Playwright

Reliable end-to-end testing for modern web apps

Nicola Iarocci MVP

“**Most companies do not have automated end-to-end tests** running as part of their CI process. We are at a point where most now have unit tests (yay!) but few have automated end-to-end tests”

Source: https://bit.ly/3BAsBmZ (testim.io)

“**Most companies we polled (85% out of 284) perform manual end-to-end tests** as part of their release process. This makes releasing software significantly slower and more error-prone”

Source: https://bit.ly/3BAsBmZ (testim.io)

open-source

Made by Microsoft with strong industry support (Google, etc.)

github.com/microsoft/playwright

cross-browser

Playwright supports all modern rendering engines including Chromium, WebKit, and Firefox

mobile web

Native mobile emulation of Google Chrome for Android and Mobile Safari

cross-platform

Test on Windows, Linux, and macOS, locally or on CI, headless or headed

cross-language

JavaScript, Python, .NET, Java

One API

The exact same API is available across all supported languages

Demo: Our first application

https://playwright.dev/dotnet/docs/intro#first-project

Create a project

Add Playwright package and install required browsers

```
# Create project
dotnet new console -n PlaywrightDemo
cd PlaywrightDemo

# Add project dependency
dotnet add package Microsoft.Playwright
# Build the project
dotnet build

# Install Playwright CLI tools (you can also use PowerShell)
dotnet tool install --global Microsoft.Playwright.CLI
# Install required browsers
playwright install
```

You can choose to only install certain browsers: ‘playwright install chrome’ will only install chromium

Our first application

Navigate to a web page and take a screenshot

```
using Microsoft.Playwright;
using System.Threading.Tasks;

class Program
{
    public static async Task Main()
    {
        using var playwright = await Playwright.CreateAsync();
        await using var browser = await playwright.Chromium.LaunchAsync();
        var page = await browser.NewPageAsync();
        await page.GotoAsync("https://playwright.dev/dotnet");
        await page.ScreenshotAsync(new PageScreenshotOptions { Path = "screenshot.png" });
    }
}
```

```
dotnet run
```

‘playwright.Firefox.LaunchAsync()’ would launch Firefox instead

Headless or headed

By default Playwright runs the browser in headless mode

```
await playwright.Chromium.LaunchAsync(new BrowserTypeLaunchOptions
{
    Headless = false,
    SlowMo = 50,
});
```

To see the browser UI, pass the Headless = false flag while launching the browser. You can also use slowMo to slow down execution.

Demo: Our first test

https://playwright.dev/dotnet/docs/test-runners

NUnit

You can choose to use NUnit test fixtures that come bundled with Playwright

```
# Create new project.
dotnet new nunit -n PlaywrightTests
cd PlaywrightTests

# Add project dependency
dotnet add package Microsoft.Playwright.NUnit
# Build the project
dotnet build

# Install required browsers
playwright install
```

Install dependencies, build project and download necessary browsers. This is only done once per project.

Test Fixtures

```
using System.Threading.Tasks;
using Microsoft.Playwright.NUnit;
using NUnit.Framework;

namespace PlaywrightTests
{
    [Parallelizable(ParallelScope.Self)]
    public class Tests : PageTest
    {
        [Test]
        public async Task ShouldAdd()
        {
            int result = await Page.EvaluateAsync<int>("() => 7 + 3");
            Assert.AreEqual(10, result);
        }

        [Test]
        public async Task ShouldMultiply()
        {
            int result = await Page.EvaluateAsync<int>("() => 7 * 3");
            Assert.AreEqual(21, result);
        }
    }
}
```

