate(() => document.title);
  + console.log(result); // prints the page title
* **close()**: Closes the page or browser.
  + await page.close();