*Create Playwright Integration Tests*

|  |  |
| --- | --- |
| Bert O’Neill | |
| Department: | Software Demo’s |
| Document Title: | Playwright and MSTest |
| Author: | Bert O’Neill |
| Date: | 28-Jan-2023 |
| Version | 1.0 |
| SharePoint | N/A |
| Document Goals | Provide a detailed overview for developers who wish to use MSTest and Playwright, to generate integration tests for their application. |

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Description | Author |
| 28-Jan-2023 | 1.0 | Initial draft | Bert O’Neill |
| 06-Feb-2023 | 1.1 | Adding project code | Bert O’Neill |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[Revision History 1](#_Toc125882981)

[Introduction 4](#_Toc125882982)

[Purpose 4](#_Toc125882983)

[Scope 4](#_Toc125882984)

[Prerequisites 4](#_Toc125882985)

[Environment Setup 5](#_Toc125882986)

[PowerShell 7+ Installation 5](#_Toc125882987)

[Playwright Installation 5](#_Toc125882988)

[Source Code 6](#_Toc125882989)

[Headless Browsers 6](#_Toc125882990)

[What is Playwright? 6](#_Toc125882991)

[Browser Types 8](#_Toc125882992)

[Playwright Advantages over Selenium 8](#_Toc125882993)

[Playwright Test Generator (Recorder) 8](#_Toc125882994)

[Debugging Your Playwright (Headless) Tests 11](#_Toc125882995)

[Misc 11](#_Toc125882996)

[Running MSTest tests in Parallel​ 12](#_Toc125882997)

[Customizing BrowserContext options​ 12](#_Toc125882998)

[Using the .runsettings file​ 13](#_Toc125882999)

[Autodetect the run settings file 14](#_Toc125883000)

[Base MSTest classes for Playwright​ 14](#_Toc125883001)

[Competitors: 15](#_Toc125883002)

[Integrations: 15](#_Toc125883003)

[Advantages of Playwright 15](#_Toc125883004)

[Playwright Architecture 16](#_Toc125883005)

[Trace Viewer 18](#_Toc125883006)

[Locators 23](#_Toc125883007)

[Locator 23](#_Toc125883008)

[Assertions With Expects 25](#_Toc125883009)

[Assertions 25](#_Toc125883010)

[List of assertions​ 25](#_Toc125883011)

[Negating Matchers​ 26](#_Toc125883012)

[Soft Assertions​ 26](#_Toc125883013)

[Custom Expect Message​ 27](#_Toc125883014)

[Polling​ 27](#_Toc125883015)

[Retrying​ 28](#_Toc125883016)

[Event Logs 28](#_Toc125883017)

# Introduction

This document provides a detailed overview on how to incorporate an MSTest project and Microsoft’s Playwright framework to generate your integration\end-2-end tests.

## Purpose

To provide a comprehensive technical overview on how to integrate Playwright into your application testing harness.

## Scope

The scope of this document is to convey the concepts needed to start using Playwright in conjunction with MSTest, from installation and setup to interrogating UI components automatically.

# Prerequisites

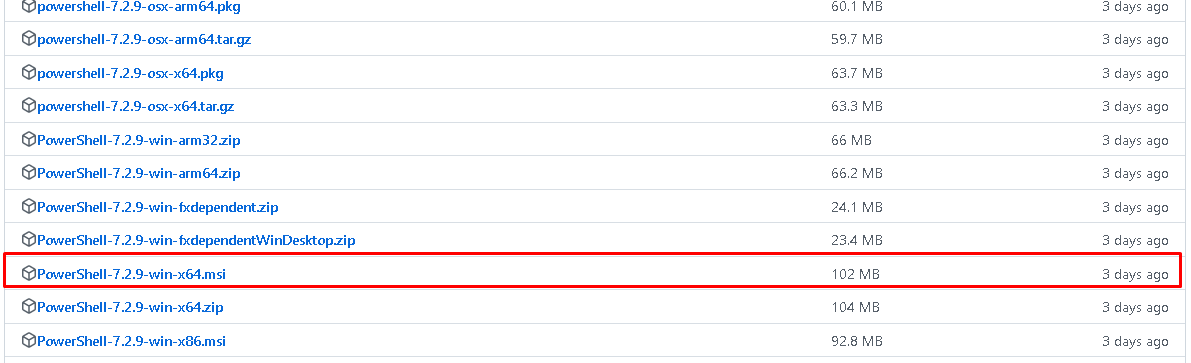
* An understanding of MSTest framework
* An understanding of integration testing
* Knowledge on using Visual Studio
* Know how to run Visual Studio Test Explorer
* Know how to find html elements in a browser’s web developer (F12)

# Environment Setup

## PowerShell 7+ Installation

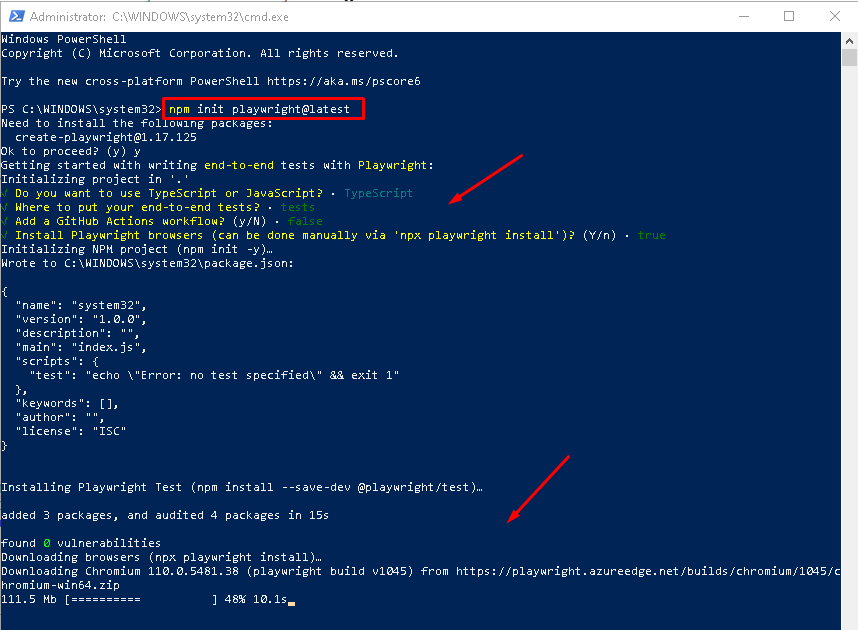
Navigate to the URL below, and install the latest version of PowerShell 7+ for your environment:

* <https://github.com/PowerShell/PowerShell/releases/tag/v7.2.9>



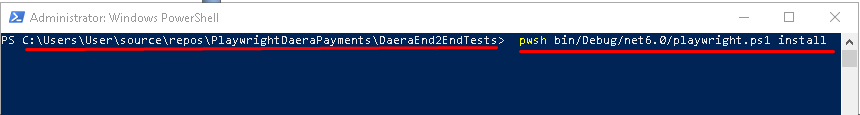
## Playwright Installation

1. Open PowerShell in Admin mode
2. Execute the command npm init playwright@latest, answering the installation questions (I selected Typescript, but you can use JavaScript) – no wrong answer to the questions ☺