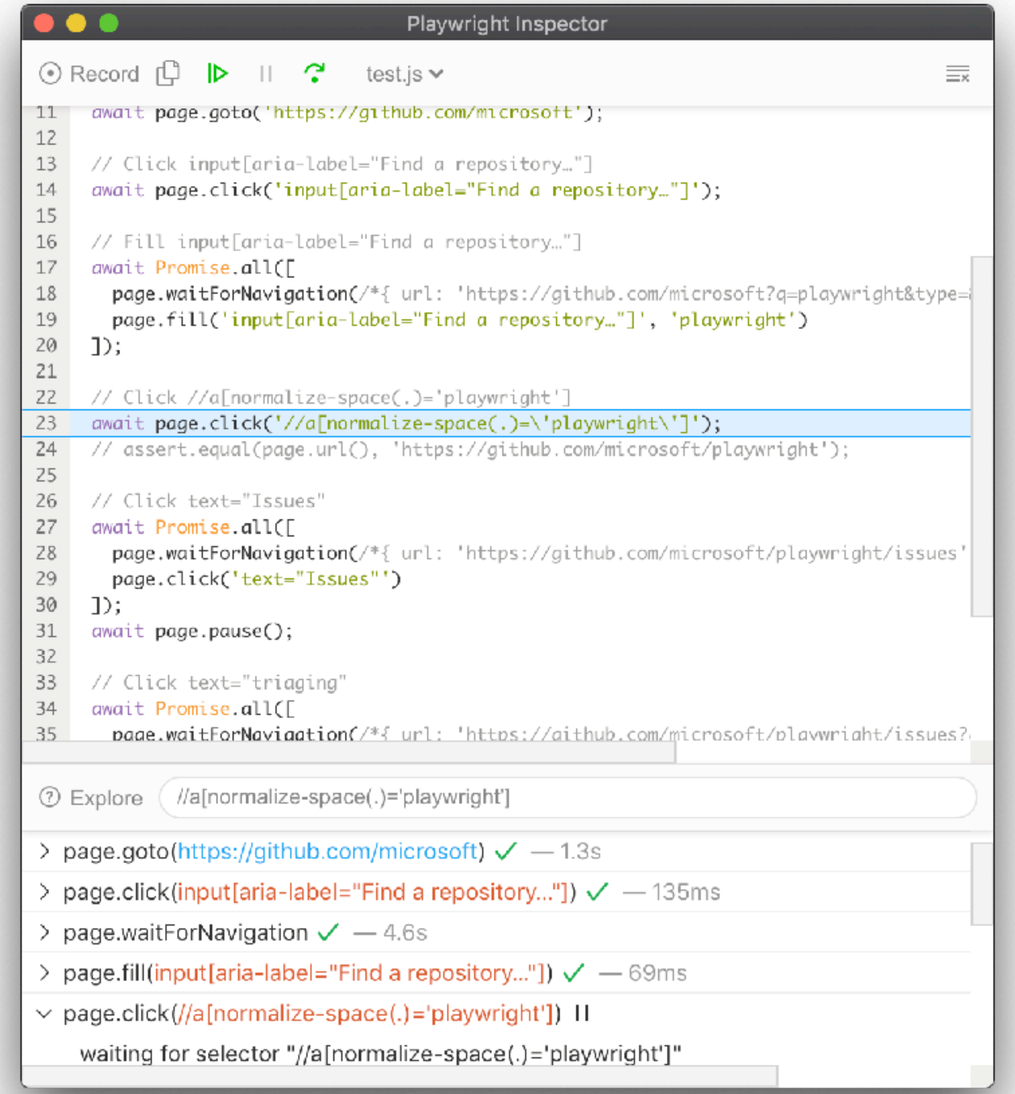
```
dotnet test -- NUnit.NumberOfTestWorkers=5
```

These fixtures support running tests on multiple browser engines in parallel, out of the box.

