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

Contents

[1. Installation and Setup 2](#_Toc178941675)

[Node.js Requirement 2](#_Toc178941676)

[**Initializing Playwright** 2](#_Toc178941677)

[**Project Structure** 2](#_Toc178941678)

[**2. Playwright Management and Updates** 2](#_Toc178941679)

[**Starting Playwright (New Project Setup)** 2](#_Toc178941680)

[**Updating Playwright** 3](#_Toc178941681)

[**Installing Browser Dependencies** 3](#_Toc178941682)

[**Checking Playwright Version** 3](#_Toc178941683)

[**4.** **Running Tests** 3](#_Toc178941684)

[**Running All Tests** 4](#_Toc178941685)

[**Running a Specific Test** 4](#_Toc178941686)

[**Viewing HTML Reports** 4](#_Toc178941687)

[**Interactive Debugging with UI** 4](#_Toc178941688)

[**4. Running Tests for Specific Browsers** 4](#_Toc178941689)

[**WebKit (Safari Engine)** 4](#_Toc178941690)

[**Firefox** 5](#_Toc178941691)

[**Chromium (Chrome/Edge)** 5](#_Toc178941692)

[**5. Additional Notes and Useful Commands** 5](#_Toc178941693)

[**Running Tests in Headed Mode** 5](#_Toc178941694)

[**Using Test Filters** 5](#_Toc178941695)

[**Debugging with Breakpoints** 5](#_Toc178941696)

[**Taking Screenshots for Visual Testing** 5](#_Toc178941697)

[1. Actions 7](#_Toc178941698)

[2. Assertions 7](#_Toc178941699)

[3. Waiting for Conditions 8](#_Toc178941700)

[4. Other Utilities 9](#_Toc178941701)

[**Areas That May Need Adjustments:** 10](#_Toc178941702)

# 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**

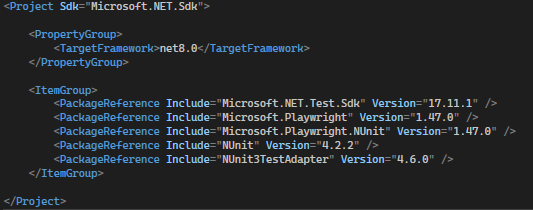
Install-Package NUnit

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();

**YML File**

**Areas That May Need Adjustments:**

1. **Project-Specific .NET Dependencies**:
   * **Step 9: Restore .NET dependencies**
     + You might need to adjust the .csproj file paths or project names if your new Blazor application has different project structures. The path to .csproj could differ based on your Blazor app structure.

**Example**:

run: dotnet restore ./YourNewBlazorApp/YourNewBlazorApp.csproj

1. **Step 10: Build the Blazor project**:
   * If the new Blazor app has a different project name or a different directory structure, you will need to update the dotnet build step to point to the correct .csproj file.

**Example**:

run: dotnet build --configuration Release ./YourNewBlazorApp/YourNewBlazorApp.csproj

1. **Step 11: Start Blazor WebAssembly App**:
   * If the new Blazor project has a different folder structure or uses a different csproj file for the app startup, you will need to adjust the path in this step:

**Example**:

run: |

nohup dotnet run --project YourNewBlazorApp/YourNewBlazorApp.csproj --urls http://localhost:5128 > app.log 2>&1 &

1. **Step 12: Wait for the WebAssembly App to be available**:
   * The port (http://localhost:5128) is hardcoded. If your new Blazor app runs on a different port or the startup configuration changes, you need to adjust the URL here.

**Example**:

run: |

curl --silent --fail http://localhost:5000 || (echo "Server not available, failing build." && exit 1)

1. **Playwright Test Scripts**:
   * The actual Playwright tests inside the tests folder will need to be adjusted to reflect the specific functionality of the new Blazor app. The .yml file doesn't need to change, but your test scripts themselves will need to be updated based on what you're testing in the new app (e.g., different pages, UI elements, workflows, etc.).