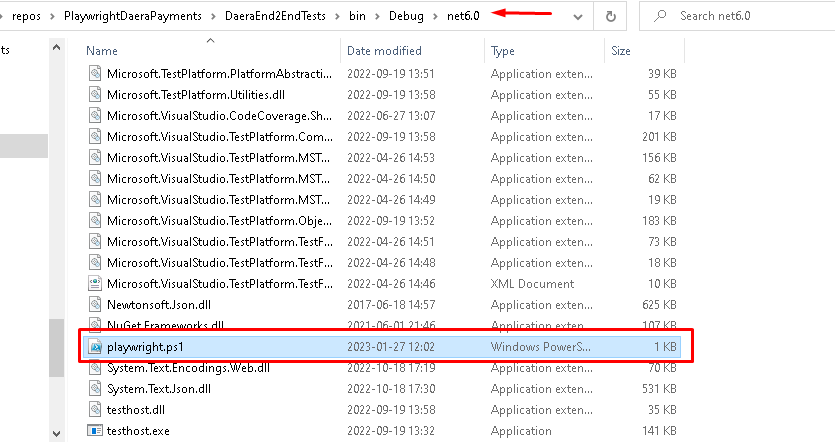


## Source Code

1. Clone Code repo below
   * <https://github.com/Bert0Neill/PlaywrightDaeraPayments>
2. Open solution in Visual Studio and compile to ensure all the packages are downloaded.
3. Open PowerShell and navigate to the project root folder of your repo and execute the following command, changing the framework version to what you have for your project:
   * pwsh bin/Debug/net6.0/playwright.ps1 install



This will create a PowerShell script file in your debug folder:



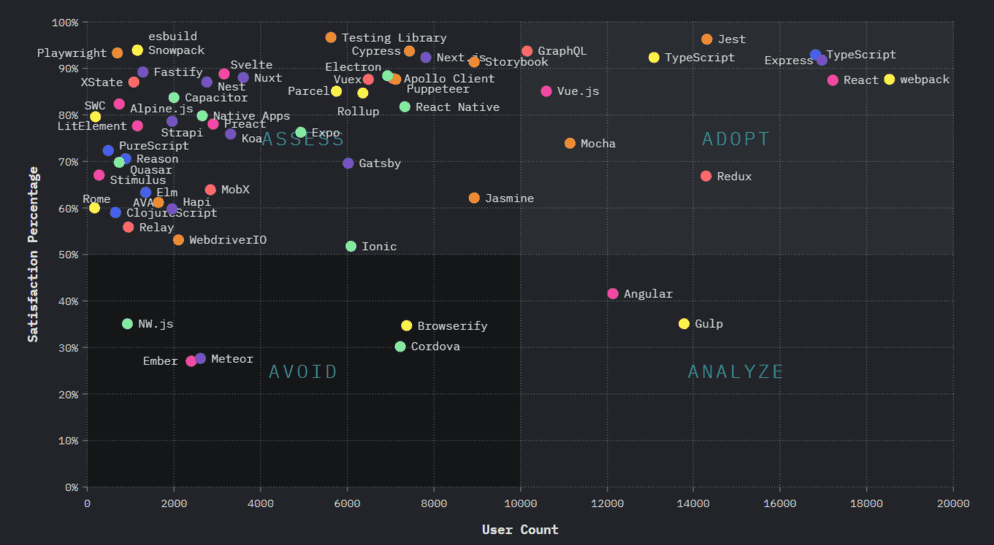
# Headless Browsers

# What is Playwright?

Built by Microsoft, Playwright is a Node.js library that, with a single API, automates Chromium, Firefox, and WebKit. These APIs can be used by developers writing JavaScript code to create new browser pages, navigate to URLs and then interact with elements on a page. In addition, since Microsoft Edge is built on the open-source Chromium web platform, Playwright can also automate Microsoft Edge.

[Playwright end-to-end testing](https://www.lambdatest.com/playwright-e2e-testing) has slowly gained popularity. Many developers consider it one of their favorite frameworks to work with.

[The State of JS survey](https://2020.stateofjs.com/en-US/technologies/) among developers shows that while Cypress adoption is on the rise, Playwright adoption is still in the early stages of development. However, [GitHub](https://github.com/microsoft/playwright) statistics show Playwright's popularity is growing (44.6k Stars and 2.2k Fork).



Playwright launches headless browsers by default. Playwright launches a headless browser by default. The command line is the only way to use a headless browser, as it does not display a UI. Playwright also supports running full (non-headless) Microsoft Edge.

# Browser Types

# Playwright Advantages over Selenium

# Playwright Test Generator (Recorder)

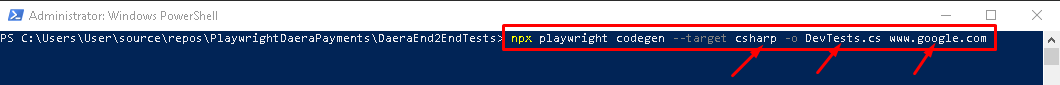
Open PowerShell and enter the following command to start the recording

npx playwright codegen --target csharp -o DevTests.cs [www.google.com](http://www.google.com)

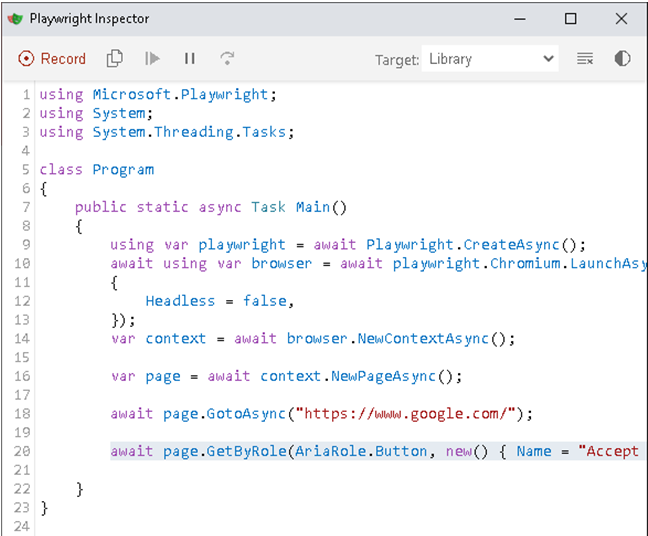
If you enter the URL at the end, this page will open, so if you wanted to generate tests for your development site, you would replace this with a localhost or your development server URL.

You will also see that we are telling Playwright to generate C# syntax tests and to save them into a file called DevTests.cs.

You may be prompted to install certain packages (do this and accept all defaults).