Inspector

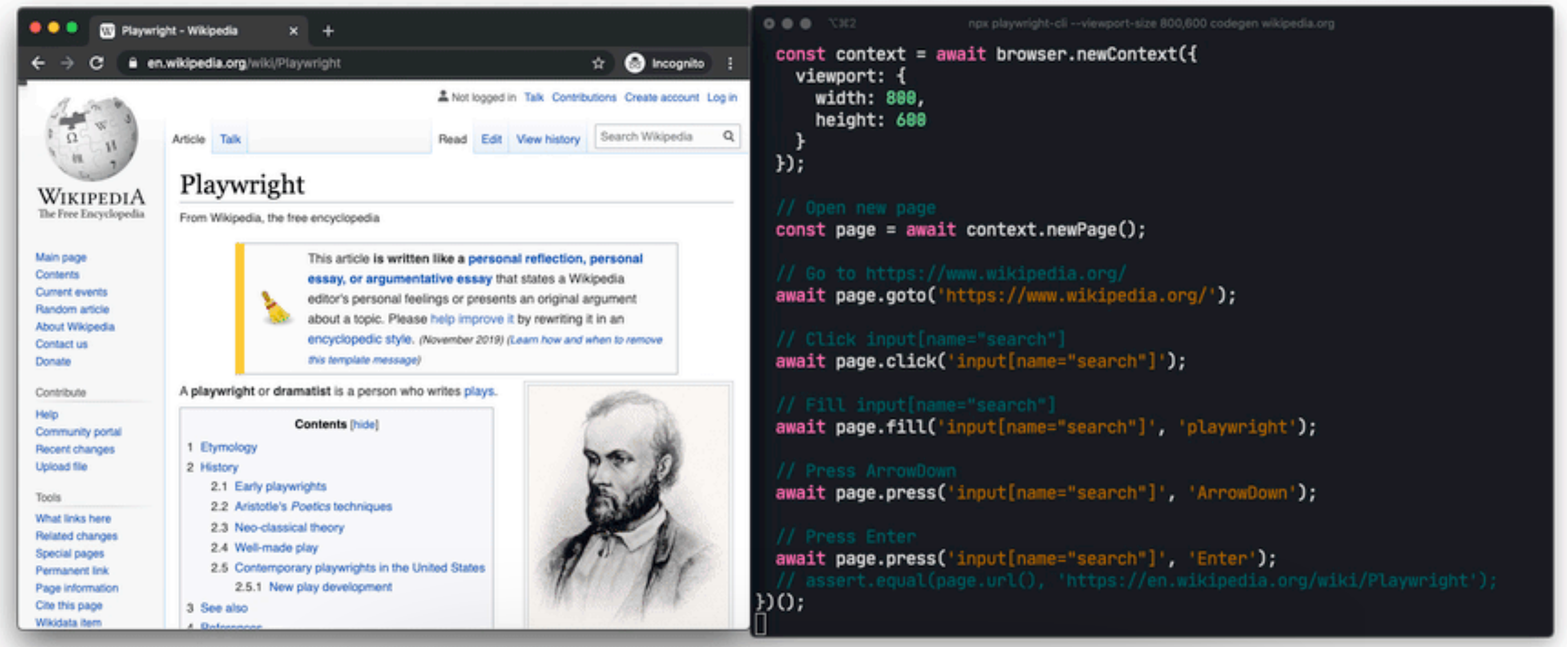
GUI Tool with Superpowers

- Codegen (automatic test authoring)
- Debugging



Codegen

Playwright comes with the ability to generate tests out of the box




```
playwright codegen wikipedia.org
```

Opens the Inspector on the target page and generates code as you interact with the page

Demo: Codegen

Browse a page and see the Inspector generate code for you

Debugging

Set the PWDEBUG environment variable to run your scripts in debug mode

```
PWDEBUG=1 dotnet test
```

This configures Playwright for debugging, opens the inspector, and launches your tests

Debugging

Set the PWDEBUG environment variable to run your scripts in debug mode

```
22  // Click //a[normalize-space(.)='playwright']
23  await page.click('//a[normalize-space(.)=\'playwright\']');
24  // assert.equal(page.url(), 'https://github.com/microsoft/playwright');
```

When PWDEBUG=1 is set, Playwright Inspector window will be opened and the script will be paused on the first Playwright statement

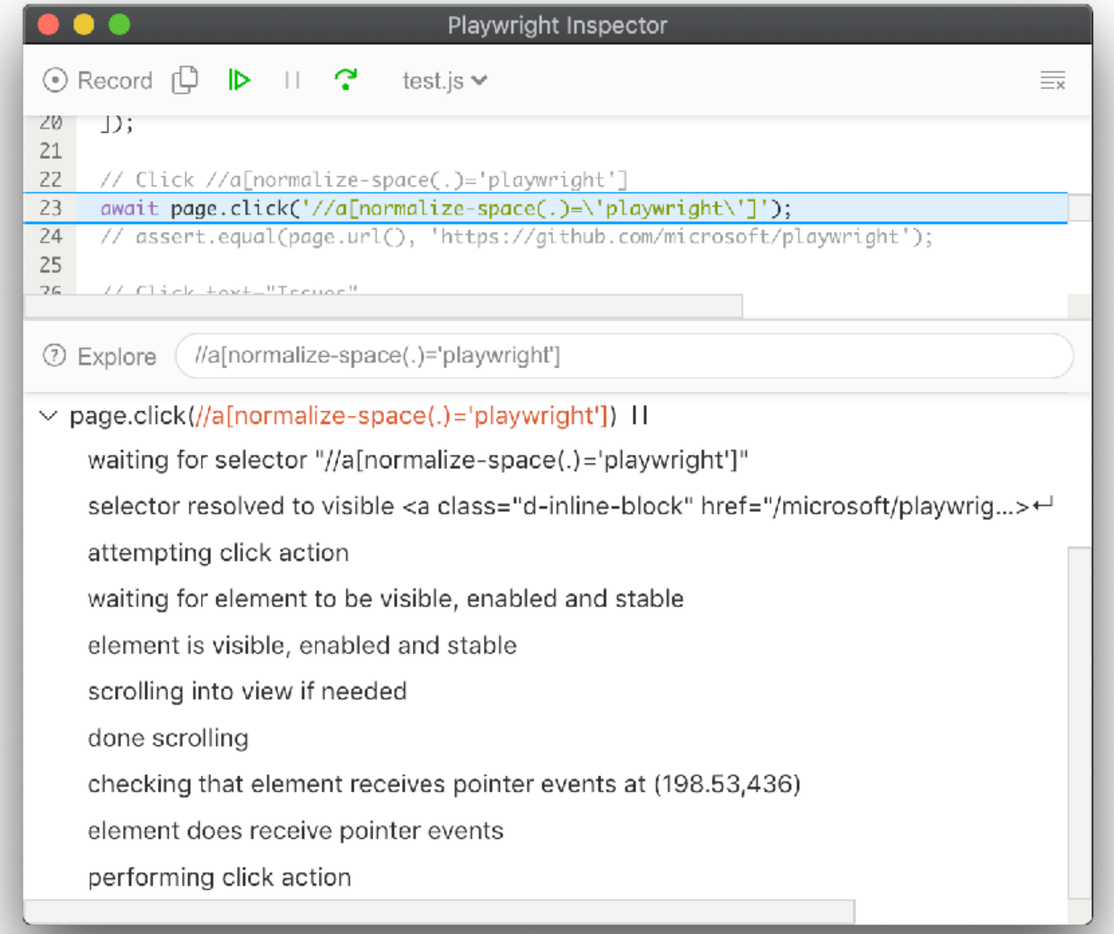
Debugging

Set the PWDEBUG environment variable to run your scripts in debug mode

When stopped on an input action such as `click`, the exact point Playwright is about to click is highlighted with the large red dot on the inspected page

Debugging

Set the PWDEBUG environment variable to run your scripts in debug mode



By the time Playwright has paused on that click action, it has already performed actionability checks that can be found in the log

Demo: Debugging

Use the Inspector to step-by-step debug your test script

Command Line

Preserve authenticated state

```
playwright codegen --save-storage=auth.json
# Perform authentication and exit.
# auth.json will contain the storage state.
```

Run codegen with *--save-storage* to save **cookies** and **localStorage** at the end of the session.
This is useful to separately record authentication step and reuse it later in the tests.

Preserve authenticated state

```
playwright open --load-storage=auth.json my.web.app
playwright codegen --load-storage=auth.json my.web.app
# Perform actions in authenticated state.
```

Run with *--load-storage* to consume previously loaded storage.
This way, all **cookies** and **localStorage** will be restored, bringing most web apps to the authenticated state.

Emulate geolocation, language and timezone

Via codegen and CLI

```
# Emulate timezone, language & location
# Once page opens, click the "my location" button to see geolocation in action
playwright codegen \
    --timezone="Europe/Rome" \
    --geolocation="41.890221,12.492348" \
    --lang="it-IT" maps.google.com
```

Emulate devices

```
# Emulate iPhone 11.
playwright codegen --device="iPhone 11" wikipedia.org
```

You can record scripts and tests while emulating a device.

Emulate color scheme and viewport size

```
# Emulate screen size and color scheme.
playwright codegen --viewport-size=800,600 --color-scheme=dark twitter.com
```

You can record scripts and tests while emulating a device.

API

Preserve authenticated state

```
// Save storage state into the file.
await context.StorageStateAsync(new BrowserContextStorageStateOptions
{
    Path = "state.json"
});

// Create a new context with the saved storage state.
var context = await browser.NewContextAsync(new BrowserNewContextOptions
{
    StorageStatePath = "state.json"
});
```

Authentication

Playwright can be used to automate scenarios that require authentication

```
var page = await context.NewPageAsync();
await page.NavigateAsync("https://github.com/login");
// Interact with login form
await page.ClickAsync("text=Login");
await page.FillAsync("input[name='login']", USERNAME);
await page.FillAsync("input[name='password']", PASSWORD);
await page.ClickAsync("text=Submit");
// Verify app is logged in
```

Emulate geolocation, language and timezone

Programmatically, via customized Browser contexts

```
using Microsoft.Playwright;
using System.Threading.Tasks;

class PlaywrightExample
{
    public static async Task Main()
    {
        using var playwright = await Playwright.CreateAsync();
        await using var browser = await playwright.Webkit.LaunchAsync();
        var options = new BrowserNewContextOptions(playwright.Devices["iPhone 11 Pro"])
        {
            Geolocation = new() { Longitude = 12.492507f, Latitude = 41.889938f },
            Permissions = new[] { "geolocation" },
            Locale = "de-DE"
        };

        await using var context = await browser.NewContextAsync(options);
        var page = await browser.NewPageAsync();
    }
}
```

Assertions

Text content

```
var content = await page.TextContentAsync("nav:first-child");
Assert.AreEqual("home", content);
```

Assertions

Inner text

```
var content = await page.InnerTextAsync(".selected");
Assert.AreEqual("value", content);
```

Assertions

Checkbox values

```
var checked = await page.IsCheckedAsync("input");
Assert.True(checked);
```

Assertions

JS expressions

```
var result = await page.EvaluateAsync<int>("([x, y]) => Promise.resolve(x * y)", new[] { 7, 8 });
Console.WriteLine(result);
```

Assertions

Inner HTML

```
var html = await page.InnerHTMLAsync("div.result");
Assert.AreEqual("<p>Result</p>", html);
```

Assertions

Visibility

```
var visibility = await page.IsVisibleAsync("input");
Assert.True(visibility);
```

Assertions

Enbaled state

```
var enabled = await page.IsEnabledAsync("input");
Assert.True(enabled);
```