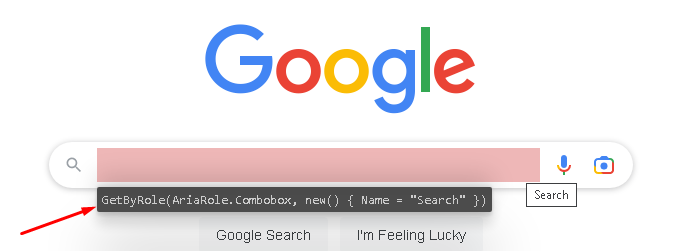


You will get the URL that you eneterd as an option loaded along with the steps recorder window (below)

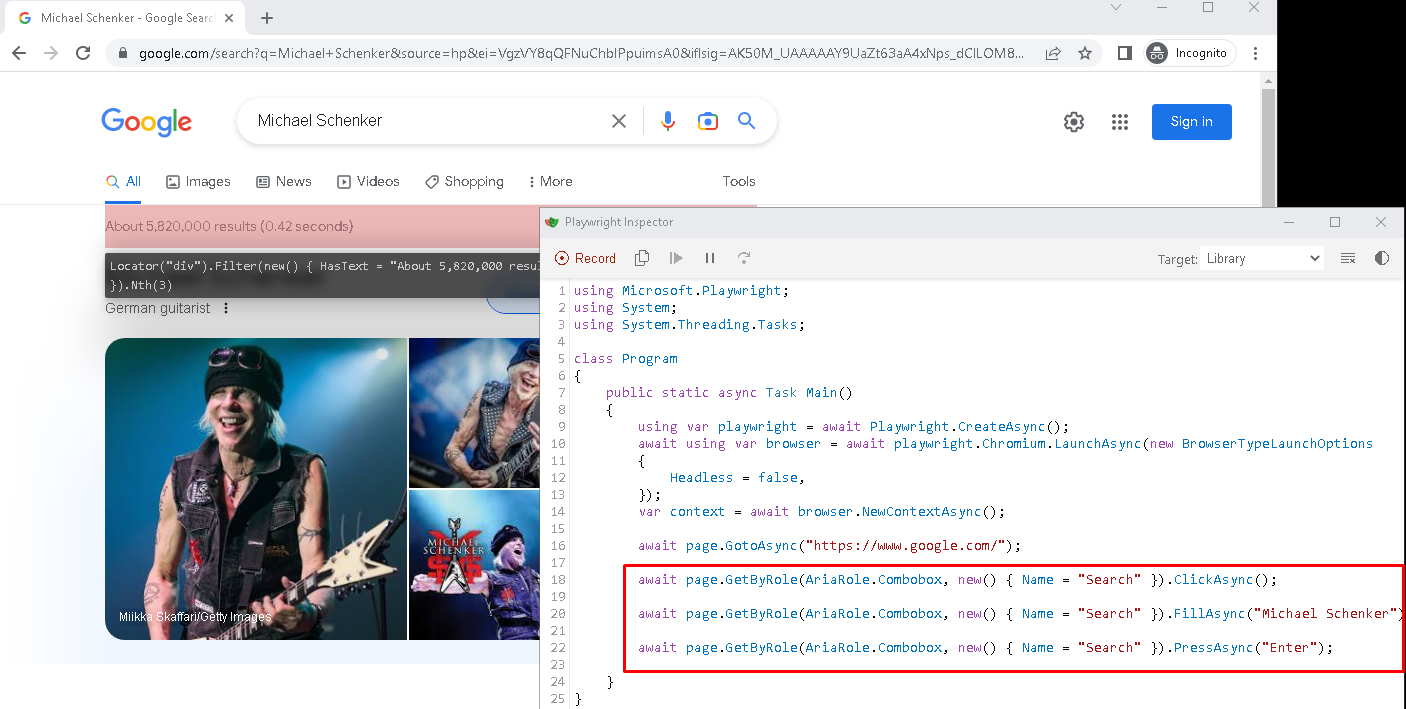


Everything that you click on the web page will be recorded and what you enter. You will also notice that Playwright will provide you with visual hints on the locators that you hover over or click into.

If I hover over Google’s search box I get the following, informing me of the element name\id, which I can use later to generate manual tests.

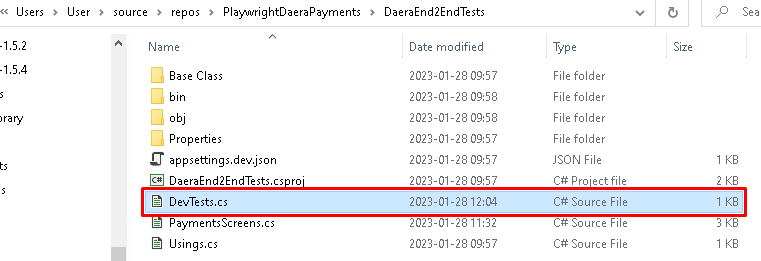


When I perform a search, it will record what I have entered.

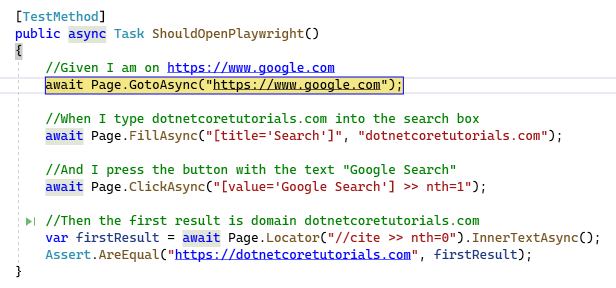


Then I can copy the steps to the clipboard and into my MSTests project, and create my assertions (or navigate more before generating my tests).

Once you have exited the Test Generator window, the file will be created for you to use.