Assertions

Custom assertions

```
// Assert local storage value
var userId = await page.EvaluateAsync<string>("() => window.localStorage.getItem('userId')");
Assert.NotNull(userId);

// Assert value for input element
var value = await page.Locator("#search").InputValueAsync();
Assert.AreEqual("query", value);

// Assert computed style
var fontSize = await page.Locator("div").EvalOnSelectorAsync<string>("el =>
window.getComputedStyle(el).fontSize");
Assert.AreEqual("16px", fontSize);

// Assert list length
var length = await page.Locator("li.selected").CountAsync();
Assert.AreEqual(3, length);
```

Browser Contexts

A BrowserContext is an isolated incognito-alike session within a browser instance

```
await using var browser = playwright.Chromium.LaunchAsync();
var context = await browser.NewContextAsync();
var page = await context.NewPageAsync();
```

Browser contexts are fast and cheap to create. If you are using Playwright Test, this happens out of the box for each test. Otherwise, you can create browser contexts manually, like above

Multiple contexts

Playwright can create multiple browser contexts within a single scenario

```
using Microsoft.Playwright;
using System.Threading.Tasks;

class Program
{
    public static async Task Main()
    {
        using var playwright = await Playwright.CreateAsync();
        // Create a Chromium browser instance
        await using var browser = await playwright.Chromium.LaunchAsync();
        await using var userContext = await browser.NewContextAsync();
        await using var adminContext = await browser.NewContextAsync();
        // Create pages and interact with contexts independently.
    }
}
```

This is useful when you want to test for multi-user functionality, like chat. Browser contexts are isolated environments on a single browser instance.

Emulation

Playwright comes with a registry of device parameters for selected mobile devices.

```
using Microsoft.Playwright;
using System.Threading.Tasks;

class Program
{
    public static async Task Main()
    {
        using var playwright = await Playwright.CreateAsync();
        await using var browser = await playwright.Chromium.LaunchAsync(new BrowserTypeLaunchOptions
        {
            Headless: False
        });
        var pixel2 = playwright.Devices["Pixel 2"];
        await using var context = await browser.NewContextAsync(pixel2);
    }
}
```

All pages created in the context above will share the same device parameters.

Events

Waiting for event

```
var waitForRequestTask = page.WaitForRequestAsync("**/*logo*.png");
await page.GotoAsync("https://wikipedia.org");
var request = await waitForRequestTask;
Console.WriteLine(request.Url);
```

Wait for a request with the specified url

Events

Waiting for event

```
var popup = await page.RunAndWaitForPopupAsync(async =>
{
    await page.EvaluateAsync("window.open()");
});
await popup.GotoAsync("https://wikipedia.org");
```

Wait for popup window

Events

Adding/removing event listener

```
page.Request += (_, request) => Console.WriteLine("Request sent: " + request.Url);
void listener(object sender, IRequest request)
{
    Console.WriteLine("Request finished: " + request.Url);
};
page.RequestFinished += listener;
await page.GotoAsync("https://wikipedia.org");

// Remove previously added listener.
page.RequestFinished -= listener;
await page.GotoAsync("https://www.openstreetmap.org/");
```

Sometimes, events happen in random time and instead of waiting for them, they need to be handled. Playwright supports traditional language mechanisms for subscribing and unsubscribing from the events.

Screenshots

```
// Page screenshot
var bytes = await page.ScreenshotAsync();
// Single element screenshot
await page.Locator(".header")
  .ScreenshotAsync(new LocatorScreenshotOptions { Path = "screenshot.png" });
```

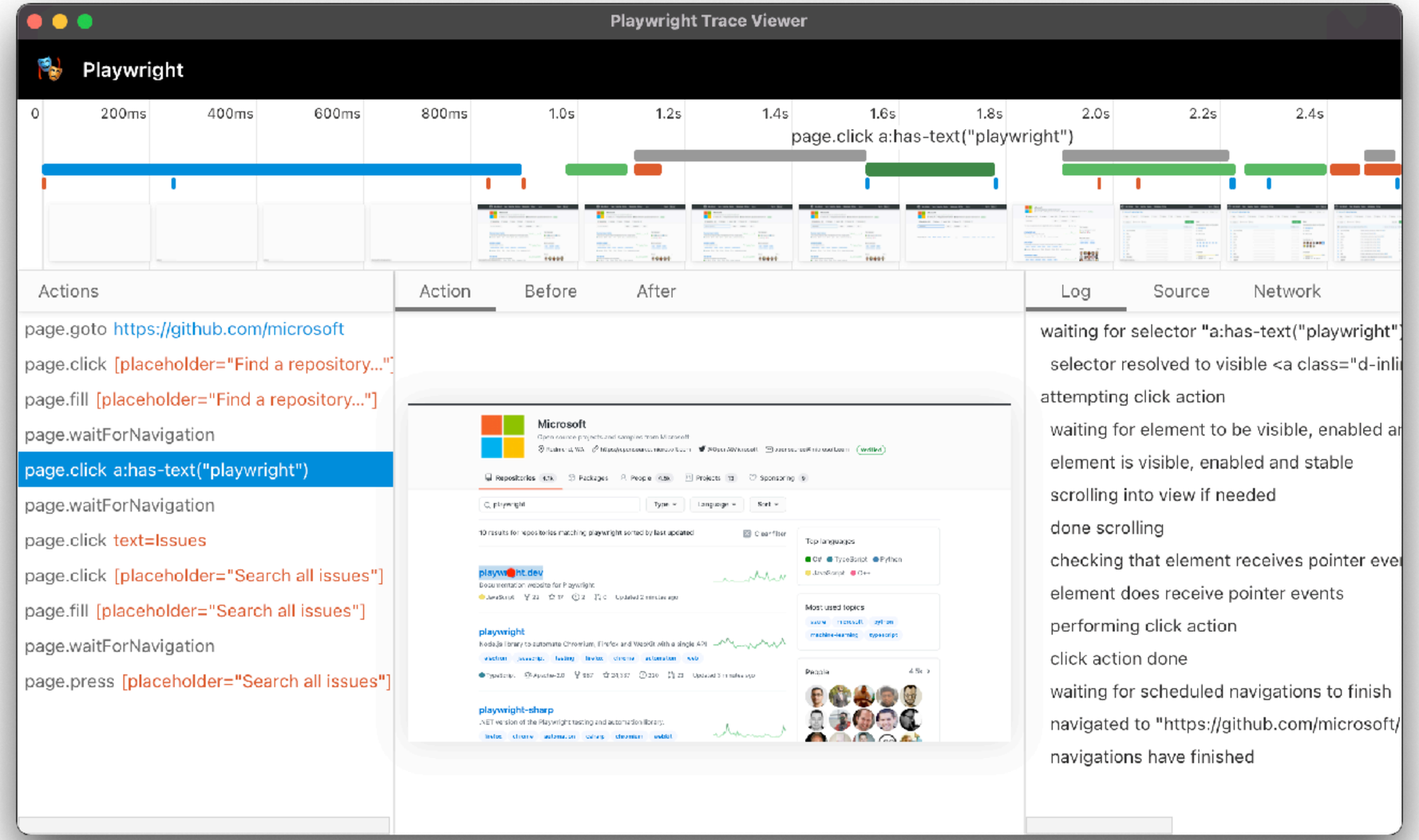
Videos

Playwright can record videos for all pages in a browser context

```
var context = await browser.NewContextAsync(new BrowserNewContextOptions
{
    RecordVideoDir = "videos/",
    RecordVideoSize = new RecordVideoSize() { Width = 640, Height = 480 }
});
// Make sure to close, so that videos are saved.
await context.CloseAsync();
```

Trace Viewer

Playwright Trace Viewer is a GUI tool that helps exploring recorded Playwright traces after the script ran