# Debugging Your Playwright (Headless) Tests



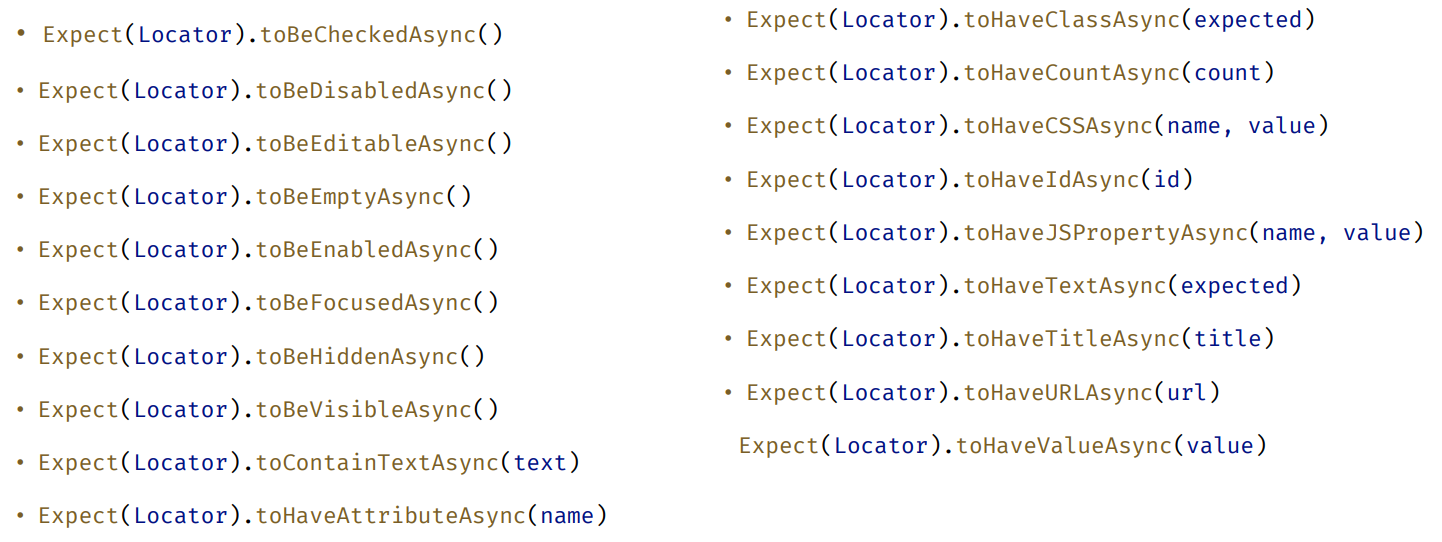
# Misc

Chrome browser is the default, but this an be changed in the Config file.

Playwright recommends to use their assertions, since they retry until either the timeout is reached or the condition is met.

Normal assertions like Assert.Equal don't retry.

https://playwright.dev/dotnet/docs/test-assertions#locator-assertions-to-have-text



Config settings

### Running MSTest tests in Parallel[​](https://playwright.dev/dotnet/docs/test-runners#running-mstest-tests-in-parallel)

By default MSTest will run all classes in parallel, while running tests inside each class sequentially (ExecutionScope.ClassLevel). It will create as many processes as there are cores on the host system. You can adjust this behavior by using the following CLI parameter or using a .runsettings file, see below. Running tests in parallel at the method level (ExecutionScope.MethodLevel) is not supported.

dotnet test --settings:.runsettings -- MSTest.Parallelize.Workers=4

### Customizing [BrowserContext](https://playwright.dev/dotnet/docs/next/api/class-browsercontext" \o "BrowserContext) options[​](https://playwright.dev/dotnet/docs/next/test-runners#customizing-browsercontext-options-1)

To customize context options, you can override the ContextOptions method of your test class derived from Microsoft.Playwright.MSTest.PageTest or Microsoft.Playwright.MSTest.ContextTest. See the following example:

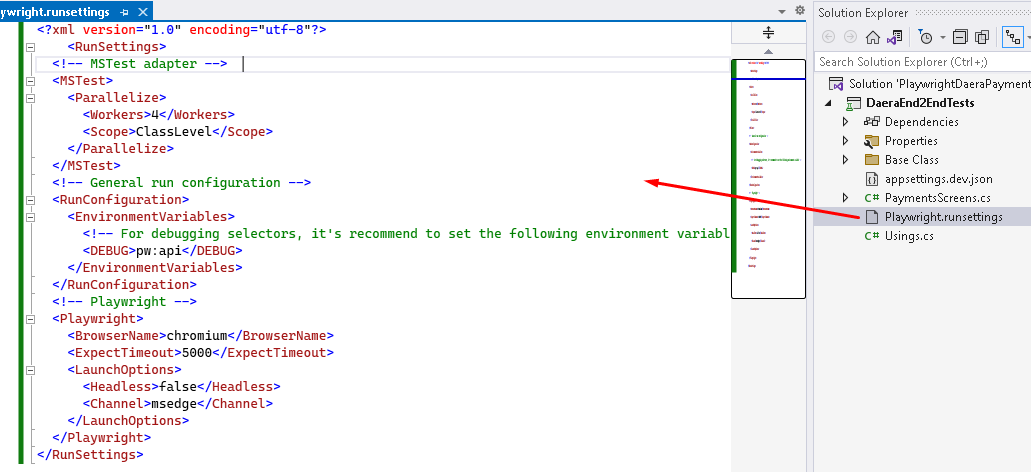
using System.Threading.Tasks;  
using Microsoft.Playwright;  
using Microsoft.Playwright.MSTest;  
using Microsoft.VisualStudio.TestTools.UnitTesting;  
  
namespace PlaywrightTests;  
  
[TestClass]  
public class UnitTest1 : PageTest  
{  
 [TestMethod]  
 public async Task TestWithCustomContextOptions()  
 {  
 *// The following Page (and BrowserContext) instance has the custom colorScheme, viewport and baseURL set:*  
 await Page.GotoAsync("/login");  
 }  
  
 public override BrowserNewContextOptions ContextOptions()  
 {  
 return new BrowserNewContextOptions()  
 {  
 ColorScheme = ColorScheme.Light,  
 ViewportSize = new()  
 {  
 Width = 1920,  
 Height = 1080  
 },  
 BaseURL = "https://github.com",  
 };  
 }  
}

### Using the .runsettings file[​](https://playwright.dev/dotnet/docs/next/test-runners#using-the-runsettings-file-1)

When running tests from Visual Studio, you can take advantage of the .runsettings file. The following shows a reference of the supported values.

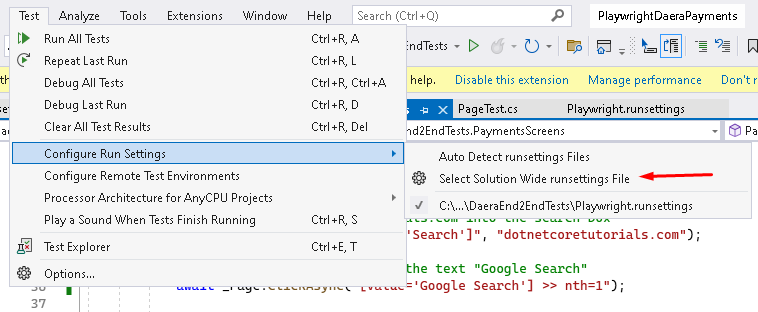
For example, to specify the number of workers, you can use MSTest.Parallelize.Workers. You can also enable DEBUG logs using RunConfiguration.EnvironmentVariables.