Recording a trace

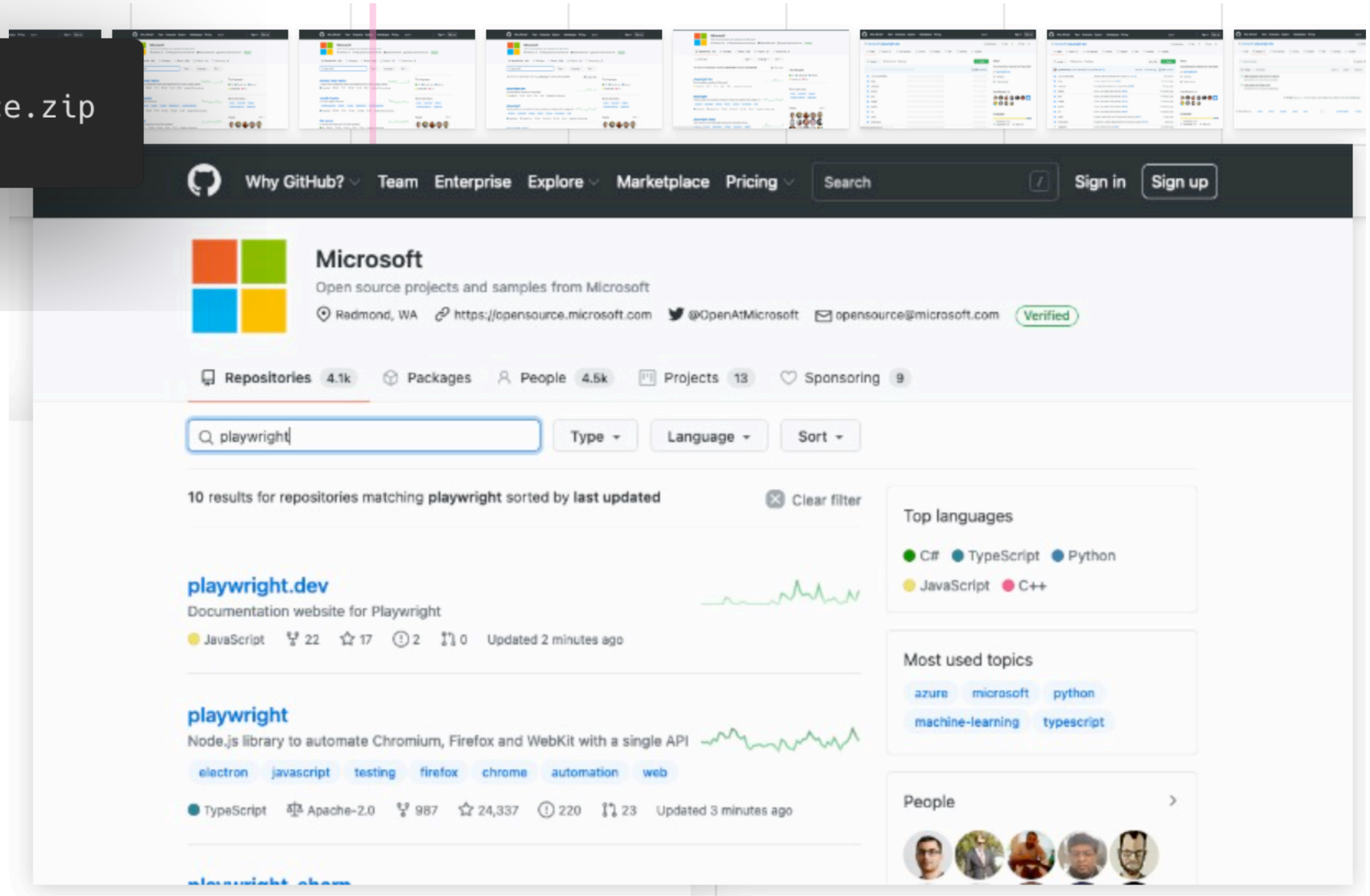
Traces can be recorded using the BrowserContext.Tracing API

```
await using var browser = playwright.Chromium.LaunchAsync();
await using var context = await browser.NewContextAsync();

// Start tracing before creating / navigating a page.
await context.Tracing.StartAsync(new TracingStartOptions
{
  Screenshots = true,
  Snapshots = true
});

var page = context.NewPageAsync();
await page.GotoAsync("https://playwright.dev");

// Stop tracing and export it into a zip archive.
await context.Tracing.StopAsync(new TracingStopOptions
{
  Path = "trace.zip"
});
```

Viewing a trace

You can open the saved trace using Playwright CLI or in your browser on trace.playwright.dev

```
playwright show-trace trace.zip
```



When tracing with the `screenshots` option turned on, each trace records screencast and renders it as a film strip. You can hover over the film strip to see a magnified image.

You can also open remote traces (produced by your CI, for example)

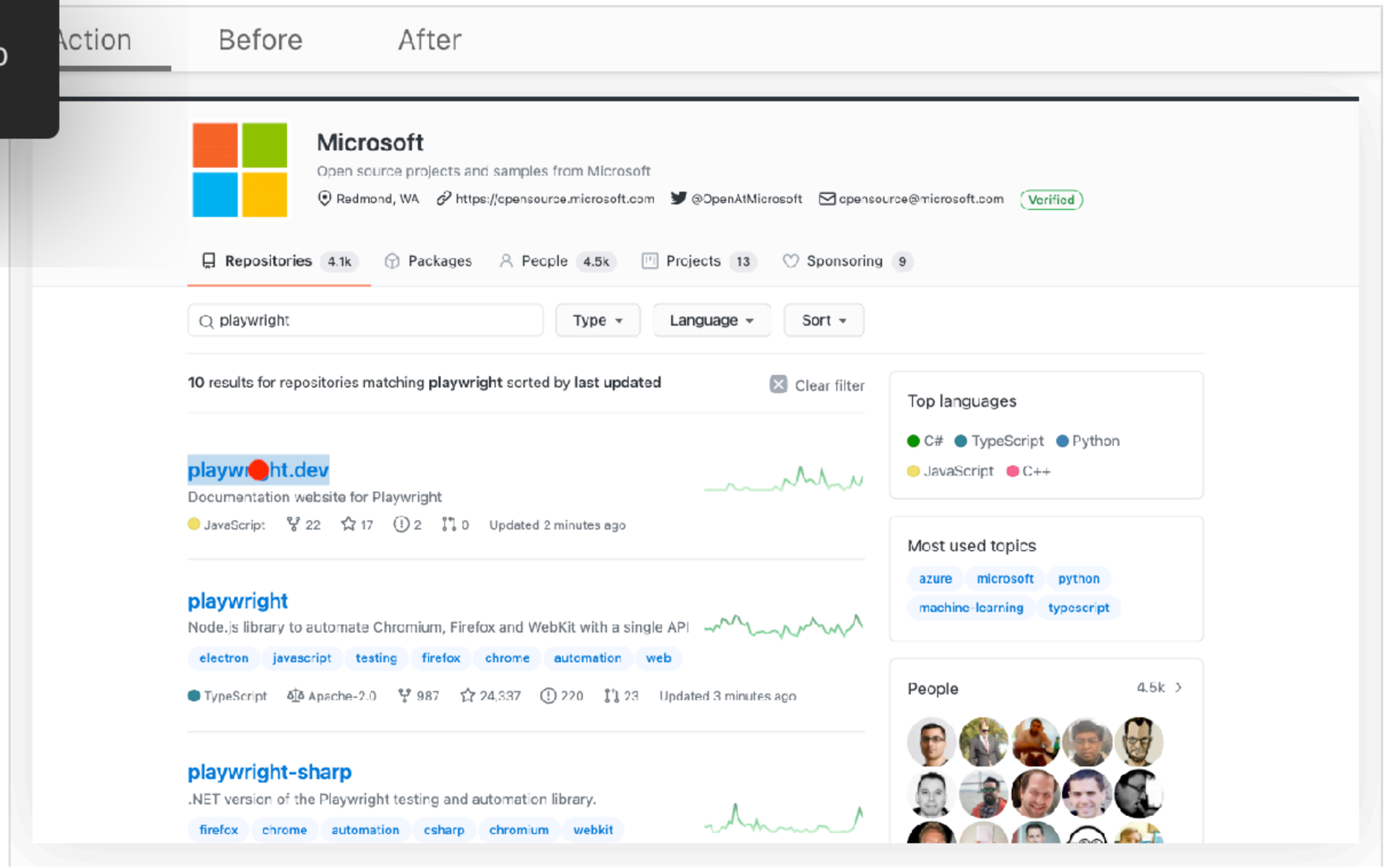
Viewing a trace

You can open the saved trace using Playwright CLI or in your browser on trace.playwright.dev

```
playwright show-trace trace.zip
```

When tracing with the snapshots option turned on, Playwright captures a set of complete DOM snapshots for each action. Notice how it highlights both, the DOM Node as well as the exact click position.

You can also open remote traces (produced by your CI, for example)



Continous Integration

Playwright tests can be executed in CI environments

```yaml
on:
  deployment_status:
jobs:
  test:
    needs: docker
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: test
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-dotnet@v1
        with:
          dotnet-version: 6.0.x
        run: dotnet restore
      - name: Install Playwright gobal tools
        run: dotnet tool install --global Microsoft.Playwright.CLI
        run: dotnet build --configuration Release --no-restore
      - name: Install playwright for project
        run: playwright install --with-deps
        run: dotnet test --no-restore --verbosity normal
        env:
          PLAYWRIGHT_TEST_BASE_URL: 'https://my-test-deployment.com'
```

Example of GitHub Action CI workflow

A note on GitHub Actions

I had to implement some try-retry logic

```
- name: test
    shell: bash --noprofile --norc {0}
    env:
      LC_ALL: en_US.utf8
    run: |
      counter=0
      exitcode=0
      reset="\e[0m"
      warn="\e[0;33m"
      while [ $counter -lt 6 ]
      do
          if [ $filter ]
          then
              echo -e "${warn}Retry $counter for $filter ${reset}"
          fi
          # run test and forward output also to a file in addition to stdout (tee command)
          dotnet test --no-build --filter=$filter | tee ./output.log
          # capture dotnet test exit status, different from tee
          exitcode=${PIPESTATUS[0]}
          if [ $exitcode == 0 ]
          then
              exit 0
          fi
          # cat outut, get failed test names, join as DisplayName=TEST with |, remove trailing |.
          filter=$(cat ./output.log | grep -o -P '(?<=\sFailed\s)\w*' | awk 'BEGIN { ORS="|" } { print("DisplayName=" $0) }' | grep -o -P
'.*(?=\|$)')
          ((counter++))
      done
      exit $exitcode
```

https://www.cazzulino.com/dotnet-test-retry.html

Feature and performance comparison

OK, I'll shortcut on this one

- Puppeteer, Selenium, Playwright, Cypress – how to choose?
  https://bit.ly/3h6NtIU
- Cypress vs Selenium vs Playwright vs Puppeteer speed comparison
  https://bit.ly/3LNmFvD
- Playwright Vs. Puppeteer Vs. Selenium: What are the differences?
  https://bit.ly/3I6qNEB

That’s all folks!

Find me at my website nicolaiarocci.com or @nicolaiarocci

Nicola Iarocci MVP