<RunSettings>  
 *<!-- MSTest adapter -->*   
 <MSTest>  
 <Parallelize>  
 <Workers>4</Workers>  
 <Scope>ClassLevel</Scope>  
 </Parallelize>  
 </MSTest>  
 *<!-- General run configuration -->*  
 <RunConfiguration>  
 <EnvironmentVariables>  
 *<!-- For debugging selectors, it's recommend to set the following environment variable -->*  
 <DEBUG>pw:api</DEBUG>  
 </EnvironmentVariables>  
 </RunConfiguration>  
 *<!-- Playwright -->*   
 <Playwright>  
 <BrowserName>chromium</BrowserName>  
 <ExpectTimeout>5000</ExpectTimeout>  
 <LaunchOptions>  
 <Headless>false</Headless>  
 <Channel>msedge</Channel>  
 </LaunchOptions>  
 </Playwright>  
</RunSettings>



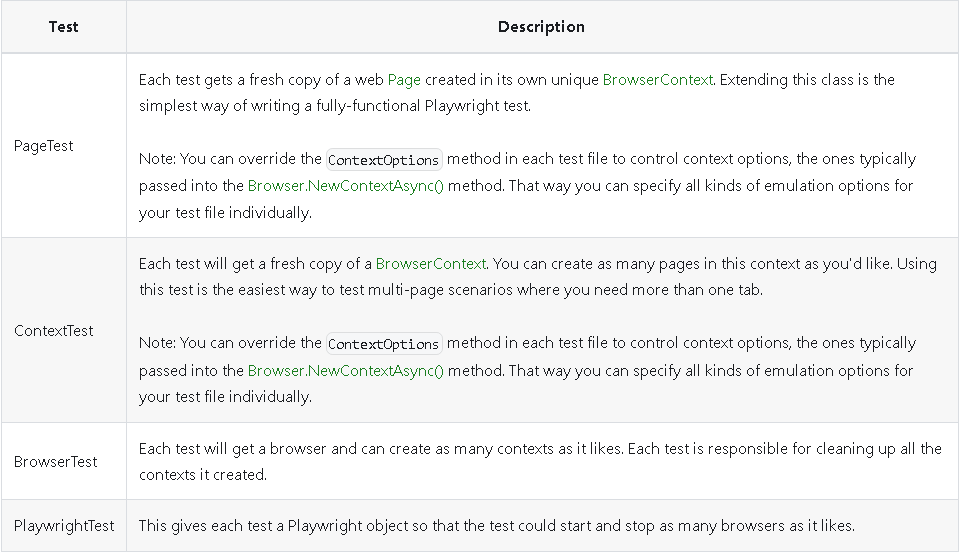
## Autodetect the run settings file

Select **Test** > **Configure Run Settings** > **Auto Detect runsettings Files**



### Base MSTest classes for Playwright[​](https://playwright.dev/dotnet/docs/next/test-runners#base-mstest-classes-for-playwright)

There are a few base classes available to you in the Microsoft.Playwright.MSTest namespace:



|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Background

<https://www.lambdatest.com/playwright>

### Competitors:

You may be weighing the pros and cons of migrating from Selenium to Cypress, and also gaining popularity in the community.

Playwright testing stands out from its competitors because it outranks other JavaScript libraries because of its intuitive syntax and the flexibility with which it can interact with the browser across multiple pages and domains. Lastly, the execution speed of test scripts on Playwright tops other automation frameworks. The [ChecklyHQ](https://blog.checklyhq.com/cypress-vs-selenium-vs-playwright-vs-puppeteer-speed-comparison/" \t "_blank) team created this benchmark report after performing in-depth execution speed monitoring of the major automation tools.

### Integrations:

Playwright comes with native integration. For example, the Playwright has Docker images, allowing you to run tests quickly in an isolated and controlled environment. Native integrations are available for the [best CI/CD tools](https://www.lambdatest.com/blog/best-ci-cd-tools/), including GitHub Actions, Azure Pipelines, CircleCI, Jenkins, and GitLab. They also support your existing JavaScript test runners, like Jest/Jasmine, AVA, and Mocha, which is helpful if you are porting from an existing code base.

Lastly, Playwright has direct integration with [Selenium grid online.](https://www.lambdatest.com/selenium-grid-online) This is essential for running larger suites of tests at scale and managing parallel execution through Selenium Grid.

## Advantages of Playwright

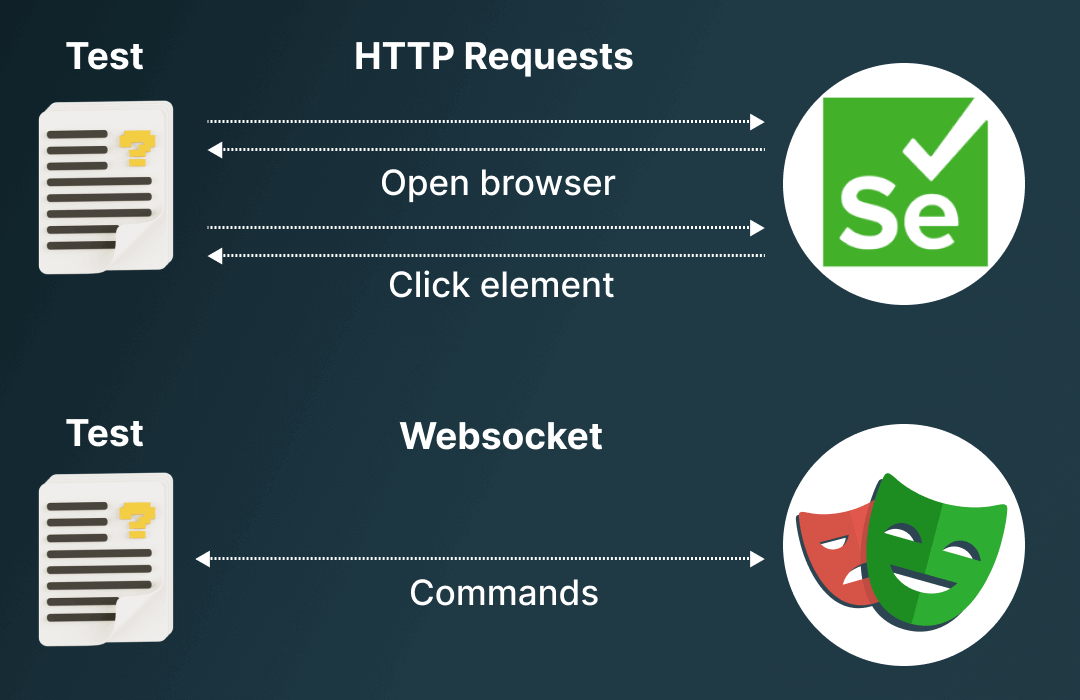
The Playwright framework helps you perform cross browser testing on sophisticated web applications. It provides accurate results and offers excellent test coverage. In addition, it has multiple advantages to automate your web testing. Some of them are as follows.

* Supports testing scenarios for multi-tab, multi-user, and iframes.
* It's available as a VS Code extension to run tests in a single click and comes with features for step-by-step debugging, exploring selectors, and recording new tests.
* Generate an HTML report to view test execution results in the browser. It contains visual mismatches and test artifacts like screenshots, traces, error logs, and video recordings.
* Installing Playwright doesn't take long, as it requires a configuration. However, the installation process can differ based on your programming language with Playwright to run the tests.
* It supports different types of testing like end-to-end, functional, and API testing.
* Offers support for automated accessibility testing using a third-party plugin.
* It provides different debugging options like Playwright Inspector, Browser Developer Tools, VSCode Debugger, and Trace Viewers Console Logs.
* It has in-built reporters like JSON, JUnit, and HTML Reporters. With Playwright, you can also create custom reporters.
* Run parallel test execution locally or on an online Selenium grid. You can also share tests between systems to run multiple tests in parallel.

## Playwright Architecture

To understand how Playwright's architecture works, we will compare its work with that of Selenium. Selenium sends each command as an independent HTTP request and receives JSON responses. Each interaction, such as opening a browser window, clicking an element, or entering text into an input box, then is sent as a separate HTTP request.

This means we have to wait longer for responses and increases the chances of errors.



Instead of communicating with each driver through a separate WebSocket connection, Playwright relies on a single WebSocket connection to communicate with all drivers, which remains in place until testing is finished. This allows commands to be sent quickly on a single connection, thus reducing the points of failure.

# Trace Viewer

A massive selling point in using a automated test runner such as Cypress.io, is that it can record videos and take screenshots of all of your test runs right out of the box. It comes with a pretty nifty viewer too that allows you to group tests by failures, and then view the actual video of the test being executed.

If we compare that to Selenium, I mean.. Selenium simply does not have that feature. It’s not to say that it can’t be done, you just don’t get it out of the box. In most cases, I’ve had automation testers simply take a screenshot on test failure and that’s it. Often the final screenshot of a failed step is enough to debug what went wrong, but not always. Additionally, there is no inbuilt tool for “viewing” these screenshots, and while MS Paint is enough to open a simple [image file](https://dotnetcoretutorials.com/2021/08/17/reading-raw-image-files-in-net/), it can get confusing managing a bunch of screenshots in your downloads folder!

[Playwright](https://playwright.dev/) is somewhere in the middle of these. While it doesn’t record actual videos, it can take screenshots of each step along the way taking before and after shots, and it provides a great viewer to pinpoint exactly went wrong. While it works out of the box, there is a tiny bit of configuration required.

I’m going to use our example from our [previous post](https://dotnetcoretutorials.com/2022/05/20/using-playwright-e2e-tests-with-c-net/) which uses MSTest, and add to it a little. However, the steps are largely the same if you are using another testing framework or no framework at all.

The full “traceable” MSTest code looks like so :

[TestClass]

public class MyUnitTests : PageTest

{

[TestInitialize]

public async Task TestSetup()

{

await Context.Tracing.StartAsync(new TracingStartOptions

{

Title = TestContext.TestName, //Note this is for MSTest only.

Screenshots = true,

Snapshots = true,

Sources = true

});

}

[TestCleanup]

public async Task TestCleanup()

{

await Context.Tracing.StopAsync(new TracingStopOptions

{

Path = TestContext.TestName + ".zip"

});

}

[TestMethod]

public async Task WhenDotNetCoreTutorialsSearchedOnGoogle\_FirstResultIsDomainDotNetCoreTutorialsDotCom()

{

//Our Test Code here. Removed for brevity.

}

}

Quite simply :

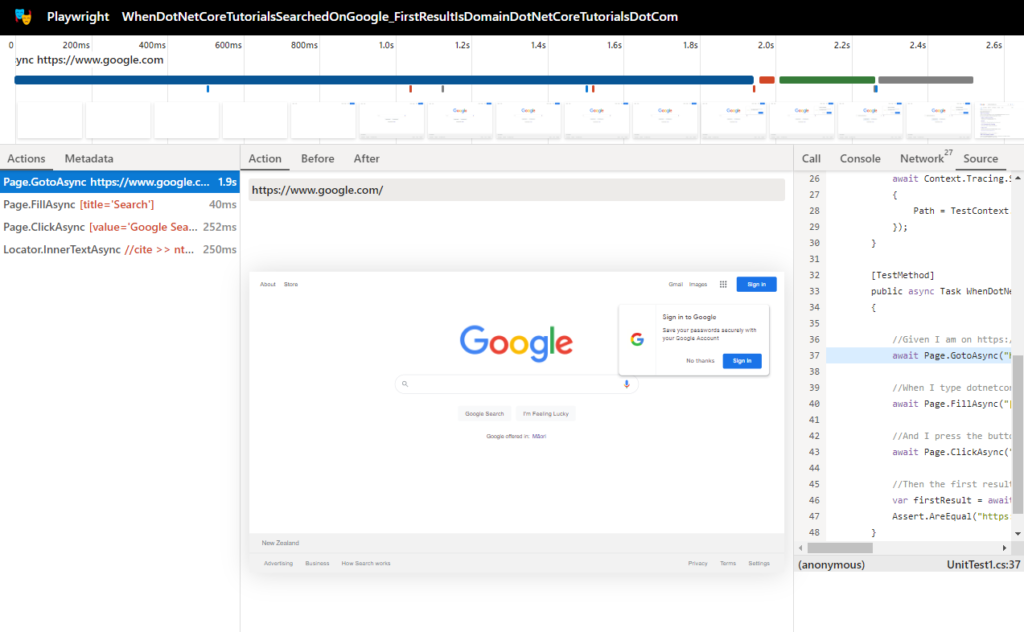
* Our TestInitialize method kicks off the tracing for us. The “TestContext” object is a MSTest specific class that can tell us which test is under execution, you can swap this out for a similar class in your test framework or just put any old string in there.
* Our TestCleanup essentially ends the trace, storing the results in a .zip file.

And that’s it!

In our bin folder, there will now be a zip file for each of our tests. Should one fail, we can go in here and retrieve the zip. Unlike Cypress, there isn’t an all encompassing viewer where we can group tests, their results and videos. This is because Playwright for .NET is relying a bit on both MSTest and Visual Studio to be test runners, and so there is a bit of a break in tooling when you then want to view traces, but it’s not that much leg work.

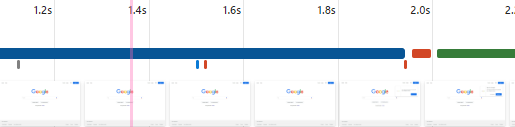
Let’s say our test broke, and we have retrieved the zip. What do we do with it? While you can download a trace viewer locally, I prefer to use Playwright’s hosted version right here <https://trace.playwright.dev/>

We simply drag and drop our zip file and tada!

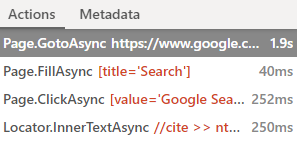
[](https://dotnetcoretutorials.com/wp-content/uploads/2022/05/PlaywrightTrace.png)

I know that’s a lot to take in, so let’s walk through it bit by bit!

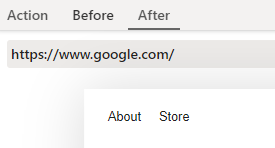
Along the top of the page, we have the timeline. This tells us over time how our test ran and the screenshots for each time period. The color coding tells us when actions changed/occurred so we can immediately jump to a certain point in our test.

[](https://dotnetcoretutorials.com/wp-content/uploads/2022/05/PlaywrightTraceTimeline.png)

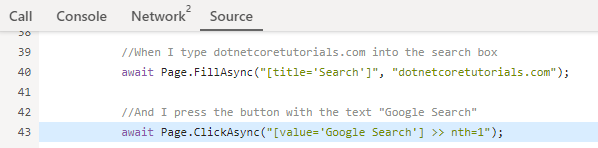
To the left of the screen, we have our executed steps, we can click on these to immediately jump in the timeline.

[](https://dotnetcoretutorials.com/wp-content/uploads/2022/05/PlaywrightTraceSteps.png)

In the centre of the page we have our screenshot. But importantly we can switch tabs for a “Before” and “After” view. This is insanely handy when inputting text or filling out other page inputs. Imagine that the test is trying to fill out a very large form, and on the very first step it doesn’t fill in a textbox correctly. The test may not fail until the form is submitted and validation occurs, but in the screenshot of the failure, we may not even be able to see the textbox itself. So this gives us a step by step view of every step occurring as it happens.

[](https://dotnetcoretutorials.com/wp-content/uploads/2022/05/PlaywrightTraceBeforeAfter.png)

To the right of the screen you’ve got a bunch of debug information including Playwright’s debug output, the console output of the browser, the network output of the browser (Similar to Chrome/Firefox dev tools), but importantly, you’ve also got a snapshot of your own code and which step is running. For instance, here I am looking at the step to fill a textbox.

[](https://dotnetcoretutorials.com/wp-content/uploads/2022/05/PlaywrightTraceSource.png)

This is **\*insanely\*** helpful. If a test fails, we can essentially know exactly where in our own code it was up to and what it was trying to do, without thinking “Well, it was trying to type text into a textbox, let me try and find where that happens in my code”.

And that’s the Playwright Test Trace Viewer. Is it as good as Cypress’ offering? Probably not quite yet. I would love to see some ability to capture a single zip for an entire test run, not per test case (And if I’ve missed that, please let me know!), but for debugging a single test failure, I think the trace viewer is crazy powerful and yet another reason to give Playwright a try if you’re currently dabbling with Selenium.

## Locators

# Locator

Locator represents a view to the element(s) on the page. It captures the logic sufficient to retrieve the element at any given moment. Locator can be created with the [page.locator(selector)](http://www.cuketest.com/playwright/docs/api/class-page" \l "page-locator) method.

*const* locator = page.locator('text=Submit');*await* locator.click();

Copy

The difference between the Locator and [ElementHandle](http://www.cuketest.com/playwright/docs/api/class-elementhandle" \o "ElementHandle) is that the latter points to a particular element, while Locator captures the logic of how to retrieve that element.

In the example below, handle points to a particular DOM element on page. If that element changes text or is used by React to render an entirely different component, handle is still pointing to that very DOM element. This can lead to unexpected behaviors.

*const* handle = *await* page.$('text=Submit');*// ...await* handle.hover();*await* handle.click();

Copy

With the locator, every time the element is used, up-to-date DOM element is located in the page using the selector. So in the snippet below, underlying DOM element is going to be located twice.

*const* locator = page.locator('text=Submit');*// ...await* locator.hover();*await* locator.click();

Copy

**Strictness**

Locators are strict. This means that all operations on locators that imply some target DOM element will throw if more than one element matches given selector.

*// Throws if there are several buttons in DOM:await* page.locator('button').click();

*// Works because we explicitly tell locator to pick the first element:await* page.locator('button').first().click();

*// Works because count knows what to do with multiple matches:await* page.locator('button').count();

Copy

* [locator.allInnerTexts()](http://www.cuketest.com/playwright/docs/api/class-locator#locator-all-inner-texts)
* [locator.allTextContents()](http://www.cuketest.com/playwright/docs/api/class-locator#locator-all-text-contents)
* [locator.boundingBox([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-bounding-box)
* [locator.check([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-check)
* [locator.click([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-click)
* [locator.count()](http://www.cuketest.com/playwright/docs/api/class-locator#locator-count)
* [locator.dblclick([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-dblclick)
* [locator.dispatchEvent(type[, eventInit, options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-dispatch-event)
* [locator.elementHandle([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-element-handle)
* [locator.elementHandles()](http://www.cuketest.com/playwright/docs/api/class-locator#locator-element-handles)
* [locator.evaluate(pageFunction[, arg, options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-evaluate)
* [locator.evaluateAll(pageFunction[, arg])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-evaluate-all)
* [locator.evaluateHandle(pageFunction[, arg, options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-evaluate-handle)
* [locator.fill(value[, options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-fill)
* [locator.first()](http://www.cuketest.com/playwright/docs/api/class-locator#locator-first)
* [locator.focus([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-focus)
* [locator.getAttribute(name[, options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-get-attribute)
* [locator.hover([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-hover)
* [locator.innerHTML([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-inner-html)
* [locator.innerText([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-inner-text)
* [locator.inputValue([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-input-value)
* [locator.isChecked([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-is-checked)
* [locator.isDisabled([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-is-disabled)
* [locator.isEditable([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-is-editable)
* [locator.isEnabled([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-is-enabled)
* [locator.isHidden([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-is-hidden)
* [locator.isVisible([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-is-visible)
* [locator.last()](http://www.cuketest.com/playwright/docs/api/class-locator#locator-last)
* [locator.locator(selector)](http://www.cuketest.com/playwright/docs/api/class-locator#locator-locator)
* [locator.nth(index)](http://www.cuketest.com/playwright/docs/api/class-locator#locator-nth)
* [locator.press(key[, options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-press)
* [locator.screenshot([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-screenshot)
* [locator.scrollIntoViewIfNeeded([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-scroll-into-view-if-needed)
* [locator.selectOption(values[, options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-select-option)
* [locator.selectText([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-select-text)
* [locator.setChecked(checked[, options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-set-checked)
* [locator.setInputFiles(files[, options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-set-input-files)
* [locator.tap([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-tap)
* [locator.textContent([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-text-content)
* [locator.type(text[, options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-type)
* [locator.uncheck([options])](http://www.cuketest.com/playwright/docs/api/class-locator#locator-uncheck)

# Assertions With Expects

# Assertions

Playwright Test uses [expect](https://jestjs.io/docs/expect) library for test assertions. This library provides a lot of matchers like toEqual, toContain, toMatch, toMatchSnapshot and many more:

expect(success).toBeTruthy();

Playwright also extends it with convenience async matchers that will wait until the expected condition is met. Consider the following example:

await expect(page.getByTestId('status')).toHaveText('Submitted');

Playwright Test will be re-testing the element with the test id of status until the fetched element has the "Submitted" text. It will re-fetch the element and check it over and over, until the condition is met or until the timeout is reached. You can either pass this timeout or configure it once via the [testConfig.expect](https://playwright.dev/docs/next/api/class-testconfig" \l "test-config-expect) value in the test config.

By default, the timeout for assertions is set to 5 seconds. Learn more about [various timeouts](https://playwright.dev/docs/next/test-timeouts).

## List of assertions[​](https://playwright.dev/docs/next/test-assertions#list-of-assertions)

| **Assertion** | **Description** |
| --- | --- |
| [expect(locator).toBeChecked()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-be-checked) | Checkbox is checked |
| [expect(locator).toBeDisabled()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-be-disabled) | Element is disabled |
| [expect(locator).toBeEditable()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-be-editable) | Element is enabled |
| [expect(locator).toBeEmpty()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-be-empty) | Container is empty |
| [expect(locator).toBeEnabled()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-be-enabled) | Element is enabled |
| [expect(locator).toBeFocused()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-be-focused) | Element is focused |
| [expect(locator).toBeHidden()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-be-hidden) | Element is not visible |
| [expect(locator).toBeVisible()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-be-visible) | Element is visible |
| [expect(locator).toContainText()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-contain-text) | Element contains text |
| [expect(locator).toHaveAttribute()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-have-attribute) | Element has a DOM attribute |
| [expect(locator).toHaveClass()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-have-class) | Element has a class property |
| [expect(locator).toHaveCount()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-have-count) | List has exact number of children |
| [expect(locator).toHaveCSS()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-have-css) | Element has CSS property |
| [expect(locator).toHaveId()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-have-id) | Element has an ID |
| [expect(locator).toHaveJSProperty()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-have-js-property) | Element has a JavaScript property |
| [expect(locator).toHaveScreenshot()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-have-screenshot-1) | Element has a screenshot |
| [expect(locator).toHaveText()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-have-text) | Element matches text |
| [expect(locator).toHaveValue()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-have-value) | Input has a value |
| [expect(locator).toHaveValues()](https://playwright.dev/docs/next/api/class-locatorassertions#locator-assertions-to-have-values) | Select has options selected |
| [expect(page).toHaveScreenshot()](https://playwright.dev/docs/next/api/class-pageassertions#page-assertions-to-have-screenshot-1) | Page has a screenshot |
| [expect(page).toHaveTitle()](https://playwright.dev/docs/next/api/class-pageassertions#page-assertions-to-have-title) | Page has a title |
| [expect(page).toHaveURL()](https://playwright.dev/docs/next/api/class-pageassertions#page-assertions-to-have-url) | Page has a URL |
| [expect(apiResponse).toBeOK()](https://playwright.dev/docs/next/api/class-apiresponseassertions#api-response-assertions-to-be-ok) | Response has an OK status |

## Negating Matchers[​](https://playwright.dev/docs/next/test-assertions#negating-matchers)

In general, we can expect the opposite to be true by adding a .not to the front of the matchers:

expect(value).not.toEqual(0);  
await expect(locator).not.toContainText("some text");

## Soft Assertions[​](https://playwright.dev/docs/next/test-assertions#soft-assertions)

By default, failed assertion will terminate test execution. Playwright also supports soft assertions: failed soft assertions **do not** terminate test execution, but mark the test as failed.

*// Make a few checks that will not stop the test when failed...*  
await expect.soft(page.getByTestId('status')).toHaveText('Success');  
await expect.soft(page.getByTestId('eta')).toHaveText('1 day');  
  
*// ... and continue the test to check more things.*  
await page.getByRole('link', { name: 'next page' }).click();  
await expect.soft(page.getByRole('heading', { name: 'Make another order' })).toBeVisible();

At any point during test execution, you can check whether there were any soft assertion failures:

*// Make a few checks that will not stop the test when failed...*  
await expect.soft(page.getByTestId('status')).toHaveText('Success');  
await expect.soft(page.getByTestId('eta')).toHaveText('1 day');  
  
*// Avoid running further if there were soft assertion failures.*  
expect(test.info().errors).toHaveLength(0);

Note that soft assertions only work with Playwright test runner.

## Custom Expect Message[​](https://playwright.dev/docs/next/test-assertions#custom-expect-message)

You can specify a custom error message as a second argument to the expect function, for example:

await expect(page.getByText('Name'), 'should be logged in').toBeVisible();

The error would look like this:

Error: should be logged in  
  
 Call log:  
 - expect.toBeVisible with timeout 5000ms  
 - waiting for "getByText('Name')"  
  
  
 2 |  
 3 | test('example test', async({ page }) => {  
 > 4 | await expect(page.getByText('Name'), 'should be logged in').toBeVisible();  
 | ^  
 5 | });  
 6 |

The same works with soft assertions:

expect.soft(value, 'my soft assertion').toBe(56);

## Polling[​](https://playwright.dev/docs/next/test-assertions#polling)

You can convert any synchronous expect to an asynchronous polling one using expect.poll.

The following method will poll given function until it returns HTTP status 200:

await expect.poll(async () => {  
 const response = await page.request.get('https://api.example.com');  
 return response.status();  
}, {  
 *// Custom error message, optional.*  
 message: 'make sure API eventually succeeds', *// custom error message*  
 *// Poll for 10 seconds; defaults to 5 seconds. Pass 0 to disable timeout.*  
 timeout: 10000,  
}).toBe(200);

You can also specify custom polling intervals:

await expect.poll(async () => {  
 const response = await page.request.get('https://api.example.com');  
 return response.status();  
}, {  
 *// Probe, wait 1s, probe, wait 2s, probe, wait 10s, probe, wait 10s, probe, .... Defaults to [100, 250, 500, 1000].*  
 intervals: [1\_000, 2\_000, 10\_000],  
 timeout: 60\_000  
}).toBe(200);

## Retrying[​](https://playwright.dev/docs/next/test-assertions#retrying)

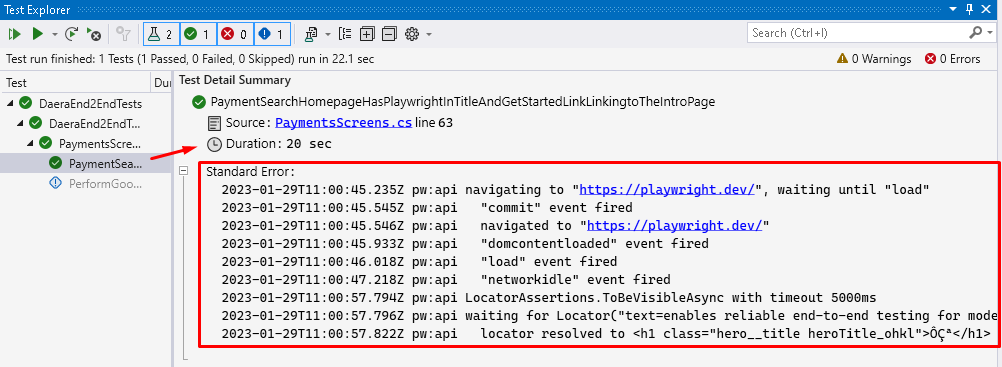
You can retry blocks of code until they are passing successfully.

await expect(async () => {  
 const response = await page.request.get('https://api.example.com');  
 expect(response.status()).toBe(200);  
}).toPass();

You can also specify custom timeout for retry intervals:

await expect(async () => {  
 const response = await page.request.get('https://api.example.com');  
 expect(response.status()).toBe(200);  
}).toPass({  
 *// Probe, wait 1s, probe, wait 2s, probe, wait 10s, probe, wait 10s, probe, .... Defaults to [100, 250, 500, 1000].*  
 intervals: [1\_000, 2\_000, 10\_000],  
 timeout: 60\_000  
});

# Event Logs



Header = false (CI pipeline)